

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ**

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Цифрова платформа бронювання коворкінг-просторів з урахуванням інтересів користувачів та підтримкою командної взаємодії на основі JavaScript»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Технології цифрового розвитку»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Дмитро КОЛПАКОВ
(підпис)

Виконав: здобувач вищої освіти групи ТЦР-41

_____ Дмитро КОЛПАКОВ

Керівник: _____ Вікторія ЖЕБКА
д-р техн. наук, проф.

Рецензент: _____

Київ 2026

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ

Завідувач кафедри

Технологій цифрового розвитку

_____ Вікторія ЖЕБКА

« _____ » _____ 2026 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

_____ Колпакову Дмитру Олеговичу

1. Тема кваліфікаційної роботи: «Цифрова платформа бронювання коворкінг-просторів з урахуванням інтересів користувачів та підтримкою командної взаємодії на основі JavaScript» керівник кваліфікаційної роботи професор Вікторія ЖЕБКА, затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 р. № 51.

2. Строк подання кваліфікаційної роботи «18» травня 2026 р.

3. Вихідні дані до кваліфікаційної роботи: офіційна документація JavaScript/TypeScript, Node.js, Express, React, Vite, Tailwind CSS, TanStack Query, Socket.IO, PostgreSQL, Prisma ORM, Redis, JWT, Swagger/OpenAPI, Docker, GitHub Actions, Jest, Playwright, i18next, Leaflet, FullCalendar; матеріали з проєктування інформаційних систем онлайн-бронювання; наукові джерела з алгоритмів рекомендацій (interest-based matching, collaborative filtering), командної взаємодії та забезпечення безпеки веб-застосунків.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

4.1. Аналіз предметної галузі бронювання коворкінг-просторів, командної роботи та існуючих платформ-аналогів.

4.2. Аналіз і обґрунтування засобів реалізації та архітектури системи (клієнт-серверна монорепо-структура, модель даних, ролі доступу, API, інтеграції).

4.3. Програмна реалізація серверної частини і веб-інтерфейсу: автентифікація та 2FA, каталог і пошук коворкінгів, бронювання зон, рекомендації за інтересами, команди/інвайти/чат у реальному часі, сповіщення, модулі для ролей OWNER/MANAGER/RECEPTION/ADMIN.

4.4. Реалізація інтерфейсу користувача, тестування та валідація результатів (unit/e2e), оцінка працездатності, безпеки та якості розробленого програмного забезпечення.

5. Перелік графічного матеріалу: презентація

5.1. Аналіз аналогів платформ бронювання коворкінгів.

5.2. Функціональні та нефункціональні вимоги до системи.

5.3. Обраний технологічний стек і обґрунтування вибору.

5.4. Архітектура системи (client/server, БД, Redis, Socket.IO, Docker-інфраструктура).

5.5. Діаграма варіантів використання (користувач платформи).

5.6. Діаграма варіантів використання (owner/manager/reception/admin).

5.7. Діаграма структури бази даних (ER-діаграма).

5.8. Діаграма бізнес-процесів: життєвий цикл бронювання, запрошення в команду, check-in/check-out.

5.9. Екранні форми веб-платформи (каталог, мапа, бронювання, команди, кабінети ролей).

5.10. Результати тестування, апробації та демонстрації роботи системи.

6. Дата видачі завдання «20» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір і аналіз науково-технічної літератури за темою бронювання коворкінгів, рекомендаційних систем і командної взаємодії	20.02–25.02.2026	
2	Аналіз існуючих платформ-аналогів та формування вимог до веб-платформи	26.02–05.03.2026	
3	Постановка задачі, формування функціональних і нефункціональних вимог, сценаріїв використання	06.03–15.03.2026	
4	Проектування архітектури системи (client/server), моделі ролей і структури API	16.03–25.03.2026	
5	Проектування бази даних PostgreSQL (Prisma), підготовка	26.03–02.04.2026	

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
	міграцій і seed-даних		
6	Реалізація серверної частини (Node.js, Express): auth, users, coworkings, zones, RBAC	03.04–10.04.2026	
7	Реалізація бізнес-логіки: бронювання, команди, інвайти, чат Socket.IO, рекомендації за інтересами	11.04–18.04.2026	
8	Реалізація клієнтської частини (React, Vite, Tailwind): каталог, мапа, бронювання, профіль, команди	19.04–25.04.2026	
9	Інтеграція додаткових модулів: сповіщення, 2FA, PDF/QR квитки, PWA, мультимовність	26.04–01.05.2026	
10	Тестування та налагодження (Jest, Playwright), оптимізація і виправлення помилок	02.05–10.05.2026	
11	Оформлення розрахунково-пояснювальної записки та підготовка графічного матеріалу	11.05–16.05.2026	
12	Підготовка доповіді, презентації, попередній захист і подання кваліфікаційної роботи	17.05–20.05.2026	

Здобувачка вищої освіти

(підпис)

Дмитро КОЛПАКОВ

Керівник

кваліфікаційної роботи

(підпис)

Вікторія ЖЕБКА

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 103 стор., 5 табл., 37 рис., 32 джерела.

Мета роботи – підвищення ефективності процесів пошуку, бронювання коворкінг-просторів та організації командної взаємодії шляхом розробки веб-платформи з персоналізованим підбором просторів за інтересами користувачів на основі JavaScript-екосистеми.

Об'єкт дослідження – процес організації роботи користувачів у коворкінг-просторах та пов'язані з ним процеси пошуку, бронювання робочих місць і командної взаємодії в онлайн-середовищі.

Предмет дослідження – моделі, методи, алгоритми та програмні засоби побудови веб-платформи бронювання коворкінг-просторів за інтересами з підтримкою командної роботи на основі екосистеми JavaScript.

Короткий зміст роботи: у кваліфікаційній роботі проаналізовано предметну галузь бронювання коворкінг-просторів, досліджено особливості сучасного ринку гнучких робочих просторів та розглянуто існуючі аналоги, зокрема Creative States, Platforma, Kooperativ і KONTORA Lviv. У результаті аналізу встановлено, що більшість наявних рішень орієнтовані переважно на презентацію простору або базове бронювання і не забезпечують персоналізованого підбору за інтересами, повноцінної командної взаємодії, внутрішнього чату та розподілу ролей між учасниками.

У роботі обґрунтовано вибