МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: «**Розробка та проєктування системи управління подіями з елементами штучного інтелекту»**

Виконав: студент 4 курсу, групи КН-42
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Пильгун Артем Юрійович

Керівник: *викладач кафедри інформаційних технологій та аналітики даних*
Мацевич Денис Володимирович

Рецензент: *кандидат технічних наук, доцент, доцент кафедри прикладної математики Донецького національного університету імені Василя Стуса*
Загоруйко Любов Василівна

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ

Завідувач кафедри інформаційних технологій та аналітики даних _______________
(проф., д.е.н. Кривицька О.Р.)

Протокол № 11 від « 20 » травня 2026 р.

Острог, 2026

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Мета кваліфікаційної роботи полягає в розробці повнофункціональної веб-платформи для управління подіями з інтегрованими елементами штучного інтелекту, яка забезпечує повний цикл взаємодії між організаторами та учасниками подій із сучасним рівнем користувацького досвіду, безпеки та продуктивності, орієнтованої на потреби українського ринку.

Основні завдання роботи включають:

- проведення аналізу існуючих платформ управління подіями та визначення їхніх переваг і недоліків;
- розробку функціональних та нефункціональних вимог до системи у розрізі трьох ролей користувачів (учасник, організатор, адміністратор);
- проектування реляційної бази даних у PostgreSQL із застосуванням строго типізованих ідентифікаторів та конвенції snake_case;
- реалізацію backend-частини на платформі .NET 10 з використанням архітектурного підходу Clean Architecture, патерну CQRS на базі бібліотеки MediatR, валідації через FluentValidation та контрольованої обробки помилок через ErrorOr;
- інтеграцію зовнішнього сервісу автентифікації Clerk з підтримкою SSO та автоматичним створенням внутрішнього профілю користувача;
- інтеграцію української платіжної системи LiqPay для обробки оплати квитків та формування виплат організаторам;
- реалізацію інтелектуального асистента на основі Microsoft Semantic Kernel з чотирма функціональними плагінами та підтримкою Function Calling для природномовного керування подіями;
- впровадження сповіщень у реальному часі та стрімінгу AI-відповідей за допомогою технології SignalR;

- реалізацію зберігання медіафайлів у об'єктному сховищі Cloudflare R2 через S3-сумісний API;
- розробку frontend-частини на фреймворку Next.js 16 з App Router, мовою TypeScript та управлінням серверним станом через TanStack Query;
- реалізацію ключових модулів системи, таких як реєстрація та автентифікація користувачів, створення, публікація та модерація подій, продаж електронних квитків з онлайн-оплатою, реєстрація учасників та формування квитків з QR-кодами, підписки для організаторів, агрегація та виплати коштів організаторам, AI-асистент для управління подіями;
- проведення модульного та інтеграційного тестування основних функціональних блоків з використанням Testcontainers;
- налаштування CI/CD pipeline через GitHub Actions з автоматизованим тестуванням web-доступності;
- розробку сучасного адаптивного інтерфейсу користувача на базі дизайн-системи, створеної у Figma;
- забезпечення безпеки, доступності та масштабованості рішення.

У роботі передбачається використання технологій: .NET 10, ASP.NET Core, Entity Framework Core, PostgreSQL, MediatR, FluentValidation, Microsoft Semantic Kernel, SignalR, Hangfire, Serilog на backend та Next.js 16 з TypeScript, React 19, Tailwind CSS, shadcn/ui, TanStack Query, React Hook Form, Zod на frontend. Інтеграції зовнішніх сервісів: Clerk (автентифікація), LiqPay (платежі), Cloudflare R2 (зберігання файлів), Groq Cloud (LLM-провайдер). Особлива увага приділяється Clean Architecture, інтеграційному тестуванню, автоматизованому тестуванню доступності та якості коду.

АНОТАЦІЯ
кваліфікаційної роботи
на здобуття освітнього ступеня бакалавра

Тема: Розробка та проєктування системи управління подіями з елементами штучного інтелекту.

Автор: Пильгун Артем Юрійович

Науковий керівник: викладач, фахівець-практик, Денис Володимирович Мацевич

Захищена «………»………………… 2026 року.

Пояснювальна записка до кваліфікаційної роботи: *191 с., 33 рис., 3 табл., 9 додатків, 31 джерело, 22 лістинги коду.*

Ключові слова: управління подіями, штучний інтелект, чиста архітектура, CQRS, Semantic Kernel, .NET 10, Next.js, SignalR, LiqPay, Clerk.

Короткий зміст праці:
Метою кваліфікаційної роботи є проєктування та розробка вебзастосунку для управління подіями «EventFlow» з інтегрованими елементами штучного інтелекту. Система забезпечує повний цикл роботи з подіями - створення, редагування, реєстрацію учасників, продаж електронних квитків з онлайн-оплатою, наявність чат-асистента на основі великих мовних моделей та обмін фото після завершення події. Серверна частина реалізована мовою C# на платформі .NET 10 з використанням архітектурного підходу Clean Architecture, патерну CQRS на базі бібліотеки MediatR та об’єктно-реляційного відображення Entity Framework Core поверх СКБД PostgreSQL. Клієнтська частина побудована на фреймворку Next.js 16 з App Router та мовою TypeScript. Для інтеграції штучного інтелекту застосовано Microsoft Semantic Kernel з чотирма функціональними плагінами. Автентифікація користувачів реалізована через зовнішній сервіс Clerk, обробка платежів - через українську платіжну систему LiqPay, доставка повідомлень у реальному часі - через SignalR. Результатом роботи є готова до експлуатації система, що відповідає сучасним підходам до проєктування програмного забезпечення.

____

ANNOTATION
of qualification paper
for bachelor's degree

Title: Development and Design of an Event Management System Incorporating Artificial Intelligence.

Author: Artem Yuriyovych Pylhun

Scientific advisor: Lecturer, Practicing Specialist, Denys Volodymyrovych Matsevych.

Defended «………»………………… 2025.

Explanatory note to the qualification thesis: 191 pages, 33 figures, 3 tables, 9 appendices, 31 references, 22 code listings.

Keywords: EVENT MANAGEMENT, ARTIFICIAL INTELLIGENCE, CLEAN ARCHITECTURE, CQRS, SEMANTIC KERNEL, .NET 10, NEXT.JS, SIGNALR, LIQPAY, CLERK

Abstract:

The aim of the qualification work is to design and develop a web application "EventFlow" for event management with integrated artificial intelligence features. The system covers the full lifecycle of events - creation, editing, attendee registration, electronic ticket sales with online payment processing, an AI chat assistant powered by large language models, and post-event photo sharing. The backend is implemented in C# on the .NET 10 platform using Clean Architecture, the CQRS pattern via the MediatR library, and Entity Framework Core ORM over PostgreSQL. The frontend is built on the Next.js 16 framework with App Router and TypeScript. Artificial intelligence is integrated through Microsoft Semantic Kernel with four functional plugins. User authentication is handled by the external Clerk service, payment processing by the Ukrainian LiqPay system, and real-time message delivery by SignalR. The result is a production-ready system that follows modern software design principles.

____

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ........ 8
ВСТУП........ 10
РОЗДІЛ 1 ЗАГАЛЬНІ ПОЛОЖЕННЯ........ 13
1.1 Опис предметного середовища........ 13
1.2 Огляд наявних аналогів........ 16
1.3 Постановка задачі........ 21
Висновки до розділу 1........ 24
РОЗДІЛ 2 ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ........ 25
2.1 Аналіз предметної області........ 25
2.2 Проєктування системи........ 27
2.2.1 Архітектурний шаблон Clean Architecture........ 27
2.2.2 Доменний шар та строго типізовані ідентифікатори........ 29
2.2.3 Шар Application: CQRS на базі MediatR........ 30
2.2.4 Шар Infrastructure: репозиторії та зовнішні сервіси........ 32
2.2.5 Шар API........ 33
2.2.6 Проєктування бази даних........ 34
2.2.7 Архітектура клієнтської частини........ 37
2.3 Алгоритмічне забезпечення........ 38
2.3.1 Алгоритм перевірки доступності квитків........ 38
2.3.2 Алгоритм формування фінальної ціни квитка........ 39
2.3.3 Алгоритм роботи AI-асистента на основі Semantic Kernel........ 39
2.3.4 Алгоритм перевірки сигнатури callback'у LiqPay........ 42
Висновки до розділу 2........ 42
РОЗДІЛ 3 ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ........ 44
3.1 Засоби розробки........ 44
3.2 Вимоги до технічного та програмного забезпечення........ 46
3.3 Опис програмної реалізації серверної частини........ 47
3.3.1 Структура рішення........ 47
3.3.2 Реалізація доменного шару........ 48
3.3.3 Реалізація шару Application........ 49
3.3.4 Реалізація шару Infrastructure........ 50
3.3.5 Реалізація API-контролерів........ 51
3.3.6 Інтеграція автентифікації через Clerk........ 52
3.3.7 Інтеграція платіжної системи LiqPay........ 54

3.3.8 AI-функціональність на базі Microsoft Semantic Kernel................................ 57
3.3.9 Real-time комунікація через SignalR..59
3.3.10 Зберігання файлів в Cloudflare R2.. 62
3.3.11 Виплати організаторам подій.. 63
3.3.12 Відновлення пароля та багатофакторна автентифікація...............................65
3.4 Опис програмної реалізації клієнтської частини.. 67
3.4.1 Стек і обґрунтування технологічних виборів.. 68
3.4.2 Процес дизайну: Figma та Figma Make.. 68
3.4.3 Кольорова палітра.. 71
3.4.4 Типографіка.. 73
3.4.5 Структура клієнтського проєкту... 74
3.4.6 Ключові реалізації клієнтської частини.. 75
3.4.7 Управління станом та оптимізації.. 81
3.5 Тестування та керівництво користувача..82
3.5.1 Тестування.. 82
3.5.2 Керівництво користувача..86
3.6 Розгортання застосунку та CI/CD.. 91
3.6.1 Контейнеризація серверної частини..91
3.6.2 Локальне розгортання інфраструктури через docker-compose.......................91
3.6.3 CI/CD pipeline через GitHub Actions...92
3.6.4 Тестування web-accessibility у CI/CD..94
3.6.5 Розгортання в production..97
Висновки до розділу 3...99
ВИСНОВКИ...101
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ...104
ДОДАТОК А..108
ДОДАТОК Б... 114
ДОДАТОК В...122
ДОДАТОК Г.. 124
ДОДАТОК Д...138
ДОДАТОК Е.. 143
ДОДАТОК Ж...160
ДОДАТОК Й...167
ДОДАТОК К...181

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

Скорочення	Розшифровка
AI	Artificial Intelligence (штучний інтелект)
API	Application Programming Interface
CQRS	Command Query Responsibility Segregation
CRUD	Create, Read, Update, Delete
DDD	Domain-Driven Design
DI	Dependency Injection
DTO	Data Transfer Object
EF	Entity Framework
ER	Entity-Relationship
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Token
LLM	Large Language Model
ORM	Object-Relational Mapping
REST	Representational State Transfer
SDK	Software Development Kit
SK	Semantic Kernel
SPA	Single Page Application
SSO	Single Sign-On
SSR	Server-Side Rendering

UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
UX	User Experience
WCAG	Web Content Accessibility Guidelines
БД	База даних
ПЗ	Програмне забезпечення
СКБД	Система керування базами даних

ВСТУП

Життя дедалі більше переходить в онлайн, тому організація подій - від невеликих мітапів до масштабних конференцій - потребує зручних і швидких цифрових інструментів. Організаторам важливо мати змогу легко створити захід, залучити аудиторію, продати квитки та постійно підтримувати зв'язок з учасниками. Відомі світові платформи, такі як Eventbrite, Meetup або Luma, пропонують багато можливостей, але вони слабко адаптовані для українського ринку. Серед головних недоліків: відсутність зручної інтеграції з нашими платіжними системами, високі комісії, неповна локалізація та брак сучасних інструментів автоматизації, які б зняли з організатора рутинну роботу.

Водночас розвиток штучного інтелекту, зокрема великих мовних моделей (LLM) та рішень на кшталт Microsoft Semantic Kernel, кардинально змінює спосіб взаємодії людини з програмами. Замість того, щоб переходити між десятками вікон і заповнювати складні форми, користувач може просто описати свій запит природною мовою - і система виконає задачу. Впровадження таких розумних помічників у платформу для управління подіями значно знижує поріг входу для новачків і суттєво економить час досвідченим фахівцям.

Саме тому метою цієї роботи є розробка власної системи управління подіями під назвою «EventFlow». Вона поєднує в собі класичний функціонал event-платформ, але доповнена елементами штучного інтелекту, адаптована під українські платіжні системи та має зручні інструменти для спілкування між організаторами й учасниками в реальному часі.

Мета роботи - спроєктувати та розробити вебзастосунок «EventFlow», що забезпечує повний цикл управління подіями з інтегрованими елементами штучного інтелекту, орієнтований на потреби українського ринку.

Задачі дослідження:

1. Провести аналіз предметної області організації подій та визначити ключові бізнес-процеси.
2. Дослідити існуючі рішення на ринку, виявити їх сильні та слабкі сторони, сформувати конкурентні переваги розроблюваної системи.
3. Спроєктувати архітектуру системи на основі підходів Clean Architecture та CQRS.
4. Спроєктувати модель даних та реалізувати її засобами PostgreSQL та Entity Framework Core.
5. Реалізувати серверну частину на платформі .NET 10 з інтеграціями зовнішніх сервісів (Clerk, LiqPay, Cloudflare R2, Semantic Kernel).
6. Реалізувати клієнтську частину, використовуючи фреймворк Next.js 16 з дизайном, створеним у Figma з використанням Figma Make.
7. Реалізувати AI-асистента та доставку повідомлень у реальному часі через SignalR.
8. Покрити критичну функціональність автоматизованими тестами.

Об'єкт дослідження - процес організації, проведення та монетизації подій із залученням сучасних інформаційних технологій.

Предмет дослідження - інструментальні засоби та архітектурні підходи (Clean Architecture, CQRS, MediatR, Microsoft Semantic Kernel, Next.js, SignalR, інтеграція з платіжною системою LiqPay) для побудови системи управління подіями з елементами штучного інтелекту.

Методи дослідження: системний аналіз для декомпозиції предметної області, об'єктно-орієнтоване проєктування для побудови моделі домену, UML-моделювання

для візуалізації архітектури, експериментальне тестування для верифікації коректності реалізації, порівняльний аналіз для оцінки конкурентного середовища.

Практична цінність роботи полягає у створенні готового до експлуатації програмного продукту, який може бути впроваджений як приватними організаторами подій, так і компаніями, що проводять корпоративні заходи. Архітектура системи є модульною та масштабованою, що дозволяє її подальший розвиток і адаптацію під специфічні потреби замовника.

Структура роботи відповідає вимогам методичних рекомендацій до виконання кваліфікаційної роботи на здобуття освітнього ступеня бакалавра. Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. У першому розділі розглядаються теоретичні засади предметної області, проводиться огляд аналогів та формулюється постановка задачі. Другий розділ присвячено інформаційному та математичному забезпеченню - аналізу даних, проєктуванню архітектури та бази даних, опису ключових алгоритмів. У третьому розділі описана програмна реалізація серверної та клієнтської частин, інтеграції зовнішніх сервісів та результати тестування.

РОЗДІЛ 1
ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1 Опис предметного середовища

Управління подіями (event management) - це окрема галузь, що охоплює планування, організацію, проведення та аналіз результатів заходів різного масштабу: від невеликих зустрічей і воркшопів до масштабних конференцій та фестивалів. Незалежно від типу заходу, організатор стикається з типовим набором задач: оформити інформаційну сторінку події, зібрати реєстрації, опрацювати оплати, повідомити учасників про зміни, провести облік присутніх та зібрати зворотний зв'язок після завершення.

З розвитком цифрових технологій усі ці процеси переходять в онлайн. За даними дослідження ринку event-management платформ, обсяг світового ринку у 2024 році становив понад 8 мільярдів доларів США, з прогнозованим середньорічним зростанням понад 13% протягом наступних п'яти років. Окремою тенденцією стає інтеграція технологій штучного інтелекту в платформи такого типу - для автоматичного підбору подій під інтереси користувача, генерації описів, рекомендацій оптимального часу проведення тощо.

З точки зору домену, ключовими бізнес-сутностями системи управління подіями є:

- **Подія (Event)** - основна одиниця системи, що містить інформацію про захід: назву, опис, дату й час початку та завершення, місце проведення (фізичну адресу або посилання для онлайн-події), кількість доступних місць, ціну, обкладинку, статус публікації та зв'язок з організатором.
- **Реєстрація (Registration)** - факт того, що конкретний користувач зареєструвався на конкретну подію. Реєстрація містить інформацію про дату подачі, статус (очікує оплати, підтверджена, скасована), посилання на квиток.

- **Квиток (Ticket)** - електронний документ, що засвідчує право участі в платній події. Квиток зберігає інформацію про оплату, статус (оплачений, повернений, використаний), унікальний ідентифікатор для перевірки та QR-код.
- **Користувач (User)** - особа, що взаємодіє з системою. Може мати ролі: учасник (Attendee), організатор (Organizer), адміністратор (Admin).
- **Підписка (Subscription)** - інструмент монетизації для організаторів, що надає розширені можливості (більше подій, додаткова аналітика, AI-функції) на основі тарифного плану.
- **Фотогалерея події (Event Photos)** - колекція зображень, завантажених організатором під час створення події.

Життєвий цикл події можна описати наступною послідовністю станів: **Draft (чернетка) → Published (опублікована) → Registration Open (відкрита реєстрація) → In Progress (триває) → Completed (завершена) → Archived (архівована)**. На кожному з цих етапів виконуються специфічні дії:

- У стані «Чернетка» організатор може редагувати опис та налаштування без обмежень;
- З моменту публікації зміни вимагають додаткового підтвердження;
- Завершена подія більше не приймає реєстрацій, але дозволяє завантаження фото;
- Архівна подія недоступна для нових дій, але зберігається для історії.

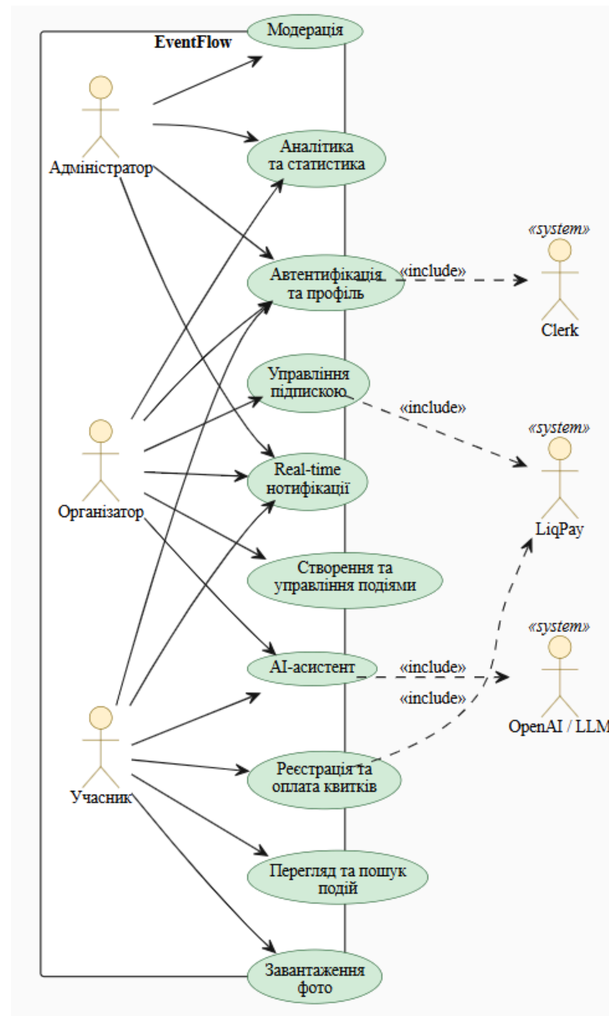

Рис. 1.1. Діаграма використання UML системи EventFlow

Джерело: розроблено автором

З точки зору регуляторного середовища, при роботі з подіями необхідно враховувати вимоги Закону України «Про захист персональних даних», який регулює обробку інформації про учасників (ім’я, контактні дані, дані оплати). Окремою важливою вимогою є відповідність стандарту PCI DSS при обробці платіжних даних - у даній роботі цю вимогу делеговано платіжній системі LiqPay, що дозволяє розроблюваному застосунку не зберігати дані карток і не проходити власну сертифікацію.

Розвиток сфери штучного інтелекту, зокрема поява великих мовних моделей (LLM) на кшталт GPT, Claude, Gemini та інструментів типу Microsoft Semantic Kernel,

дозволяє якісно будувати взаємодію з системами керування подіями. Замість того, щоб користувач самостійно проходив через серію форм для створення події, він може звернутися до AI-асистента природною мовою: «Створи мені подію на 50 людей про React-розробку наступної суботи в Києві о 14:00». Асистент ставить уточнюючі питання, заповнює потрібні поля та створює подію - все за одну розмову. Аналогічно AI допомагає шукати події за неточними критеріями або виконувати масові операції типу скасування реєстрації.

1.2 Огляд наявних аналогів

На ринку event-management платформ існує значна кількість рішень, що відрізняються за функціональністю, цільовою аудиторією та регіональним фокусом. Розглянемо основних конкурентів розроблюваної системи EventFlow.

Eventbrite - глобальний лідер у сфері продажу квитків на події, заснований у 2006 році. Платформа підтримує події будь-якого масштабу - від невеликих майстер-класів до масштабних конференцій з десятками тисяч учасників. Серед переваг - велика база користувачів, інтеграції з популярними сервісами (Zoom, Slack, MailChimp), гнучкі налаштування квитків, аналітика. Основні недоліки з точки зору українського користувача: висока комісія (близько 3,5% + фіксована плата за квиток), відсутність нативної інтеграції з українськими платіжними системами, неповна локалізація українською мовою, відсутність AI-асистента для управління подіями. Інтерфейс часто перенавантажений рекламою інших подій, що відволікає від цілі користувача.

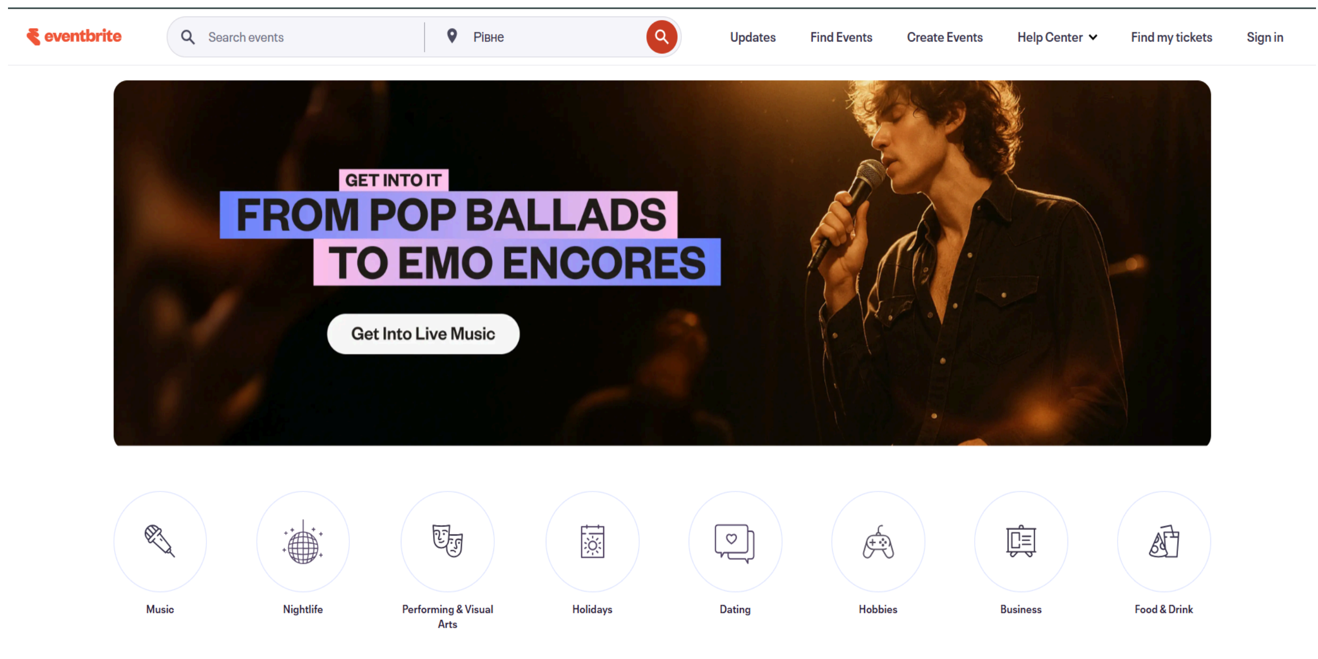

Рис. 1.2. UI-дизайн платформи Eventbrite

Джерело: https://www.eventbrite.com/

Luma - сучасна платформа з фокусом на спільнотні події, тех-конференції та зустрічі стартапів. Відрізняється мінімалістичним та елегантним інтерфейсом, швидкістю роботи, інтеграцією з Calendar додатками. Серед переваг - простота створення подій, якісні email-комунікації, безкоштовний базовий тариф. Проте Luma не підтримує оплату в гривнях через локальні платіжні системи (тільки Stripe з картками міжнародного зразка), не має повноцінного AI-асистента, обмежено локалізована українською мовою.

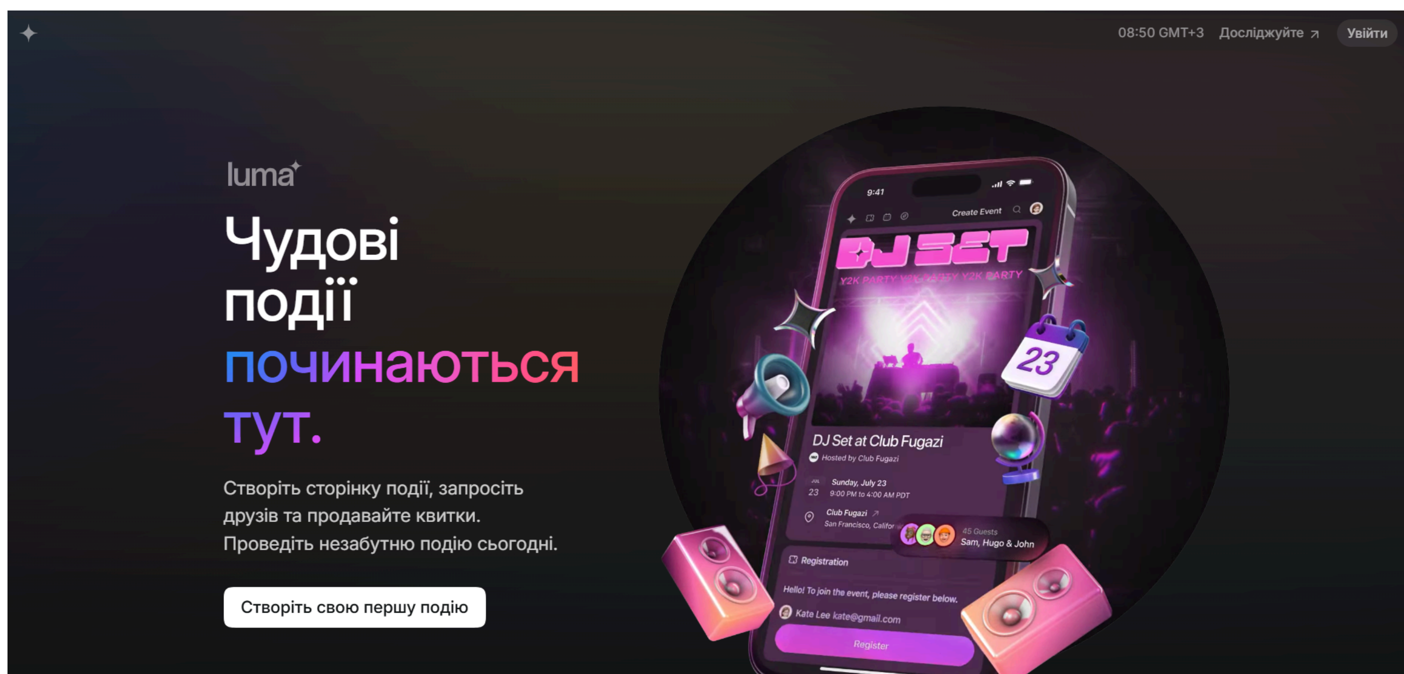

Рис. 1.3. UI-дизайн платформи Luma

Джерело: https://luma.com

Concert.ua, Karabas та подібні українські аґрегатори квитків - це не платформи самостійного створення подій, а маркетплейси, де організатор може розмістити свій захід після проходження модерації. Вони добре підходять для масових концертів та офіційних заходів, але не для невеликих мітапів, воркшопів чи приватних подій. Створити подію самостійно за кілька хвилин неможливо.

На основі проведеного огляду сформовано порівняльну таблицю функціональності існуючих рішень та розроблюваної системи EventFlow.

Таблиця 1.1

Порівняння функціоналу EventFlow з аналогами

Функція	EventFlow	Eventbrite	Luma	Meetup
Самостійне створення події за хвилини	+	+	+	+

Українська локалізація	+	частково	-	-
Оплата в UAH через LiqPay	+	-	-	-
AI-асистент керування подіями	+	-	-	-
Real-time нотифікації	+	+	+	-
Фотогалерея події	+	-	+	+
Підписки для організаторів	+	+ (платні плани)	+	+
Безкоштовний базовий план	+	+	+	-
Інтеграція з Clerk SSO	+	-	+ (Google)	-
Open-source / самостійне розгортання	+	-	-	-

Джерело: складено автором на основі офіційної документації згаданих платформ

На основі проведеного аналізу побудовано SWOT-аналіз розроблюваної системи EventFlow.

Таблиця 1.2

SWOT-аналіз системи EventFlow

Сильні сторони	**Слабкі сторони**
Інтеграція з LiqPay для українського ринку	Відсутність початкової бази користувачів
AI-асистент на базі Semantic Kernel	Обмежені маркетингові ресурси
Сучасна архітектура (Clean Architecture, CQRS)	Залежність від зовнішніх сервісів (Clerk, AI-провайдер)
Real-time комунікація через SignalR	Відсутність мобільного застосунку на старті
Адаптивний UI на базі Next.js та Tailwind	
Можливості	**Загрози**
Зростання попиту на офлайн-події після пандемії	Поява нових глобальних конкурентів з AI
Перехід організаторів від глобальних до локальних платформ	Зміна тарифної політики LiqPay або Clerk
Розширення на європейський ринок	Економічна нестабільність, що зменшує бюджети на події
Інтеграція з державними сервісами (Дія)	Зміни в законодавстві щодо персональних даних

Джерело: складено автором

Проведений аналіз дозволяє зробити висновок, що ніша системи з нативною інтеграцією платіжного інструментарію для України та повноцінним AI-асистентом залишається не повністю заповненою. Це створює ринкові можливості для впровадження EventFlow.

1.3 Постановка задачі

На основі аналізу предметної області та огляду наявних рішень сформульовано вимоги до розроблюваної системи EventFlow.

Функціональні вимоги структуровано за ролями користувачів.

Учасник (Attendee):

- реєстрація та автентифікація через Clerk (email, Google, інші провайдери);
- перегляд каталогу опублікованих подій з фільтрами за категорією, датою, локацією, ціною;
- детальний перегляд інформації про подію (опис, програма, організатор, локація на мапі);
- реєстрація на безкоштовну подію;
- купівля квитка на платну подію через LiqPay з підтримкою Apple Pay, Google Pay, банківських карток;
- скасування реєстрації з поверненням коштів (відповідно до політики організатора);
- перегляд особистого кабінету з історією подій та квитків;
- завантаження фото в галерею після події;
- спілкування з AI-асистентом для пошуку подій та інших запитів;
- отримання real-time нотифікацій про важливі події (підтвердження реєстрації, нагадування).

Організатор (Organizer):

- усі можливості учасника;
- створення подій з повним набором параметрів (назва, опис, дати, локація, ціна, ліміт учасників, обкладинка);
- редагування та видалення власних подій;
- перегляд списку зареєстрованих учасників;
- доступ до аналітичної панелі з показниками подій (продажі, відсоток продажів, середній чек);
- управління підпискою (вибір тарифу, оплата, скасування);
- використання AI-асистента для виконання адміністративних дій (створення, редагування, аналіз).

Адміністратор (Admin):

- усі можливості організатора;
- перегляд глобальної статистики системи;
- управління користувачами та ролями;
- отримання нотифікацій про підозрілу активність та масові реєстрації.

Нефункціональні вимоги:

- ***Продуктивність***: середній час відповіді API-ендпоінтів - менше 300 мс при типовому навантаженні; підтримка мінімум 100 одночасних користувачів без деградації;
- ***Безпека***: автентифікація через Clerk SSO; авторизація на основі ролей; делегування обробки платіжних даних LiqPay для відповідності PCI DSS; зберігання чутливої конфігурації у змінних середовища, а не в коді;
- ***Масштабованість***: stateless архітектура серверної частини, що дозволяє горизонтальне масштабування; винесення сесій SignalR у Redis при потребі; зберігання файлів у Cloudflare R2 без прив’язки до сервера застосунку;

- ***Доступність***: адаптивний інтерфейс, що коректно працює на десктопі, планшеті, смартфоні; відповідність стандарту WCAG AA для контрастності та клавіатурної навігації;
- ***Підтримуваність***: чітка модульна архітектура (Clean Architecture); покриття критичної функціональності тестами; типізований код на TypeScript та C# з нульовими попередженнями компілятора;
- ***Локалізація***: інтерфейс українською та англійською мовами;
- ***Сумісність***: підтримка сучасних браузерів Chrome, Firefox, Safari, Edge у двох останніх стабільних версіях.

Обмеження проєкту:

- використання безкоштовних тарифних планів зовнішніх сервісів (Clerk Free, LiqPay тестове середовище для розробки, Cloudflare R2 у безкоштовній квоті);
- монолітна архітектура серверної частини без розбиття на мікросервіси (для зменшення складності розгортання та підтримки на етапі розробки);
- розгортання у вигляді одного Docker-контейнера серверної частини плюс окремого контейнера БД;
- мінімальна інтеграція з третіми сервісами поза переліком (без автоматичних email-розсилок, push-нотифікацій на пристрій тощо).

Критерії успіху проєкту:

1. Користувач може успішно зареєструватися, створити подію, опублікувати її та отримати реєстрацію іншого користувача.
2. Платіжний flow «купівля квитка → callback від LiqPay → видача квитка» проходить без помилок.
3. AI-асистент коректно виконує мінімум п’ять типових сценаріїв (створення події, пошук, реєстрація, скасування, статистика).

4. Real-time нотифікація про реєстрацію доставляється організатору в межах 1 секунди.
5. Загальне покриття тестами критичних шляхів - не менше 60%.

Висновки до розділу 1

У першому розділі проведено комплексний аналіз предметної області event-management. Описано ключові сутності та бізнес-процеси, що мають бути реалізовані в системі: подія, реєстрація, квиток, користувач, підписка, фотогалерея.

Здійснено огляд трьох основних конкурентів - Eventbrite, Luma, Meetup - та виділено їх сильні і слабкі сторони. На основі порівняльного аналізу та SWOT-аналізу обґрунтовано конкурентні переваги розроблюваної системи EventFlow: інтеграція з LiqPay, повноцінний AI-асистент, сучасний стек технологій та орієнтація на український ринок.

Сформульовано функціональні та нефункціональні вимоги до системи у розрізі трьох ролей користувачів (Attendee, Organizer, Admin), визначено обмеження проєкту та критерії успіху. Постановка задачі є основою для проєктування архітектури в наступному розділі.

РОЗДІЛ 2
ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1 Аналіз предметної області

На основі сформованих у першому розділі вимог виділено ключові інформаційні сутності системи EventFlow та потоки даних між ними. Доменна модель складається з дев'яти основних сутностей, кожна з яких відповідає певному аспекту бізнес-логіки: Event (подія), EventRegistration (реєстрація на подію), Ticket (квиток), User (користувач), Role (роль), UserRole (зв'язок користувача з роллю), Subscription (підписка), EventPhoto (фото події) та ChatConversation (історія розмов з AI-асистентом).

Вхідні дані системи надходять з кількох джерел:

1. *Дані від користувача через UI*: форма створення/редагування події (назва, опис, дати, локація, ціна, ліміт учасників, обкладинка), форма реєстрації на подію, форма редагування профілю, текстові повідомлення в AI-чат, файли зображень для галереї.
2. *Дані від зовнішнього сервісу автентифікації Clerk*: JWT-токен у заголовку запиту, ClerkUserId як стійкий ідентифікатор користувача, профільна інформація (email, ім'я, аватар) під час першого входу.
3. *Дані від платіжної системи LiqPay*: webhook-callback після успішної або неуспішної транзакції, що містить статус платежу, ідентифікатор замовлення, суму, валюту, сигнатуру для перевірки автентичності повідомлення.
4. *Файли мультимедіа*: зображення подій та фотогалерей, що завантажуються через multipart-форми та зберігаються в Cloudflare R2 з поверненням URL.

Вихідні дані системи включають:

- REST API відповіді у форматі JSON для синхронної комунікації з клієнтом;

- real-time повідомлення через WebSocket (SignalR) для асинхронних подій;
- редіректи на платіжний шлюз LiqPay для ініціації оплати;
- pre-signed URLs для прямого завантаження файлів у S3;
- стрімінг чанків відповіді AI-асистента у реальному часі.

Обмеження на дані реалізовані через шар валідації на базі бібліотеки FluentValidation:

- назва події - від 3 до 200 символів, обов’язкова;
- опис події - до 5000 символів;
- дата завершення події має бути пізніше за дату початку;
- ціна квитка - невід’ємна; для безкоштовних подій дорівнює нулю;
- максимальна кількість учасників - додатне ціле число;
- email - за RFC 5321;
- розмір завантаженого зображення - не більше 10 МБ;
- формати зображень - JPEG, PNG, WebP.

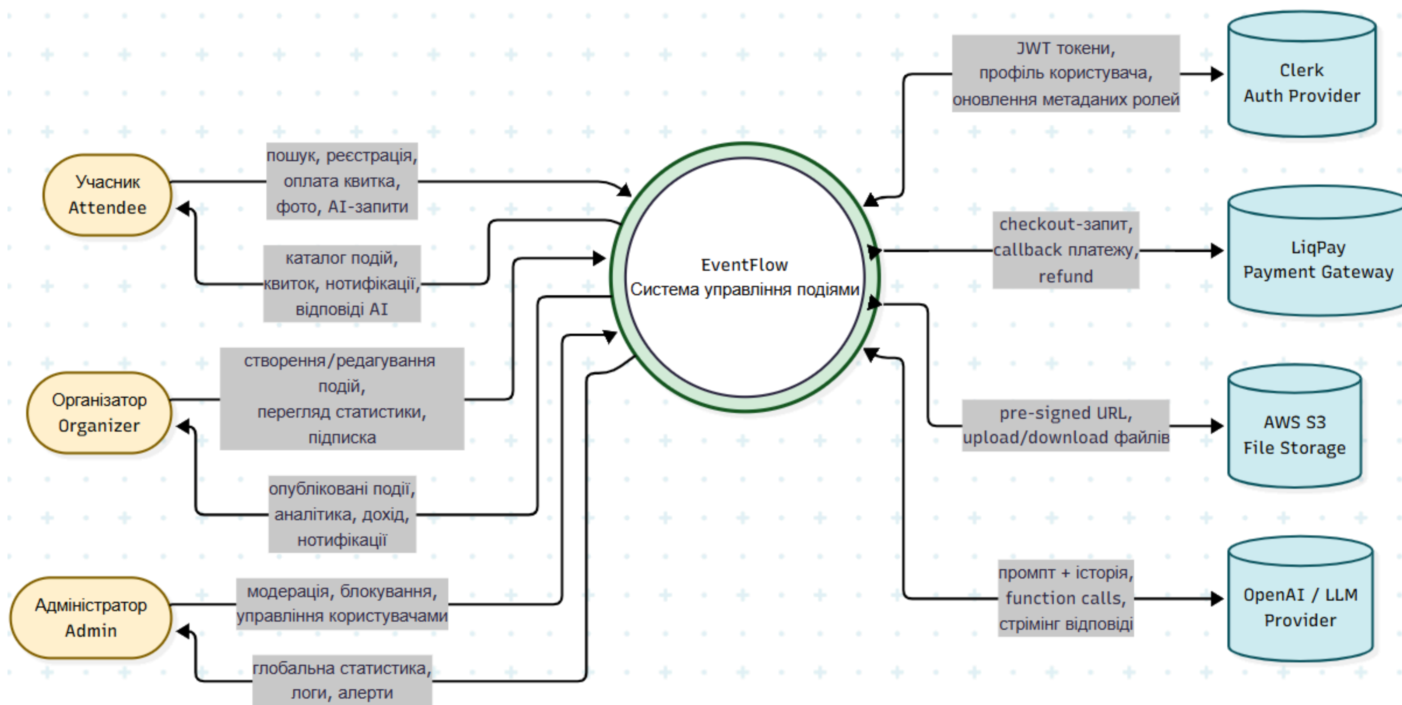

Рис. 2.1. Контекстна діаграма потоків даних системи EventFlow

Джерело: розроблено автором

2.2 Проєктування системи

2.2.1 Архітектурний шаблон Clean Architecture

Архітектурною основою серверної частини системи обрано підхід Clean Architecture, запропонований Робертом Мартіном. Згідно з цим підходом, програмний код організовується у концентричні шари, де внутрішні шари не залежать від зовнішніх. Такий підхід забезпечує:

- незалежність бізнес-логіки від фреймворків та зовнішніх сервісів;
- можливість тестування бізнес-правил без піднімання БД, веб-сервера чи зовнішніх API;
- легку заміну компонентів інфраструктури (наприклад, заміна LiqPay на Stripe не потребує змін у бізнес-логіці);
- чітке розмежування відповідальностей між частинами системи.

Архітектура EventFlow складається з чотирьох шарів:

1. **Domain** - внутрішнє ядро системи. Містить доменні сутності (Event, User, Ticket тощо), value objects (наприклад, EventId, TicketId - строго типізовані ідентифікатори), доменні події та інваріанти бізнес-правил. Цей шар не залежить ні від чого, окрім стандартної бібліотеки .NET.
2. **Application** - шар прикладної логіки. Містить use-cases у вигляді команд (Commands) та запитів (Queries) відповідно до патерну CQRS, обробники цих команд та запитів, валідатори, інтерфейси інфраструктурних сервісів (без реалізацій). Залежить тільки від Domain.
3. **Infrastructure** - шар реалізацій. Містить реалізацію репозиторіїв через Entity Framework Core, інтеграції з зовнішніми сервісами (Clerk, LiqPay, Cloudflare R2,

Semantic Kernel), реалізацію SignalR хабів, конфігурацію БД. Залежить від Application та Domain.

4. **API (Presentation)** - точка входу. Містить контролери ASP.NET Core, middleware, конфігурацію DI-контейнера, налаштування Swagger, маршрутизацію SignalR-хабів. Залежить від Application та Infrastructure.

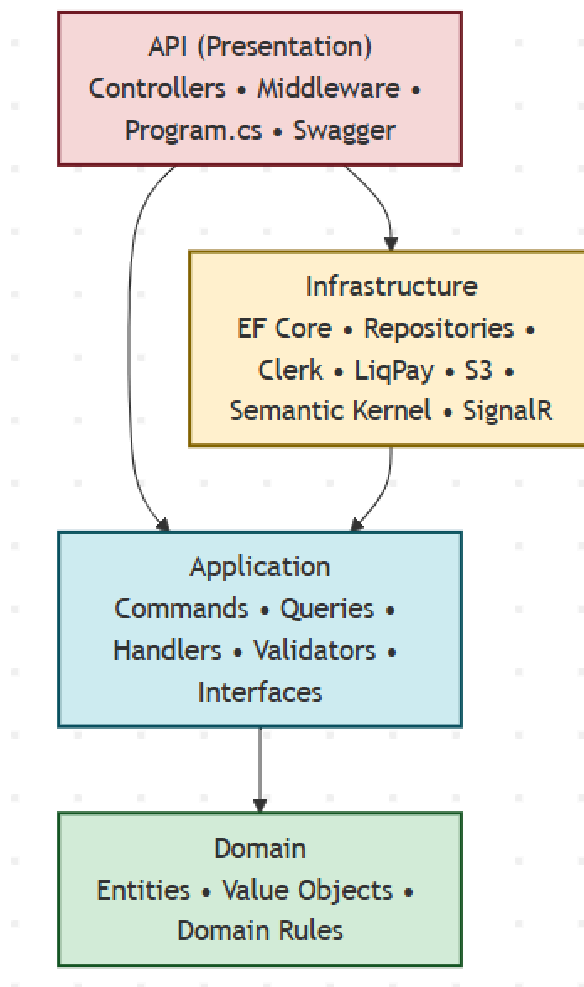

Рис. 2.2. Схема Clean Architecture системи EventFlow з чотирма шарами

Джерело: розроблено автором

Обґрунтування вибору Clean Architecture над альтернативами: класична трирівнева архітектура (UI / Business Logic / Data Access) недостатньо гнучка для проєкту з кількома зовнішніми інтеграціями та активною AI-функціональністю -

будь-яка зміна в інтеграційному шарі ризикує зачепити бізнес-логіку. Hexagonal Architecture концептуально подібна до Clean, але слабше поширена в .NET-екосистемі, що ускладнює онбординг нових розробників. Вертикальні слайси (Vertical Slice Architecture) ефективні для невеликих проєктів, але при зростанні кількості функцій починають дублювати код. Clean Architecture знаходить оптимальний баланс між строгістю розмежування та читабельністю.

2.2.2 Доменний шар та строго типізовані ідентифікатори

У доменному шарі реалізовано дев’ять основних сутностей. Однією з ключових проєктних рішень є використання строго типізованих ідентифікаторів замість сирих типів Guid або int. Кожна сутність має власний тип ідентифікатора у вигляді **readonly record struct**:

Лістинг коду 2.1 Приклад ідентифікатора для кожної сутності

```
public readonly record struct EventId(Guid Value);
public readonly record struct UserId(Guid Value);
public readonly record struct TicketId(Guid Value);
public readonly record struct RegistrationId(Guid Value);
public readonly record struct SubscriptionId(Guid Value);
```

Перевагою такого підходу є неможливість випадково передати, наприклад, UserId в метод, що очікує EventId - компілятор C# виявить помилку на етапі збірки. Це усуває цілий клас потенційних багів, які були б важко виявляються в продакшені.

Доменні сутності містять не тільки дані, а й бізнес-правила у вигляді методів. Наприклад, сутність Event має метод Publish(), який змінює статус події з Draft на Published з перевіркою попередніх умов (заповнені обов’язкові поля, дата початку в майбутньому). Сутність Ticket має методи MarkAsPaid(), Refund(), MarkAsUsed(), кожен з яких перевіряє можливість переходу зі поточного стану.

Повний код сутності "Event" з усіма доменними методами ("Publish", "Cancel", "AddPhoto", "UpdateDetails", "GetRegistrationCount") наведено у Додатку А.

2.2.3 Шар Application: CQRS на базі MediatR

У шарі Application застосовано патерн CQRS - розділення відповідальностей між командами (Commands), які змінюють стан системи, та запитами (Queries), які тільки читають дані. Для оркестрації застосовано бібліотеку MediatR, що реалізує патерн Mediator та надає інтерфейси IRequest<TResponse> для команд/запитів і IRequestHandler<TRequest, TResponse> для їх обробників.

Структура папок шару Application відображає бізнес-домени:

```
EventFlow.Application/
├── Common/
│   ├── Behaviors/      (pipeline behaviors)
│   ├── Interfaces/     (інтерфейси інфраструктурних сервісів)
│   └── Mappings/
├── Events/
│   ├── Commands/
│   │   ├── CreateEvent/
│   │   ├── UpdateEvent/
│   │   ├── PublishEvent/
│   │   ├── DeleteEvent/
│   │   ├── RegisterForEvent/
│   │   ├── CreateTicketCheckout/
│   │   └── ProcessTicketPaymentCallback/
│   └── Queries/
│       ├── GetEventById/
│       ├── GetEventsList/
│       └── GetMyEvents/
├── Subscriptions/
└── Users/
```

Рис. 2.3. Структура папок шару Application

Джерело: розроблено автором

Кожна команда інкапсульована у власній папці разом з її обробником, валідатором та DTO відповіді. Це дозволяє швидко знаходити пов’язаний код і не допускає розпорошення логіки за різними файлами.

Для реалізації бізнес-логіки з контрольованою обробкою помилок без використання винятків застосовано бібліотеку ErrorOr. Замість того, щоб кидати виняток при помилці бізнес-правила, обробник повертає ErrorOr<TResponse> - об’єднання типу «успішний результат або список помилок». Це підвищує читабельність та продуктивність (виняток у .NET - дорога операція).

Для крос-функціональних задач (валідація, логування) застосовані pipeline behaviors MediatR. ValidationBehavior автоматично запускає всі зареєстровані валідатори FluentValidation для команди перед викликом її обробника та повертає помилку валідації, якщо вона виникає. LoggingBehavior записує в логи назву команди, час її виконання та результат.

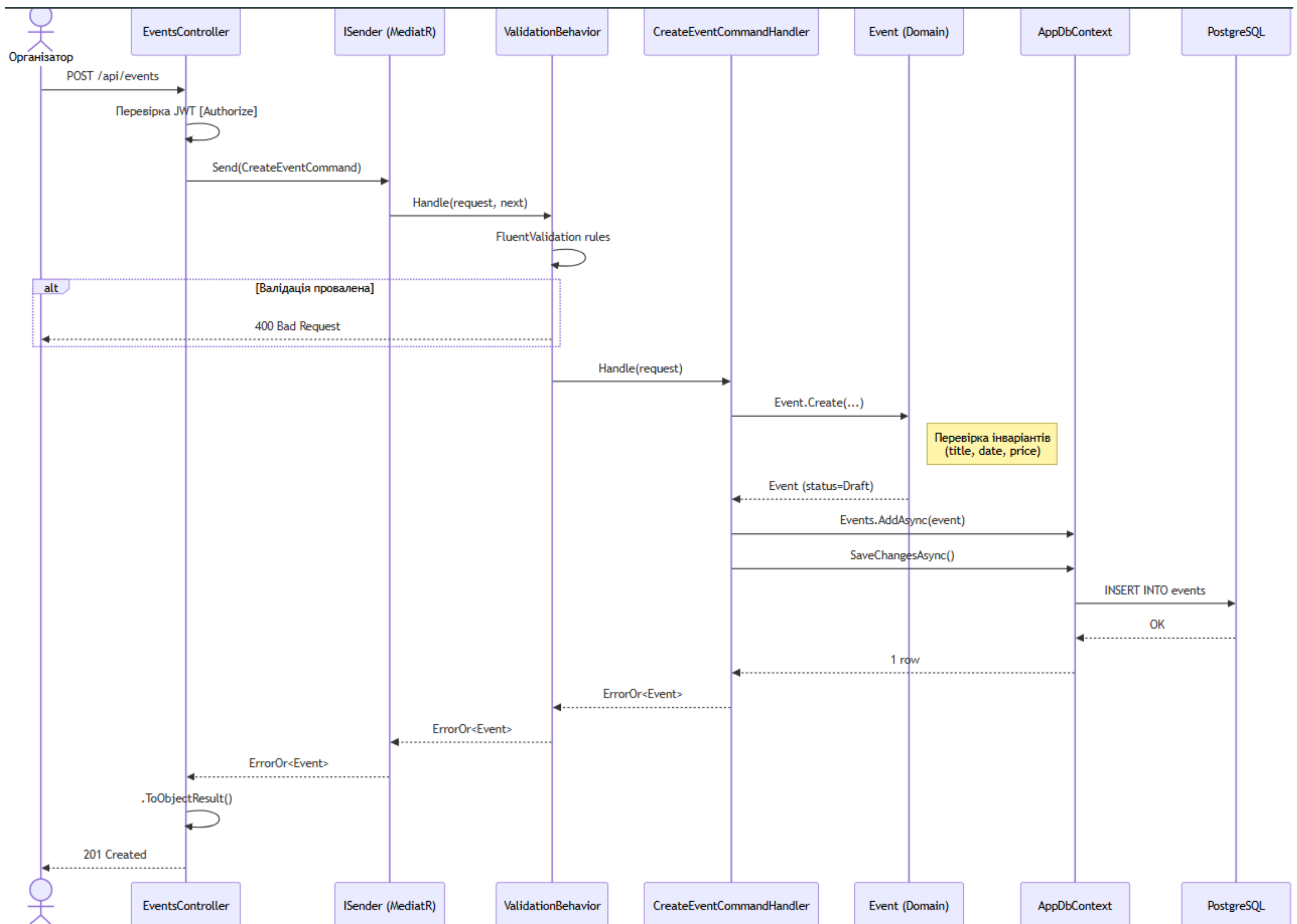

Рис. 2.3. Sequence-діаграма обробки команди створення події

Джерело: розроблено автором

Приклад реалізації команди "CreateEventCommand" та її обробника наведено у Додатку Б.

2.2.4 Шар Infrastructure: репозиторії та зовнішні сервіси

Шар Infrastructure містить реалізації всіх інтерфейсів, оголошених у шарі Application, та технічні деталі взаємодії з зовнішнім світом. Основні групи компонентів:

- ***Persistence***: AppDbContext - клас контексту Entity Framework Core, що містить DbSet<> для кожної агрегатної сутності, конфігурації мапінгу через Fluent API,

конвертери для типів DateTime (всі дати зберігаються у форматі UTC через DateTimeUtcConverter), реалізація AppDbContextInitializer для seed-даних та автоматичних міграцій при старті.

- ***AI***: AiChatService - оркестратор AI-розмов, чотири плагіни Semantic Kernel (EventManagementPlugin, EventRegistrationPlugin, OrganizerDashboardPlugin, EventSearchPlugin), CaptureFunctionResultFilter для інспекції результатів виконання kernel-функцій, конфігурація DI (ConfigureAI.cs).
- ***Payment***: LiqPayService - реалізація інтеграції з платіжною системою LiqPay, генерація сигнатури запитів, перевірка сигнатури callback-ів.
- ***Storage***: інтеграція з Cloudflare R2 через офіційний SDK для зберігання зображень.
- ***Hubs***: NotificationsHub, AiChatHub - реалізації SignalR-хабів для real-time комунікації.
- ***Services***: SignalRNotificationService, UserContextService, EventRefundService - інфраструктурні сервіси, що реалізують відповідні інтерфейси з шару Application.

Доступ до даних організовано через DbContext напряму у обробниках Queries (для оптимальності читання) та через спеціалізовані репозиторії для Commands (для інкапсуляції складної логіки збереження агрегатів).

2.2.5 Шар API

Шар API є точкою входу зовнішніх запитів. Складається з:

- *Controllers* - REST-контролери для кожного бізнес-домену (EventsController, SubscriptionsController, AdminController, UsersController). Контролери максимально тонкі: вони отримують HTTP-запит, формують команду чи запит MediatR, надсилають через ISender та повертають відповідь з відповідним HTTP-статусом.

- *Hubs* - точки входу для WebSocket-з’єднань SignalR (AiChatHub, NotificationsHub).
- *Middleware* - UserContextMiddleware витягує заголовок X-User-Id (значення Clerk ID), резолвить внутрішнього користувача через IUserContextService і прокидає в контекст запиту; ExceptionHandlerMiddleware перетворює неперехоплені винятки на структуровані HTTP-відповіді з кодами стану.
- *Program.cs* - конфігурація DI-контейнера, реєстрація шарів через extension-методи AddApplication(), AddInfrastructure(), AddAI(), налаштування Swagger, CORS, JWT-автентифікації, SignalR.

Для перетворення ErrorOr<T> у відповідь ASP.NET Core використано extension-метод .ToObjectResult(), що автоматично формує коректну HTTP-відповідь з відповідним статусом (200 OK для успіху, 400/404/409 для різних типів помилок).

2.2.6 Проєктування бази даних

База даних побудована на СКБД PostgreSQL. Вибір зумовлений її надійністю, відкритим кодом, потужною підтримкою JSON-полів (для зберігання гнучких даних типу налаштувань профілю), а також безкоштовністю для комерційного використання.

Іменування таблиць та стовпців виконано у форматі snake_case через відповідну конвенцію іменування Entity Framework Core. Це покращує читабельність SQL-запитів при дебагінгу та виконанні adhoc-запитів через pgAdmin.

Перелік основних таблиць:

- events - головна таблиця подій;
- event_registrations - реєстрації на події;
- tickets - квитки (зв’язок 1:1 з реєстрацією для платних подій);
- users - внутрішні користувачі (з посиланням на Clerk через clerk_user_id);
- roles - довідник ролей (Attendee, Organizer, Admin);

- user_roles - зв’язок «багато-до-багатьох» між користувачами та ролями;
- subscriptions - активні та історичні підписки організаторів;
- event_photos - посилання на завантажені у S3 фото;
- chat_conversations - історія взаємодій користувача з AI-асистентом;
- organizer_payout_accounts - платіжні реквізити організаторів (1:1 з users), з підтримкою методів manual_bank_transfer та liqpay_merchant;
- payouts - розрахункові записи виплат організаторам (агрегація проданих квитків за період), з масивом ticket_ids у форматі JSONB.

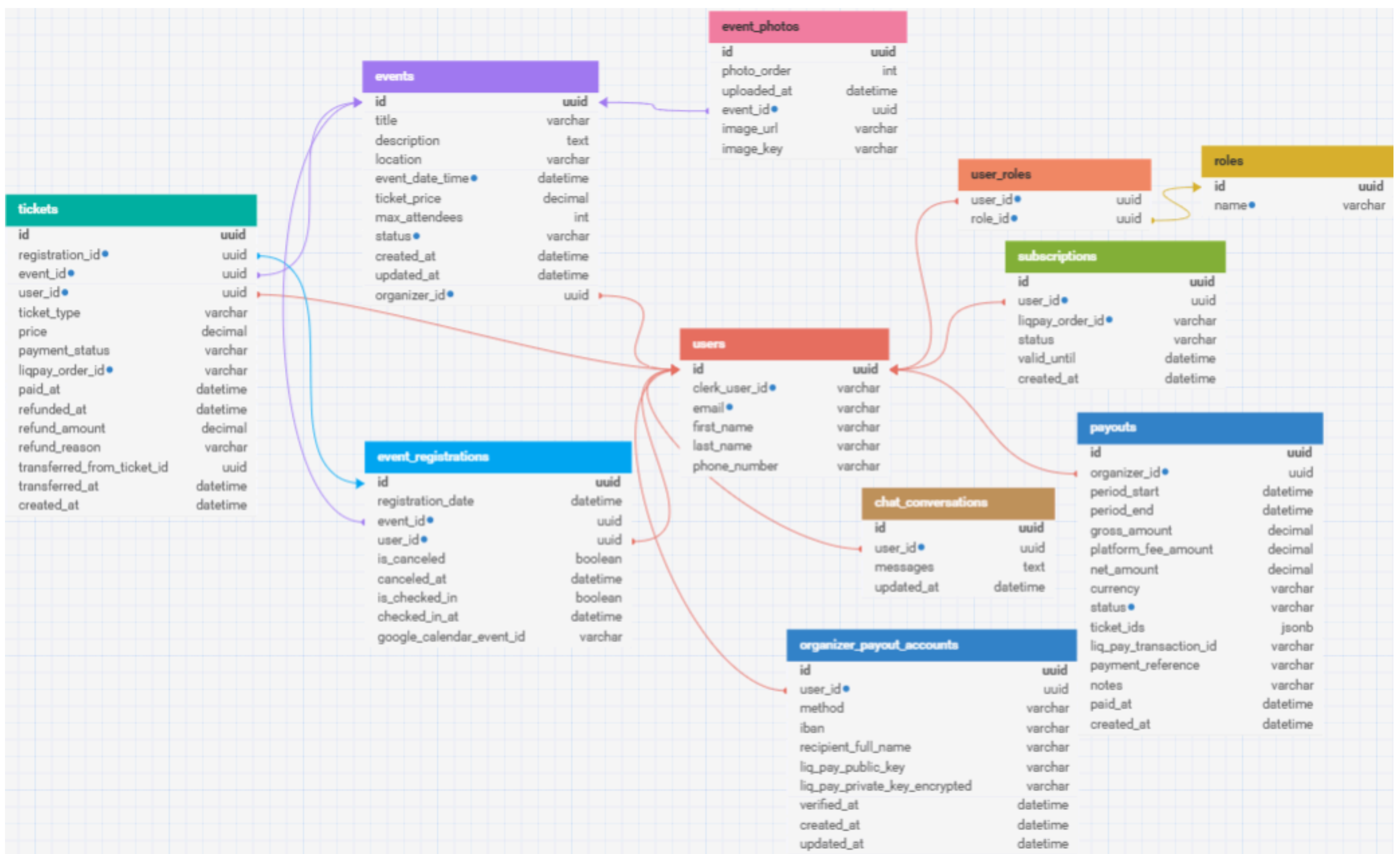

Рис. 2.4. ER-діаграма повної бази даних EventFlow

Джерело: розроблено автором

Ключові індекси:

- events.organizer_id - для швидкої вибірки подій конкретного організатора;
- events.status, events.start_date - для фільтрації опублікованих майбутніх подій;

- event_registrations.event_id, event_registrations.user_id - унікальний індекс, що запобігає подвійній реєстрації;
- tickets.payment_id - для пошуку квитка за ідентифікатором платежу при обробці колбеків LiqPay;
- users.clerk_user_id - унікальний індекс для швидкого резолвінгу зовнішнього ідентифікатора у внутрішній.
- organizer_payout_accounts.user_id - унікальний індекс, забезпечує співвідношення 1:1 між організатором та платіжним акаунтом;
- payouts.organizer_id, payouts.status - для швидкої фільтрації виплат у адмін-панелі та підрахунку зведеної статистики у профілі організатора.

Таблиця 2.1

Атрибути сутності Event

Атрибут	**Тип**	**Обмеження**	**Опис**
id	UUID	PK, NOT NULL	Унікальний ідентифікатор
title	varchar(200)	NOT NULL	Назва події
description	text	NULL	Розгорнутий опис
start_date	timestamptz	NOT NULL	Дата та час початку
end_date	timestamptz	NOT NULL	Дата та час завершення
location	varchar(500)	NULL	URL онлайн-події
price	decimal(18,2)	NOT NULL, ≥ 0	Ціна квитка в UAH
max_attendees	int	NOT NULL, > 0	Ліміт учасників

status	varchar(50)	NOT NULL	Статус події
organizer_id	UUID	FK → users.id	Посилання на організатора
cover_image_url	varchar(2048)	NULL	URL обкладинки в S3
is_blocked	boolean	NOT NULL	Чи заблокована адміном
created_at	timestamptz	NOT NULL	Час створення
updated_at	timestamptz	NOT NULL	Час останнього оновлення

Джерело: розроблено автором.

2.2.7 Архітектура клієнтської частини

Клієнтська частина побудована на фреймворку Next.js 16 з App Router. Структура проєкту відповідає принципам функціонального розділення з елементами Feature-Sliced Design:

```
eventflow-client/src/
 ├── app/          (роутинг App Router, layout, error boundaries)
 ├── features/     (бізнес-фічі: створення події, чат, тощо)
 ├── entities/     (UI-представлення доменних сутностей)
 ├── widgets/      (композитні UI-блоки)
 ├── shared/       (універсальні утиліти, UI-кіт, API-клієнт)
 ├── hooks/        (кастомні React-хуки)
 ├── contexts/     (React-контексти)
 ├── lib/          (інтеграції з Clerk, SignalR, тощо)
 ├── i18n/        (українська/англійська локалізація)
 └── types/        (глобальні TypeScript-типи)
```

Рис. 2.5. Структура клієнтської частини

Джерело: розроблено автором

Управління серверним станом реалізовано через React Query (TanStack Query): автоматичне кешування, фонове оновлення, optimistic updates. Локальний UI-стан керується через React Context для глобальних значень (тема, локаль, поточний користувач) та хук useState для локальних компонентних станів. Для складних форм з валідацією використано React Hook Form у поєднанні з Zod-схемами, що дозволяє переиспользовувати ту саму схему валідації між клієнтом та серверним кодом TypeScript-утиліт.

2.3 Алгоритмічне забезпечення

2.3.1 Алгоритм перевірки доступності квитків

При паралельних спробах декількох користувачів зареєструватися на подію з обмеженою кількістю місць виникає класична задача конкурентного доступу. Якщо обробка наївно перевіряє кількість зайнятих місць, потім вставляє нову реєстрацію - між цими двома операціями інший потік може вставити свою реєстрацію, що призведе до перевищення ліміту.

Для усунення проблеми застосовано підхід оптимістичного блокування на рівні бази даних:

1. Зчитується поточний стан події (включно з полем version або timestamp updated_at).
2. Перевіряється, чи поточна кількість зареєстрованих менша за max_attendees.
3. Виконується INSERT у таблицю реєстрацій з умовою на унікальний індекс (event_id, user_id).
4. Виконується UPDATE events SET registered_count = registered_count + 1, version = version + 1 WHERE id = @id AND version = @oldVersion.

5. Якщо UPDATE повертає 0 рядків - інша транзакція встигла раніше, виконується ретрай або повертається помилка EVENT_FULL.

Цей алгоритм гарантує консистентність без використання важких ексклюзивних блокувань таблиці.

2.3.2 Алгоритм формування фінальної ціни квитка

Базова ціна квитка може бути модифікована рядом факторів: знижка від організатора, комісія платіжної системи. Формула розрахунку:

$$P_{final} = P_{base} \times (1 - D_{discount}) + F_{liqpay} \quad (2.1)$$

де: P_{final} - кінцева ціна для користувача; P_{base} - базова ціна квитка від організатора; $D_{discount}$ - частка знижки, $0 \leq D_{discount} \leq 1$; F_{liqpay} - фіксована комісія LiqPay (наразі 0, оскільки делегується організатору).

2.3.3 Алгоритм роботи AI-асистента на основі Semantic Kernel

AI-асистент EventFlow використовує підхід Function Calling - у запит до LLM передається список доступних функцій (kernel functions з плагінів) разом з їх описами та схемами параметрів. Модель самостійно вирішує, чи відповідати текстом, чи викликати одну або кілька функцій.

Алгоритм обробки запиту користувача:

1. Користувач надсилає текстове повідомлення через AI-чат.
2. AiChatService отримує повідомлення, додає його до історії розмови.
3. Сервіс формує запит до LLM (через адаптер Semantic Kernel) з системним промптом, історією розмови, поточним повідомленням та переліком доступних функцій з чотирьох плагінів.
4. LLM повертає відповідь, що може містити:

   a. - текстову частину (стрімиться клієнту через AiChatHub у реальному часі);
   b. - функцій (function calls).
5. Якщо є виклики функцій, для кожної функції:
   a. - перевіряється авторизація поточного користувача всередині плагіна (наприклад, EventManagementPlugin дозволяє редагувати тільки власні події);
   b. - виконується, її результат фіксується через CaptureFunctionResultFilter;
   c. - результат повертається до LLM як новий контекст.
6. LLM формує фінальну текстову відповідь з урахуванням результатів функцій.
7. Відповідь зберігається в історію та надсилається клієнту.

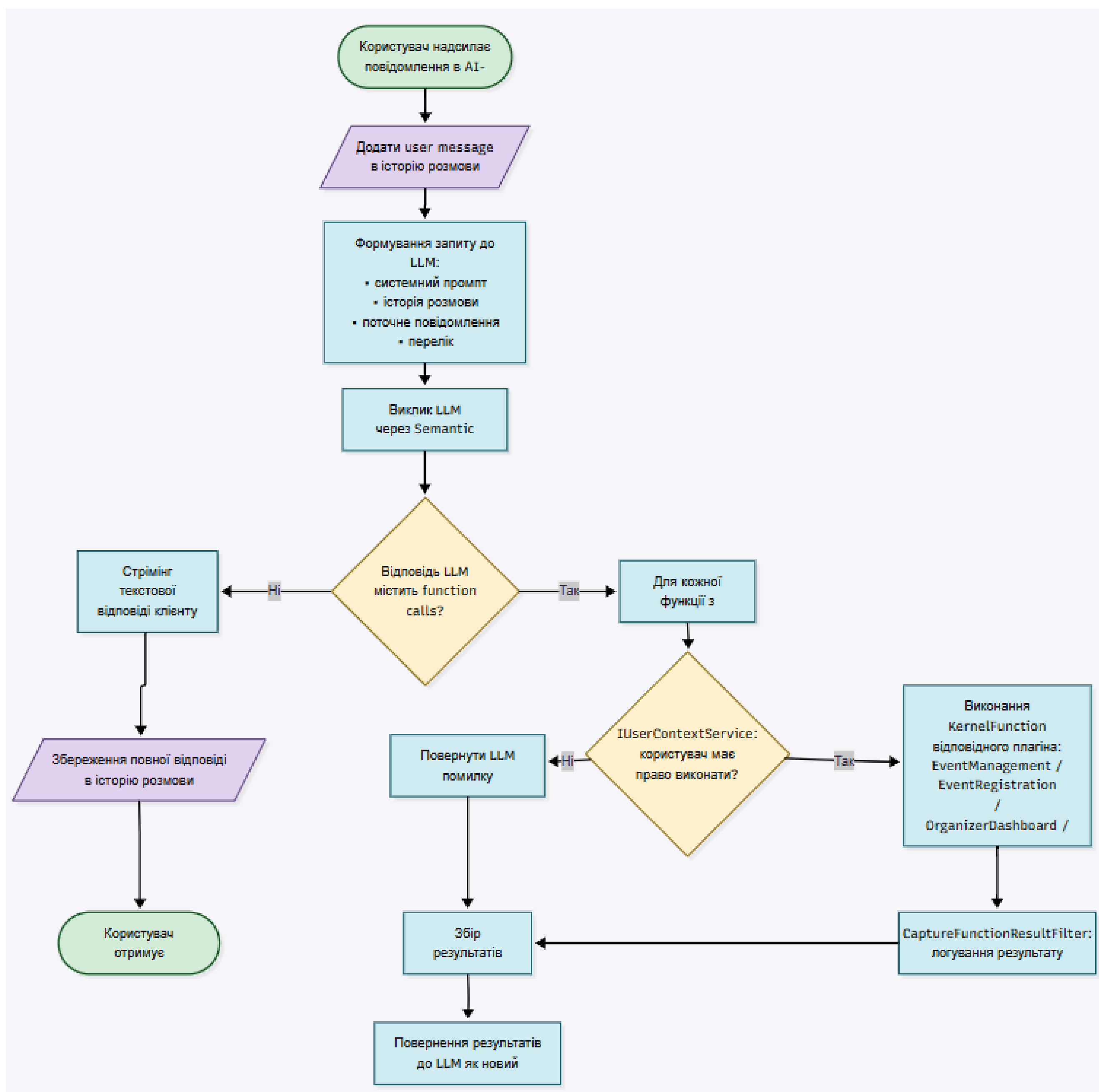

Рис. 2.6. Блок-схема алгоритму обробки AI-запиту

Джерело: розроблено автором

Архітектура з чотирма плагінами (EventManagement, EventRegistration, OrganizerDashboard, EventSearch) дозволяє не передавати в LLM усі функції одразу - Semantic Kernel автоматично вибирає релевантні на основі контексту запиту. Це знижує споживання токенів та підвищує точність вибору функцій.

2.3.4 Алгоритм перевірки сигнатури callback'у LiqPay

При отриманні callback від LiqPay про статус платежу необхідно переконатися, що повідомлення дійсно надіслане LiqPay, а не зловмисником. LiqPay використовує наступну схему:

1. У запиті передаються поля data (Base64-енкодований JSON з даними платежу) та signature.
2. Алгоритм перевірки:

$$signature_{expected} = Base64\big(SHA1(privateKey + data + privateKey)\big) \quad (2.2)$$

3. Якщо signature з запиту збігається з signature_expected - повідомлення автентичне.
4. У разі збігу декодується data, з нього витягується статус платежу та order_id, оновлюється відповідний Ticket в базі даних.
5. У разі неавтентичності повідомлення повертається 400 Bad Request, зловмисний запит логується.

Висновки до розділу 2

У другому розділі здійснено повний цикл проєктування системи EventFlow.

Виділено дев'ять основних доменних сутностей та описано потоки вхідних та вихідних даних із зазначенням обмежень валідації. Обґрунтовано вибір Clean Architecture як основного архітектурного шаблону та описано чотири її шари: Domain, Application, Infrastructure, API.

Спроєктовано шар Application на основі патерну CQRS з використанням бібліотеки MediatR, ErrorOr для контрольованої обробки помилок та pipeline behaviors для крос-функціональної логіки. Описано структуру шару Infrastructure з його ключовими компонентами: AppDbContext, AI-плагіни, LiqPayService, SignalR-хаби.

Розроблено модель бази даних на PostgreSQL з дев’ятьма основними таблицями, описано ключові індекси та продемонстровано детальний опис атрибутів на прикладі таблиці events.

Запропоновано та формалізовано чотири основні алгоритми системи: перевірка доступності квитків з оптимістичним блокуванням, розрахунок фінальної ціни, обробка AI-запитів через Function Calling Semantic Kernel, перевірка сигнатури LiqPay.

Спроєктовані моделі та архітектура є основою для програмної реалізації, описаної в наступному розділі.

РОЗДІЛ 3
ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1 Засоби розробки

Для реалізації системи EventFlow обрано сучасний стек технологій, що забезпечує продуктивність розробки, типобезпеку, тестованість та зручність розгортання.

Серверна частина:

- *.NET 10* - остання LTS-версія платформи на момент початку розробки. Забезпечує високу продуктивність, native-AOT компіляцію, мінімалістичні API через Minimal APIs, покращену підтримку системного коду.
- *C# 14* - мова програмування з підтримкою primary-конструкторів, collection expressions, raw string literals, що значно покращують читабельність.
- *ASP.NET Core* - фреймворк для побудови HTTP API з нативною підтримкою DI, middleware, маршрутизації.
- *Entity Framework Core 8* - ORM з підтримкою LINQ, міграцій, value converters, Fluent API для конфігурації.
- *PostgreSQL 16* - реляційна СКБД, підключена через провайдер Npgsql.EntityFrameworkCore.PostgreSQL.
- *MediatR* - реалізація патерну Mediator для CQRS.
- *FluentValidation* - декларативна валідація вхідних даних.
- *ErrorOr* - типобезпечне представлення результатів виконання з можливими помилками.
- *Microsoft Semantic Kernel* - фреймворк для інтеграції LLM у застосунок з підтримкою Function Calling, плагінів, керування історією розмов.
- *Microsoft.AspNetCore.SignalR* - real-time комунікація через WebSocket з абстракцією над протоколами.

- *AWS SDK for .NET (S3)* - офіційний SDK для роботи з об’єктним сховищем.
- *Swashbuckle.AspNetCore* - автоматична генерація Swagger/OpenAPI документації.

Клієнтська частина:

- *Next.js 16* - React-фреймворк з App Router, який підтримує серверні компоненти, server actions, streaming рендерингу, optimization images.
- *React 19* - бібліотека UI з підтримкою Concurrent Mode, Suspense, новими хуками типу useTransition.
- *TypeScript 5* - мова програмування зі статичною типізацією, що працює поверх JavaScript.
- *Tailwind CSS* - utility-first CSS-фреймворк для швидкого стилізування.
- *shadcn/ui* - колекція копіювально-вставних компонентів на базі Radix UI та Tailwind.
- *TanStack Query (React Query)* - управління серверним станом.
- *React Hook Form + Zod* - форми з валідацією.
- *Microsoft SignalR Client (@microsoft/signalr)* - клієнт для real-time комунікації.
- *Clerk SDK для Next.js* - інтеграція автентифікації Clerk.

Інфраструктура та інструменти:

- *Docker, Docker Compose* - контейнеризація та локальне розгортання.
- *GitHub Actions* - CI/CD pipeline.
- *Cloudflare R2* - зберігання файлів.
- *Clerk* - зовнішній сервіс автентифікації.
- *LiqPay* - платіжна система.
- *Groq Cloud* - провайдер LLM для AI-функцій.

Середовища розробки та супровід:

- *JetBrains Rider* та *Cursor*- IDE для C# та TypeScript коду відповідно.
- *Dbeaver* - графічний клієнт для роботи з PostgreSQL.

- *Figma* - проєктування UI/UX, з використанням AI-функції *Figma Make* для пришвидшення генерації варіантів дизайну.
- *Git, GitHub* - контроль версій.

3.2 Вимоги до технічного та програмного забезпечення

Серверні вимоги для продакшен-розгортання:

- процесор: 2 vCPU або більше;
- оперативна пам’ять: 4 ГБ або більше;
- дисковий простір: 20 ГБ SSD;
- операційна система: Linux Ubuntu 22.04 LTS або сумісна;
- встановлений Docker Engine 24+ та Docker Compose;
- мережа: статична IP-адреса, відкриті порти 80, 443; вихідний доступ до Clerk API, LiqPay API, Cloudflare R2, Groq API.

Серверні вимоги для розробки:

- .NET 10 SDK;
- Node.js 20+ та pnpm;
- Docker Desktop;
- Git.

Клієнтські вимоги (для кінцевого користувача):

- сучасний браузер з підтримкою ES2022 та WebSocket: Chrome 120+, Firefox 120+, Safari 17+, Edge 120+;
- стабільне інтернет-з’єднання (мінімум 1 Мбіт/с для коректної роботи real-time функцій);
- роздільна здатність екрана від 320×568 пікселів (адаптивний інтерфейс).

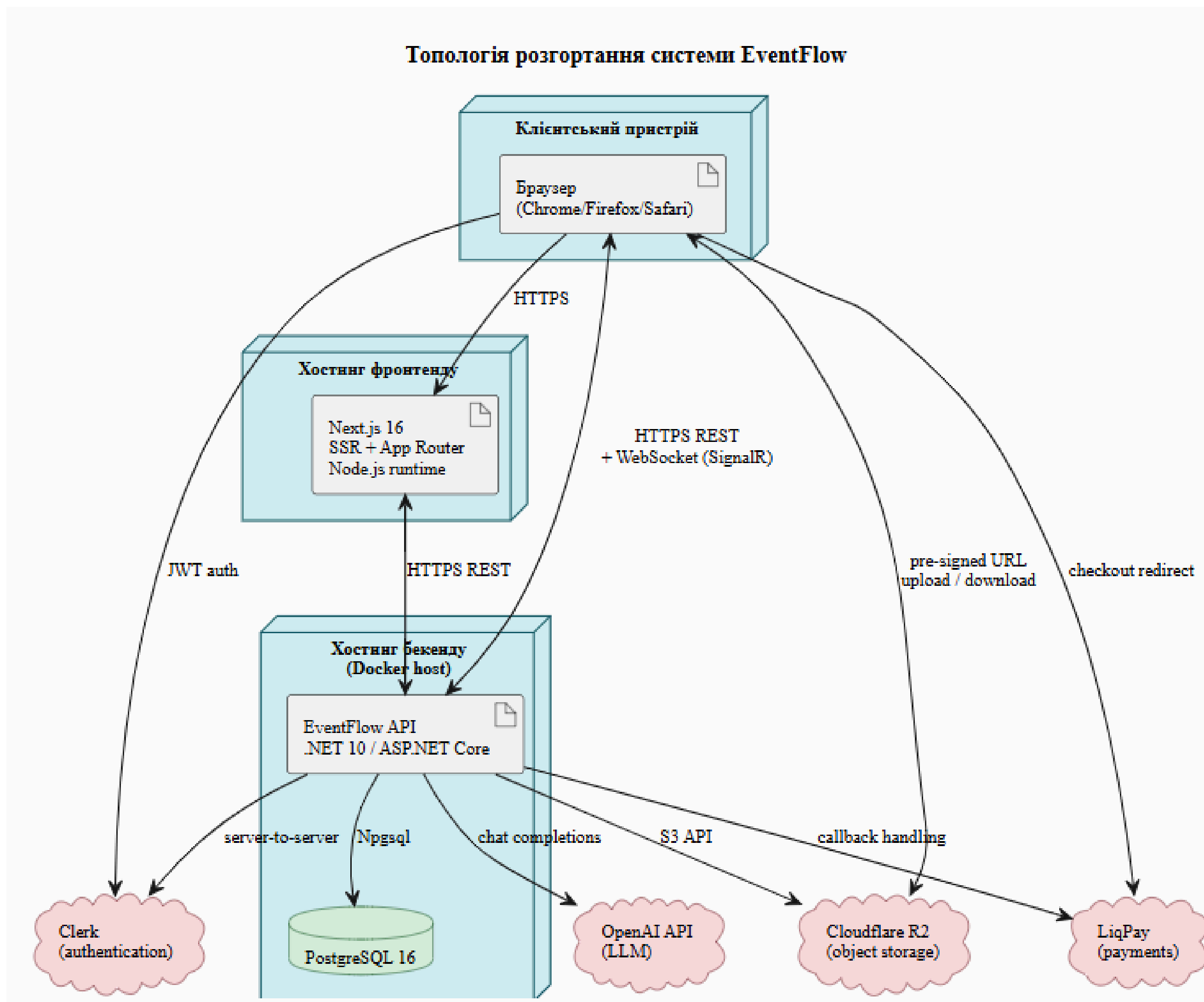

Рис. 3.1. Топологія розгортання системи EventFlow - заглушка для рисунка

Джерело: розроблено автором

3.3 Опис програмної реалізації серверної частини

3.3.1 Структура рішення

Рішення EventFlow.Backend.sln складається з чотирьох основних проєктів та п’яти тестових проєктів:

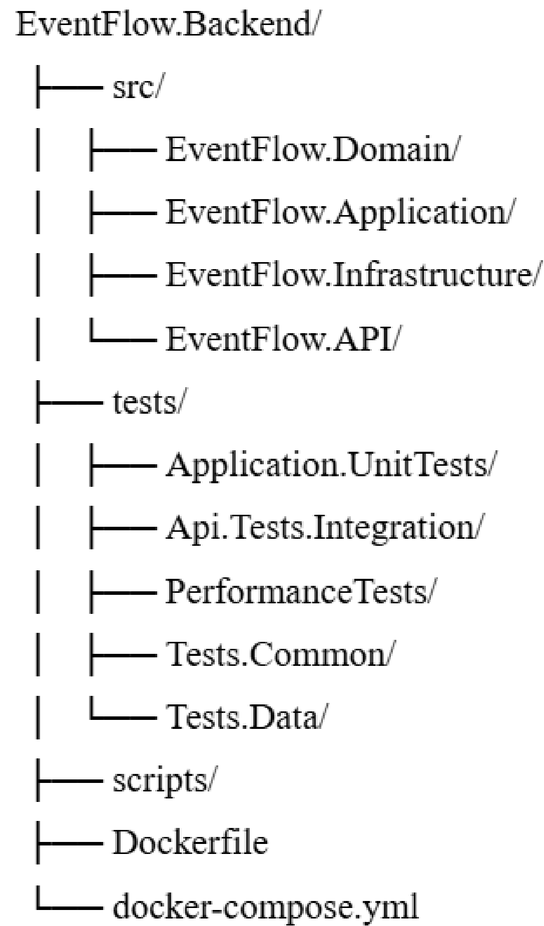

Рис. 3.2. Структура solution в IDE

Джерело: розроблено автором

Граф залежностей між проєктами повністю відповідає правилу Clean Architecture: внутрішні шари не залежать від зовнішніх. EventFlow.Domain не залежить ні від чого. EventFlow.Application залежить тільки від Domain. EventFlow.Infrastructure залежить від Application та Domain. EventFlow.API залежить від Application та Infrastructure (для реєстрації DI).

3.3.2 Реалізація доменного шару

Доменний шар містить дев’ять сутностей, розташованих у папці Entities: ChatConversation, Event, EventPhoto, EventRegistration, Role, Subscription, Ticket, User,

UserRole. Окрема папка ValueObjects містить строго типізовані ідентифікатори та неперіодичні значення.

Особливістю реалізації є приватні сетери всіх властивостей сутностей - стан можна змінювати лише через бізнес-методи, що гарантує дотримання інваріантів. У класі Event пряма зміна Status неможлива; для переходу між статусами існують методи Publish(), Cancel() з відповідними перевірками попередніх умов (наприклад, не можна опублікувати скасовану подію або подію без жодної фотографії). Створення нової події відбувається через статичний фабричний метод Event.Create(), який валідує доменні інваріанти: непорожня назва, дата у майбутньому, невід'ємна ціна. Повний код сутності Event з усіма доменними методами (Cancel, UpdateDetails, AddPhoto, RemovePhoto, GetRegistrationCount) та інших восьми сутностей наведено у Додатку А.

3.3.3 Реалізація шару Application

Шар Application побудований навколо MediatR. Кожна команда або запит реалізована як record, що імплементує IRequest<TResponse>. Обробник реалізує IRequestHandler<TRequest, TResponse>. Валідатор успадковує AbstractValidator<TRequest> з FluentValidation.

Кожна команда оголошена як record, що імплементує IRequest<TResponse>, де TResponse - це ErrorOr<TResult>. Команда CreateEventCommand приймає всі параметри запиту (назва, опис, дата, ціна, ліміт учасників, посилання на користувача-організатора) та повертає або створену сутність події, або список доменних помилок без використання винятків. Обробник CreateEventCommandHandler реалізує бізнес-логіку: резолвить внутрішнього користувача через IUserContextService, перевіряє роль (тільки організатор або адмін може створити подію), застосовує ліміти підписки (Free-план - до 3 активних подій, Pro-план - до 50 подій на місяць), створює сутність через Event.Create() та зберігає її в БД. Повний код команди разом з валідатором та обробником наведено у Додатку Б.

Інтеграція FluentValidation з MediatR-конвеєром реалізована через ValidationBehaviour<TRequest, TResponse> - pipeline behavior, що автоматично запускається перед кожним обробником. Behaviour отримує всі валідатори, зареєстровані для конкретного типу команди, через DI-контейнер, запускає їх паралельно через Task.WhenAll, і у разі наявності помилок кидає ValidationException, який далі перехоплюється middleware і конвертується в HTTP 400 з детальним переліком проблем. Завдяки цій абстракції розробникам обробників не треба викликати валідацію вручну - вона вмикається автоматично для будь-якої команди, для якої існує клас-валідатор. Повний код pipeline behaviours наведено у Додатку Б.

Контроль доступу реалізовано через сервіс IUserContextService, що надає поточного автентифікованого користувача обробникам команд. Обробники, що змінюють чи читають дані конкретного користувача, перевіряють його через цей інтерфейс.

3.3.4 Реалізація шару Infrastructure

AppDbContext є центральним класом доступу до даних. У ньому оголошено DbSet<> для кожної агрегатної сутності, перевизначено OnModelCreating для застосування Fluent-конфігурацій з папки Persistence/Configurations, налаштовано конвертери для DateTime (всі дати у форматі UTC).

Метод OnModelCreating у класі AppDbContext застосовує всі Fluent API конфігурації мапінгу сутностей з папки Persistence/Configurations через modelBuilder.ApplyConfigurationsFromAssembly(). Кожна сутність має окремий клас-конфігурацію, що задає первинні ключі, відношення між таблицями, обмеження довжини рядків, унікальні індекси та конвертери для строго типізованих ідентифікаторів (наприклад, EventId ↔ uuid). Окрема глобальна конвенція DateTimeUtcConverter забезпечує, що всі дати при читанні з БД отримують DateTimeKind.Utc, а при записі - конвертуються в UTC, що усуває проблеми зі

змішуванням часових зон. Повний код AppDbContext та всіх класів-конфігурацій наведено у Додатку В.

Клас AppDbContextInitializer запускається при старті застосунку та виконує:

1. Перевірку наявності бази даних, її створення за потреби.
2. Застосування міграцій EF Core (Database.MigrateAsync()).
3. Заповнення довідникових даних (Roles) при першому запуску.

3.3.5 Реалізація API-контролерів

Контролери ASP.NET Core реалізують REST-ендпоінти. Контролер EventsController оголошує наступні маршрути:

- GET /api/events - список опублікованих подій з пагінацією та фільтрами;
- GET /api/events/{id} - деталі однієї події;
- POST /api/events - створення нової події (потрібна автентифікація);
- PUT /api/events/{id} - оновлення події (тільки організатор);
- DELETE /api/events/{id} - видалення (тільки організатор або адмін);
- POST /api/events/{id}/publish - публікація події;
- POST /api/events/{id}/register - реєстрація поточного користувача;
- POST /api/events/{id}/checkout - створення платіжного запиту LiqPay;
- POST /api/events/payment-callback - обробник callback від LiqPay.

Контролери ASP.NET Core максимально тонкі: вони отримують HTTP-запит, формують команду чи запит MediatR, надсилають її через ISender та повертають результат через extension-метод .ToObjectResult(), що автоматично конвертує ErrorOr<T> у HTTP-відповідь з відповідним статусом (200 OK для успіху, 400 для помилок валідації, 404 для відсутніх ресурсів, 409 для конфліктів). EventsController оголошує REST-ендпоінти для всіх операцій з подіями: створення (POST /api/events), оновлення, видалення, публікація, реєстрація учасника, ініціація платежу через LiqPay, обробка callback-ів від LiqPay. Жодної бізнес-логіки в контролері немає - вся вона

зосереджена в обробниках команд та запитів у шарі Application. Повний код EventsController та інших контролерів наведено у Додатку Г.

Документація API згенерована автоматично через Swagger/OpenAPI на базі XML-коментарів та атрибутів. Розробники отримують інтерактивну сторінку /swagger, через яку можна виконувати запити безпосередньо в браузері.

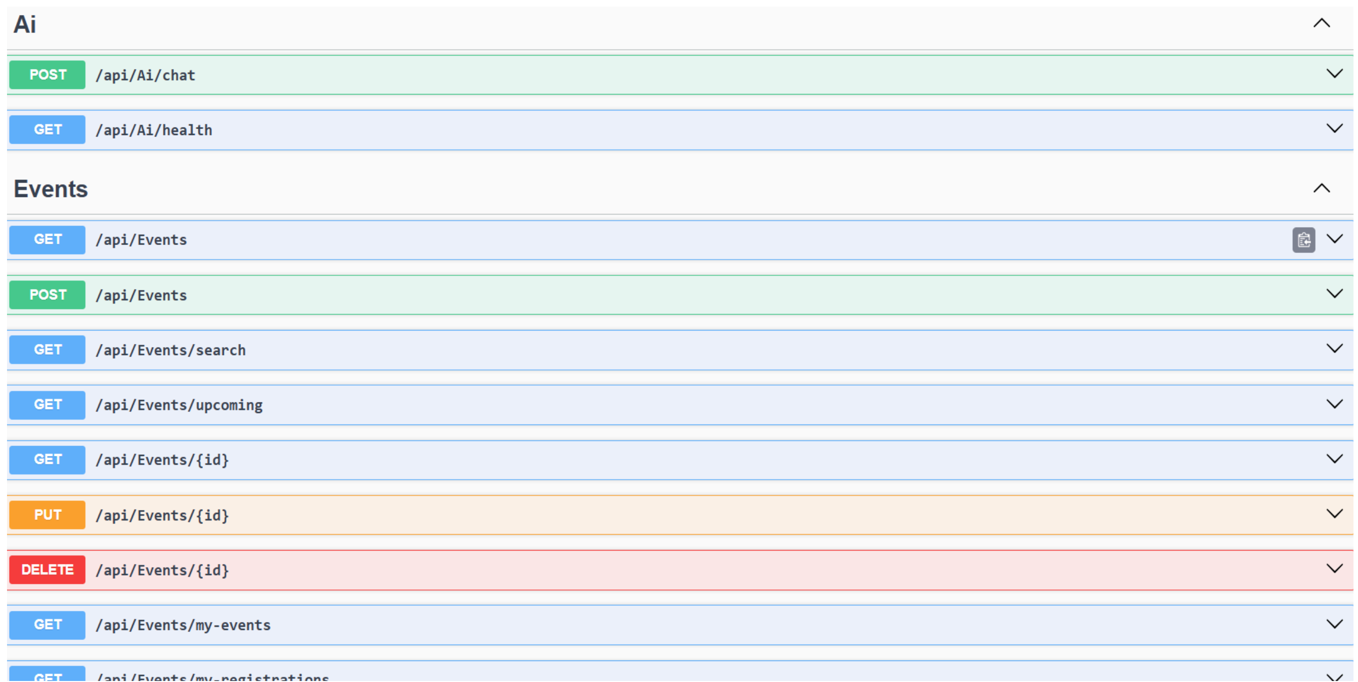

Рис. 3.3. Інтерфейс Swagger UI

Джерело: розроблено автором

3.3.6 Інтеграція автентифікації через Clerk

Clerk - це SaaS-сервіс автентифікації та управління користувачами, що надає готові UI-компоненти для входу/реєстрації, підтримує SSO через Google, GitHub, Apple, email magic links, MFA. Вибір Clerk над альтернативами (Auth0, IdentityServer, власна реалізація) обґрунтовано: швидкою інтеграцією, безкоштовним тарифом до 10 000 активних користувачів, готовими компонентами для Next.js, відсутністю необхідності зберігати паролі в нашій БД.

Архітектура автентифікації в EventFlow:

1. Користувач відкриває клієнтський застосунок та натискає «Увійти».
2. Завантажується UI-компонент Clerk у вигляді модального вікна або на окремій сторінці.
3. Користувач автентифікується через обраний провайдер.
4. Clerk видає JWT, який автоматично прикріплюється до всіх запитів до бекенду через Clerk SDK.
5. На бекенді ASP.NET Core валідує JWT через JWT Bearer middleware (публічний ключ Clerk завантажується з jwks endpoint).
6. Кастомний UserContextMiddleware витягує clerkUserId з claims токена та резолвить внутрішнього User через UserContextService (з кешуванням та created-on-first-access логікою).

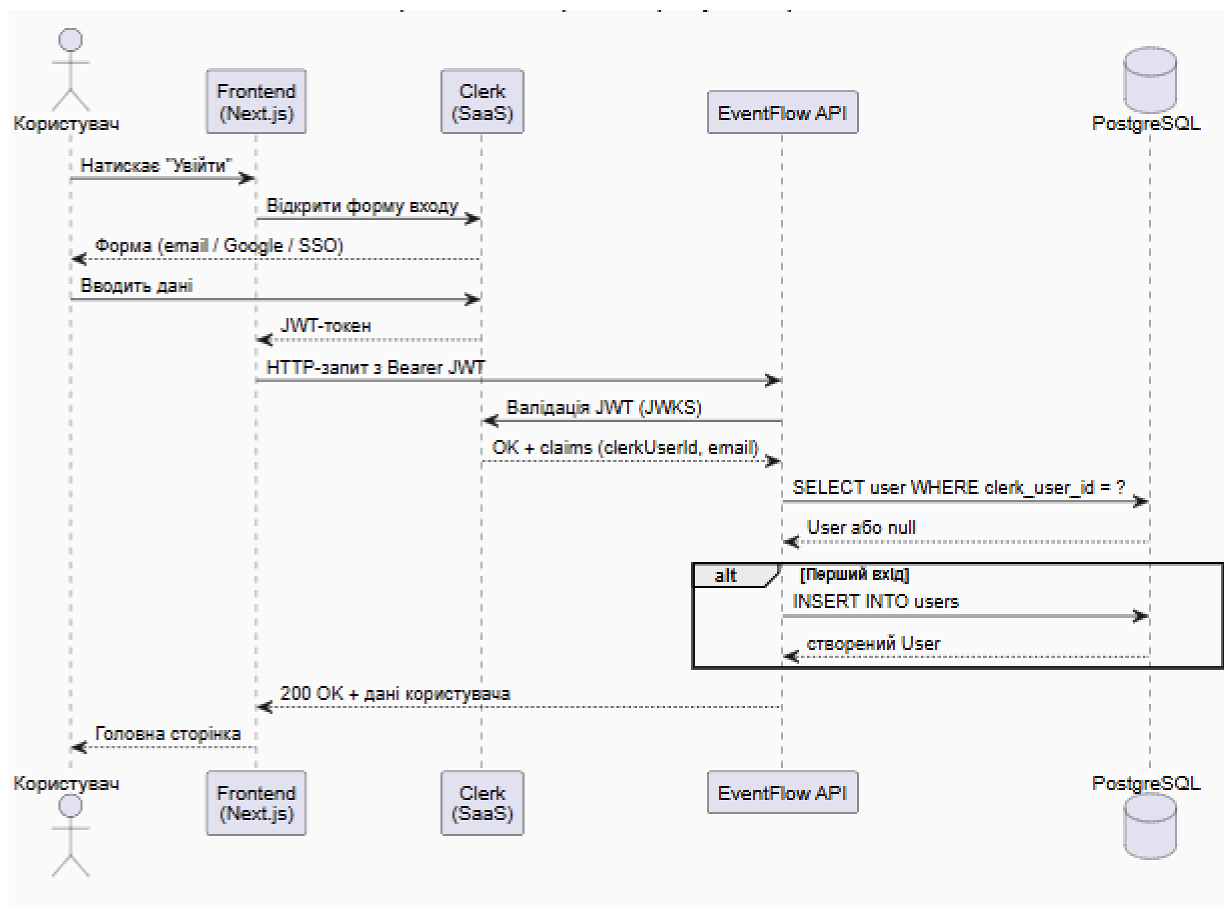

Рис. 3.4. Sequence-діаграма автентифікації через Clerk

Джерело: розроблено автором

Інтеграція автентифікації Clerk зі внутрішньою моделлю користувачів реалізована через два компоненти: middleware UserContextMiddleware, що витягує clerkUserId з claims JWT-токена, та сервіс UserContextService, що резолвить внутрішнього User за цим ClerkUserId через IUserRepository. Якщо користувач з таким ClerkUserId ще не існує у внутрішній БД (тобто це його перший вхід після реєстрації в Clerk), сервіс автоматично створює відповідний запис, копіюючи email та ім'я з claims токена. Це забезпечує безшовний onboarding без додаткових форм реєстрації. Повний код middleware та сервісу наведено у Додатку Г.

При першому вході користувача в систему UserContextService автоматично створює відповідний запис у таблиці users, копіюючи email та ім'я з claims Clerk-токена. Це забезпечує seamless onboarding без додаткових форм реєстрації.

Окремо реалізовано параметризований сидінг адміністратора. Початкова версія AppDbContextInitializer.SeedAdminAsync() використовувала фіксований ідентифікатор seed_admin_001, який не відповідав жодному реальному обліковому запису Clerk, через що адміністратор фактично не міг увійти в систему. Поточна реалізація читає з конфігурації значення Admin:ClerkUserId (через appsettings.Development.json у режимі розробки або змінну середовища Admin__ClerkUserId у продакшен-середовищі) та виконує одну з двох дій: створює нового користувача з роллю Admin, або підвищує існуючого користувача (синхронізованого через /api/users/sync) до ролі Admin. Якщо ж конфігурація відсутня - сидінг пропускається з попередженням у журналі, щоб уникнути створення «фантомного» адміністратора.

3.3.7 Інтеграція платіжної системи LiqPay

LiqPay - це платіжна система від ПриватБанку, що дозволяє приймати платежі картками Visa, Mastercard, Apple Pay, Google Pay, готівкою через термінали, з можливістю приймати кошти в гривнях, доларах, євро. Перевага LiqPay для проєкту:

широке розповсюдження в Україні, відносно низька комісія (1-2,75%), наявність тестового середовища без необхідності верифікації організації.

Архітектура платіжного flow в EventFlow:

1. Користувач натискає «Купити квиток» на сторінці платної події.
2. Клієнт надсилає POST /api/events/{id}/checkout із зазначенням події.
3. Серверний обробник CreateTicketCheckoutCommand:
    a. перевіряє наявність вільних місць;
    b. створює запис Ticket зі статусом Pending;
    c. формує параметри для LiqPay: amount, currency, order_id (= TicketId), description, result_url (повернення клієнта), server_url (callback);
    d. викликає LiqPayService.GenerateCheckoutData() для створення data (Base64) та signature.
4. Клієнт отримує два рядки data та signature і виконує POST-форму на сторінку LiqPay.
5. LiqPay показує інтерфейс оплати, користувач вводить дані картки.
6. Після успішної (чи неуспішної) оплати LiqPay надсилає callback на server_url (POST /api/events/payment-callback).
7. Серверний обробник ProcessTicketPaymentCallbackCommand:
    a. перевіряє сигнатуру callback за алгоритмом, описаним у пункті 2.3.4;
    b. декодує data, витягує статус;
    c. якщо status = "success" - оновлює Ticket.Status на Paid, надсилає real-time нотифікацію учаснику та організатору через INotificationService;
    d. якщо платіж неуспішний - встановлює статус Failed, користувач отримує можливість повторити спробу.

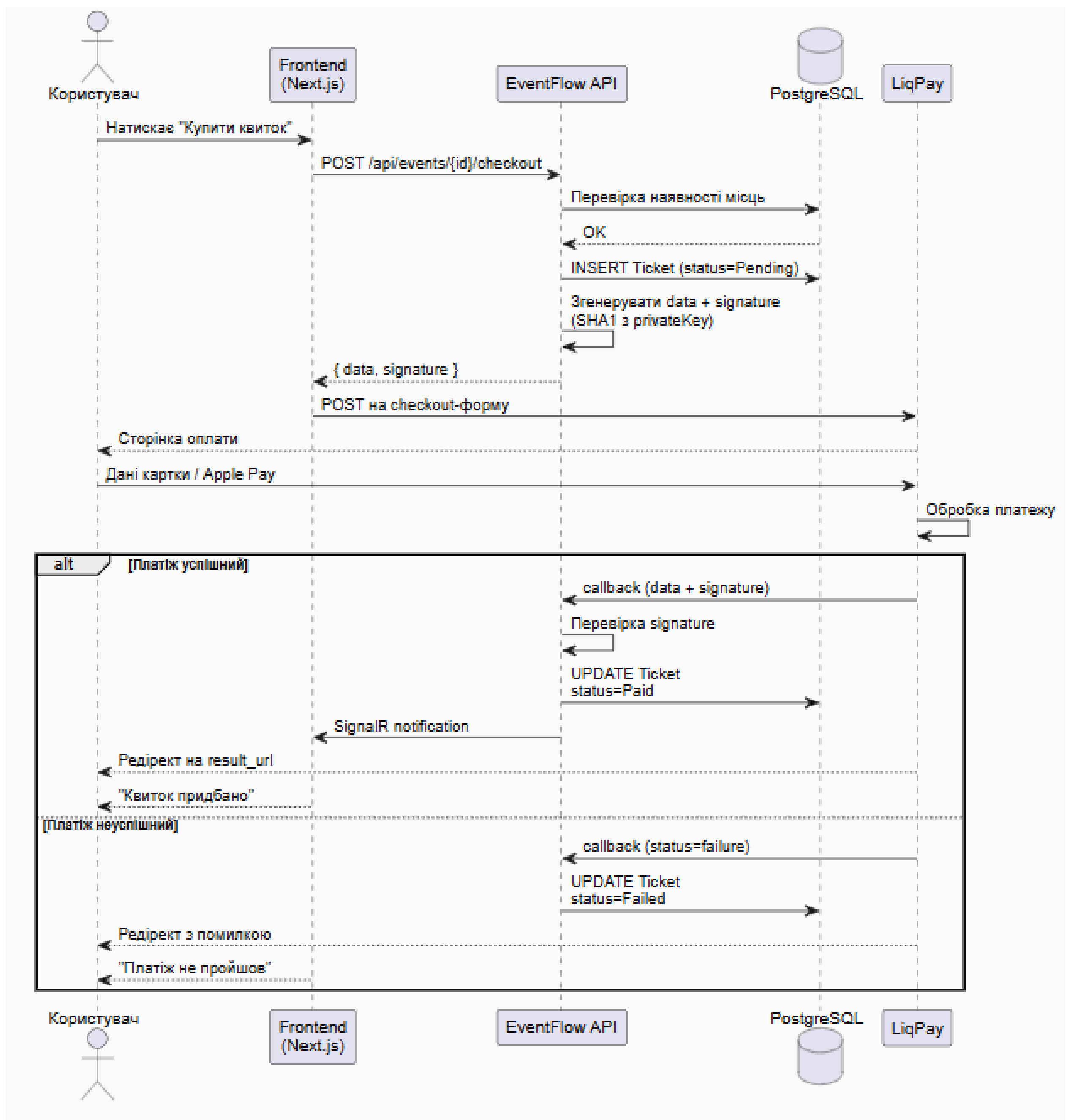

Рис. 3.5. Sequence-діаграма оплати квитка через LiqPay

Джерело: розроблено автором

Сервіс LiqPayService інкапсулює два ключові алгоритми взаємодії з LiqPay: формування платіжного запиту та перевірку сигнатури вхідних callback-ів. Метод GenerateCheckoutData створює JSON з параметрами платежу (сума, валюта, ідентифікатор замовлення, посилання для повернення клієнта та callback-у сервера),

кодує його в Base64 та обчислює сигнатуру за алгоритмом Base64(SHA1(privateKey + data + privateKey)). При отриманні callback від LiqPay про статус платежу метод VerifySignature обчислює очікувану сигнатуру за тим самим алгоритмом і порівнює з присланою - це гарантує, що повідомлення дійсно надіслане LiqPay, а не зловмисником. Повний код LiqPayService з усіма методами наведено у Додатку Д.

Для повернення коштів реалізовано окремий сервіс EventRefundService, що використовує API LiqPay refund для скасування транзакції.

3.3.8 AI-функціональність на базі Microsoft Semantic Kernel

Інтелектуальна підсистема EventFlow побудована на Microsoft Semantic Kernel. Цей фреймворк надає абстракцію над LLM-провайдерами (OpenAI, Groq Cloud, локальні моделі), підтримує патерн Function Calling, дозволяє організовувати функції в плагіни.

В EventFlow реалізовано чотири плагіни:

1. **EventManagementPlugin** - управління подіями від імені організатора. Функції: створення події, оновлення параметрів, публікація, скасування. Усередині кожної функції перевіряється, що поточний користувач є організатором цієї події.
2. **EventRegistrationPlugin** - реєстрація на події та керування реєстраціями. Функції: реєстрація на безкоштовну подію, перегляд власних реєстрацій, скасування реєстрації.
3. **OrganizerDashboardPlugin** - аналітика для організаторів. Функції: статистика по конкретній події, загальний дохід організатора, найпопулярніші події.
4. **EventSearchPlugin** - пошук подій. Функції: пошук за ключовими словами, за датою, за категорією, за локацією, рекомендації подібних подій.

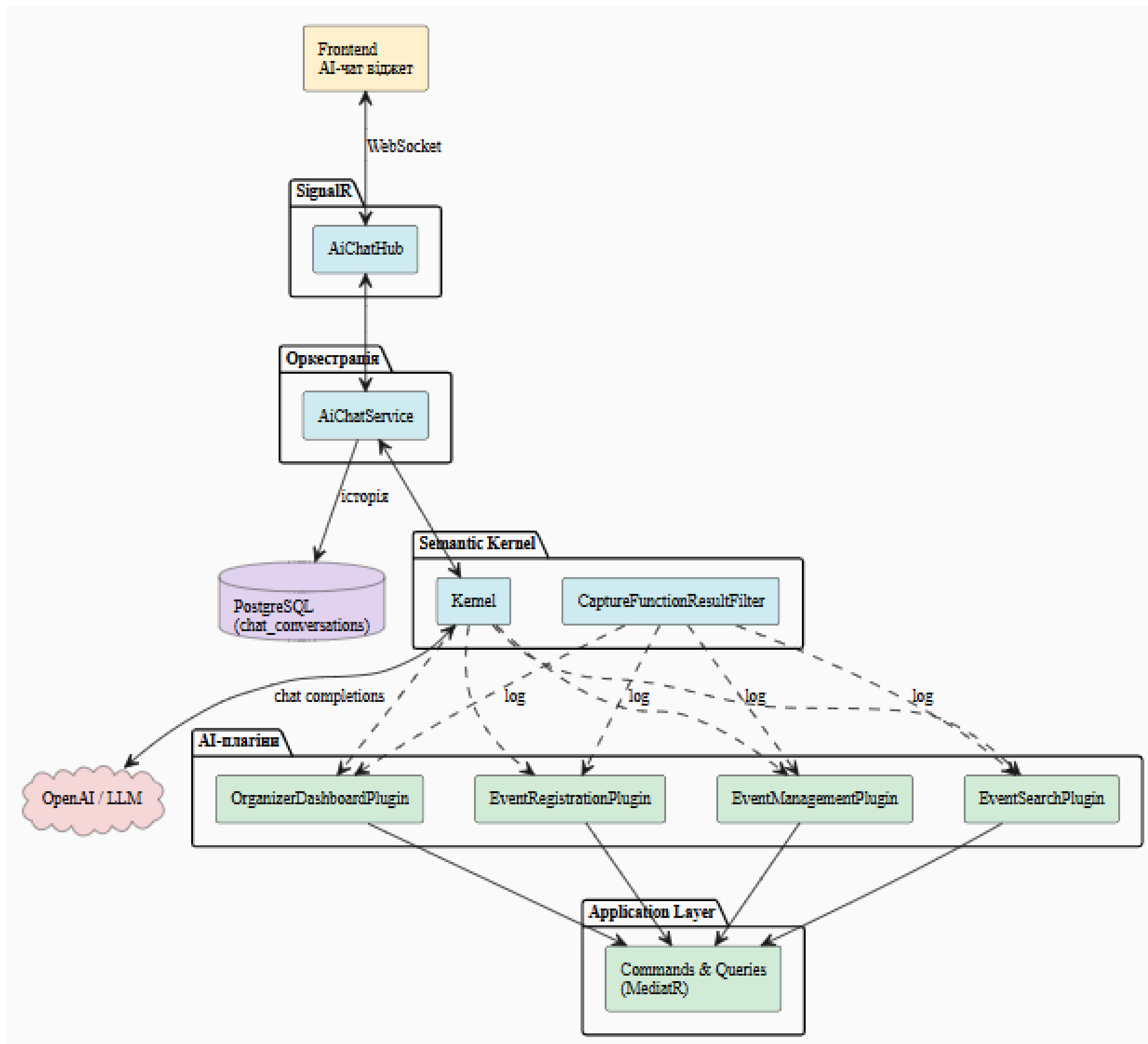

Рис. 3.6. Архітектура AI-pipeline EventFlow

Джерело: розроблено автором

Кожна функція AI-плагіна оголошена з атрибутом [KernelFunction] та має детальний опис у [Description], що дозволяє LLM зрозуміти, коли її викликати, та які параметри передати. Параметри функцій також супроводжуються [Description], що допомагає моделі коректно витягти значення з природномовного запиту користувача. Усередині кожної функції перевіряється авторизація поточного користувача через IUserContextService - наприклад, EventManagementPlugin дозволяє редагувати тільки події, в яких поточний користувач є організатором. Сервіс AiChatService оркеструє

розмову: зберігає історію в таблиці chat_conversations, формує системний промпт, викликає Semantic Kernel з реєстрованими плагінами, обробляє відповідь LLM (як текст, так і function calls), стрімить текстову частину клієнту через AiChatHub чанками. Повний код чотирьох плагінів та сервісу AiChatService наведено у Додатку Е.

Безпека AI-функцій забезпечена кількома рівнями:

- автентифікація: AI-чат доступний тільки автентифікованим користувачам;
- авторизація на рівні плагінів: кожна функція перевіряє, чи має поточний користувач право виконати дію (наприклад, редагувати тільки свої події);
- ізоляція контексту: AI бачить тільки доступні поточному користувачу дані, не може отримати дані інших користувачів через function calls.

3.3.9 Real-time комунікація через SignalR

Для real-time комунікації використано Microsoft SignalR - фреймворк, що надає абстракцію над WebSocket з автоматичним fallback на Server-Sent Events чи long polling для несумісних мереж. SignalR підтримує групи з’єднань, що дозволяє ефективно розсилати повідомлення підмножині клієнтів.

В EventFlow реалізовано два хаби:

AiChatHub (/hubs/ai-chat) - використовується виключно для стрімінгу відповідей AI-асистента. Коли AI-сервіс отримує від LLM чергову порцію тексту, вона негайно надсилається підключеному клієнту через цей хаб. Це створює ефект «AI пише в реальному часі», як у популярних чат-інтерфейсах.

NotificationsHub (/hubs/notifications) - забезпечує доставку in-app нотифікацій (не email і не push на пристрій, а саме сповіщення в інтерфейсі застосунку, доки в користувача відкрита вкладка). При підключенні клієнт передає в query параметрах свій userId та role. Хаб додає підключення в групи:

- user:{clerkUserId} - для отримання персональних нотифікацій;

- organizer:{clerkUserId} - якщо користувач має роль організатора чи адміна, для отримання нотифікацій про активність на його подіях;
- admin - глобальна група адмінів для отримання системних повідомлень.

Інтерфейс INotificationService оголошено у шарі Application, реалізація SignalRNotificationService - у шарі Infrastructure. Це забезпечує незалежність бізнес-логіки від технології real-time доставки: за потреби можна замінити SignalR на інший механізм без змін в Application.

На момент написання роботи реалізовано один сценарій нотифікацій - про реєстрацію учасника на подію. У обробнику RegisterForEventCommand після успішного створення реєстрації паралельно надсилаються три нотифікації: учаснику (підтвердження реєстрації), організатору (новий учасник), адмінам (для статистики). Архітектурно через типи payload (registration, cancellation, checkin, payment) система готова до розширення на інші сценарії без зміни Application Layer.

Рис. 3.7. Архітектура SignalR-хабів та інтеграції з Application Layer

Джерело: розроблено автором

При підключенні клієнта до NotificationsHub метод OnConnectedAsync витягує з query-параметрів userId (Clerk ID) та role користувача, після чого додає його SignalR-з’єднання у відповідні групи: user:{clerkUserId} для отримання персональних нотифікацій, organizer:{clerkUserId} для організаторів та адмінів, окрему групу admin для адміністраторів. Реалізація SignalRNotificationService (інтерфейс INotificationService з шару Application) надсилає повідомлення в ці групи через метод IHubContext.Clients.Group().SendAsync("ReceiveNotification", payload). Така схема

дозволяє таргетувати нотифікації без перебору з'єднань вручну. Повний код хабу та сервісу наведено у Додатку Ж.

3.3.10 Зберігання файлів в Cloudflare R2

Для зберігання обкладинок подій та фотографій з фотогалерей використано Cloudflare R2 - масштабоване об’єктне сховище. Інтеграція реалізована через офіційний AWS SDK для .NET.

Реалізовано схему з pre-signed URL: клієнт запитує у бекенді короткоживучу URL для прямого завантаження файлу в S3, минаючи серверний застосунок. Це знижує навантаження на сервер та зменшує затримки при завантаженні великих файлів. Після успішного завантаження клієнт повідомляє бекенд про публічну URL отриманого файла.

Метод GetPresignedUploadUrlAsync сервісу S3StorageService генерує підписану URL з обмеженим часом життя (15 хвилин) для прямого завантаження файлу клієнтом у Cloudflare R2, минаючи серверний застосунок. Це знижує навантаження на бекенд та пришвидшує завантаження великих файлів. Метод UploadAsync буферизує вхідний потік у MemoryStream перед відправкою у R2 - це необхідно, оскільки Cloudflare R2 (на відміну від AWS S3) відхиляє chunked transfer encoding із помилкою сигнатури, а ASP.NET Core's IFormFile.OpenReadStream() повертає не-seekable потік, що змушує AWS SDK переходити саме в chunked-режим. Повний код сервісу наведено у Додатку Ж.

Окремий компонент TicketCleanupJob (у папці Email, історично) виконує фонову задачу очищення прострочених Pending квитків - якщо користувач ініціював checkout, але не оплатив протягом 30 хвилин, відповідне місце звільняється для інших.

3.3.11 Виплати організаторам подій

Однією з ключових бізнес-задач системи управління подіями з продажем квитків є передача отриманих коштів організаторам. У EventFlow обрано модель «маркетплейс єдиного продавця» (англ. *marketplace of one merchant*), за якої платформа є зареєстрованим продавцем у системі LiqPay та отримує всі платежі на власний мерчант-акаунт, а організатори отримують свою частку коштів за результатами планового розрахунку. Така модель була обрана з огляду на те, що альтернативний підхід - розщеплення платежу між мерчантами LiqPay (англ. *split payment*) - вимагає від кожного організатора реєстрації власного бізнес-акаунту в Приват24, що передбачає наявність ФОП або юридичної особи в Україні. Для широкого кола неспеціалізованих організаторів така вимога створює надмірний бар'єр входу.

Архітектура розрахунків з організаторами реалізована через дві нові доменні сутності:

- OrganizerPayoutAccount - платіжні реквізити організатора (1:1 з User). Атрибути: Method (метод виплати - банківський переказ або LiqPay-мерчант), Iban, RecipientFullName, опційні LiqPayPublicKey та LiqPayPrivateKeyEncrypted (зашифрований за алгоритмом AES-256 для майбутньої підтримки split-payment режиму), VerifiedAt (мітка підтвердження адміністратором).
- Payout - фактичний розрахунковий запис. Атрибути: OrganizerId, PeriodStart/PeriodEnd, GrossAmount (валова сума), PlatformFeeAmount (комісія платформи), NetAmount (сума до виплати), Status (Pending → Approved → Paid, або Failed/Rejected), TicketIds (JSONB-масив ідентифікаторів квитків, що покриваються цим розрахунком), PaidAt, LiqPayTransactionId (для split-payment режиму).

Бізнес-логіка виплат реалізована у вигляді фонової задачі WeeklyPayoutAggregationJob, що зареєстрована в Hangfire як recurring-задача з cron-розкладом «щопонеділка о 02:00 UTC». Алгоритм роботи задачі:

1. Знайти всі квитки зі статусом Paid, дата події яких є меншою за now - PayoutDelayDays (за замовчуванням 3 дні - буфер для опрацювання потенційних chargeback-запитів) та які ще не входять до жодного Payout.
2. Згрупувати знайдені квитки за організатором події.
3. Для кожного організатора розрахувати: gross = SUM(ticket.amount), fee = gross × PlatformFeePercentage / 100, net = gross − fee.
4. Створити запис Payout зі статусом Pending та надіслати email-сповіщення організатору, а також SignalR-нотифікацію адміністраторам про необхідність опрацювання нових виплат.

Адміністративна частина flow винесена у AdminController: ендпоінт GET /api/admin/payouts?status=Pending повертає список очікуваних виплат, POST /api/admin/payouts/{id}/mark-paid фіксує факт ручного банківського переказу та зберігає референс транзакції, POST /api/admin/payouts/{id}/reject повертає квитки до пулу нерозподілених для повторної агрегації. Окремо реалізовано два транзакційні email-шаблони: SendPayoutPendingAsync (надсилається організатору при створенні нового Payout фоновою задачею) та SendPayoutPaidAsync (надсилається після підтвердження виплати адміністратором, з референсом банківської транзакції в тілі листа). Обидва шаблони відправляються через ту саму інфраструктуру Hangfire-задач, що й попередні листи (підтвердження реєстрації, оплати, скасування) - це забезпечує ретраїбельність на випадок тимчасових збоїв SMTP-сервера.

Для організаторів передбачено окремий розділ профілю з трьома елементами: (1) форма платіжних реквізитів з валідацією IBAN за алгоритмом MOD-97, (2) картка зведеної інформації про доходи (за весь час, в очікуванні виплати, останній виплачено), (3) історія виплат з можливістю фільтрування. Розрахунок комісії платформи параметризовано через секцію Payouts у appsettings.json, що дозволяє адміністратору налаштовувати ставку без зміни коду.

Спроєктована модель є відкритою для розширення: за умови, що організатор у майбутньому надасть LiqPay-мерчант ключі, система зможе перейти у режим

автоматичного split-payment без міграції даних - поля LiqPayPublicKey та LiqPayPrivateKeyEncrypted вже передбачені в схемі.

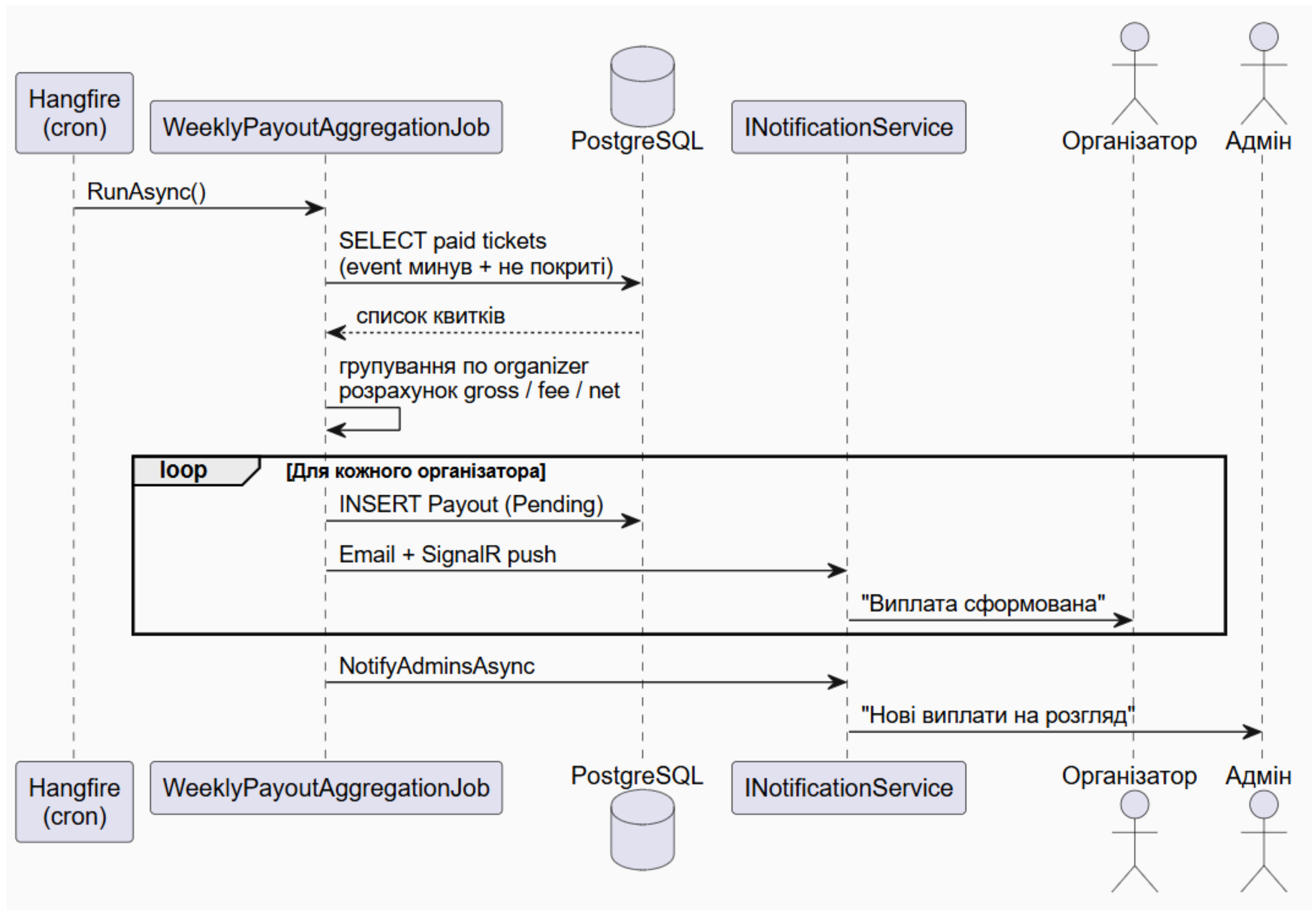

Рис. 3.8. Sequence-діаграма агрегації виплати організатору

Джерело: розроблено автором

3.3.12 Відновлення пароля та багатофакторна автентифікація

Початкова реалізація автентифікації через Clerk (підрозділ 3.3.6) обмежувалася SSO-провайдерами та простим email/password входом. Для підвищення безпеки облікових записів, особливо для користувачів, що реєструються через email/password (без захисту, який забезпечує OAuth-провайдер на кшталт Google), у систему додано два механізми: відновлення пароля та обов'язкову багатофакторну автентифікацію (MFA, англ. *multi-factor authentication*).

Відновлення пароля. Сторінку входу реалізовано на базі компонента <SignIn /> від Clerk, який автоматично відображає посилання «Забули пароль?» за умови, що в Clerk Dashboard у розділі User & Authentication активовано password-стратегію з механізмом доставки коду відновлення через email. Сам процес повністю обробляється на стороні Clerk: користувач отримує одноразовий код на свою електронну пошту, вводить його у форму, після чого може встановити новий пароль. Жодного коду на боці серверної частини EventFlow для цієї функціональності не потрібно - цей підхід є типовим для архітектури з делегованою автентифікацією та зменшує поверхню атаки на власну інфраструктуру.

Багатофакторна автентифікація. У Clerk Dashboard активовано два типи другого фактора: одноразові коди з authenticator-додатків (TOTP - Time-based One-Time Password, RFC 6238) як основний та SMS-коди як резервний. Політика налаштована таким чином, що MFA є обов'язковою для користувачів, які зареєструвалися без верифікованого соціального провайдера - тобто саме для тих, хто потенційно вразливіший до компрометації пароля. Користувачі, що увійшли через Google, додатково не зобов'язуються налаштовувати другий фактор, оскільки Google уже забезпечує власний рівень MFA.

На рівні бекенду додано перевірку наявності claim'a mfa у JWT-токені для критичних endpoint-ів (адміністративні дії, керування виплатами, видалення подій). Перевірка реалізована через ASP.NET Core authorization policy:

Лістинг 3.1 Перевірка клейма mfa

```
options.AddPolicy("MfaRequired", policy =>
      policy.RequireAssertion(ctx =>
          ctx.User.HasClaim(c =>
                 (c.Type == "mfa"
               || c.Type == "two_factor"
               || c.Type == "amr")
               && (c.Value.Equals("true",
StringComparison.OrdinalIgnoreCase)
```

```
                || c.Value.Contains("mfa",
StringComparison.OrdinalIgnoreCase)
                || c.Value.Contains("totp",
StringComparison.OrdinalIgnoreCase)))));
```

Контролери, що оперують чутливими операціями, маркуються атрибутом [Authorize(Policy = "MfaRequired")], що гарантує: навіть скомпрометований основний фактор не дає змоги виконати такі дії без другого. Розширена форма перевірки враховує те, що Clerk залежно від версії API може повертати інформацію про другий фактор у різних клеймах: mfa (булеве значення), two_factor (булеве або текстове) або amr (Authentication Methods References, RFC 8176, що містить значення на кшталт mfa, otp, totp). Така толерантність до формату захищає систему від збоїв при майбутніх змінах у клеймах Clerk.

Інтерфейс керування безпекою. На стороні клієнтського застосунку у профіль користувача додано окремий розділ «Безпека», що монтує компонент <UserProfile /> від Clerk. Цей компонент надає користувачу єдиний інтерфейс для зміни пароля, керування MFA-факторами (додавання/видалення TOTP-пристрою, перегляд recovery codes), перегляду активних сесій з можливістю їх примусового завершення та керування підключеними соціальними обліковими записами. Імплементація на стороні Next.js фактично зводиться до десятка рядків TSX-коду, тоді як уся складна логіка криптографії та зберігання чутливих даних залишається на стороні Clerk.

Такий розподіл відповідальностей між власною кодовою базою та зовнішнім сервісом ілюструє один із принципів сучасної інженерії програмного забезпечення: не реалізовувати самостійно те, що вже якісно вирішено спеціалізованими сервісами, особливо у безпеково-критичних доменах.

3.4 Опис програмної реалізації клієнтської частини

3.4.1 Стек і обґрунтування технологічних виборів

Клієнтська частина реалізована на фреймворку Next.js 16 з App Router. Цей вибір обґрунтовано наступним:

- *SSR (Server-Side Rendering)* для сторінок подій критично важливий для SEO. Сторінки з SSR індексуються пошуковими системами без затримки, на відміну від чистих SPA, де рендеринг відбувається на клієнті.
- *Server Components* дозволяють виконувати запити до API безпосередньо на сервері при рендерингу, без передачі чутливих токенів на клієнт.
- *Server Actions* спрощують роботу з формами, виконуючи мутації на сервері без явних API-ендпоінтів.
- *Streaming рендерингу* дозволяє показувати інтерфейс користувачу частинами, не очікуючи повного завантаження даних.
- *Image Optimization* автоматично оптимізує зображення, що особливо важливо для сторінок з обкладинками подій та фотогалерей.

Альтернативи: Create React App застаріла та більше не підтримується; Vite + React надає лише SPA без SSR; Remix був гідною альтернативою, але після злиття з React Router у 2024 році її розвиток уповільнився.

TypeScript забезпечує типобезпеку на всьому шляху від API-відповідей до UI-компонентів. Tailwind CSS у поєднанні з shadcn/ui (колекцією копіювально-вставних компонентів на базі Radix UI) забезпечує консистентний дизайн з мінімальним обсягом кастомного CSS.

3.4.2 Процес дизайну: Figma та Figma Make

Процес створення UI/UX дизайну системи EventFlow складався з кількох етапів.

Етап 1. Дослідження конкурентів та збір референсів. Проаналізовано візуальні рішення Eventbrite, Luma, Meetup, а також референси з Dribbble та Behance за

тематикою «event management dashboard». Зібрано moodboard у Figma з понад 30 прикладами успішних інтерфейсів.

Етап 2. Низькодеталізовані макети (wireframes). На цьому етапі визначено базову структуру кожного основного екрану: головна сторінка, каталог подій, деталі події, особистий кабінет, AI-чат, адмін-панель. Wireframes створювались монохромно, з фокусом на ієрархію інформації та користувацькі сценарії.

Етап 3. Високодеталізовані макети. На основі затверджених wireframes створено повноколірні макети з реальним контентом, типографікою та компонентами.

Етап 4. Використання Figma Make. Особливістю процесу стало активне використання інструменту *Figma Make* - функції генерації варіантів інтерфейсів за допомогою штучного інтелекту, інтегрованої безпосередньо в Figma. Замість тривалого ручного перебору варіантів окремих компонентів (картка події, форма реєстрації, віджет AI-чату) ставилися текстові промпти типу «event card with image, title, date, registration button, modern minimalistic style» - Figma Make генерував кілька альтернатив, з яких обиралися найкращі для подальшого доопрацювання. Це скоротило час підбору фінального вигляду компонентів у 3-4 рази порівняно з ручним проєктуванням.

Етап 5. Дизайн-система. На основі затверджених макетів створено дизайн-систему - структурований набір компонентів, кольорів, шрифтів та правил їх використання. Це забезпечує консистентність інтерфейсу та пришвидшує подальший розвиток продукту.

Етап 6. Передача в розробку. Фінальні макети та дизайн-токени (кольори, відступи, типографіка) експортовано у формат, придатний для прямої трансляції в Tailwind CSS конфігурацію та CSS-змінні.

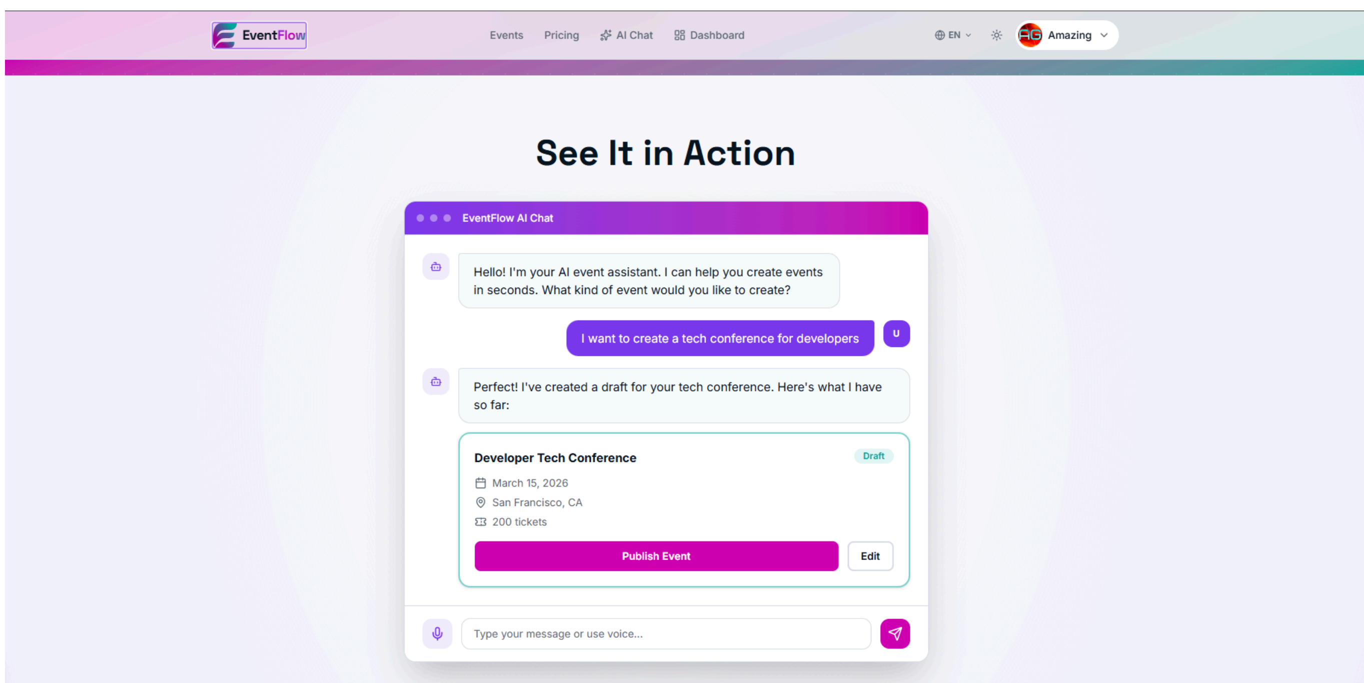

Рис 3.8. Секція лендингу

Джерело: розроблено автором

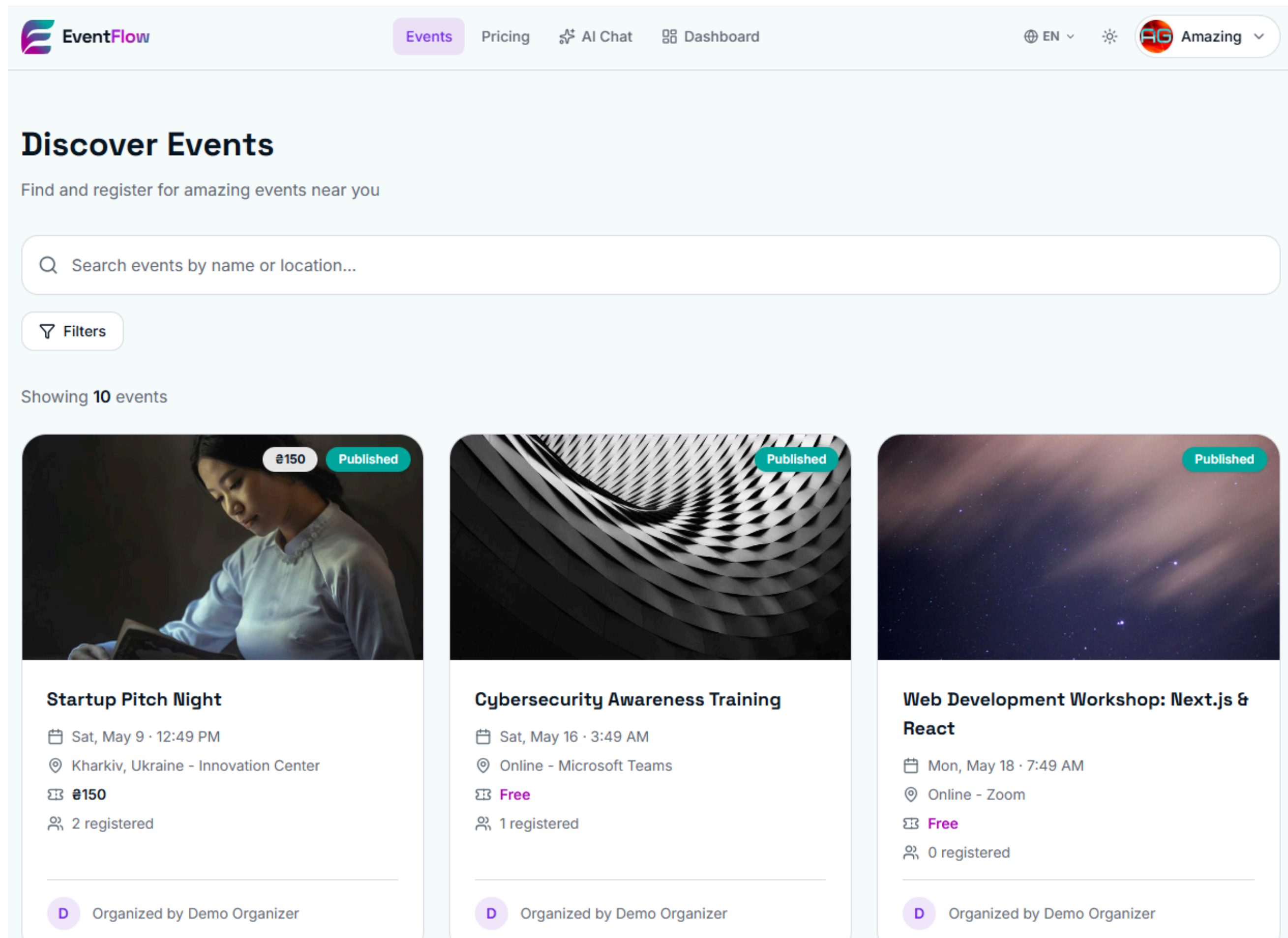

Рис. 3.9. Екран переліку подій

Джерело: розроблено автором

3.4.3 Кольорова палітра

Кольорова палітра системи EventFlow побудована за принципом семантичної структуризації. Вона складається з кількох груп.

Primary (основні кольори бренду) - задають візуальну ідентичність продукту, використовуються для головних кнопок дій, посилань, акцентів.

Secondary (додаткові кольори) - для другорядних елементів інтерфейсу, неактивних станів кнопок, заголовків блоків.

Accent (акцентні кольори) - для виділення особливих елементів: AI-функціональності, нових подій, спеціальних пропозицій.

Semantic (семантичні кольори) - мають фіксований смисл і використовуються однаково по всьому інтерфейсу:

- success (зелений) - успішні операції, підтверджена реєстрація, оплачений квиток;
- warning (оранжевий) - попередження, наближення дедлайну реєстрації;
- error (червоний) - помилки, неуспішна оплата, заблоковані події;
- info (синій) - інформаційні повідомлення.

Neutral (нейтральна шкала) - від білого до чорного, через декілька градацій сірого. Використовується для тексту, фону, рамок, розділювачів.

Реалізація палітри в коді виконана через CSS Custom Properties (змінні) у файлі globals.css, що дозволяє автоматично перемикати між Light та Dark темами без перевантаження. Tailwind CSS конфігурація розширена кастомними кольорами, посилаючись на ці змінні.

При виборі кольорів враховано психологію сприйняття для event-індустрії: енергійні, але не нав’язливі акцентні тони, що передають відчуття активності та руху, в поєднанні з нейтральним фоном для тривалого читання списків подій. Усі поєднання кольорів перевірено на відповідність стандарту WCAG AA для контрастності тексту на фоні (співвідношення не менше 4.5:1 для основного тексту).

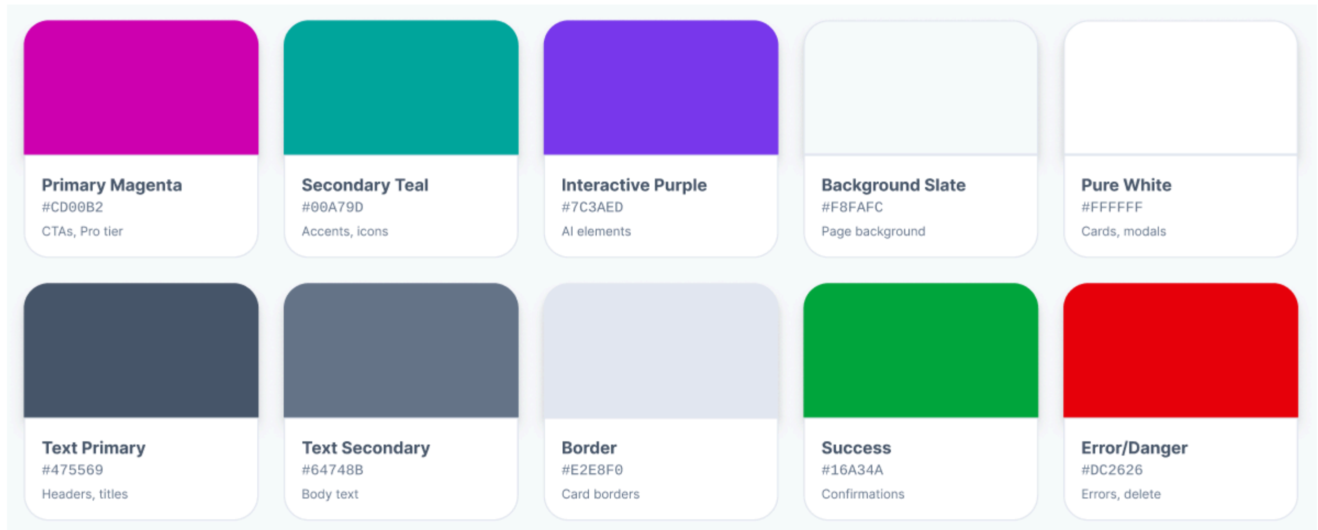

Рис. 3.10. Кольорова палітра EventFlow з HEX-кодами та токенами Tailwind

Джерело: розроблено автором

3.4.4 Типографіка

Система типографіки спроєктована для забезпечення читабельності на тривалих сторінках з описами подій та коментарями, з підтримкою кирилиці.

Основний шрифт використовується для всього UI-тексту - кнопок, форм, списків, описів. Обрано сучасний sans-serif шрифт зі змінною товщиною, що дозволяє використовувати єдиний файл шрифту для всіх ваг (від 100 до 900) і оптимізує продуктивність завантаження.

Акцентний шрифт використовується для великих заголовків (H1, H2 на лендінгах та головних сторінках) для створення візуального контрасту та виразності.

Type scale структуровано за рівнями:

- Display (48-72px) - герой-секції на лендінгу;
- H1 (36-48px) - заголовки сторінок;
- H2 (28-32px) - заголовки секцій;
- H3 (22-24px) - заголовки карток;

- H4 (18-20px) - підзаголовки;
- Body (16px) - основний текст;
- Small (14px) - допоміжний текст, метадані;
- Caption (12px) - підписи, мітки.

Підключення шрифтів реалізовано через next/font - нативний механізм Next.js для self-hosted шрифтів. Це усуває проблему layout shift під час завантаження, не передає дані стороннім сервісам (Google Fonts) та забезпечує оптимальне кешування.

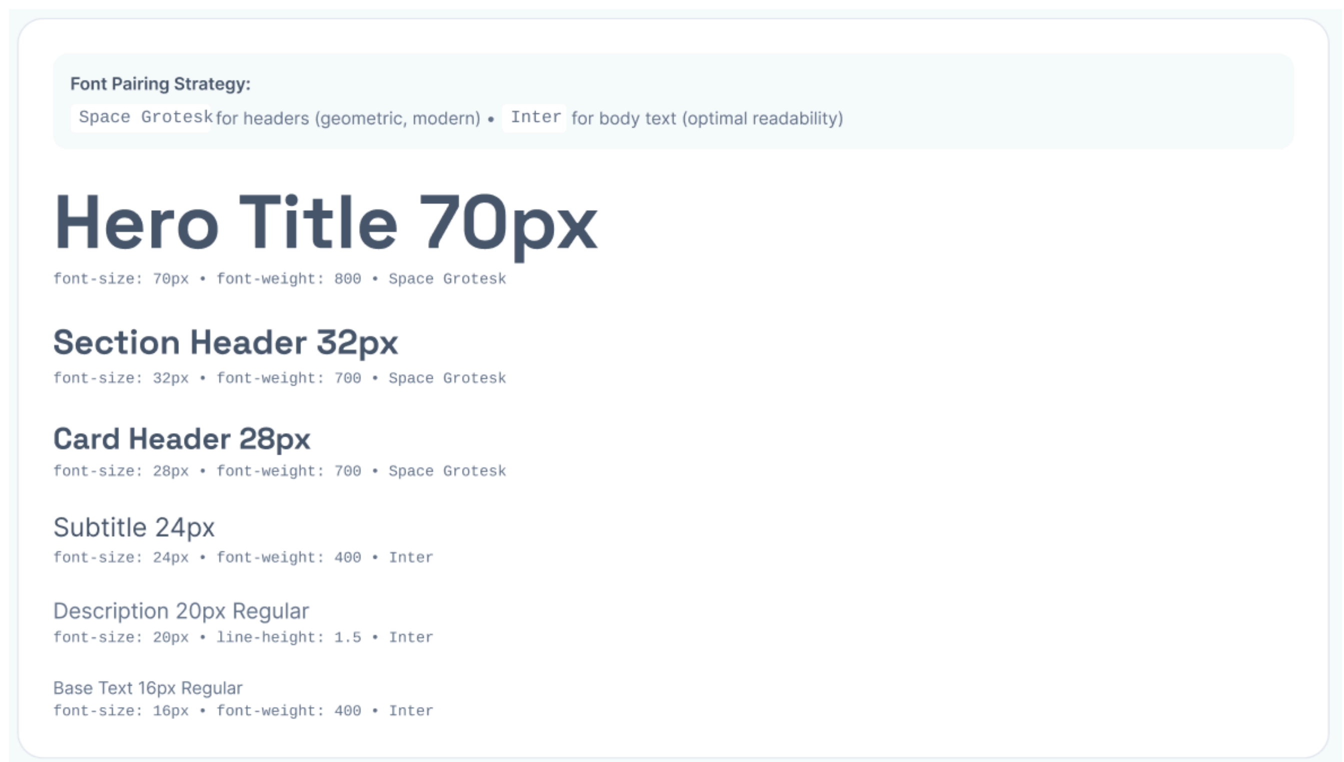

Рис. 3.11. Type system EventFlow з прикладами використання

Джерело: розроблено автором

3.4.5 Структура клієнтського проєкту

Клієнтський проєкт організовано за принципами функціонального розділення з елементами Feature-Sliced Design.

```
eventflow-client/src/
 ├── app/
 |    ├── (auth)/           (група маршрутів автентифікації)
 |    ├── events/           (каталог та деталі подій)
 |    ├── dashboard/        (особистий кабінет)
 |    ├── profile/
 |    |    ├── page.tsx                (особистий кабінет)
 |    |    ├── security/page.tsx       (безпека: <UserProfile /> від Clerk)
 |    |    └── payouts/page.tsx        (виплати: реквізити, дохід, історія)
 |    ├── admin/            (адмін-панель)
 |    ├── layout.tsx        (кореневий layout)
 |    └── globals.css       (CSS-змінні, теми)
 ├── features/
 |    ├── create-event/
 |    ├── ai-chat/
 |    ├── event-registration/
 |    ├── photo-upload/
 |    └── subscription/
 ├── entities/
 |    ├── event/
 |    ├── user/
 |    └── ticket/
 ├── widgets/
 |    ├── header/
 |    ├── footer/
 |    ├── event-card/
 |    └── notification-toaster/
 ├── shared/
 |    ├── ui/               (кнопки, форми, модалки - shadcn/ui)
 |    ├── api/              (типізований API-клієнт)
 |    ├── lib/              (утиліти)
 |    └── config/
 ├── hooks/                 (useAiChat, useNotifications, тощо)
 ├── contexts/              (ThemeContext, LocaleContext)
 ├── lib/                   (інтеграції з Clerk, SignalR)
 ├── i18n/                  (uk, en переклади)
 └── types/                 (глобальні TypeScript типи)
```

Рис. 3.12. Client app структура

Джередо: розроблено автором

3.4.6 Ключові реалізації клієнтської частини

Інтеграція з Clerk. У кореневому layout-і застосунок обгорнуто в <ClerkProvider> з налаштуваннями локалізації (українська та англійська). Захист маршрутів реалізовано через middleware.ts, що використовує clerkMiddleware для

перевірки автентифікації перед рендерингом приватних шляхів. Список приватних і публічних маршрутів задано через matcher-конфігурацію Next.js. Для серверних запитів до бекенду JWT-токен Clerk автоматично прикріплюється через axios-інтерсептор у клієнтському API-модулі. Повний код layout, middleware та інтерсептора наведено у Додатку Й.

Форма створення події. Форма створення події побудована на React Hook Form у поєднанні з Zod-схемою валідації. Та сама схема використовується як для клієнтської валідації перед відправкою запиту, так і для типізації даних форми через z.infer<typeof eventSchema>. Це усуває розсинхронізацію типів між валідатором і компонентом. Форма складається з кількох логічних блоків (основна інформація, дата та локація, ціна та ліміт учасників, обкладинка) з покроковим навігатором для покращення UX на великих формах. Повний код Zod-схеми та компонента наведено у Додатку Й.

Create New Event

Fill in the details below or use AI to create your event faster

Try AI Event Creation

Create events in seconds using voice or text chat

Use AI Assistant

Event Title *

e.g., AI & Machine Learning Summit 2026

Description *

Tell attendees what your event is about...

Event Date *

Select date

Start Time *

Start Time

Location

e.g., Convention Center, San Francisco, CA

Ticket Price (UAH)

0 (free)

Free plan: free events only. Upgrade to Pro →

Max Attendees

Up to 30

Free plan: up to 30 attendees. Upgrade to Pro →

Рис. 3.13. Інтерфейс форми створення події

Джерело: розроблено автором

Каталог подій. Реалізований як Server Component для оптимального SEO. Запит до API виконується безпосередньо на сервері Next.js. Фільтри та пагінація використовують механізм URL-параметрів через useSearchParams, що забезпечує можливість поділитися URL-ом з конкретним станом фільтрів.

Сторінка події. Server Component з SEO-оптимізацією через generateMetadata (динамічний <title>, Open Graph теги для шеринга в соцмережах). Кнопка реєстрації або купівлі квитка є Client Component, оскільки потребує доступу до автентифікованого користувача та інтерактивності.

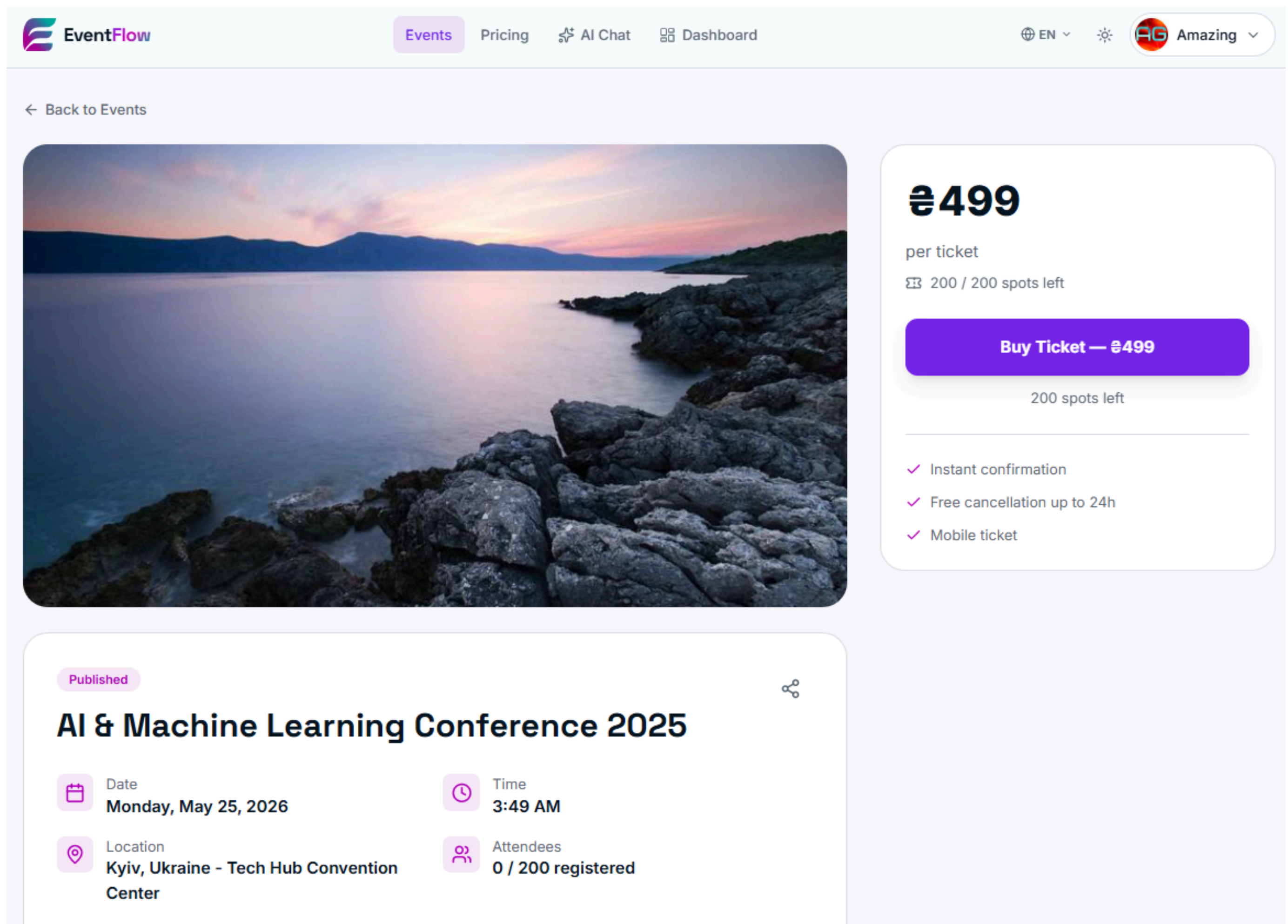

Рис. 3.14. Сторінка події з кнопкою реєстрації

Джерело: розроблено автором

Інтеграція оплати. При натисканні «Купити квиток» клієнт викликає API POST /api/events/{id}/checkout, отримує data та signature від сервера, формує приховану форму з action="https://www.liqpay.ua/api/3/checkout" та виконує submit. Користувач перенаправляється на сторінку LiqPay. Після оплати LiqPay повертає його на result_url з

параметрами стану. Тим часом сервер обробляє callback паралельно і оновлює стан квитка.

QR-код для оплати

Скануйте з Приват24

Тестовий режим

LIQPAY

Ticket for test

До оплати 10.00 UAH

Pay

Оплатити через G Pay

Pay

або

Номер картки

0000 0000 0000 0000

Термін дії

MM / PP

CVV2 / CVC2

123

Email для отримання квитанції

example@domain.com

Натискаючи на кнопку «Оплатити», ви підтверджуєте що ознайомлені з переліком інформації про послугу та надаєте згоду з умовами публічного договору

Рис. 3.15. Сторінка оплати на LiqPay

Джерело: розроблено автором

AI-чат-віджет. Реалізований як плаваюча кнопка в правому нижньому куті, що відкриває панель чату. Кастомний React-хук useAiChat керує підключенням до

AiChatHub через @microsoft/signalr. При надсиланні повідомлення викликається API-ендпоінт; відповідь стрімиться чанками через хаб і поступово відображається в UI з ефектом «typing» - кожен новий чанок додається до поточного буфера повідомлення в стейті. Хук інкапсулює логіку з’єднання, обробки помилок з'єднання та автоматичного відновлення при розриві. Історія розмови зберігається на сервері та автоматично завантажується при відкритті чату. Повний код хука наведено у Додатку Й.

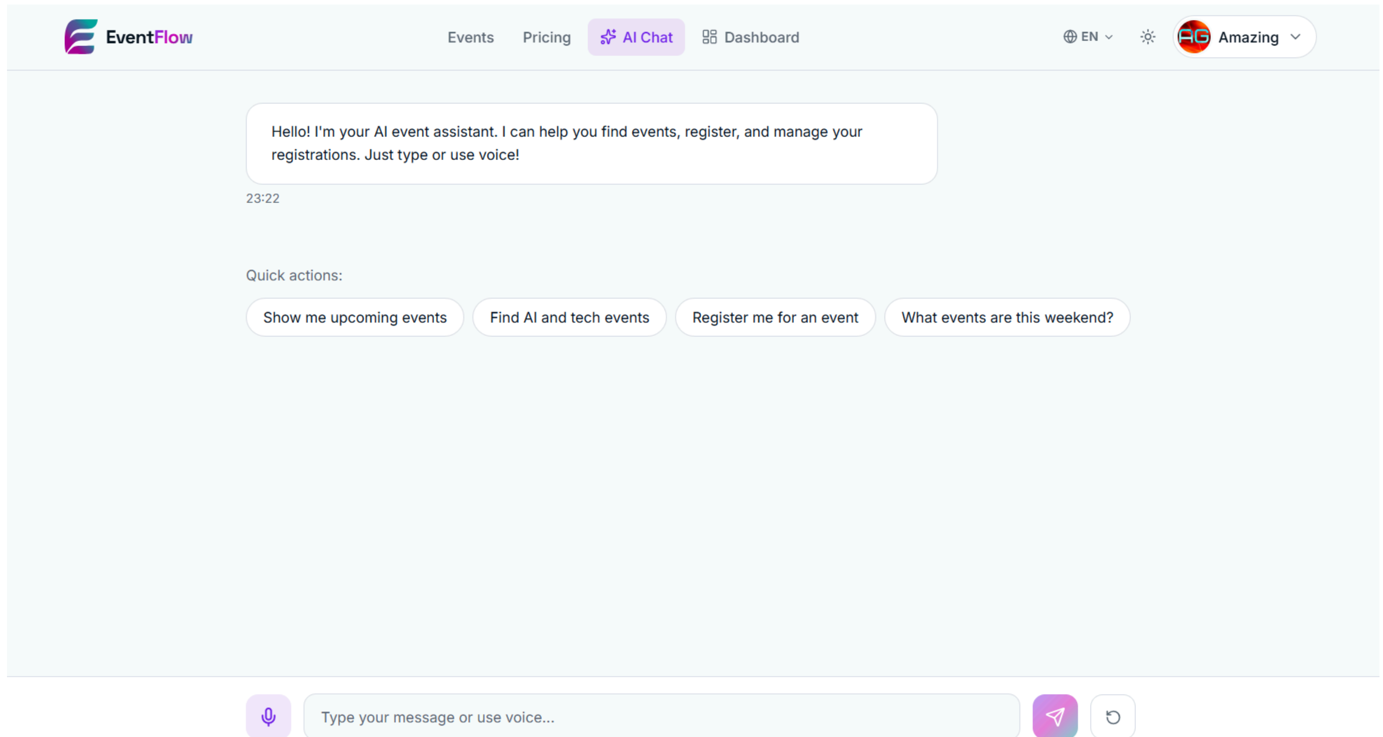

Рис. 3.16. AI-чат-віджет у дії

Джерело: розроблено автором

Завантаження фото. Компонент PhotoUploader реалізує drag-and-drop інтерфейс із прев'ю обраних файлів. При підтвердженні клієнт виконує триетапний flow: запит у бекенді pre-signed URL для конкретного файла, прямий PUT-запит у Cloudflare R2 за отриманою URL, повідомлення бекенду про публічну URL отриманого файла для збереження в БД. Такий підхід знижує навантаження на серверний застосунок (файл не проходить через нього) і пришвидшує завантаження. Повний код компонента та інтеграції з API наведено у Додатку Й.

Локалізація (i18n). Підтримка двох мов - української та англійської. Використано бібліотеку next-intl, що інтегрується з App Router. Перемикач мов розташовано в шапці. Усі рядки винесено у JSON-файли в папку i18n/messages/. Локаль зберігається в cookie для повторних візитів.

Real-time нотифікації. Кастомний хук useNotifications встановлює з’єднання з NotificationsHub при автентифікації користувача та слухає подію ReceiveNotification. При отриманні повідомлення відображається toast-нотифікація через бібліотеку shadcn-toast. Це забезпечує миттєві оновлення інтерфейсу - наприклад, організатор бачить новий запис на свою подію без перезавантаження сторінки.

Сторінка виплат організатора. Сторінка побудована на TanStack Query з трьома незалежними запитами:

- getEarnings() (зведена статистика);
- getPayoutAccount() (поточні реквізити);
- getMyPayouts() (історія);

Форма редагування реквізитів реалізована з умовним рендерингом полів - для методу manual_bank_transfer відображається IBAN та ПІБ отримувача, для liqpay_merchant - пара public/private ключів, причому private-ключ зберігається у полі <input type="password"> для захисту від випадкового експонування. Після успішного збереження реквізитів кеш TanStack Query інвалідується, що автоматично перезавантажує дані без перезавантаження сторінки.

Сторінка безпеки. Імплементація фактично делегує всю логіку Clerk: один TSX-компонент монтує <UserProfile routing="hash" />.

3.4.7 Управління станом та оптимізації

Серверний стан керується через TanStack Query. Цей вибір забезпечує:

- автоматичне кешування з інвалідацією за тегами;

- background refetching для забезпечення актуальності даних;
- optimistic updates для миттєвого відгуку UI;
- DevTools для зручного дебагінгу.

Локальний UI-стан обробляється стандартними React-хуками. Для глобальних значень (тема, локаль, поточна сесія) використано React Context з мемоізацією, щоб уникнути зайвих ререндерів.

Розділення Server vs Client Components. За замовчуванням всі компоненти в App Router є Server Components. Директива "use client" додається тільки коли компонент потребує інтерактивності, доступу до браузерних API, хуків стану. Це мінімізує кількість JavaScript, що передається клієнту, і покращує продуктивність та SEO.

Оптимізації продуктивності:

- ліниве завантаження (dynamic import) для важких компонентів типу AI-чату;
- prefetch посилань на видимі картки подій (<Link prefetch>);
- оптимізовані зображення через <Image> з автоматичним підбором розміру та формату (WebP/AVIF);
- code splitting на рівні маршрутів (нативно через App Router).

3.5 Тестування та керівництво користувача

3.5.1 Тестування

Тестування системи EventFlow реалізовано в п’яти проєктах, що відповідають різним рівням пірамідного підходу до тестування.

Application.UnitTests - модульні тести для шару Application. Покривають обробники команд та запитів, валідатори, бізнес-логіку. Залежності (БД, зовнішні

сервіси) замокані через бібліотеку NSubstitute. Тести швидкі (виконуються за мілісекунди), не потребують запуску інфраструктури.

Api.Tests.Integration - інтеграційні тести через EventFlowWebFactory (наслідник WebApplicationFactory<Program> з ASP.NET Core). Запускають реальний інстанс застосунку з реальною PostgreSQL у Docker-контейнері (через Testcontainers). Це дозволяє перевіряти повний шлях запиту: HTTP → Middleware → Controller → MediatR → Handler → DbContext → PostgreSQL. Окремо реалізовано тести інтеграції з S3 та AI-сценарії з використанням FakeChatCompletionService для детермінованих результатів.

PerformanceTests - навантажувальні тести для критичних API-ендпоінтів. Дозволяють виявити проблеми продуктивності до їх потрапляння в продакшен.

Tests.Common - спільні утиліти, базові класи для тестів, конфігурація Testcontainers.

Tests.Data - тестові фікстури та білдери даних. Клас EventsData надає готові комбінації даних подій для типових сценаріїв.

Unit-тести Application шару використовують xUnit як test framework, FluentAssertions для виразних перевірок та NSubstitute для мокування залежностей (репозиторіїв, зовнішніх сервісів). Кожен обробник тестується ізольовано: задається стан моків, викликається Handle, перевіряється результат і взаємодія з моками. Integration-тести через EventFlowWebFactory (наслідник WebApplicationFactory<Program>) запускають реальний інстанс застосунку з реальною PostgreSQL у Docker-контейнері, що піднімається через Testcontainers. Це дозволяє перевіряти повний шлях запиту: HTTP → Middleware → Controller → MediatR → Handler → DbContext → PostgreSQL. AI-сценарії покриті окремими integration-тестами, що використовують FakeChatCompletionService - детермінований замінник реального LLM, який повертає фіксовані відповіді для перевірних промптів. Окрему категорію

складають тести фінансової логіки: тести валідатора IBAN (за алгоритмом ISO 13616 MOD-97 з перевіркою на стандартних векторах GB/DE/FR та українському IBAN), тести коректного округлення сум виплат до двох знаків після коми, тести state-машини сутності Payout (заборона повторного MarkPaid, заборона переведення з Rejected у Paid), а також тести криптографічного round-trip для PayoutEncryptionService (зокрема перевірка того, що шифрування одного й того самого plaintext двічі дає різні ciphertext через випадковий nonce, та що модифікація останнього байта ciphertext призводить до CryptographicException під час декодування). Приклади unit-тестів та integration-тестів наведено у Додатку К.

Integration-тести через EventFlowWebFactory (наслідник WebApplicationFactory<Program>) запускають реальний інстанс застосунку з реальною PostgreSQL у Docker-контейнері, що піднімається через Testcontainers. Це дозволяє перевіряти повний шлях запиту: HTTP → Middleware → Controller → MediatR → Handler → DbContext → PostgreSQL - на відміну від unit-тестів, де всі залежності замокані. Перед кожним тестом база приводиться до чистого стану через міграції та seed-дані, що гарантує детермінованість результатів. AI-сценарії покриті окремими integration-тестами, які використовують FakeChatCompletionService - детермінований замінник реального LLM, що повертає фіксовані відповіді для перевірних промптів. Приклади integration-тестів наведено у Додатку К.

Елемент	Покриття	Непокрито/Всього
Total	79%	1778/8491
tests	96%	107/2880
Application.UnitTests	99%	15/1594
Api.Tests.Integration	93%	92/1286
src	70%	1671/5611
EventFlow.Application	75%	193/771
{} EventFlow.Application	75%	193/771
ConfigureApplication	100%	0/9
{} Events.Commands	87%	64/477
{} Subscriptions.Commands	78%	21/95
{} Common	74%	11/43
{} Users.Commands	34%	97/147
EventFlow.Infrastructure	73%	979/3640
EventFlow.Domain	70%	135/456
{} EventFlow.Domain	70%	135/456
{} Entities	73%	114/426
{} ValueObjects	30%	21/30
EventFlow.API	51%	364/744
Program	73%	22/82
{} EventFlow.API	48%	342/662

Рис. 3.17. Звіт про покриття коду після прогону тестів

Джерело: розроблено автором

Стратегія тестування за пірамідою: значна частка - швидкі unit-тести Application шару, менша - інтеграційні тести ключових сценаріїв (реєстрація, оплата, AI-сценарії), зверху - невелика кількість performance-тестів. Такий підхід забезпечує швидкий feedback loop для розробників та достатнє покриття для виявлення регресій.

3.5.2 Керівництво користувача

Розглянемо основні сценарії використання системи EventFlow з точки зору кінцевого користувача.

Реєстрація та вхід. При першому відкритті система пропонує авторизуватися через Clerk. Підтримуються вхід через email з паролем, magic link та Google. Після успішної автентифікації користувач автоматично створюється у внутрішній БД EventFlow.

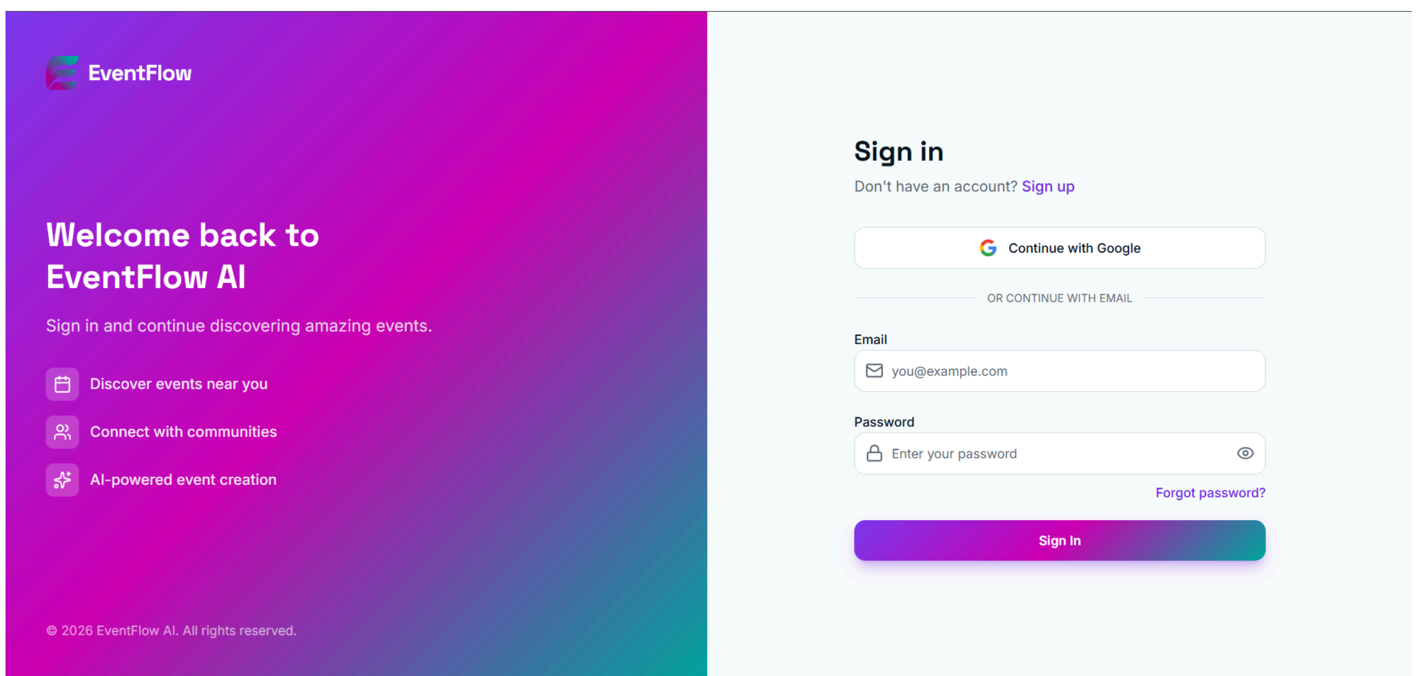

Рис. 3.18. Сторінка входу

Джерело: розроблено автором

Головна сторінка. Відображається лендінг з описом продукту, секцією переваг, прикладами актуальних подій, закликом до дії. Зверху - навігаційна панель з посиланнями на каталог, особистий кабінет, AI-чат.

Каталог подій. Список усіх опублікованих подій з фільтрами за категорією, датою, ціною, локацією та сортуванням. Кожна подія представлена у вигляді картки з обкладинкою, назвою, датою, локацією, ціною та кнопкою «Детальніше».

Детальна сторінка події. Великий банер з обкладинкою, повний опис, інформація про організатора, дату, локацію, кількість вільних місць, ціну. Кнопка «Зареєструватися» (для безкоштовних) або «Купити квиток» (для платних).

Створення події (для організаторів). Форма з кількома кроками: основна інформація, дата та локація, квитки та ціна, обкладинка та публікація. Валідація на кожному кроці, можливість зберегти як чернетку.

Купівля квитка. Натискання «Купити квиток» переадресовує на сторінку LiqPay. Після успішної оплати користувач повертається у застосунок, бачить підтвердження покупки, отримує real-time нотифікацію про активацію квитка.

Особистий кабінет. Розділ «Мої події» (для організаторів) - список створених подій зі статистикою. Розділ «Мої квитки» (для всіх) - придбані та заброньовані квитки з QR-кодами для пред’явлення на події.

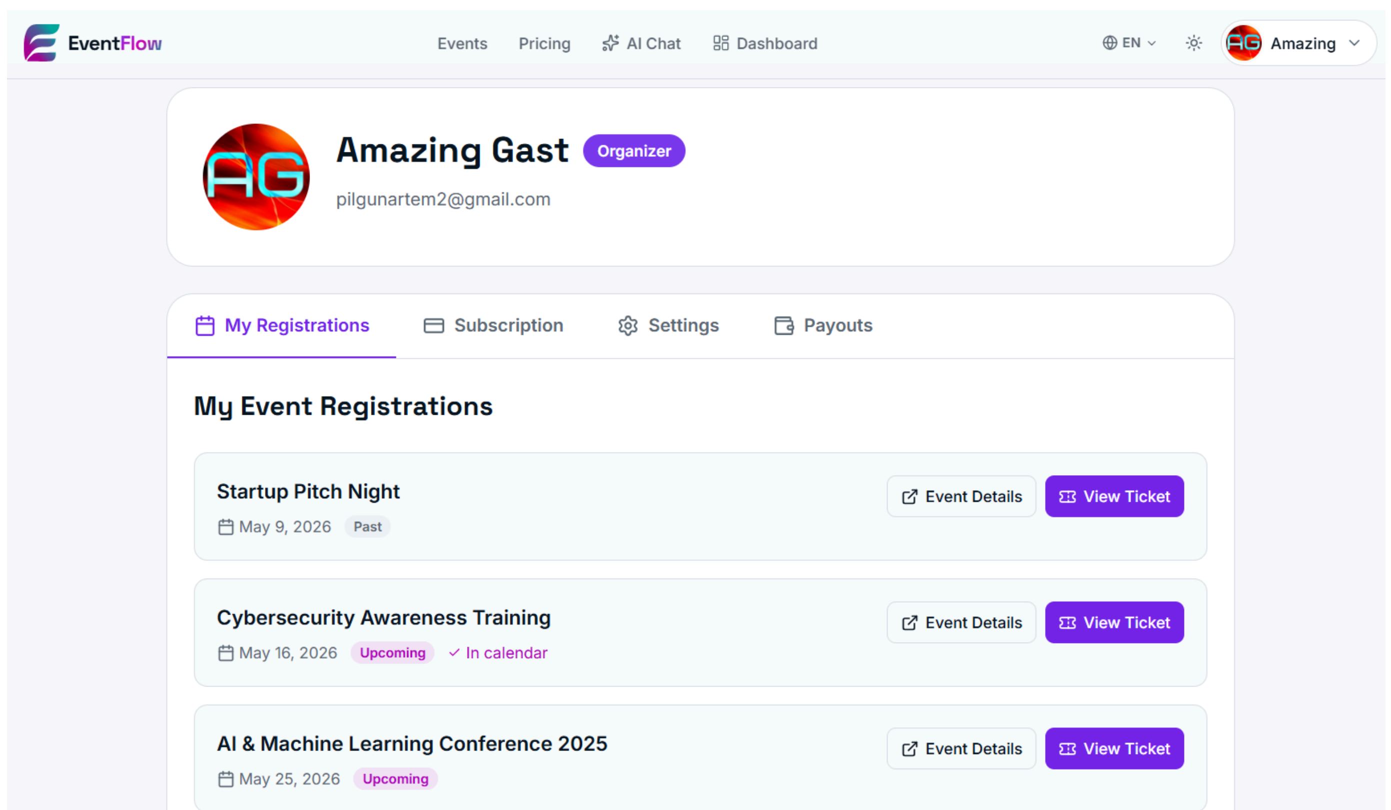

Рис. 3.19. Особистий кабінет - мої події

Джерело: розроблено автором

AI-асистент. Доступний з будь-якої сторінки через плаваючу кнопку. Користувач описує своє завдання природною мовою - асистент уточнює деталі, виконує дії, повертає результат.

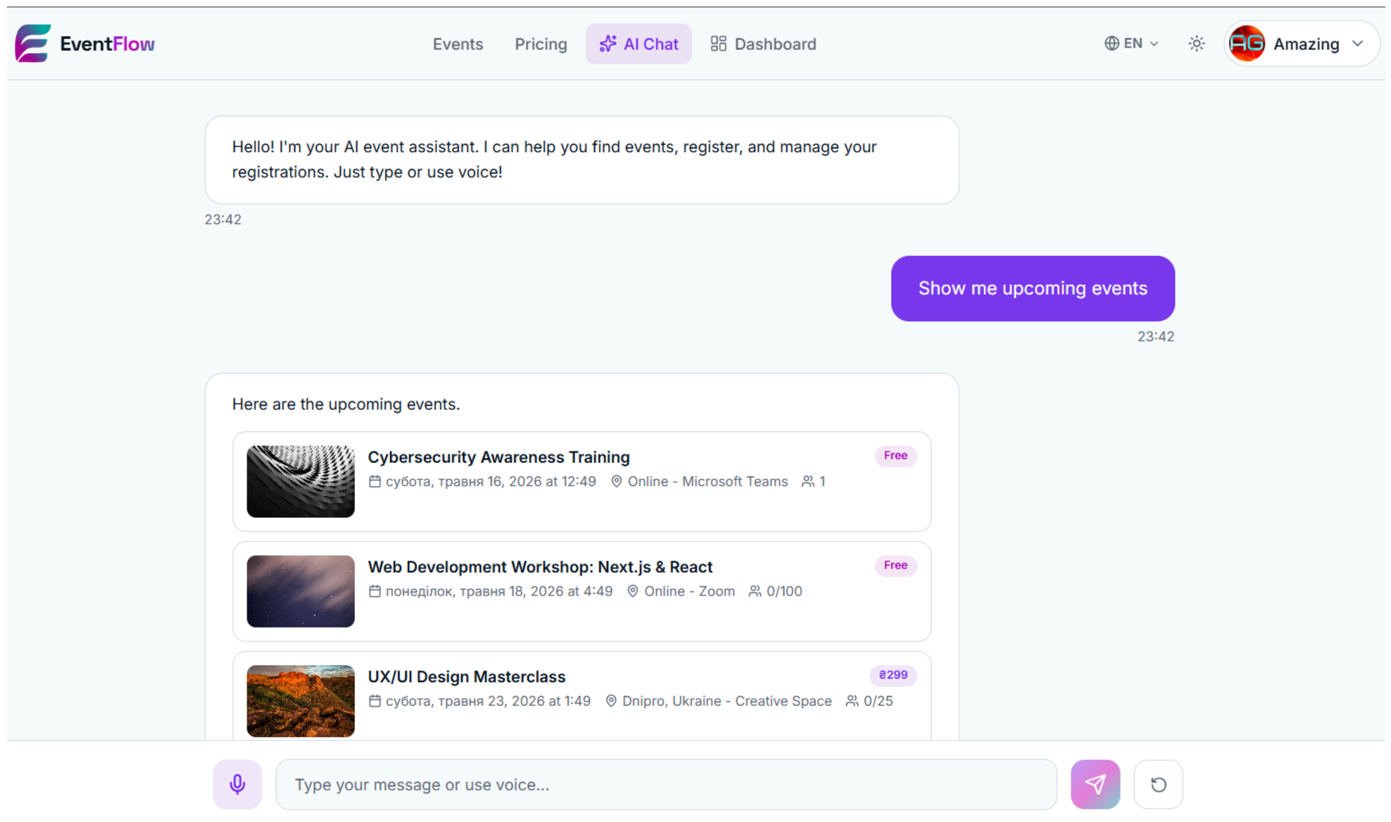

Рис. 3.20. AI-чат у дії: створення події через текстовий запит

Джерело: розроблено автором

Фотогалерея події. Організатор може завантажувати власні фото через drag-and-drop або вибір файлів.

Адмін-панель. Доступна тільки користувачам з роллю Admin. Містить розділи: модерація подій (перегляд), управління користувачами, глобальна статистика системи, виплати організаторам. Учасники переглядають галерею у read-only режимі.

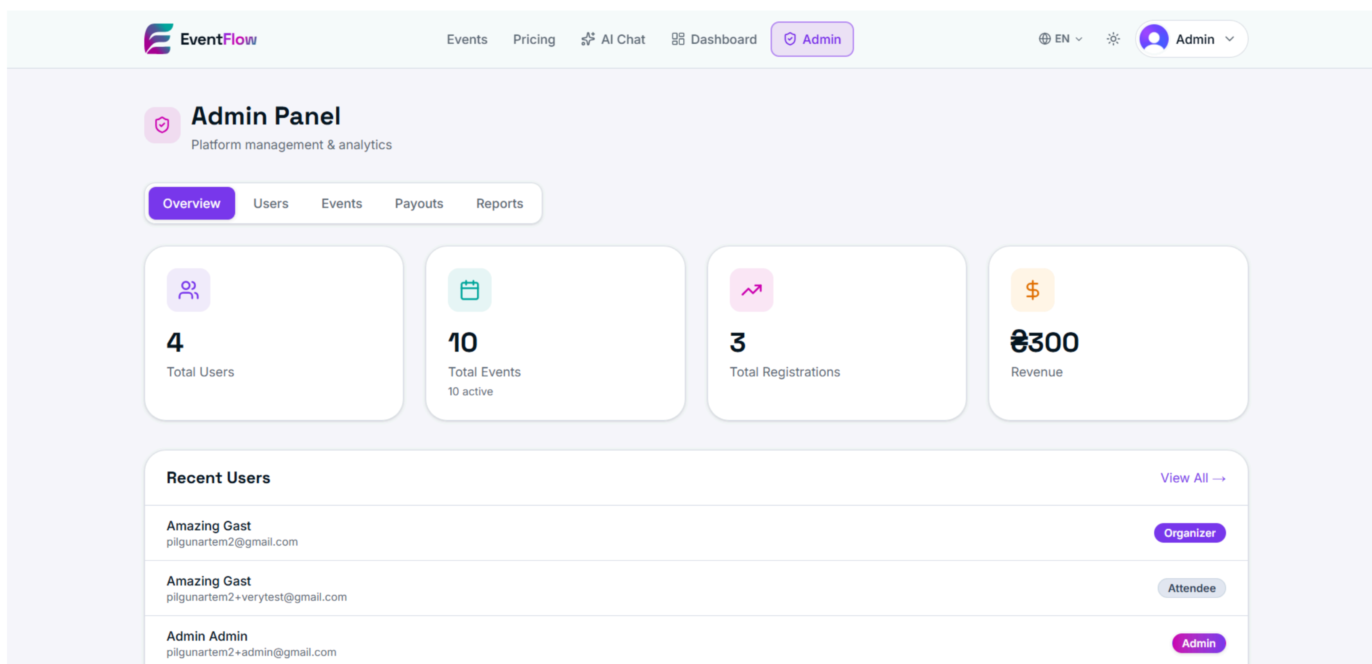

Рис. 3.21. Адмін-панель - модерація подій

Джерело: розроблено автором

Виплати організатору. Після створення платної події та продажу хоча б одного квитка організатор переходить у розділ «Виплати» особистого кабінету (/profile/payouts). Перший крок - введення IBAN та ПІБ отримувача; форма виконує клієнтську валідацію за алгоритмом MOD-97 ще до відправки на сервер. Введені реквізити очікують підтвердження адміністратором (статус «Очікує підтвердження»). Після кожної завершеної події (з 3-денним буфером на chargeback) фонова задача автоматично формує запис виплати, який організатор бачить у списку зі статусом «Очікує», а у картці «Дохід» зростає сума «В очікуванні підтвердження». Після ручного банківського переказу адміністратор фіксує референс транзакції у адмін-панелі, після чого статус виплати змінюється на «Виплачено», а організатору надсилається відповідне email-сповіщення.

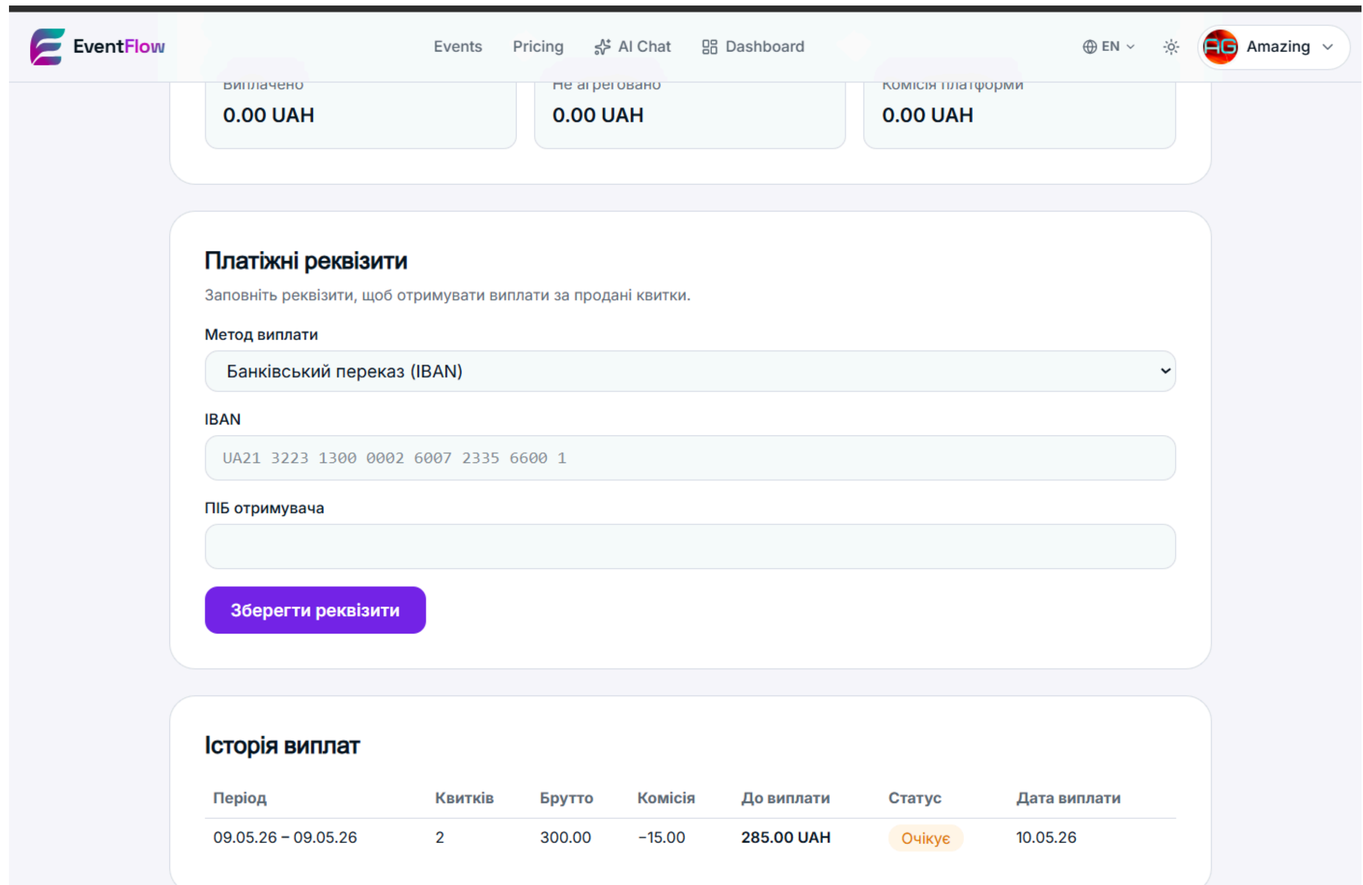

Рис. 3.22. Інтерфейс розділу «Виплати» в особистому кабінеті

Джерело: розроблено автором

Налаштування безпеки акаунта. У розділі «Безпека» (/profile/security) користувач може змінити пароль, увімкнути двофакторну автентифікацію через TOTP-додаток (Google Authenticator, Authy тощо), переглянути та примусово завершити активні сесії на інших пристроях, а також відключити прив'язані соціальні облікові записи. Інтерфейс надається безпосередньо компонентом <UserProfile /> від Clerk.

Розгортання системи EventFlow побудоване на принципах Infrastructure as Code: усі залежності, версії та налаштування описано в конфігураційних файлах, що зберігаються разом з кодом проєкту в системі контролю версій. Такий підхід забезпечує відтворюваність середовища на будь-якій машині - від локального розробницького ноутбука до продакшен-серверу.

3.6 Розгортання застосунку та CI/CD

3.6.1 Контейнеризація серверної частини

Серверна частина EventFlow упакована в Docker-контейнер на основі офіційного образу mcr.microsoft.com/dotnet/aspnet:10.0. Це гарантує однакову поведінку застосунку незалежно від хост-системи (Windows, Linux, macOS). У реалізації використано підхід multi-stage build: окремий контейнер на основі mcr.microsoft.com/dotnet/sdk:10.0 для збирання застосунку, далі значно компактніший runtime-контейнер на базі aspnet для його запуску. Це зменшує розмір фінального образу приблизно у п'ять разів - з ~700 МБ до ~150 МБ.

Структура Dockerfile містить чітко визначені шари: копіювання csproj-файлів усіх чотирьох проєктів (Domain, Application, Infrastructure, API), відновлення NuGet-пакетів через dotnet restore, копіювання вихідного коду, виклик dotnet publish з конфігурацією Release, налаштування ENTRYPOINT для запуску EventFlow.API.dll та EXPOSE 8080 для відкриття порту контейнера. Завдяки впорядкованому використанню Docker layer caching повторна збірка після зміни лише вихідного коду без зміни залежностей займає секунди замість хвилин.

Змінна оточення ASPNETCORE_URLS=http://+:8080 налаштовує Kestrel слухати на всіх інтерфейсах контейнера на порту 8080, що дозволяє reverse-proxy або балансувальнику навантаження легко направити трафік без додаткової конфігурації.

3.6.2 Локальне розгортання інфраструктури через docker-compose

Для розгортання повного локального середовища розробки використовується docker-compose.yml, що одночасно піднімає два інфраструктурні сервіси:

- eventflow-db - PostgreSQL 16-alpine на порту 5433 хост-машини (мапінг на 5432 контейнера) з налаштованим named volume eventflow-db-data для збереження

даних між перезапусками контейнера, що уникає втрати тестових даних при кожному рестарті;

- eventflow-localstack - емулятор AWS-сумісного S3-сховища на порту 4567 з активованим тільки одним сервісом S3 (SERVICES=s3) та персистентністю через volume eventflow-localstack-data. Скрипт ініціалізації scripts/localstack-init.sh автоматично створює бакет eventflow-photos при першому старті, тож розробнику не потрібно вручну налаштовувати S3-bucket.

Сам застосунок EventFlow.API запускається локально через dotnet run поза docker-compose - це навмисне рішення, що пришвидшує цикл розробки: hot-reload, debugger, точки зупину працюють у звичайному режимі без шару контейнерізації. Контейнеризований запуск API використовується лише в продакшен-середовищі.

Розробник запускає інфраструктурний стек однією командою docker-compose up -d, а зупиняє через docker-compose down. Параметр restart: always гарантує, що сервіси автоматично перезапускаються після рестарту хост-машини.

3.6.3 CI/CD pipeline через GitHub Actions

Безперервна інтеграція (Continuous Integration) реалізована через GitHub Actions з workflow-файлом .github/workflows/ci.yml. Pipeline спрацьовує на кожен push у гілки main та development, а також на кожен pull_request до головної гілки. Налаштування concurrency забезпечує автоматичне скасування попередніх запусків при пушах у ту саму гілку - це економить ресурси GitHub Actions та пришвидшує feedback розробнику.

Pipeline складається з чотирьох логічних job-ів, що виконуються паралельно та послідовно (рис. 3.18).

Job 1: backend-build - збірка та юніт-тести бекенду

Цей job встановлює .NET 10 SDK, налаштовує кешування NuGet-пакетів через actions/cache@v4 (ключ кешу базується на хеші всіх csproj-файлів), відновлює залежності та збирає solution у release-конфігурації. Після успішної збірки запускаються юніт-тести проєкту Application.UnitTests зі збором покриття у форматі Cobertura. На останньому етапі через danielpalme/ReportGenerator-GitHub-Action@v5 генерується HTML-звіт про покриття коду, який публікується як артефакт workflow під назвою BackendCoverageReport. Розробник може завантажити цей звіт з GitHub UI після завершення прогону.

Job 2: backend-integration - інтеграційні тести з Testcontainers

Цей job залежить від успішного завершення backend-build через директиву needs: backend-build. Перед запуском тестів виконується паралельне попереднє завантаження Docker-образів через docker pull ... & команди - postgres:16-alpine, localstack/localstack:3.8, testcontainers/ryuk:0.11.0. Це усуває холодний старт Testcontainers, який в іншому випадку додав би ~90 секунд до часу прогону тестів.

Інтеграційні тести запускаються з таймаутом 15 хвилин та діагностичними прапорцями --blame-hang-timeout 5m --blame-hang-dump-type mini, що автоматично створюють mini-dump процесу у випадку зависання тесту - це критично для діагностики проблем у CI-середовищі, де неможливо приєднати дебагер. Тести використовують секрети LIQPAY_PUBLIC_KEY та LIQPAY_PRIVATE_KEY через GitHub Secrets для тестування sandbox-режиму LiqPay без зберігання чутливих даних у репозиторії. Результати тестів у форматі TRX публікуються як артефакт IntegrationTestResults.

Job 3: frontend-checks - перевірки клієнтської частини

Паралельно з бекенд-тестами запускається job для клієнтської частини. Виконується встановлення Node.js 22 з кешуванням npm-залежностей за хешем package-lock.json, встановлення пакетів через npm ci --legacy-peer-deps, лінтінг через

ESLint з обмеженням --max-warnings 50, прогін фронтенд-тестів через npm test -- --ci --coverage та фінальна продакшен-збірка через npm run build з підставленими секретами Clerk.

Job 4: performance-tests - навантажувальне тестування через k6

Цей job залежить від backend-integration і запускається останнім у послідовності. Тут вже не використовується Testcontainers - натомість Postgres підіймається як GitHub Actions service з health-check через pg_isready. Pipeline застосовує EF Core міграції через dotnet ef database update, запускає API в фоновому режимі з логуванням у api_log.txt, чекає 15 секунд на старт, після чого виконує навантажувальні тести через k6 за скриптом tests/PerformanceTests/test-events.js. У разі провалу будь-якого тесту останні 50 рядків логу API публікуються в Actions output для діагностики. API процес зупиняється через pkill -f dotnet у блоці always(), що гарантує очистку ресурсів навіть при помилках.

3.6.4 Тестування web-accessibility у CI/CD

Веб-доступність (web accessibility, a11y) є однією з ключових нефункціональних вимог, сформульованих у Розділі 1.3. Відповідність стандарту WCAG 2.1 рівня AA перевіряється не лише вручну при дизайні, але й автоматично у CI/CD pipeline на кожен PR. Це запобігає регресіям, коли новий компонент чи зміна стилю випадково ламає доступність раніше працюючих сторінок.

Інструментарій тестування a11y

Для автоматичних перевірок використовується комбінація двох інструментів. Перший - **axe-core**, open-source бібліотека від Deque Systems, що сканує сторінку на відповідність понад 90 правилам WCAG 2.0/2.1 та найкращих практик ARIA. Інтегрована у вигляді npm-пакету @axe-core/playwright. Другий - **Lighthouse CI**, інструмент Google, що проводить комплексний аудит сторінок з показниками за

чотирма категоріями: Performance, Accessibility, Best Practices, SEO. Окремий accessibility score має досягати щонайменше 95 балів зі 100 для проходження pipeline.

Архітектура accessibility-тестів

Тестовий проєкт eventflow-client/tests/a11y/ містить набір Playwright-сценаріїв, що автоматично відкривають ключові сторінки застосунку та запускають для кожної сканування через axe-core: головна сторінка для анонімного користувача, каталог подій з фільтрами, сторінка деталей події, форма створення події з мокованою organizer-сесією, AI-чат у відкритому стані, особистий кабінет учасника, адмін-панель.

Для кожної сторінки axe-core повертає JSON-звіт зі списком знайдених порушень з прив'язкою до конкретних DOM-елементів та правил WCAG. Тест провалюється, якщо знайдено хоч одне порушення рівня serious або critical. Порушення рівня minor і moderate записуються в лог, але не блокують pipeline - їх виправлення планується в окремих ітераціях.

Інтеграція в GitHub Actions

Окремий job accessibility-tests у CI-workflow виконує послідовність команд: запуск фронтенду в режимі production-build на ефемерному порту, прогін Playwright-сценаріїв з axe-core, генерація HTML-звіту, публікація його як артефакта workflow. Розробник може завантажити цей звіт прямо з GitHub UI після завершення CI-прогону. Додатковий крок виконує Lighthouse CI через офіційний treosh/lighthouse-ci-action, що формує бейдж з accessibility score та публікує його в коментарі до PR. Якщо score падає нижче 95 - pipeline провалюється.

Лістинг 3.18. Фрагмент конфігурації accessibility-тесту з axe-core

```
import { test, expect } from "@playwright/test";
import AxeBuilder from "@axe-core/playwright";
```

```
test.describe("Accessibility audit", () => {
  test("Homepage has no critical a11y violations", async ({ page
}) => {
    await page.goto("/");
    await page.waitForLoadState("networkidle");

    const results = await new AxeBuilder({ page })
      .withTags(["wcag2a", "wcag2aa", "wcag21aa"])
      .analyze();

    const criticalViolations = results.violations.filter(
      (v) => v.impact === "critical" || v.impact === "serious",
    );

    expect(criticalViolations).toEqual([]);
  });

  test("Event details page is keyboard navigable", async ({ page
}) => {
    await page.goto("/events/sample-event-id");
    await page.keyboard.press("Tab");
    await expect(page.locator(":focus")).toBeVisible();
  });
});
```

Як видно з лістингу, AxeBuilder ініціалізується з тегами wcag2a, wcag2aa, wcag21aa, що покриває обов'язкові вимоги стандарту WCAG 2.1 рівня AA. Результат фільтрується по impact-рівню, і тест провалюється тільки при наявності serious або critical порушень. Другий тест перевіряє клавіатурну навігацію - натискання Tab має переводити фокус на видимий інтерактивний елемент.

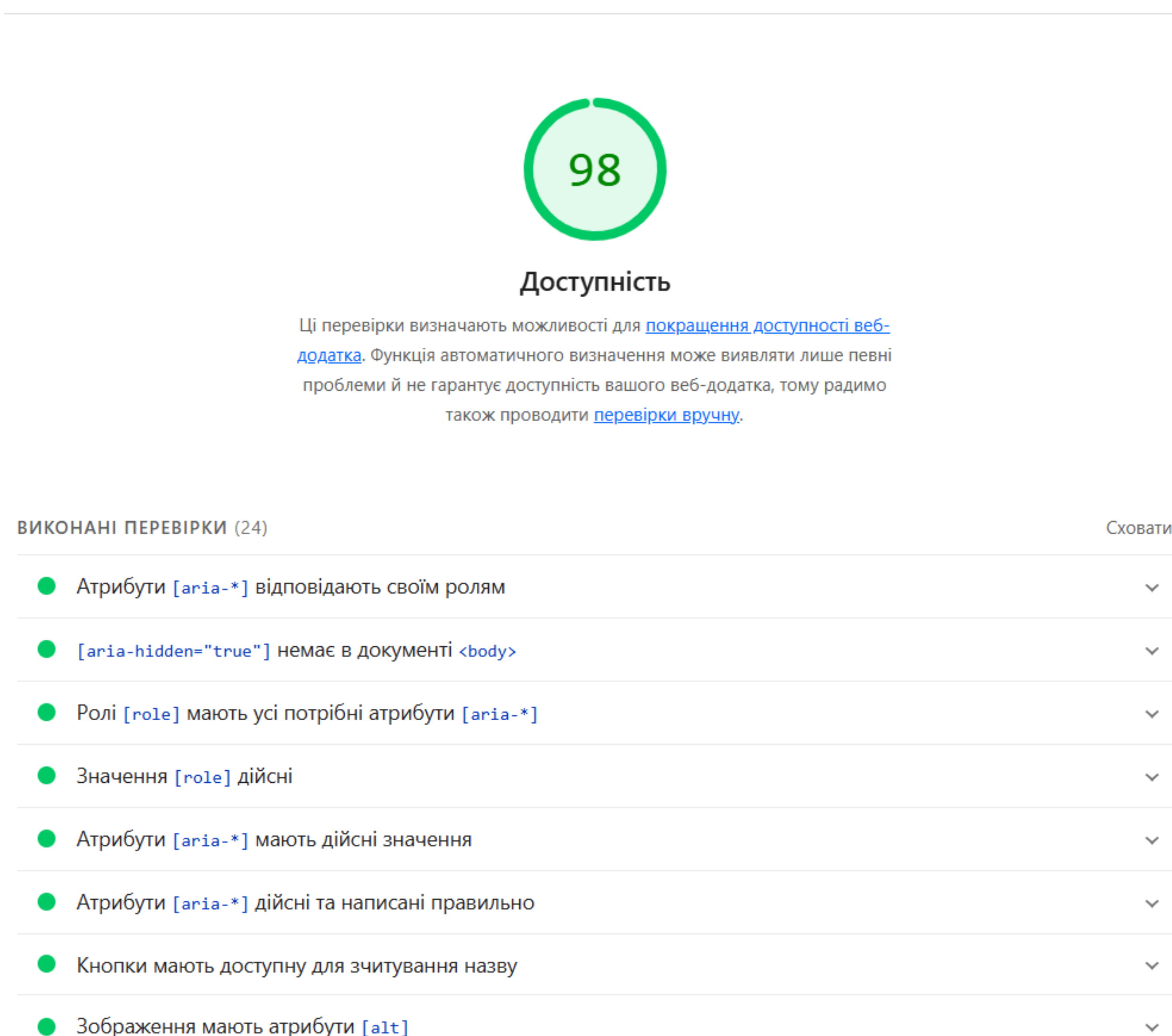

Рис. 3.18. Скріншот HTML-звіту Lighthouse CI з показниками доступності EventFlow

Джерело: розроблено автором

На рисунку 3.18 наведено результат прогону Lighthouse CI на головній сторінці застосунку: accessibility score становить понад 95 балів зі 100, з виокремленими підпунктами перевірки - наявність alt-текстів для всіх зображень, коректний контраст за WCAG AA, валідна семантична розмітка, відсутність дублікатів id-атрибутів, ARIA-атрибути для всіх інтерактивних елементів.

3.6.5 Розгортання в production

Production-розгортання EventFlow побудоване за двокомпонентною моделлю.

Серверна частина запакована в Docker-образ і розгорнута на VPS під управлінням Ubuntu 22.04 LTS. Запуск виконується через docker-compose з production-overlay-файлом, що відрізняється від dev-конфігурації: відсутність LocalStack (замінений на Cloudflare R2 через перемикання ServiceUrl у StorageOptions), реальні credentials до Clerk та LiqPay через змінні середовища, увімкнений HTTPS через reverse-proxy Nginx з Let's Encrypt-сертифікатом, обмежені ресурси контейнера через директиви mem_limit: 2g та cpus: 1.5.

Клієнтська частина розгортається на хостингу для статичних сайтів та edge-функцій - Vercel або Cloudflare Pages. Next.js застосунок будується в режимі standalone, що дає оптимізований deployment з нативною підтримкою SSR і ISR (Incremental Static Regeneration) для каталогу подій. Розгортання автоматизоване через інтеграцію Vercel з GitHub: кожен merge у main гілку автоматично тригерить новий production-build.

Моніторинг та логи

Логування реалізоване через Serilog зі структурованим JSON-форматом, що дозволяє інтеграцію з системами централізованого логування - Grafana Loki або ELK stack. Health-check endpoint /health повертає статус застосунку та залежностей (PostgreSQL, Cloudflare R2, Groq Cloud) у форматі, придатному для автоматичного моніторингу через Uptime Kuma або інші подібні інструменти.

Стратегія оновлень

Оновлення продакшен-версії виконується за принципом zero-downtime deployment: новий образ контейнера завантажується паралельно з працюючим, після успішного health-check трафік перемикається на новий контейнер через Nginx upstream-peconfiguration, старий зупиняється. Для бази даних застосовуються non-breaking EF Core міграції - кожна міграція спочатку додає нові колонки/таблиці, далі код використовує і старі, і нові поля, після підтвердження стабільності нової версії

старі поля можна видалити в наступному релізі. Така стратегія дозволяє відкочуватись (rollback) на попередню версію застосунку без втрати даних.

Висновки до розділу 3

У третьому розділі наведено детальний опис програмної реалізації системи EventFlow.

Описано повний стек технологій серверної та клієнтської частин, обґрунтовано вибір кожного інструменту. Наведено вимоги до апаратного та програмного забезпечення для розгортання системи в продакшен-середовищі.

Серверна частина реалізована за принципами Clean Architecture з чотирма шарами та CQRS-патерном на базі MediatR. Реалізовано інтеграції з зовнішніми сервісами: автентифікація через Clerk з SSO-підтримкою, платежі через LiqPay для українського ринку, AI-функціональність через Microsoft Semantic Kernel з чотирма плагінами, real-time комунікація через два SignalR-хаби, зберігання файлів в Cloudflare R2.

Клієнтська частина реалізована на Next.js 16 з App Router. Описано детальний процес створення дизайну в Figma з використанням AI-функції Figma Make, що значно скоротило час підбору фінального вигляду компонентів. Розроблено повноцінну дизайн-систему з кольоровою палітрою (з підтримкою Light/Dark тем та відповідністю WCAG AA), системою типографіки на базі next/font та структурованим набором UI-компонентів. Описано ключові реалізації: інтеграція з Clerk, форма створення події, каталог, детальна сторінка, AI-чат-віджет, завантаження фото, локалізація, real-time нотифікації.

Тестування організовано в п’яти проєктах з застосуванням пірамідного підходу: швидкі unit-тести Application шару, інтеграційні тести з реальною БД через EventFlowWebFactory та Testcontainers, performance-тести критичних ендпоінтів.

Окремо реалізовано тестування AI-сценаріїв з детермінованими результатами через FakeChatCompletionService.

Реалізовано повноцінний CI/CD pipeline через GitHub Actions з чотирма паралельно-послідовними job-ами: збірка з юніт-тестами та звітом про покриття, інтеграційні тести через Testcontainers, перевірки клієнтської частини та навантажувальні тести через k6. Окремо налаштовано автоматизоване тестування web-accessibility через axe-core та Lighthouse CI, що запобігає регресіям доступності на кожному pull-request. Контейнеризація серверної частини через multi-stage Docker build та інфраструктурний docker-compose для PostgreSQL і LocalStack забезпечують відтворюваність середовища від ноутбука розробника до продакшен-серверу.

Описано керівництво користувача для всіх основних сценаріїв роботи з системою з ілюстраціями екранів інтерфейсу.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи розроблено систему управління подіями EventFlow з елементами штучного інтелекту, що повністю відповідає поставленим у вступі задачам.

За результатами першого розділу:

- Проведено аналіз предметної області event-management, виділено дев'ять ключових бізнес-сутностей та описано життєвий цикл події.
- Розглянуто три основні аналоги - Eventbrite, Luma, Meetup - їх сильні та слабкі сторони.
- Обґрунтовано конкурентні переваги EventFlow: інтеграція з LiqPay, AI-асистент, сучасний стек технологій, орієнтація на український ринок.
- Сформульовано функціональні та нефункціональні вимоги, обмеження проєкту та критерії успіху.

За результатами другого розділу:

- Спроєктовано архітектуру системи на основі підходу Clean Architecture з чотирма шарами (Domain, Application, Infrastructure, API), що забезпечує тестованість, гнучкість та чітке розмежування відповідальностей.
- Спроєктовано шар Application на основі CQRS з використанням MediatR, ErrorOr та pipeline behaviors.
- Розроблено модель бази даних на PostgreSQL з дев'ятьма таблицями, ключовими індексами та обмеженнями цілісності.
- Формалізовано чотири основні алгоритми: перевірка доступності квитків з оптимістичним блокуванням, формування фінальної ціни, обробка AI-запитів через Function Calling, перевірка сигнатури LiqPay.

За результатами третього розділу:

- Реалізовано серверну частину на платформі .NET 10 з усіма запланованими інтеграціями: Clerk для автентифікації, LiqPay для платежів, Cloudflare R2 для файлів, Semantic Kernel для AI-функцій, SignalR для real-time комунікації.
- Реалізовано чотири AI-плагіни (EventManagement, EventRegistration, OrganizerDashboard, EventSearch), що покривають основні сценарії взаємодії користувача з системою через природну мову.
- Реалізовано клієнтську частину на Next.js 16 з App Router, з повноцінною дизайн-системою, створеною в Figma з використанням AI-функції Figma Make.
- Розроблено комплексну стратегію тестування з п’ятьма проєктами тестів - від unit до performance.

Технічні характеристики реалізованого продукту:

- 50 REST API ендпоінтів;
- два SignalR-хаби для real-time функцій;
- чотири AI-плагіни з понад десятком функцій;
- понад 30 React-компонентів інтерфейсу;
- підтримка двох мов інтерфейсу - української та англійської;
- адаптивний дизайн для всіх типів пристроїв.

Практична цінність роботи полягає у створенні готової до експлуатації системи, що може бути впроваджена як приватними організаторами подій, так і компаніями. Архітектура є модульною, що дозволяє подальший розвиток.

Перспективи розвитку:

- розробка нативного мобільного застосунку (iOS, Android) з використанням React Native;
- еволюція до мікросервісної архітектури при зростанні навантаження;
- впровадження ML-рекомендаційної системи на основі історії відвідуваних подій користувача;

- розширення AI-функціональності: автоматична генерація обкладинок, описів подій, маркетингових текстів;
- інтеграція з державними сервісами України (Дія) для верифікації організаторів та автоматичного формування електронних квитанцій.

Поставлена мета кваліфікаційної роботи досягнута повністю.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Microsoft .NET 10 Documentation. URL: https://learn.microsoft.com/en-us/dotnet/ (дата звернення: 15.10.2025).
2. ASP.NET Core Documentation. URL: https://learn.microsoft.com/en-us/aspnet/core/ (дата звернення: 18.10.2025).
3. C# Programming Guide. Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/csharp/ (дата звернення: 22.10.2025).
4. Entity Framework Core Documentation. URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 28.10.2025).
5. Common Web Application Architectures. Microsoft .NET Architecture Guides. URL: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures (дата звернення: 02.11.2025).
6. Taylor J. Clean Architecture Solution Template for .NET. GitHub. URL: https://github.com/jasontaylordev/CleanArchitecture (дата звернення: 05.11.2025).
7. MediatR Wiki - In-process messaging with no dependencies. GitHub. URL: https://github.com/jbogard/MediatR/wiki (дата звернення: 10.11.2025).
8. FluentValidation Documentation. URL: https://docs.fluentvalidation.net/ (дата звернення: 12.11.2025).
9. ErrorOr - A simple, fluent discriminated union of an error or a result. GitHub. URL: https://github.com/amantinband/error-or (дата звернення: 14.11.2025).
10. CQRS Pattern. Microsoft Cloud Design Patterns. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs (дата звернення: 18.11.2025).
11. Domain-Driven Design Fundamentals. Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/ (дата звернення: 22.11.2025).
12. Microsoft Semantic Kernel Documentation. URL: https://learn.microsoft.com/en-us/semantic-kernel/ (дата звернення: 28.11.2025).

13. Function Calling with Semantic Kernel. Microsoft Learn. URL: https://learn.microsoft.com/en-us/semantic-kernel/concepts/ai-services/chat-completion/function-calling/ (дата звернення: 02.12.2025).
14. Groq Cloud API Reference. URL: https://console.groq.com/docs/api-reference (дата звернення: 05.12.2025).
15. SignalR Overview for ASP.NET Core. Microsoft Learn. URL: https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction (дата звернення: 10.12.2025).
16. Hangfire Documentation - Background jobs for .NET. URL: https://docs.hangfire.io/ (дата звернення: 14.12.2025).
17. Serilog - Diagnostic logging library for .NET. URL: https://serilog.net/ (дата звернення: 18.12.2025).
18. PostgreSQL 16 Documentation. URL: https://www.postgresql.org/docs/16/ (дата звернення: 22.12.2025).
19. Npgsql Entity Framework Core Provider. URL: https://www.npgsql.org/efcore/ (дата звернення: 26.12.2025).
20. Testcontainers for .NET. URL: https://dotnet.testcontainers.org/ (дата звернення: 04.01.2026).
21. xUnit.net Documentation. URL: https://xunit.net/docs/getting-started/ (дата звернення: 06.01.2026).
22. NSubstitute - A friendly substitute for .NET mocking libraries. URL: https://nsubstitute.github.io/ (дата звернення: 08.01.2026).
23. Next.js Documentation - App Router. URL: https://nextjs.org/docs/app (дата звернення: 12.01.2026).
24. React Documentation - Reference. URL: https://react.dev/reference/react (дата звернення: 14.01.2026).
25. TypeScript Handbook. URL: https://www.typescriptlang.org/docs/handbook/intro.html (дата звернення: 16.01.2026).

26. Tailwind CSS v4 Documentation. URL: https://tailwindcss.com/docs (дата звернення: 18.01.2026).
27. shadcn/ui - Re-usable components built with Radix UI and Tailwind CSS. URL: https://ui.shadcn.com/ (дата звернення: 20.01.2026).
28. TanStack Query Documentation. URL: https://tanstack.com/query/latest/docs/framework/react/overview (дата звернення: 22.01.2026).
29. React Hook Form - Performant, flexible forms with easy-to-use validation. URL: https://react-hook-form.com/ (дата звернення: 24.01.2026).
30. Zod - TypeScript-first schema validation. URL: https://zod.dev/ (дата звернення: 26.01.2026).
31. Microsoft SignalR Client for JavaScript. NPM. URL: https://www.npmjs.com/package/@microsoft/signalr (дата звернення: 28.01.2026).
32. Clerk Documentation - Authentication and user management. URL: https://clerk.com/docs дата звернення: 02.02.2026).
33. LiqPay API Documentation. URL: https://www.liqpay.ua/documentation (дата звернення: 06.02.2026).
34. AWSSDK.S3 NuGet Package. URL: https://www.nuget.org/packages/AWSSDK.S3 (дата звернення: 10.02.2026).
35. Cloudflare R2 Documentation. URL: https://developers.cloudflare.com/r2/ (дата звернення: 12.02.2026).
36. Cloudflare R2 - S3 API compatibility. URL: https://developers.cloudflare.com/r2/api/s3/api/ (дата звернення: 14.02.2026).
37. Docker Documentation. URL: https://docs.docker.com/ (дата звернення: 18.02.2026).
38. GitHub Actions Documentation. URL: https://docs.github.com/en/actions (дата звернення: 22.02.2026).
39. Playwright Documentation - End-to-end testing for modern web apps. URL: https://playwright.dev/docs/intro (дата звернення: 26.02.2026).

40. axe-core - Accessibility engine for automated Web UI testing. GitHub. URL: https://github.com/dequelabs/axe-core (дата звернення: 02.03.2026).
41. Google Lighthouse - Automated tool for improving the quality of web pages. URL: https://developer.chrome.com/docs/lighthouse/overview (дата звернення: 04.03.2026).

ДОДАТОК А

Лістинг А.1. Повний код сутності Event

```
using EventFlow.Domain.ValueObjects;

namespace EventFlow.Domain.Entities;

/// <summary>
/// Represents an event that can be created by organizers and
attended by users.
/// </summary>
public class Event
{
    public EventId Id { get; private set; }
    public string Title { get; private set; } = string.Empty;
    public string Description { get; private set; } =
string.Empty;
    public string? Location { get; private set; }
    public DateTime EventDateTime { get; private set; }
    public decimal TicketPrice { get; private set; }
    public int? MaxAttendees { get; private set; }
    public string Status { get; private set; } =
EventStatus.Draft;
    public DateTime CreatedAt { get; private set; }
    public DateTime? UpdatedAt { get; private set; }
    public UserId OrganizerId { get; private set; }
    public User Organizer { get; private set; } = null!;

    private readonly List<EventRegistration> _registrations =
new();
    public IReadOnlyCollection<EventRegistration> Registrations =>
_registrations.AsReadOnly();

    private readonly List<EventPhoto> _photos = new();
    public IReadOnlyCollection<EventPhoto> Photos =>
```

```
_photos.AsReadOnly();

    private Event() { }

    public bool IsFree => TicketPrice == 0m;

    public bool HasAvailableSpots() =>
        MaxAttendees is null || GetRegistrationCount() <
MaxAttendees;

    public static Event Create(
        string title,
        string description,
        DateTime eventDateTime,
        UserId organizerId,
        string? location = null,
        decimal ticketPrice = 0m,
        int? maxAttendees = null)
    {
        if (string.IsNullOrWhiteSpace(title))
            throw new ArgumentException("Event title cannot be
null or empty.", nameof(title));

        if (string.IsNullOrWhiteSpace(description))
            throw new ArgumentException("Event description cannot
be null or empty.", nameof(description));

        if (organizerId.Value == Guid.Empty)
            throw new ArgumentException("OrganizerId cannot be
empty.", nameof(organizerId));

        if (eventDateTime <= DateTime.UtcNow)
            throw new ArgumentException("Event date and time must
be in the future.", nameof(eventDateTime));
```

```
        if (ticketPrice < 0)
            throw new ArgumentException("Ticket price cannot be
negative.", nameof(ticketPrice));

        if (maxAttendees is not null && maxAttendees <= 0)
            throw new ArgumentException("Max attendees must be
greater than zero.", nameof(maxAttendees));

        return new Event
        {
            Id = EventId.New(),
            Title = title,
            Description = description,
            EventDateTime = eventDateTime,
            OrganizerId = organizerId,
            Location = location,
            TicketPrice = ticketPrice,
            MaxAttendees = maxAttendees,
            Status = EventStatus.Draft,
            CreatedAt = DateTime.UtcNow
        };
    }

    public void UpdateDetails(
        string title,
        string description,
        DateTime eventDateTime,
        string? location,
        decimal ticketPrice = 0m,
        int? maxAttendees = null)
    {
        if (string.IsNullOrWhiteSpace(title))
            throw new ArgumentException("Event title cannot be
null or empty.", nameof(title));
```

```
        if (string.IsNullOrWhiteSpace(description))
            throw new ArgumentException("Event description cannot
be null or empty.", nameof(description));

        if (eventDateTime <= DateTime.UtcNow)
            throw new ArgumentException("Event date and time must
be in the future.", nameof(eventDateTime));

        if (ticketPrice < 0)
            throw new ArgumentException("Ticket price cannot be
negative.", nameof(ticketPrice));

        Title = title;
        Description = description;
        EventDateTime = eventDateTime;
        Location = location;
        TicketPrice = ticketPrice;
        MaxAttendees = maxAttendees;
        UpdatedAt = DateTime.UtcNow;
    }

    public void Publish()
    {
        if (Status == EventStatus.Canceled)
            throw new InvalidOperationException("Cannot publish a
canceled event.");

        if (!_photos.Any())
            throw new InvalidOperationException("Cannot publish an
event without at least one photo.");

        Status = EventStatus.Published;
        UpdatedAt = DateTime.UtcNow;
    }
```

```
    public void Cancel()
    {
        if (Status == EventStatus.Canceled)
            throw new InvalidOperationException("Event is already
canceled.");

        Status = EventStatus.Canceled;
        UpdatedAt = DateTime.UtcNow;
    }

    public void AddPhoto(EventPhoto photo)
    {
        if (photo == null) throw new
ArgumentNullException(nameof(photo));
        _photos.Add(photo);
        UpdatedAt = DateTime.UtcNow;
    }

    public void RemovePhoto(EventPhoto photo)
    {
        if (photo == null) throw new
ArgumentNullException(nameof(photo));
        _photos.Remove(photo);
        UpdatedAt = DateTime.UtcNow;
    }

    public int GetRegistrationCount() =>
        IsFree
            ? _registrations.Count(r => !r.IsCanceled)
            : _registrations.Count(r => !r.IsCanceled && r.Ticket
is not null && r.Ticket.IsPaid);

    public static class EventStatus
    {
        public const string Draft = "Draft";
```

```
        public const string Published = "Published";
        public const string Canceled = "Canceled";
    }
}
```

Сутність Event ілюструє підхід Domain-Driven Design: усі властивості мають приватні сетери, фабричний метод Create() валідує доменні інваріанти, а перехід між статусами (Draft → Published → Canceled) виконується через бізнес-методи з перевіркою попередніх умов.

ДОДАТОК Б

Лістинг Б.1. Команда CreateEventCommand з валідатором та обробником

```
using EventFlow.Application.Common.Interfaces.Queries;
using EventFlow.Application.Common.Interfaces.Repositories;
using EventFlow.Application.Common.Services;
using EventFlow.Domain.Entities;
using FluentValidation;
using MediatR;
using Microsoft.Extensions.Logging;

namespace EventFlow.Application.Events.Commands.CreateEvent;

// ── Command ──
public record CreateEventCommand(
    string Title,
    string Description,
    DateTime EventDateTime,
    string? Location,
    string? ImageUrl,
    bool Publish,
    decimal TicketPrice,
    int? MaxAttendees,
    string UserIdHeader) : IRequest<ErrorOr<CreateEventResult>>;

// ── Result ──
public record CreateEventResult(Guid EventId, Event CreatedEvent);

// ── Validator ──
public class CreateEventCommandValidator :
AbstractValidator<CreateEventCommand>
{
    public CreateEventCommandValidator()
    {
        RuleFor(x => x.Title).NotEmpty().WithMessage("Title is
```

```
required.");
        RuleFor(x =>
x.Description).NotEmpty().WithMessage("Description is required.");
        RuleFor(x =>
x.EventDateTime).GreaterThan(DateTime.UtcNow).WithMessage("Event
date must be in the future.");
        RuleFor(x =>
x.TicketPrice).GreaterThanOrEqualTo(0).WithMessage("Ticket price
cannot be negative.");
        RuleFor(x => x.MaxAttendees).GreaterThan(0).When(x =>
x.MaxAttendees.HasValue)
            .WithMessage("Max attendees must be greater than
zero.");
        RuleFor(x => x.UserIdHeader).NotEmpty().WithMessage("User
ID is required.");
    }
}

// ── Handler ──
public class CreateEventCommandHandler(
    IEventRepository eventRepository,
    IUserRepository userRepository,
    IEventQueries eventQueries,
    ISubscriptionQueries subscriptionQueries,
    IUserResolverService userResolver,
    ILogger<CreateEventCommandHandler> logger)
    : IRequestHandler<CreateEventCommand,
ErrorOr<CreateEventResult>>
{
    private const int FreeActiveEventLimit = 3;
    private const int ProMonthlyEventLimit = 50;
    private const int FreeMaxAttendeesLimit = 30;

    public async Task<ErrorOr<CreateEventResult>> Handle(
        CreateEventCommand request, CancellationToken
```

```
cancellationToken)
    {
        var userIdResult = await
userResolver.ResolveUserIdAsync(request.UserIdHeader,
cancellationToken);
        if (userIdResult.IsError)
            return userIdResult.Errors;

        var userId = userIdResult.Value;

        var isOrganizer = await
userRepository.HasRoleAsync(userId, "Organizer",
cancellationToken);
        var isAdmin = await userRepository.HasRoleAsync(userId,
"Admin", cancellationToken);

        if (!isOrganizer && !isAdmin)
            return Error.Forbidden("User.NotOrganizer",
                "Only organizers can create events. Become an
organizer first via your profile.");

        var effectiveMaxAttendees = request.MaxAttendees;
        if (!isAdmin)
        {
            var subscription = await
subscriptionQueries.GetByUserIdAsync(userId, cancellationToken);
            var hasPro = subscription is not null &&
subscription.IsActive();

            if (hasPro)
            {
                var eventsThisMonth = await
eventQueries.CountByOrganizerThisMonthAsync(userId,
cancellationToken);
                if (eventsThisMonth >= ProMonthlyEventLimit)
```

```
                    return
Error.Validation("Event.MonthlyLimitReached",
                        $"You have reached the monthly limit of
{ProMonthlyEventLimit} events.");
            }
            else
            {
                var activeEvents = await
eventQueries.CountActiveByOrganizerAsync(userId,
cancellationToken);
                if (activeEvents >= FreeActiveEventLimit)
                    return
Error.Validation("Event.FreeLimitReached",
                        $"Free plan allows up to
{FreeActiveEventLimit} active events.");

                if (request.TicketPrice > 0)
                    return
Error.Validation("Event.PaidNotAllowed",
                        "Paid events require a Pro
subscription.");

                if (request.MaxAttendees.HasValue &&
request.MaxAttendees.Value > FreeMaxAttendeesLimit)
                    return
Error.Validation("Event.MaxAttendeesExceeded",
                        $"Free plan caps events at
{FreeMaxAttendeesLimit} attendees.");

                if (request.MaxAttendees is null)
                    effectiveMaxAttendees = FreeMaxAttendeesLimit;
            }
        }

        var evt = Event.Create(
```

```
            request.Title,
            request.Description,
            request.EventDateTime,
            userId,
            request.Location,
            request.TicketPrice,
            effectiveMaxAttendees);

        if (!string.IsNullOrEmpty(request.ImageUrl))
        {
            var photo = EventPhoto.Create(evt.Id,
request.ImageUrl, $"events/{evt.Id.Value}/0", 0);
            evt.AddPhoto(photo);
        }

        if (request.Publish)
        {
            if (!evt.Photos.Any())
                return Error.Validation("Event.NoPhoto", "Cannot
publish an event without at least one photo.");
            evt.Publish();
        }

        var created = await eventRepository.AddAsync(evt,
cancellationToken);
        logger.LogInformation("Event {EventId} created by user
{UserId}", created.Id.Value, userId);

        return new CreateEventResult(created.Id.Value, created);
    }
}
```

Команда CreateEventCommand демонструє патерн CQRS у дії: запит інкапсульований у record, валідація реалізована окремим класом на FluentValidation, а

обробник застосовує бізнес-правила (перевірка ролі, ліміти Free/Pro підписки) перед делегуванням створення сутності доменному фабричному методу Event.Create().

Лістинг Б.2. Pipeline behaviour ValidationBehaviour

```
using FluentValidation;
using MediatR;

namespace EventFlow.Application.Common.Behaviours;

public class ValidationBehaviour<TRequest, TResponse> :
IPipelineBehavior<TRequest, TResponse>
    where TRequest : IRequest<TResponse>
{
    private readonly IEnumerable<IValidator<TRequest>>
_validators;

    public ValidationBehaviour(IEnumerable<IValidator<TRequest>>
validators)
    {
        _validators = validators;
    }

    public async Task<TResponse> Handle(
        TRequest request,
        RequestHandlerDelegate<TResponse> next,
        CancellationToken cancellationToken)
    {
        if (!_validators.Any())
            return await next();

        var context = new ValidationContext<TRequest>(request);

        var validationResults = await Task.WhenAll(
            _validators.Select(v => v.ValidateAsync(context,
cancellationToken)));

        var failures = validationResults
            .SelectMany(r => r.Errors)
```

```
            .Where(f => f != null)
            .ToList();

        if (failures.Count != 0)
            throw new ValidationException(failures);

        return await next();
    }
}
```

ДОДАТОК В

Лістинг В.1. Клас AppDbContext

```
using EventFlow.Domain.Entities;
using Microsoft.EntityFrameworkCore;
using System.Reflection;

namespace EventFlow.Infrastructure.Persistence;

/// <summary>
/// Application database context for EventFlow.
/// Manages all entities and their relationships using Entity
Framework Core.
/// </summary>
public class AppDbContext : DbContext
{
    public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options)
    {
    }

    // Entity DbSets
    public DbSet<User> Users => Set<User>();
    public DbSet<Role> Roles => Set<Role>();
    public DbSet<UserRole> UserRoles => Set<UserRole>();
    public DbSet<Subscription> Subscriptions =>
Set<Subscription>();
    public DbSet<Event> Events => Set<Event>();
    public DbSet<EventPhoto> EventPhotos => Set<EventPhoto>();
    public DbSet<EventRegistration> EventRegistrations =>
Set<EventRegistration>();
    public DbSet<Ticket> Tickets => Set<Ticket>();
    public DbSet<ChatConversation> ChatConversations =>
Set<ChatConversation>();
```

```
    public DbSet<OrganizerPayoutAccount> OrganizerPayoutAccounts
=> Set<OrganizerPayoutAccount>();
    public DbSet<Payout> Payouts => Set<Payout>();

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

        // Apply all IEntityTypeConfiguration from this assembly

modelBuilder.ApplyConfigurationsFromAssembly(Assembly.GetExecuting
Assembly());
    }
}
```

ValidationBehaviour - це pipeline behaviour MediatR, що автоматично запускає всі зареєстровані для типу команди валідатори через Task.WhenAll ще до виклику обробника. Невалідна команда викидає ValidationException, який централізовано перехоплюється middleware і перетворюється на HTTP 400 з переліком помилок.

ДОДАТОК Г

Лістинг Г.1. Контролер EventsController

```
using EventFlow.Application.Common.Interfaces.Queries;
using EventFlow.Application.Common.Services;
using EventFlow.Application.Events.Commands.CancelRegistration;
using EventFlow.Application.Events.Commands.CheckInAttendee;
using EventFlow.Application.Events.Commands.ClearCalendarEventId;
using EventFlow.Application.Events.Commands.CreateEvent;
using EventFlow.Application.Events.Commands.CreateTicketCheckout;
using EventFlow.Application.Events.Commands.DeleteEvent;
using EventFlow.Application.Events.Commands.DeleteEventPhoto;
using EventFlow.Application.Events.Commands.UploadEventPhoto;
using
EventFlow.Application.Events.Commands.ProcessTicketPaymentCallback
;
using EventFlow.Application.Events.Commands.RegisterForEvent;
using EventFlow.Application.Events.Commands.SetCalendarEventId;
using EventFlow.Application.Events.Commands.UpdateEvent;
using EventFlow.API.DTOs.Events;
using EventFlow.API.DTOs.Requests;
using EventFlow.API.Modules.Errors;
using MediatR;
using EventFlow.Domain.ValueObjects;
using DomainEventId = EventFlow.Domain.ValueObjects.EventId;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace EventFlow.API.Controllers;

[ApiController]
[Route("api/[controller]")]
[Authorize]
public class EventsController(
```

```
    ISender sender,
    IEventQueries eventQueries,
    IEventRegistrationQueries registrationQueries,
    IUserResolverService userResolver,
    IConfiguration configuration,
    ILogger<EventsController> logger) : ControllerBase
{
    // Queries (read side)
    [HttpGet, AllowAnonymous]
    public async Task<IActionResult> GetEvents(
        [FromQuery] string? search = null,
        [FromQuery] string? location = null,
        [FromQuery] DateTime? dateFrom = null,
        [FromQuery] DateTime? dateTo = null,
        [FromQuery] string? status = null,
        [FromQuery] int page = 1,
        [FromQuery] int pageSize = 12,
        CancellationToken cancellationToken = default)
    {
        if (pageSize < 1) pageSize = 1;
        if (pageSize > 100) pageSize = 100;

        var totalCount = await eventQueries.CountEventsAsync(
            search, location, dateFrom, dateTo, status,
cancellationToken);

        var events = await eventQueries.SearchEventsAsync(
            search, location, dateFrom, dateTo, status,
            limit: pageSize, page: page, cancellationToken:
cancellationToken);

        var dtos =
events.Select(EventDto.FromDomainModel).ToList();
        var totalPages = (int)Math.Ceiling((double)totalCount /
pageSize);
```

```
        return Ok(new EventListResponse(dtos, totalCount, page,
pageSize, totalPages));
    }

    [HttpGet("{id:guid}"), AllowAnonymous]
    public async Task<IActionResult> GetById(Guid id,
CancellationToken cancellationToken = default)
    {
        var evt = await eventQueries.GetByIdAsync(new
DomainEventId(id), cancellationToken);
        if (evt is null)
            return NotFound(new { error = "Event not found" });

        return Ok(EventDto.FromDomainModel(evt));
    }

    [HttpGet("my-events")]
    public async Task<IActionResult> GetMyEvents(CancellationToken
cancellationToken = default)
    {
        var userIdHeader =
Request.Headers["X-User-Id"].FirstOrDefault();
        if (string.IsNullOrEmpty(userIdHeader))
            return BadRequest(new { error = "User ID is required"
});

        var userIdResult = await
userResolver.ResolveUserIdAsync(userIdHeader, cancellationToken);
        if (userIdResult.IsError)
            return NotFound(new { error = "User not found" });

        var events = await
eventQueries.GetByOrganizerAsync(userIdResult.Value,
cancellationToken);
        var dtos =
```

```
events.Select(EventDto.FromDomainModel).ToList();
        return Ok(new EventListResponse(dtos, dtos.Count, 1,
dtos.Count, 1));
    }

    [HttpGet("{id:guid}/registration-status")]
    public async Task<IActionResult> GetRegistrationStatus(Guid
id, CancellationToken cancellationToken = default)
    {
        var userIdHeader =
Request.Headers["X-User-Id"].FirstOrDefault();
        if (string.IsNullOrEmpty(userIdHeader))
            return Ok(new { isRegistered = false, isCanceled =
false });

        var userIdResult = await
userResolver.ResolveUserIdAsync(userIdHeader, cancellationToken);
        if (userIdResult.IsError)
            return Ok(new { isRegistered = false, isCanceled =
false });

        var registration = await
registrationQueries.GetByEventAndUserAsync(
            new DomainEventId(id), userIdResult.Value,
cancellationToken);

        if (registration is null)
            return Ok(new { isRegistered = false, isCanceled =
false });

        var isPaidEvent = registration.Event is not null &&
!registration.Event.IsFree;
        var isActuallyRegistered = !registration.IsCanceled &&
(!isPaidEvent || registration.IsPaid);
```

```
        return Ok(new
        {
            isRegistered = isActuallyRegistered,
            registrationId = registration.Id.Value,
            isCanceled = registration.IsCanceled,
            isPaid = registration.IsPaid,
            paymentPending = isPaidEvent &&
!registration.IsCanceled && !registration.IsPaid,
            paymentStatus = registration.Ticket?.PaymentStatus,
            googleCalendarEventId =
registration.GoogleCalendarEventId
        });
    }

    [HttpGet("{id:guid}/registrations")]
    public async Task<IActionResult> GetEventRegistrations(Guid
id, CancellationToken cancellationToken = default)
    {
        var userIdHeader =
Request.Headers["X-User-Id"].FirstOrDefault();
        if (string.IsNullOrEmpty(userIdHeader))
            return BadRequest(new { error = "User ID is required"
});

        var userIdResult = await
userResolver.ResolveUserIdAsync(userIdHeader, cancellationToken);
        if (userIdResult.IsError)
            return Unauthorized();

        var evt = await eventQueries.GetByIdAsync(new
DomainEventId(id), cancellationToken);
        if (evt is null) return NotFound();
        if (evt.OrganizerId != userIdResult.Value) return
Forbid();
```

```
        var registrations = await
registrationQueries.GetByEventAsync(new DomainEventId(id),
cancellationToken);
        var dtos =
registrations.Select(EventAttendeeDto.FromDomainModel).ToList();
        return Ok(dtos);
    }

    // Commands (write side)

    [HttpPost]
    public async Task<IActionResult> CreateEvent(
        [FromBody] CreateEventRequest request,
        CancellationToken cancellationToken = default)
    {
        var userIdHeader =
Request.Headers["X-User-Id"].FirstOrDefault();
        if (string.IsNullOrEmpty(userIdHeader))
            return BadRequest(new { error = "User ID is required"
});

        var result = await sender.Send(
            new CreateEventCommand(
                request.Title, request.Description,
request.EventDateTime,
                request.Location, request.ImageUrl,
request.Publish,
                request.TicketPrice, request.MaxAttendees,
userIdHeader),
            cancellationToken);

        return result.Match<IActionResult>(
            value => CreatedAtAction(nameof(GetById),
                new { id = value.EventId },
                EventDto.FromDomainModel(value.CreatedEvent)),
```

```
            errors => result.ToObjectResult());
    }

    [HttpPut("{id:guid}")]
    public async Task<IActionResult> UpdateEvent(
        Guid id,
        [FromBody] UpdateEventRequest request,
        CancellationToken cancellationToken = default)
    {
        var userIdHeader =
Request.Headers["X-User-Id"].FirstOrDefault();
        if (string.IsNullOrEmpty(userIdHeader))
            return BadRequest(new { error = "User ID is required"
});

        var result = await sender.Send(
            new UpdateEventCommand(
                id, request.Title, request.Description,
request.EventDateTime,
                request.Location, request.ImageUrl,
request.Status,
                request.TicketPrice, request.MaxAttendees,
userIdHeader),
            cancellationToken);

        return result.ToObjectResult(v =>
EventDto.FromDomainModel(v.UpdatedEvent));
    }

    [HttpDelete("{id:guid}")]
    public async Task<IActionResult> DeleteEvent(Guid id,
CancellationToken cancellationToken = default)
    {
        var userIdHeader =
Request.Headers["X-User-Id"].FirstOrDefault();
```

```
        if (string.IsNullOrEmpty(userIdHeader))
            return BadRequest(new { error = "User ID is required"
});

        var result = await sender.Send(
            new DeleteEventCommand(id, userIdHeader),
cancellationToken);

        return result.Match<IActionResult>(
            _ => NoContent(),
            errors => result.ToObjectResult());
    }

    [HttpPost("{id:guid}/register")]
    public async Task<IActionResult> RegisterForEvent(Guid id,
CancellationToken cancellationToken = default)
    {
        var userIdHeader =
Request.Headers["X-User-Id"].FirstOrDefault();
        if (string.IsNullOrEmpty(userIdHeader))
            return BadRequest(new { error = "User ID is required.
Please log in." });

        var result = await sender.Send(
            new RegisterForEventCommand(id, userIdHeader),
cancellationToken);

        return result.ToObjectResult();
    }

    [HttpDelete("{id:guid}/register")]
    public async Task<IActionResult> CancelRegistration(Guid id,
CancellationToken cancellationToken = default)
    {
        var userIdHeader =
```

```
Request.Headers["X-User-Id"].FirstOrDefault();
        if (string.IsNullOrEmpty(userIdHeader))
            return BadRequest(new { error = "User ID is required"
});

        var result = await sender.Send(
            new CancelRegistrationCommand(id, userIdHeader),
cancellationToken);

        return result.ToObjectResult();
    }

    [HttpPost("checkin/{registrationId:guid}")]
    public async Task<IActionResult> CheckIn(Guid registrationId,
CancellationToken cancellationToken = default)
    {
        var userIdHeader =
Request.Headers["X-User-Id"].FirstOrDefault();
        if (string.IsNullOrEmpty(userIdHeader))
            return BadRequest(new { error = "User ID is required"
});

        var result = await sender.Send(
            new CheckInAttendeeCommand(registrationId,
userIdHeader), cancellationToken);

        return result.ToObjectResult(v => new { success =
v.Success, alreadyCheckedIn = v.AlreadyCheckedIn, checkedInAt =
v.CheckedInAt });
    }

    /// <summary>
    /// Creates a LiqPay checkout for a paid event ticket.
    /// </summary>
    [HttpPost("{id:guid}/checkout")]
```

```
    public async Task<IActionResult> CreateTicketCheckout(Guid id,
CancellationToken cancellationToken = default)
    {
        var userIdHeader =
Request.Headers["X-User-Id"].FirstOrDefault();
        if (string.IsNullOrEmpty(userIdHeader))
            return BadRequest(new { error = "User ID is required"
});

        var frontendUrl = configuration["FrontendUrl"] ??
"http://localhost:3000";
        var callbackBaseUrl =
configuration["LiqPay:CallbackBaseUrl"]
            ?? $"{Request.Scheme}://{Request.Host}";

        var result = await sender.Send(
            new CreateTicketCheckoutCommand(id, userIdHeader,
frontendUrl, callbackBaseUrl),
            cancellationToken);

        return result.ToObjectResult();
    }

    // Event photos

    [HttpPost("{id:guid}/photos"), RequestSizeLimit(10 * 1024 *
1024)]
    public async Task<IActionResult> UploadPhoto(
        Guid id,
        IFormFile file,
        CancellationToken cancellationToken = default)
    {
        var userIdHeader =
Request.Headers["X-User-Id"].FirstOrDefault();
        if (string.IsNullOrEmpty(userIdHeader))
```

```
            return BadRequest(new { error = "User ID is required"
});

        if (file is null || file.Length == 0)
            return BadRequest(new { error = "No file uploaded" });

        await using var stream = file.OpenReadStream();
        var result = await sender.Send(
            new UploadEventPhotoCommand(id, stream,
file.ContentType, file.FileName, userIdHeader),
            cancellationToken);

        return result.ToObjectResult(v => new
        {
            id = v.PhotoId,
            imageUrl = v.ImageUrl,
            imageKey = v.ImageKey,
            order = v.Order
        });
    }

    [HttpDelete("{id:guid}/photos/{photoId:guid}")]
    public async Task<IActionResult> DeletePhoto(
        Guid id, Guid photoId,
        CancellationToken cancellationToken = default)
    {
        var userIdHeader =
Request.Headers["X-User-Id"].FirstOrDefault();
        if (string.IsNullOrEmpty(userIdHeader))
            return BadRequest(new { error = "User ID is required"
});

        var result = await sender.Send(
            new DeleteEventPhotoCommand(id, photoId,
userIdHeader), cancellationToken);
```

```
        return result.Match<IActionResult>(
            _ => NoContent(),
            errors => result.ToObjectResult());
    }

    /// <summary>
    /// LiqPay server-to-server callback for ticket payments.
    /// </summary>
    [HttpPost("ticket-callback"), AllowAnonymous]
    public async Task<IActionResult>
TicketPaymentCallback(CancellationToken cancellationToken =
default)
    {
        var data = Request.Form["data"].FirstOrDefault();
        var signature =
Request.Form["signature"].FirstOrDefault();

        if (string.IsNullOrEmpty(data) ||
string.IsNullOrEmpty(signature))
            return BadRequest("Missing data or signature");

        var result = await sender.Send(
            new ProcessTicketPaymentCallbackCommand(data,
signature), cancellationToken);

        return result.Match<IActionResult>(
            _ => Ok(),
            errors => result.ToObjectResult());
    }
}
```

Контролер EventsController - тонкий шар над MediatR: кожен endpoint конвертує HTTP-запит у команду або запит, надсилає її через ISender і повертає результат через

extension-метод .ToObjectResult(). Жодної бізнес-логіки в контролері немає - це принцип Clean Architecture, що ізолює доменні правила від технічних деталей HTTP.

Лістинг Г.2. Сервіс UserContextService

```
using EventFlow.Application.Common.Interfaces.Services;
using EventFlow.Domain.ValueObjects;
using Microsoft.AspNetCore.Http;

namespace EventFlow.Infrastructure.Services;

/// <summary>
/// Service for accessing the current user context from HTTP
context.
/// Used by AI plugins to perform actions on behalf of the
authenticated user.
/// </summary>
public class UserContextService : IUserContextService
{
    private readonly IHttpContextAccessor _httpContextAccessor;

    private const string ClerkUserIdClaim = "sub";
    private const string InternalUserIdClaim = "internal_user_id";

    private UserId? _explicitUserId;

    public UserContextService(IHttpContextAccessor
httpContextAccessor)
    {
        _httpContextAccessor = httpContextAccessor;
    }

    public void SetExplicitUserId(UserId? userId) =>
_explicitUserId = userId;
```

```
    public UserId? GetCurrentUserId()
    {
        if (_explicitUserId.HasValue)
            return _explicitUserId;

        var userIdClaim =
_httpContextAccessor.HttpContext?.User?.FindFirst(InternalUserIdCl
aim)?.Value;

        if (string.IsNullOrEmpty(userIdClaim))
            return null;

        if (Guid.TryParse(userIdClaim, out var guid))
            return new UserId(guid);

        return null;
    }

    public string? GetCurrentClerkUserId()
    {
        return
_httpContextAccessor.HttpContext?.User?.FindFirst(ClerkUserIdClaim
)?.Value;
    }

    public bool IsAuthenticated()
    {
        return
_httpContextAccessor.HttpContext?.User?.Identity?.IsAuthenticated
?? false;
    }
}
```

Сервіс UserContextService забезпечує доступ до поточного користувача з будь-якого шару Application через інтерфейс IUserContextService. Підтримує два

режими: автоматичний (зчитує internal_user_id claim з JWT) та explicit (для SignalR-хабів, де HTTP-контекст недоступний у звичайному вигляді).

ДОДАТОК Д

Лістинг Д.1. Сервіс LiqPayService

```
using System.Security.Cryptography;
using System.Text;
using System.Text.Json;
using EventFlow.Application.Common.Interfaces.Services;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;

namespace EventFlow.Infrastructure.Payment;

/// <summary>
/// LiqPay payment service implementation.
/// Handles checkout form data generation and callback validation.
/// </summary>
public class LiqPayService : ILiqPayService
{
    private readonly LiqPaySettings _settings;
    private readonly ILogger<LiqPayService> _logger;

    public LiqPayService(IOptions<LiqPaySettings> settings,
ILogger<LiqPayService> logger)
    {
        _settings = settings.Value;
        _logger = logger;
    }

    public (string Data, string Signature) CreateCheckoutData(
        string orderId, decimal amount, string description,
        string resultUrl, string serverUrl)
```

```
    {
        var requestData = new Dictionary<string, object>
        {
            ["public_key"] = _settings.PublicKey,
            ["version"] = 3,
            ["action"] = "pay",
            ["amount"] = amount,
            ["currency"] = "UAH",
            ["description"] = description,
            ["order_id"] = orderId,
            ["result_url"] = resultUrl,
            ["server_url"] = serverUrl,
        };

        if (_settings.IsSandbox)
        {
            requestData["sandbox"] = 1;
        }

        var jsonData = JsonSerializer.Serialize(requestData);
        var base64Data =
Convert.ToBase64String(Encoding.UTF8.GetBytes(jsonData));
        var signature = GenerateSignature(base64Data);

        return (base64Data, signature);
    }

    public async Task<bool> RefundAsync(string orderId, decimal
amount)
    {
        var requestData = new Dictionary<string, object>
        {
            ["public_key"] = _settings.PublicKey,
            ["version"] = 3,
            ["action"] = "refund",
```

```
            ["order_id"] = orderId,
            ["amount"] = amount,
        };

        var jsonData = JsonSerializer.Serialize(requestData);
        var base64Data =
Convert.ToBase64String(Encoding.UTF8.GetBytes(jsonData));
        var signature = GenerateSignature(base64Data);

        using var httpClient = new HttpClient();
        var content = new FormUrlEncodedContent(new
Dictionary<string, string>
        {
            ["data"] = base64Data,
            ["signature"] = signature,
        });

        try
        {
            var response = await
httpClient.PostAsync("https://www.liqpay.ua/api/request",
content);
            var responseBody = await
response.Content.ReadAsStringAsync();

            using var doc = JsonDocument.Parse(responseBody);
            var result = doc.RootElement.TryGetProperty("result",
out var r) ? r.GetString() : null;
            var status = doc.RootElement.TryGetProperty("status",
out var s) ? s.GetString() : null;

            var isSuccess = result == "ok" && (status ==
"reversed" || status == "success");

            if (isSuccess)
```

```
            {
                _logger.LogInformation(
                    "LiqPay refund OK: orderId={OrderId}, 
amount={Amount}, status={Status}",
                    orderId, amount, status);
            }
            else
            {
                _logger.LogWarning(
                    "LiqPay refund FAILED: orderId={OrderId}, 
amount={Amount}, fullResponse={Body}",
                    orderId, amount, responseBody);
            }

            return isSuccess;
        }
        catch (Exception ex)
        {
            _logger.LogError(ex,
                "LiqPay refund request threw exception: 
orderId={OrderId}, amount={Amount}",
                orderId, amount);
            return false;
        }
    }

    public bool ValidateCallback(string data, string signature)
    {
        var expectedSignature = GenerateSignature(data);
        return CryptographicOperations.FixedTimeEquals(
            Encoding.UTF8.GetBytes(expectedSignature),
            Encoding.UTF8.GetBytes(signature));
    }

    public string DecodeCallbackData(string data)
```

```
    {
        var bytes = Convert.FromBase64String(data);
        return Encoding.UTF8.GetString(bytes);
    }

    private string GenerateSignature(string base64Data)
    {
        var signString = _settings.PrivateKey + base64Data +
_settings.PrivateKey;
        var sha1Bytes =
SHA1.HashData(Encoding.UTF8.GetBytes(signString));
        return Convert.ToBase64String(sha1Bytes);
    }
}
```

Сервіс LiqPayService інкапсулює інтеграцію з платіжним шлюзом ПриватБанку. Метод CreateCheckoutData формує платіжний запит з підписом за алгоритмом Base64(SHA1(privateKey + data + privateKey)). Метод ValidateCallback використовує CryptographicOperations.FixedTimeEquals для запобігання timing-атакам при перевірці сигнатури callback-ів від LiqPay.

ДОДАТОК Е

Лістинг Е.1. AI-плагін EventManagementPlugin

```
using System.ComponentModel;
using System.Text;
using EventFlow.Application.Common.Interfaces.Queries;
using EventFlow.Application.Common.Interfaces.Repositories;
using EventFlow.Application.Common.Interfaces.Services;
using EventFlow.Domain.Entities;
using EventFlow.Domain.ValueObjects;
using Microsoft.SemanticKernel;

namespace EventFlow.Infrastructure.AI.Plugins;

/// <summary>
/// Semantic Kernel plugin for organizer event management.
/// Enables the AI to create, update, publish, and delete events
on behalf of the organizer.
/// </summary>
public class EventManagementPlugin
{
    private const int FreeActiveEventLimit = 3;
    private const int ProMonthlyEventLimit = 50;

    private readonly IEventRepository _eventRepository;
    private readonly IEventQueries _eventQueries;
    private readonly IUserContextService _userContextService;
    private readonly IUserRepository _userRepository;
    private readonly ISubscriptionQueries _subscriptionQueries;
    private readonly IStorageService _storageService;
    private readonly IEventRefundService _refundService;

    public EventManagementPlugin(
        IEventRepository eventRepository,
```

```
        IEventQueries eventQueries,
        IUserContextService userContextService,
        IUserRepository userRepository,
        ISubscriptionQueries subscriptionQueries,
        IStorageService storageService,
        IEventRefundService refundService)
    {
        _eventRepository = eventRepository;
        _eventQueries = eventQueries;
        _userContextService = userContextService;
        _userRepository = userRepository;
        _subscriptionQueries = subscriptionQueries;
        _storageService = storageService;
        _refundService = refundService;
    }

    [KernelFunction("create_event")]
    [Description("Create a new event as a draft. Use this when the
organizer wants to create an event.")]
    public async Task<string> CreateEventAsync(
        [Description("Title of the event")] string title,
        [Description("Full description of the event")] string
description,
        [Description("Date and time of the event (format:
yyyy-MM-dd HH:mm). Must be in the future.")] string eventDateTime,
        [Description("Location of the event")] string? location =
null,
        [Description("Ticket price in UAH. Use 0 for free events
(default: 0).")] decimal ticketPrice = 0,
        [Description("Maximum number of attendees. Leave empty for
unlimited.")] int? maxAttendees = null)
    {
        var userId = _userContextService.GetCurrentUserId();
        if (userId == null)
            return "You need to be logged in to create events.
```

```
Please sign in first.";

        var isOrganizer = await
_userRepository.HasRoleAsync(userId.Value, "Organizer");
        var isAdmin = await
_userRepository.HasRoleAsync(userId.Value, "Admin");

        if (!isOrganizer && !isAdmin)
            return "Only organizers can create events. Please
upgrade to a Pro subscription to unlock event creation.";

        if (!isAdmin)
        {
            var subscription = await
_subscriptionQueries.GetByUserIdAsync(userId.Value);
            var hasPro = subscription is not null &&
subscription.IsActive();

            if (hasPro)
            {
                var eventsThisMonth = await
_eventQueries.CountByOrganizerThisMonthAsync(userId.Value);
                if (eventsThisMonth >= ProMonthlyEventLimit)
                    return $"You have reached the monthly limit of
{ProMonthlyEventLimit} events.";
            }
            else
            {
                return "AI event creation is a Pro feature. You
can create events manually via the Dashboard.";
            }
        }

        if (!DateTime.TryParse(eventDateTime, out var
parsedDateTime))
```

```
            return "Invalid date format. Please use format:
yyyy-MM-dd HH:mm.";

        if (parsedDateTime <= DateTime.UtcNow)
            return "Event date must be in the future.";

        if (ticketPrice < 0)
            return "Ticket price cannot be negative.";

        if (maxAttendees is <= 0)
            return "Maximum attendees must be greater than zero.";

        var evt = Event.Create(
            title, description, parsedDateTime, userId.Value,
            location, ticketPrice, maxAttendees);

        var created = await _eventRepository.AddAsync(evt);

        return $"Event **\"{title}\"** has been created as a
draft.\n\n" +
               $"**To publish this event**, you need to add at
least one photo first.\n" +
               $"Please go to the [Edit Event
page](/events/{created.Id.Value}/edit) to upload a photo.";
    }

    [KernelFunction("update_event")]
    [Description("Update an existing event's details. Pass the
event ID (GUID) OR the event title.")]
    public async Task<string> UpdateEventAsync(
        [Description("The event ID (GUID) OR exact title of the
event to update")] string eventId,
        [Description("New title (leave empty to keep current)")]
string? title = null,
        [Description("New description (leave empty to keep
```

```
current)")] string? description = null,
        [Description("New date/time (format: yyyy-MM-dd HH:mm)")]
string? eventDateTime = null,
        [Description("New location")] string? location = null,
        [Description("New ticket price in UAH")] decimal?
ticketPrice = null,
        [Description("New maximum attendees")] int? maxAttendees =
null)
    {
        var userId = _userContextService.GetCurrentUserId();
        if (userId == null) return "You need to be logged in to
manage events.";

        var isAdmin = await
_userRepository.HasRoleAsync(userId.Value, "Admin");
        if (!isAdmin)
        {
            var subscription = await
_subscriptionQueries.GetByUserIdAsync(userId.Value);
            if (subscription is null || !subscription.IsActive())
                return "AI event editing is a Pro feature.";
        }

        var evt = await ResolveOrganizerEventAsync(eventId,
userId.Value);
        if (evt is null)
            return "I couldn't find that event among the ones you
organized.";

        DateTime? parsedDateTime = null;
        if (!string.IsNullOrEmpty(eventDateTime))
        {
            if (!DateTime.TryParse(eventDateTime, out var dt))
                return "Invalid date format. Please use yyyy-MM-dd
HH:mm.";
```

```
            if (dt <= DateTime.UtcNow)
                return "Event date must be in the future.";
            parsedDateTime = dt;
        }

        var newPrice = ticketPrice ?? evt.TicketPrice;
        if (newPrice < 0) return "Ticket price cannot be
negative.";

        evt.UpdateDetails(
            title ?? evt.Title,
            description ?? evt.Description,
            parsedDateTime ?? evt.EventDateTime,
            location ?? evt.Location,
            newPrice,
            maxAttendees ?? evt.MaxAttendees);

        await _eventRepository.UpdateAsync(evt);

        return $"Event **\"{evt.Title}\"** has been updated.";
    }

    [KernelFunction("publish_event")]
    [Description("Publish a draft event. The event must have at
least one photo.")]
    public async Task<string> PublishEventAsync(
        [Description("The event ID (GUID) OR exact title")] string
eventId)
    {
        var userId = _userContextService.GetCurrentUserId();
        if (userId == null) return "You need to be logged in.";

        var isAdmin = await
_userRepository.HasRoleAsync(userId.Value, "Admin");
        if (!isAdmin)
```

```
        {
            var subscription = await
_subscriptionQueries.GetByUserIdAsync(userId.Value);
            if (subscription is null || !subscription.IsActive())
                return "AI event publishing is a Pro feature.";
        }

        var evt = await ResolveOrganizerEventAsync(eventId,
userId.Value);
        if (evt is null) return "I couldn't find that event.";

        if (evt.Status == Event.EventStatus.Published)
            return $"Event **\"{evt.Title}\"** is already
published.";

        if (evt.Status == Event.EventStatus.Canceled)
            return "Cannot publish a canceled event.";

        if (!evt.Photos.Any())
            return $"Event **\"{evt.Title}\"** needs at least one
photo before publishing.";

        evt.Publish();
        await _eventRepository.UpdateAsync(evt);

        return $"Event **\"{evt.Title}\"** is now **published**
and visible to attendees!";
    }

    [KernelFunction("get_my_events")]
    [Description("Get a list of events organized by the current
user.")]
    public async Task<string> GetMyEventsAsync()
    {
        var userId = _userContextService.GetCurrentUserId();
```

```
        if (userId == null) return "You need to be logged in.";

        var events = await
_eventQueries.GetByOrganizerAsync(userId.Value);
        if (!events.Any())
            return "You haven't created any events yet.";

        var sb = new StringBuilder();
        sb.AppendLine($"You have **{events.Count}** event(s):\n");

        foreach (var evt in events)
        {
            var statusLabel = evt.Status switch
            {
                Event.EventStatus.Draft => "Draft",
                Event.EventStatus.Published => "Published",
                Event.EventStatus.Canceled => "Canceled",
                _ => evt.Status
            };
            sb.AppendLine($"**{evt.Title}** -- {statusLabel}");
            sb.AppendLine($"  - Date: {evt.EventDateTime:dddd,
MMMM d, yyyy 'at' h:mm tt}");
            sb.AppendLine($"  - Registrations:
{evt.GetRegistrationCount()}");
            sb.AppendLine();
        }
        return sb.ToString();
    }

    [KernelFunction("delete_event")]
    [Description("Delete an event. Refunds all paid attendees
first.")]
    public async Task<string> DeleteEventAsync(
        [Description("The event ID (GUID) OR exact title")] string
eventId)
```

```
    {
        var userId = _userContextService.GetCurrentUserId();
        if (userId == null) return "You need to be logged in.";

        var evt = await ResolveOrganizerEventAsync(eventId,
userId.Value);
        if (evt is null) return "I couldn't find that event.";

        var domainId = evt.Id;
        var title = evt.Title;

        var refundSummary = await
_refundService.RefundAllRegistrationsAsync(
            evt, $"Event '{title}' was deleted by the organizer");

        foreach (var photo in evt.Photos)
        {
            await _storageService.DeleteAsync(photo.ImageKey);
        }

        await _eventRepository.DeleteAsync(domainId);

        return $"Event **\"{title}\"** has been deleted.";
    }

    private async Task<Event?> ResolveOrganizerEventAsync(string
eventIdOrTitle, UserId userId)
    {
        Event? evt = null;

        if (Guid.TryParse(eventIdOrTitle, out var guid))
        {
            evt = await
_eventRepository.GetByIdWithDetailsAsync(new EventId(guid));
        }
```

```
        if (evt is null)
        {
            var myEvents = await
_eventQueries.GetByOrganizerAsync(userId);
            evt = myEvents.FirstOrDefault(e =>
                     e.Title.Equals(eventIdOrTitle,
StringComparison.OrdinalIgnoreCase))
                  ?? myEvents.FirstOrDefault(e =>
                     e.Title.Contains(eventIdOrTitle,
StringComparison.OrdinalIgnoreCase));

            if (evt is not null)
                evt = await
_eventRepository.GetByIdWithDetailsAsync(evt.Id);
        }

        if (evt is null) return null;
        if (evt.OrganizerId == userId) return evt;

        var isAdmin = await _userRepository.HasRoleAsync(userId,
"Admin");
        return isAdmin ? evt : null;
    }
}
```

AI-плагін EventManagementPlugin оголошує функції створення, оновлення, публікації, перегляду та видалення подій з атрибутами [KernelFunction] та [Description], які Semantic Kernel автоматично перетворює на tool-definitions для LLM. Кожна функція перевіряє права через IUserContextService та підписку через ISubscriptionQueries, що ізолює дозволи від моделі.

Лістинг Е.2. Сервіс-оркестратор AiChatService

```
using System.Runtime.CompilerServices;
using EventFlow.Application.Common.Interfaces.Repositories;
```

```
using EventFlow.Application.Common.Interfaces.Services;
using EventFlow.Domain.ValueObjects;
using EventFlow.Infrastructure.AI.Plugins;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using Microsoft.SemanticKernel;
using Microsoft.SemanticKernel.ChatCompletion;
using Microsoft.SemanticKernel.Connectors.OpenAI;

namespace EventFlow.Infrastructure.AI;

/// <summary>
/// AI Chat Service using Semantic Kernel with native function
calling.
/// Plugins are invoked automatically by the kernel when the LLM
requests them.
/// </summary>
public class AiChatService : IAiChatService
{
    private readonly Kernel _kernel;
    private readonly AiOptions _options;
    private readonly EventSearchPlugin _eventSearchPlugin;
    private readonly EventRegistrationPlugin
_eventRegistrationPlugin;
    private readonly EventManagementPlugin _eventManagementPlugin;
    private readonly OrganizerDashboardPlugin
_organizerDashboardPlugin;
    private readonly IUserContextService _userContextService;
    private readonly IUserRepository _userRepository;
    private readonly ILogger<AiChatService> _logger;
    private readonly AiResponseDataCollector _dataCollector;

    private const string SystemPromptTemplate = """
        You are EventFlow AI, a friendly assistant for an event
management platform.
```

```
        You help users find events, register, and manage
registrations.
        You also help organizers create and manage events.

        === CRITICAL RULE ===
        You have tools available. You MUST call a tool for any
request that reads or changes
        event/registration data. NEVER invent results.

        TOOL ROUTING:
        - "show me events" → search_events
        - "show me details about X" → get_event_details
        - "register me for X" → register_for_event
        - "cancel my registration for X" → cancel_registration
        - "my registrations" → get_my_registrations
        - "show my events" → get_my_events
        - "delete X event" → delete_event
        - "publish X event" → publish_event
        - "update X event" → update_event

        Current date: {current_datetime}
        """;

    public AiChatService(
        Kernel kernel,
        IOptions<AiOptions> options,
        EventSearchPlugin eventSearchPlugin,
        EventRegistrationPlugin eventRegistrationPlugin,
        EventManagementPlugin eventManagementPlugin,
        OrganizerDashboardPlugin organizerDashboardPlugin,
        AiResponseDataCollector dataCollector,
        IUserContextService userContextService,
        IUserRepository userRepository,
        ILogger<AiChatService> logger)
    {
```

```
        _kernel = kernel;
        _options = options.Value;
        _eventSearchPlugin = eventSearchPlugin;
        _eventRegistrationPlugin = eventRegistrationPlugin;
        _eventManagementPlugin = eventManagementPlugin;
        _organizerDashboardPlugin = organizerDashboardPlugin;
        _dataCollector = dataCollector;
        _userContextService = userContextService;
        _userRepository = userRepository;
        _logger = logger;
    }

    public async Task<AiChatResponse> ProcessMessageAsync(
        string? userId,
        string message,
        IReadOnlyList<ChatMessage>? conversationHistory = null,
        CancellationToken cancellationToken = default)
    {
        string? lastFunctionResult = null;

        try
        {
            // Resolve Clerk userId → database UserId so plugins
can act on behalf of user
            await ResolveAndSetUserIdAsync(userId,
cancellationToken);

            // Clone the singleton kernel and attach scoped
plugins for this request
            var kernel = _kernel.Clone();

            // Merge all plugin functions into a SINGLE plugin
named "EventFlow"
            var allFunctions = new List<KernelFunction>();
            foreach (var pluginInstance in new object[]
```

```
                     {
                         _eventSearchPlugin,
                         _eventRegistrationPlugin,
                         _eventManagementPlugin,
                         _organizerDashboardPlugin,
                     })
            {
                var extracted =
KernelPluginFactory.CreateFromObject(pluginInstance);
                allFunctions.AddRange(extracted);
            }
            kernel.Plugins.AddFromFunctions("EventFlow",
allFunctions);

            // Function invocation filter -- capture the result of
each tool call
            kernel.FunctionInvocationFilters.Add(
                new CaptureFunctionResultFilter(result =>
lastFunctionResult = result));

            var systemPrompt = SystemPromptTemplate
                .Replace("{current_datetime}",
DateTime.Now.ToString("f"));
            var chatHistory = new ChatHistory(systemPrompt);

            if (conversationHistory != null)
            {
                var recentHistory = conversationHistory.Count > 4
                    ?
conversationHistory.Skip(conversationHistory.Count - 4).ToList()
                    : conversationHistory;

                foreach (var msg in recentHistory)
                {
                    if (msg.Role == "user")
```

```
                        chatHistory.AddUserMessage(msg.Content);
                    else

chatHistory.AddAssistantMessage(msg.Content);
                }
            }

            chatHistory.AddUserMessage(message);

            // FunctionChoiceBehavior.Auto() -- kernel
automatically:
            // 1. Send plugin definitions to LLM as tools
            // 2. Detect when LLM requests a tool call
            // 3. Invoke the matching [KernelFunction] method
            // 4. Append result and call LLM again for final
response
#pragma warning disable SKEXP0001
            var settings = new OpenAIPromptExecutionSettings
            {
                FunctionChoiceBehavior =
FunctionChoiceBehavior.Auto(),
                Temperature = _options.Temperature,
                MaxTokens = _options.MaxTokens
            };
#pragma warning restore SKEXP0001

            var chatCompletion =
kernel.GetRequiredService<IChatCompletionService>();

            var result = await
chatCompletion.GetChatMessageContentAsync(
                chatHistory, settings, kernel, cancellationToken);

            var responseText = result.Content ?? "I'm sorry, I
couldn't generate a response.";
```

```
            var structuredData = _dataCollector.HasData
                ? _dataCollector.Get()
                : (Type: (string?)null, Data: (object?)null);

            return new AiChatResponse(responseText, Success: true,
                StructuredData: structuredData.Data is not null
                    ? new { type = structuredData.Type, data = 
structuredData.Data }
                    : null);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Error processing AI chat 
message");
            return new AiChatResponse(
                "I apologize, but I encountered an error 
processing your request. Please try again.",
                Success: false,
                Error: ex.Message);
        }
    }

    public async IAsyncEnumerable<string> 
ProcessMessageStreamAsync(
        string? userId,
        string message,
        IReadOnlyList<ChatMessage>? conversationHistory = null,
        [EnumeratorCancellation] CancellationToken 
cancellationToken = default)
    {
        var response = await ProcessMessageAsync(userId, message, 
conversationHistory, cancellationToken);

        var words = response.Message.Split(' ');
```

```
        foreach (var word in words)
        {
            yield return word + " ";
            await Task.Delay(20, cancellationToken);
        }
    }

    private async Task ResolveAndSetUserIdAsync(string? userId,
CancellationToken cancellationToken)
    {
        if (string.IsNullOrEmpty(userId)) return;

        if (userId.StartsWith("user_"))
        {
            var user = await
_userRepository.GetByClerkIdAsync(userId, cancellationToken);
            if (user != null)
            {
                _userContextService.SetExplicitUserId(user.Id);
            }
        }
        else if (Guid.TryParse(userId, out var guid))
        {
            _userContextService.SetExplicitUserId(new
UserId(guid));
        }
    }
}
```

Сервіс AiChatService оркеструє AI-розмову: клонує kernel, об'єднує чотири плагіни в один логічний tool-namespace EventFlow-*, формує системний промпт з поточним часом, передає історію розмови (останні 4 повідомлення) та надсилає запит на LLM з налаштуванням FunctionChoiceBehavior.Auto(), що дозволяє моделі самостійно вирішувати, які функції викликати.

ДОДАТОК Ж

Лістинг Ж.1. SignalR хаб NotificationsHub

```
using Microsoft.AspNetCore.SignalR;

namespace EventFlow.Infrastructure.Hubs;

/// <summary>
/// Real-time notifications hub.
/// Clients connect with ?userId=<clerkUserId>&role=<role>
/// and join the appropriate groups to receive targeted
notifications.
/// </summary>
public class NotificationsHub : Hub
{
    public override async Task OnConnectedAsync()
    {
        var query = Context.GetHttpContext()?.Request.Query;
        var userId = query?["userId"].FirstOrDefault();
        var role   = query?["role"].FirstOrDefault();

        if (!string.IsNullOrEmpty(userId))
            await Groups.AddToGroupAsync(Context.ConnectionId,
$"user:{userId}");

        if (role == "admin")
            await Groups.AddToGroupAsync(Context.ConnectionId,
"admin");

        if (role is "organizer" or "admin" &&
!string.IsNullOrEmpty(userId))
            await Groups.AddToGroupAsync(Context.ConnectionId,
$"organizer:{userId}");
```

```
        await base.OnConnectedAsync();
    }
}
```

Хаб NotificationsHub додає підключені клієнти в SignalR-групи на основі query-параметрів userId і role: персональна група user:{clerkId} для нотифікацій про власні дії, organizer:{clerkId} для організаторських подій та глобальна admin для системних повідомлень. Це дозволяє таргетувати розсилку без перебору з'єднань.

Лістинг Ж.2. Реалізація SignalRNotificationService

```
using EventFlow.Application.Common.Interfaces.Services;
using EventFlow.Infrastructure.Hubs;
using Microsoft.AspNetCore.SignalR;

namespace EventFlow.Infrastructure.Services;

public class SignalRNotificationService : INotificationService
{
    private readonly IHubContext<NotificationsHub> _hub;

    public
SignalRNotificationService(IHubContext<NotificationsHub> hub) =>
_hub = hub;

    public Task NotifyUserAsync(string clerkUserId,
NotificationPayload payload, CancellationToken ct = default) =>

_hub.Clients.Group($"user:{clerkUserId}").SendAsync("ReceiveNotifi
cation", payload, ct);

    public Task NotifyOrganizerAsync(string organizerClerkUserId,
NotificationPayload payload, CancellationToken ct = default) =>
```

```
_hub.Clients.Group($"organizer:{organizerClerkUserId}").SendAsync(
"ReceiveNotification", payload, ct);

    public Task NotifyAdminsAsync(NotificationPayload payload,
CancellationToken ct = default) =>

_hub.Clients.Group("admin").SendAsync("ReceiveNotification",
payload, ct);
}
```

Сервіс SignalRNotificationService реалізує інтерфейс INotificationService з шару Application через IHubContext<NotificationsHub>. Бізнес-логіка не знає про SignalR - вона викликає NotifyUserAsync / NotifyOrganizerAsync / NotifyAdminsAsync на абстракції, що дозволяє замінити real-time транспорт без змін в Application Layer.

Лістинг Ж.3. Сервіс S3StorageService

```
using Amazon.S3;
using Amazon.S3.Model;
using EventFlow.Application.Common.Interfaces.Services;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;

namespace EventFlow.Infrastructure.Storage;

/// <summary>
/// S3-compatible object storage implementation. Works with AWS
S3, LocalStack, and Cloudflare R2.
/// </summary>
public class S3StorageService : IStorageService
{
    private readonly IAmazonS3 _s3;
    private readonly StorageOptions _options;
    private readonly ILogger<S3StorageService> _logger;

```

```
    public S3StorageService(
        IAmazonS3 s3,
        IOptions<StorageOptions> options,
        ILogger<S3StorageService> logger)
    {
        _s3 = s3;
        _options = options.Value;
        _logger = logger;
    }

    public async Task<StorageUploadResult> UploadAsync(
        Stream content,
        string contentType,
        string keyPrefix,
        string originalFileName,
        CancellationToken cancellationToken = default)
    {
        if
(!StorageOptions.AllowedContentTypes.Contains(contentType,
StringComparer.OrdinalIgnoreCase))
            throw new InvalidOperationException(
                $"Content type '{contentType}' is not allowed.");

        // Buffer into MemoryStream so AWSSDK computes SHA256
instead of chunked encoding.
        // Cloudflare R2 rejects chunked uploads with signature
errors.
        byte[] bytes;
        using (var buffer = new MemoryStream())
        {
            await content.CopyToAsync(buffer, cancellationToken);
            bytes = buffer.ToArray();
        }

        if (bytes.Length > StorageOptions.MaxFileSizeBytes)
```

```
            throw new InvalidOperationException(
                $"File size {bytes.Length} exceeds max
{StorageOptions.MaxFileSizeBytes} bytes.");

        if (bytes.Length == 0)
            throw new InvalidOperationException("File is empty.");

        var extension = GetExtension(originalFileName,
contentType);
        var key =
$"{keyPrefix.TrimEnd('/')}/{Guid.NewGuid():N}{extension}";

        using var uploadStream = new MemoryStream(bytes);
        var request = new PutObjectRequest
        {
            BucketName = _options.BucketName,
            Key = key,
            InputStream = uploadStream,
            ContentType = contentType,
        };

        // Disable payload signing for HTTPS endpoints (R2, AWS)
to avoid streaming-sha256 path
        if (!string.IsNullOrWhiteSpace(_options.ServiceUrl) &&
            _options.ServiceUrl.StartsWith("https://",
StringComparison.OrdinalIgnoreCase))
        {
            request.DisablePayloadSigning = true;
        }

        try
        {
            await _s3.PutObjectAsync(request, cancellationToken);
        }
        catch (AmazonS3Exception ex)
```

```
        {
            _logger.LogError(ex,
                "Failed to upload {Key} to bucket {Bucket}",
                key, _options.BucketName);
            throw;
        }

        var url = $"{_options.PublicUrlBase.TrimEnd('/')}/{key}";
        _logger.LogInformation("Uploaded {Key} ({ContentType}) to
{Bucket}", key, contentType, _options.BucketName);
        return new StorageUploadResult(key, url);
    }

    public async Task DeleteAsync(string key, CancellationToken
cancellationToken = default)
    {
        if (string.IsNullOrWhiteSpace(key)) return;

        try
        {
            await _s3.DeleteObjectAsync(new DeleteObjectRequest
            {
                BucketName = _options.BucketName,
                Key = key
            }, cancellationToken);
            _logger.LogInformation("Deleted {Key} from {Bucket}",
key, _options.BucketName);
        }
        catch (Exception ex)
        {
            // Best-effort -- deletion must never break the caller
            _logger.LogWarning(ex, "Failed to delete {Key} from
{Bucket}", key, _options.BucketName);
        }
    }
```

```
    private static string GetExtension(string fileName, string
contentType)
    {
        var ext = Path.GetExtension(fileName);
        if (!string.IsNullOrWhiteSpace(ext)) return
ext.ToLowerInvariant();

        return contentType.ToLowerInvariant() switch
        {
            "image/jpeg" => ".jpg",
            "image/png" => ".png",
            "image/webp" => ".webp",
            _ => ".bin"
        };
    }
}
```

Сервіс S3StorageService працює з S3-сумісним API через AWS SDK для .NET, що дозволяє використовувати один код для трьох провайдерів: AWS S3, Cloudflare R2 (продакшен) та LocalStack (розробка). Буферизація потоку в MemoryStream обходить специфіку Cloudflare R2 - він відхиляє chunked transfer encoding, у яке AWS SDK переходить за замовчуванням для не-seekable потоків з IFormFile.

ДОДАТОК Й

Лістинг Й.1. middleware Clerk (proxy.ts)

```
typescript
import { clerkMiddleware, createRouteMatcher } from
"@clerk/nextjs/server";

const isPublicRoute = createRouteMatcher([
  "/",
  "/chat",
  "/events",
  "/events/(.*)",
  "/pricing",
  "/sign-in(.*)",
  "/sign-up(.*)",
  "/sso-callback",
  "/api/webhooks(.*)",
]);

export default clerkMiddleware(async (auth, req) => {
  if (!isPublicRoute(req)) {
    await auth.protect();
  }
});

export const config = {
  matcher: [
    // Skip Next.js internals and all static files, unless found
in search params

"/((?!_next|[^?]*\\.(?:html?|css|js(?!on)|jpe?g|webp|png|gif|svg|t
tf|woff2?|ico|csv|docx?|xlsx?|zip|webmanifest)).*)",
    // Always run for API routes
    "/(api|trpc)(.*)",
```

```
  ],
};
```

Файл proxy.ts (відповідник middleware.ts у Next.js App Router) використовує clerkMiddleware для перевірки автентифікації на всіх непублічних маршрутах. Перелік публічних шляхів задається через createRouteMatcher. Matcher-конфігурація виключає Next.js internals та статичні файли з обробки middleware для оптимізації продуктивності.

Лістинг Й.2. Сторінка створення події (фрагмент)

```
typescript
"use client";

import { useState } from "react";
import { useRouter } from "next/navigation";
import { useUser } from "@clerk/nextjs";
import { useMutation, useQueryClient } from
"@tanstack/react-query";
import { toast } from "sonner";
import { createEvent, updateEvent, uploadEventPhoto } from
"@/lib/api/events";
import { getMySubscription } from "@/lib/api/subscriptions";
import { DatePicker, TimePicker, PhotoUploader } from
"@/components/index";
import { useLanguage } from "@/contexts/LanguageContext";
import { useRole } from "@/hooks/useRole";
import { useQuery } from "@tanstack/react-query";

export default function CreateEventPage() {
  const router = useRouter();
  const queryClient = useQueryClient();
  const { user, isSignedIn } = useUser();
  const { t } = useLanguage();
```

```
  const role = useRole();

  const { data: subscription } = useQuery({
    queryKey: ["my-subscription"],
    queryFn: getMySubscription,
    enabled: isSignedIn && (role === "organizer" || role ===
"admin"),
  });
  const isProActive = subscription?.plan === "pro" &&
subscription?.isActive;
  const isAdmin = role === "admin";
  const showFreeLimits = !isAdmin && !isProActive;
  const FREE_MAX_ATTENDEES = 30;

  const [formData, setFormData] = useState({
    title: "",
    description: "",
    date: "",
    startTime: "",
    location: "",
    ticketPrice: "",
    maxAttendees: "",
  });
  const [stagedPhotos, setStagedPhotos] = useState<File[]>([]);
  const [fieldErrors, setFieldErrors] = useState<Record<string,
string>>({});

  const mutation = useMutation({
    mutationFn: async (publish: boolean) => {
      const dateTime = new
Date(`${formData.date}T${formData.startTime}`);
      const price = parseFloat(formData.ticketPrice) || 0;
      const maxAtt = formData.maxAttendees ?
parseInt(formData.maxAttendees, 10) : null;
```

```
      // 1. Create as draft first so we have an event ID to attach
photos to
      const created = await createEvent({
        title: formData.title,
        description: formData.description,
        eventDateTime: dateTime.toISOString(),
        location: formData.location || undefined,
        ticketPrice: price,
        maxAttendees: maxAtt,
        publish: false,
      });

      // 2. Upload staged photos
      for (const file of stagedPhotos) {
        await uploadEventPhoto(created.id, file);
      }

      // 3. Publish if requested (full payload required by
UpdateEventCommand validator)
      if (publish) {
        await updateEvent(created.id, {
          title: formData.title,
          description: formData.description,
          eventDateTime: dateTime.toISOString(),
          location: formData.location || undefined,
          ticketPrice: price,
          maxAttendees: maxAtt,
          status: "Published",
        });
      }

      return created;
    },
    onSuccess: (data, publish) => {
      queryClient.invalidateQueries({ queryKey: ["my-events"] });
```

```
      toast.success(publish ? "Event published!" : "Event saved as draft.");
      router.push(`/events/${data.id}/edit`);
    },
  });

  // Form validation + render omitted for brevity -- see full file in repository
  // Full form contains: title, description, date+time pickers, location,
  // ticket price (Pro-only), max attendees (capped at 30 for Free), photo uploader

  return (
    <main className="container mx-auto max-w-3xl py-8">
      {/* JSX form structure -- full version in repository */}
    </main>
  );
}
```

Сторінка створення події використовує useMutation з TanStack Query для оркестрації багатокрокового флоу: створення події як чернетки → послідовне завантаження прив'язаних фотографій у Cloudflare R2 → опційна публікація через окремий UPDATE-запит. Trifaza-pattern необхідна тому, що EventPhoto сутності потребують вже існуючого EventId.

Лістинг Й.3. Хук інтеграції з SignalR - useNotifications

```
typescript
"use client";

import { useEffect, useRef } from "react";
import * as signalR from "@microsoft/signalr";
import { toast } from "sonner";
```

```
interface NotificationPayload {
  type: "registration" | "cancellation" | "checkin" | "payment";
  title: string;
  message: string;
  eventId?: string;
  registrationId?: string;
}

const BACKEND_URL = process.env.NEXT_PUBLIC_BACKEND_URL ??
"http://localhost:5215";

export function useNotifications(
  userId?: string | null,
  role?: string | null,
  getToken?: () => Promise<string | null>,
) {
  const connectionRef = useRef<signalR.HubConnection |
null>(null);

  useEffect(() => {
    if (!userId) return;

    const connection = new signalR.HubConnectionBuilder()
      .withUrl(`${BACKEND_URL}/hubs/notifications`, {
        accessTokenFactory: getToken
          ? () => getToken().then(t => t ?? "")
          : undefined,
      })
      .withAutomaticReconnect()
      .configureLogging(signalR.LogLevel.Warning)
      .build();

    connection.on("ReceiveNotification", (payload:
NotificationPayload) => {
```

```
      switch (payload.type) {
        case "registration":
          toast.success(payload.title, { description:
payload.message });
          break;
        case "cancellation":
          toast.info(payload.title, { description: payload.message
});
          break;
        case "checkin":
          toast.success(payload.title, { description:
payload.message });
          break;
        case "payment":
          toast.success(payload.title, { description:
payload.message });
          break;
        default:
          toast(payload.title, { description: payload.message });
      }
    });

    connection.start().catch((err) =>
      console.warn("[Notifications] Connection failed:", err),
    );

    connectionRef.current = connection;

    return () => {
      connection.stop();
    };
  }, [userId, role, getToken]);
}
```

Хук useNotifications керує SignalR-з'єднанням з NotificationsHub через @microsoft/signalr. Передає JWT-токен Clerk через accessTokenFactory для автентифікації WebSocket-з'єднання. При отриманні події ReceiveNotification показує toast-повідомлення через бібліотеку Sonner, диференціюючи стиль (success/info/default) залежно від типу нотифікації.

Лістинг Й.4. Компонент PhotoUploader з drag-and-drop

```
typescript
"use client";

import { useRef, useState, useCallback } from "react";
import { Upload, X, Image as ImageIcon, Loader2 } from
"lucide-react";
import { toast } from "sonner";
import { uploadEventPhoto, deleteEventPhoto } from
"@/lib/api/events";
import type { EventPhoto } from "@/entities/event/types";
import { ConfirmModal } from "./ConfirmModal";

const MAX_FILE_SIZE_BYTES = 5 * 1024 * 1024;
const ALLOWED_TYPES = ["image/jpeg", "image/png", "image/webp"];
const MAX_PHOTOS = 5;

export interface StagedPhoto {
  file: File;
  previewUrl: string;
}

interface PhotoUploaderProps {
  eventId?: string | null;
  existingPhotos?: EventPhoto[];
  onPhotosChange?: (photos: EventPhoto[]) => void;
  onStagedChange?: (files: File[]) => void;
```

```
  hasError?: boolean;
  disabled?: boolean;
}

export function PhotoUploader({
  eventId,
  existingPhotos = [],
  onPhotosChange,
  onStagedChange,
  hasError,
  disabled,
}: PhotoUploaderProps) {
  const inputRef = useRef<HTMLInputElement>(null);
  const [staged, setStaged] = useState<StagedPhoto[]>([]);
  const [livePhotos, setLivePhotos] =
useState<EventPhoto[]>(existingPhotos);
  const [uploading, setUploading] = useState(false);
  const [progress, setProgress] = useState(0);
  const [deleteTarget, setDeleteTarget] = useState<EventPhoto |
null>(null);

  const isLive = !!eventId;
  const photoCount = isLive ? livePhotos.length : staged.length;
  const atLimit = photoCount >= MAX_PHOTOS;

  const validate = (file: File): string | null => {
    if (!ALLOWED_TYPES.includes(file.type))
      return `"${file.name}": only JPEG, PNG, and WebP images are
allowed.`;
    if (file.size > MAX_FILE_SIZE_BYTES)
      return `"${file.name}": exceeds the 5 MB size limit.`;
    return null;
  };

  const handleFiles = useCallback(
```

```
    async (fileList: FileList | File[]) => {
      const files = Array.from(fileList);
      const accepted: File[] = [];
      for (const f of files) {
        if (photoCount + accepted.length >= MAX_PHOTOS) {
          toast.error(`Maximum ${MAX_PHOTOS} photos per event.`);
          break;
        }
        const err = validate(f);
        if (err) { toast.error(err); continue; }
        accepted.push(f);
      }
      if (accepted.length === 0) return;

      if (!isLive) {
        // Staged mode -- collect files locally until parent
uploads them
        const newStaged: StagedPhoto[] = accepted.map((file) => ({
          file,
          previewUrl: URL.createObjectURL(file),
        }));
        const merged = [...staged, ...newStaged];
        setStaged(merged);
        onStagedChange?.(merged.map((s) => s.file));
        return;
      }

      // Live mode -- upload immediately
      setUploading(true);
      setProgress(0);
      try {
        const uploaded: EventPhoto[] = [];
        for (const file of accepted) {
          const result = await uploadEventPhoto(eventId!, file,
(p) => setProgress(p));
```

```
          uploaded.push({
            id: result.id,
            imageUrl: result.imageUrl,
            imageKey: result.imageKey,
            order: result.order,
          });
        }
        const merged = [...livePhotos, ...uploaded];
        setLivePhotos(merged);
        onPhotosChange?.(merged);
        toast.success(`${uploaded.length} photo${uploaded.length >
1 ? "s" : ""} uploaded.`);
      } catch {
        // Global axios interceptor shows error toast
      } finally {
        setUploading(false);
        setProgress(0);
      }
    },
    [isLive, eventId, livePhotos, staged, photoCount,
onPhotosChange, onStagedChange],
  );

  const removeStaged = (index: number) => {
    URL.revokeObjectURL(staged[index].previewUrl);
    const next = staged.filter((_, i) => i !== index);
    setStaged(next);
    onStagedChange?.(next.map((s) => s.file));
  };

  const confirmDeleteLive = async () => {
    if (!deleteTarget || !eventId) return;
    try {
      await deleteEventPhoto(eventId, deleteTarget.id);
      const merged = livePhotos.filter((p) => p.id !==
```

```
deleteTarget.id);
      setLivePhotos(merged);
      onPhotosChange?.(merged);
      toast.success("Photo deleted.");
    } catch { /* handled globally */ }
    finally { setDeleteTarget(null); }
  };

  const onDrop = (e: React.DragEvent<HTMLDivElement>) => {
    e.preventDefault();
    if (disabled || atLimit) return;
    if (e.dataTransfer.files?.length)
handleFiles(e.dataTransfer.files);
  };

  return (
    <div className="space-y-3">
      <div
        onDrop={onDrop}
        onDragOver={(e) => e.preventDefault()}
        className={`flex flex-col items-center justify-center
gap-2 rounded-xl border-2 border-dashed px-6 py-8 ${
          hasError ? "border-red-500 bg-red-500/5"
          : atLimit ? "border-border bg-muted/20 opacity-60"
          : "border-border bg-card hover:border-primary
hover:bg-primary/5 cursor-pointer"
        }`}
        onClick={() => !disabled && !atLimit &&
inputRef.current?.click()}
        role="button"
        tabIndex={0}
      >
        {uploading ? (
          <>
            <Loader2 className="h-8 w-8 animate-spin text-primary"
```

```
/>
            <p className="text-sm font-medium">Uploading...
{progress}%</p>
          </>
        ) : (
          <>
            <Upload className="h-8 w-8 text-muted-foreground" />
            <p className="text-sm font-medium">
              {atLimit ? `Maximum ${MAX_PHOTOS} photos reached` :
"Drop photos here or click to upload"}
            </p>
            <p className="text-xs text-muted-foreground">
              JPEG, PNG, or WebP · max 5 MB · up to {MAX_PHOTOS}
photos
            </p>
          </>
        )}
        <input
          ref={inputRef}
          type="file"
          accept={ALLOWED_TYPES.join(",")}
          multiple
          className="hidden"
          disabled={disabled || atLimit || uploading}
          onChange={(e) => {
            if (e.target.files?.length)
handleFiles(e.target.files);
            e.target.value = "";
          }}
        />
      </div>

      {/* Thumbnails grid + ConfirmModal -- full version in
repository */}
    </div>
```

```
  );
}
```

Компонент PhotoUploader підтримує два режими роботи: staged (для форми створення події, де eventId ще немає - файли зберігаються локально через URL.createObjectURL) та live (для редагування існуючої події - файли завантажуються одразу через uploadEventPhoto). Drag-and-drop інтерфейс реалізовано через нативні DOM-події onDrop і onDragOver.

ДОДАТОК К

Лістинг К.1. Базовий клас EventFlowWebFactory для integration-тестів

```
using System.Runtime.CompilerServices;
using System.Security.Claims;
using System.Text.Encodings.Web;
using EventFlow.Application.Common.Interfaces.Services;
using EventFlow.Infrastructure.Persistence;
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Mvc.Testing;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;
using Npgsql;
using Testcontainers.PostgreSql;

namespace Api.Tests.Integration.Infrastructure;

/// <summary>
/// Custom WebApplicationFactory that spins up a PostgreSQL
Testcontainer
/// and overrides the application's DB connection for isolated
integration tests.
/// </summary>
public class EventFlowWebFactory : WebApplicationFactory<Program>,
IAsyncLifetime
{
    private readonly PostgreSqlContainer _postgres = new
PostgreSqlBuilder()
        .WithImage("postgres:16-alpine")
```

```
        .WithDatabase("eventflow_test")
        .WithUsername("test")
        .WithPassword("test")
        .Build();

    private string _connectionString = string.Empty;
    public string ConnectionString => _connectionString;

    protected override void ConfigureWebHost(IWebHostBuilder
builder)
    {
        // Development environment triggers seed data
        builder.UseEnvironment("Development");

        builder.ConfigureTestServices(services =>
        {
            // Remove existing DbContext options (real connection
string)
            var descriptor = services.SingleOrDefault(
                d => d.ServiceType ==
typeof(DbContextOptions<AppDbContext>));
            if (descriptor != null) services.Remove(descriptor);

            // Re-register AppDbContext pointing at the
Testcontainer database
            var dataSourceBuilder = new
NpgsqlDataSourceBuilder(_connectionString);
            dataSourceBuilder.EnableDynamicJson();
            var dataSource = dataSourceBuilder.Build();

            services.AddDbContext<AppDbContext>(options =>

options.UseNpgsql(dataSource).UseSnakeCaseNamingConvention());

            // Replace AI / SignalR / Storage with no-op test
```

```
doubles
            ReplaceService<IAiChatService,
NoOpAiChatService>(services);
            ReplaceService<INotificationService,
NoOpNotificationService>(services);
            ReplaceService<IStorageService,
InMemoryStorageService>(services);

            // Test auth scheme -- always-succeeding handler
            services.AddAuthentication("Test")
                .AddScheme<AuthenticationSchemeOptions,
TestAuthHandler>("Test", _ => { });
            services.AddAuthorization(options =>
            {
                options.DefaultPolicy = new
AuthorizationPolicyBuilder("Test")
                    .RequireAssertion(_ => true)
                    .Build();
                options.FallbackPolicy = null;
            });
        })
        .ConfigureAppConfiguration((_, config) =>
        {
            config.AddJsonFile("appsettings.Test.json", optional:
false, reloadOnChange: true)
                  .AddEnvironmentVariables();
        });
    }

    private static void ReplaceService<TService,
TImpl>(IServiceCollection services)
        where TService : class where TImpl : class, TService
    {
        var descriptor = services.SingleOrDefault(d =>
d.ServiceType == typeof(TService));
```

```
        if (descriptor is not null) services.Remove(descriptor);
        services.AddScoped<TService, TImpl>();
    }

    public async Task InitializeAsync()
    {
        await _postgres.StartAsync();
        _connectionString = _postgres.GetConnectionString();
    }

    public new async Task DisposeAsync()
    {
        await _postgres.DisposeAsync();
        await base.DisposeAsync();
    }
}

/// <summary>Fake auth handler that always succeeds.</summary>
file sealed class TestAuthHandler(
    IOptionsMonitor<AuthenticationSchemeOptions> options,
    ILoggerFactory logger,
    UrlEncoder encoder)
    : AuthenticationHandler<AuthenticationSchemeOptions>(options,
logger, encoder)
{
    protected override Task<AuthenticateResult>
HandleAuthenticateAsync()
    {
        var identity = new ClaimsIdentity("Test");
        var principal = new ClaimsPrincipal(identity);
        var ticket = new AuthenticationTicket(principal, "Test");
        return
Task.FromResult(AuthenticateResult.Success(ticket));
    }
}
```

Клас EventFlowWebFactory наслідує WebApplicationFactory<Program> і піднімає реальну PostgreSQL у Docker-контейнері черезTestcontainers перед кожним прогоном тестів. У ConfigureWebHost зовнішні залежності (AI-сервіс, SignalR, S3) замінюються на no-op double-и, а автентифікація - на завжди-успішний TestAuthHandler, що дозволяє тестам перевіряти повний HTTP-flow без реальних зовнішніх API.

Лістинг К.2. Приклад unit-тесту плагіна (EventSearchPluginTests)

```
using EventFlow.Application.Common.Interfaces.Queries;
using EventFlow.Domain.Entities;
using EventFlow.Domain.ValueObjects;
using EventFlow.Infrastructure.AI;
using EventFlow.Infrastructure.AI.Plugins;

namespace Application.UnitTests.AI;

public class EventSearchPluginTests
{
    private readonly IEventQueries _eventQueries =
Substitute.For<IEventQueries>();
    private readonly EventSearchPlugin _sut;

    public EventSearchPluginTests()
    {
        _sut = new EventSearchPlugin(_eventQueries, new
AiResponseDataCollector());
    }

    private static Event CreateTestEvent(string title = "Test
Event", string? location = "Kyiv")
    {
        var organizerId = UserId.New();
        return Event.Create(title, "Test description",
            DateTime.UtcNow.AddDays(7), organizerId, location,
```

```
100m, 50);
    }

    [Fact]
    public async Task
SearchEvents_WithResults_ReturnsFormattedList()
    {
        // Arrange
        var evt = CreateTestEvent("AI Conference");
        _eventQueries.SearchEventsAsync(
                Arg.Any<string?>(), Arg.Any<string?>(),
                Arg.Any<DateTime?>(), Arg.Any<DateTime?>(),
                Arg.Any<string?>(), Arg.Any<int>(),
Arg.Any<int>(),
                Arg.Any<CancellationToken>())
            .Returns(new List<Event> { evt }.AsReadOnly());

        // Act
        var result = await _sut.SearchEventsAsync(searchTerm:
"AI");

        // Assert
        result.ShouldContain("displayed as visual cards");
    }

    [Fact]
    public async Task
SearchEvents_NoResults_ReturnsNoEventsMessage()
    {
        // Arrange
        _eventQueries.SearchEventsAsync(
                Arg.Any<string?>(), Arg.Any<string?>(),
                Arg.Any<DateTime?>(), Arg.Any<DateTime?>(),
                Arg.Any<string?>(), Arg.Any<int>(),
Arg.Any<int>(),
```

```
                Arg.Any<CancellationToken>())
            .Returns(new List<Event>().AsReadOnly());

        // Act
        var result = await _sut.SearchEventsAsync(searchTerm:
"nonexistent");

        // Assert
        result.ShouldContain("No events found");
    }

    [Fact]
    public async Task SearchEvents_WithDateRange_ParsesDates()
    {
        // Arrange
        _eventQueries.SearchEventsAsync(
                Arg.Any<string?>(), Arg.Any<string?>(),
                Arg.Any<DateTime?>(), Arg.Any<DateTime?>(),
                Arg.Any<string?>(), Arg.Any<int>(),
Arg.Any<int>(),
                Arg.Any<CancellationToken>())
            .Returns(new List<Event>().AsReadOnly());

        // Act
        await _sut.SearchEventsAsync(startDate: "2026-01-01",
endDate: "2026-12-31");

        // Assert
        await _eventQueries.Received(1).SearchEventsAsync(
            Arg.Any<string?>(), Arg.Any<string?>(),
            Arg.Is<DateTime?>(d => d.HasValue),
            Arg.Is<DateTime?>(d => d.HasValue),
            Arg.Any<string?>(), Arg.Any<int>(), Arg.Any<int>(),
            Arg.Any<CancellationToken>());
    }
```

```
}
```

Unit-тест EventSearchPluginTests ілюструє pyramid-of-testing підхід: залежності (IEventQueries) замокані через NSubstitute, тестується тільки логіка плагіна без БД та LLM. Assertions виконуються через Shouldly для виразного синтаксису. Кожен тест перевіряє один аспект поведінки (з результатами, без результатів, з датою) - атомарність дозволяє швидко локалізувати причину помилки.

Лістинг К.3. Приклад integration-тесту реєстрації на подію

```
using System.Net;
using Api.Tests.Integration.Infrastructure;
using EventFlow.API.DTOs.Events;
using EventFlow.Application.Events.Commands.RegisterForEvent;

namespace Api.Tests.Integration.Events;

/// <summary>
/// Integration tests for event registration and cancellation
endpoints.
/// Each test method that modifies data cleans up via
CleanupTestDataAsync().
/// </summary>
public class RegisterForEventTests(EventFlowWebFactory factory)
    : IntegrationTestBase(factory), IAsyncLifetime
{
    private string _attendeeClerkId = string.Empty;
    private Guid _eventId = Guid.Empty;

    public async Task InitializeAsync()
    {
        // Seed a fresh attendee user and a published event before
each test class run
        _attendeeClerkId = await SyncUserAsync(firstName:
```

```
"Attendee", lastName: "Tester");
        (_, _eventId) = await
SeedPublishedEventAsync("Registration Test Event");
    }

    public async Task DisposeAsync()
    {
        await CleanupTestDataAsync();
    }

    [Fact]
    public async Task
RegisterForEvent_Returns200_WithSuccessMessage()
    {
        // Arrange
        Client.DefaultRequestHeaders.Remove("X-User-Id");
        Client.DefaultRequestHeaders.Add("X-User-Id",
_attendeeClerkId);

        // Act
        var response = await
Client.PostAsync($"/api/events/{_eventId}/register", null);

        // Assert
        response.StatusCode.Should().Be(HttpStatusCode.OK);

        var body = await
response.ToResponseModel<RegistrationResponse>();
        body.Success.Should().BeTrue();
        body.Message.Should().Contain("registered");
    }

    [Fact]
    public async Task
RegisterForEvent_Returns400_WhenNoUserHeader()
```

```
    {
        // Arrange -- no X-User-Id header
        Client.DefaultRequestHeaders.Remove("X-User-Id");

        // Act
        var response = await
Client.PostAsync($"/api/events/{_eventId}/register", null);

        // Assert

response.StatusCode.Should().Be(HttpStatusCode.BadRequest);
    }

    [Fact]
    public async Task
RegisterForEvent_Returns200_AlreadyRegistered_WhenCalledTwice()
    {
        // Arrange -- sync a unique user for this specific test
        var uniqueClerkId = await SyncUserAsync(firstName:
"Double", lastName: "Register");
        Client.DefaultRequestHeaders.Remove("X-User-Id");
        Client.DefaultRequestHeaders.Add("X-User-Id",
uniqueClerkId);

        // Act -- register twice
        var first = await
Client.PostAsync($"/api/events/{_eventId}/register", null);
        var second = await
Client.PostAsync($"/api/events/{_eventId}/register", null);

        // Assert -- both succeed (second is idempotent and
reports "already registered")
        first.StatusCode.Should().Be(HttpStatusCode.OK);
        second.StatusCode.Should().Be(HttpStatusCode.OK);
```

```
        var secondBody = await
second.ToResponseModel<RegistrationResponse>();
        secondBody.Message.Should().ContainAny("already",
"Already");
    }
}
```

Integration-тест RegisterForEventTests запускається через EventFlowWebFactory і виконує реальні HTTP-запити до власного інстансу застосунку з реальною PostgreSQL. Метод InitializeAsync готує тестові дані (синхронізує користувача, створює опубліковану подію), DisposeAsync гарантує очищення після класу. Тести перевіряють як happy path (успішна реєстрація), так і error cases (відсутній заголовок, повторна реєстрація).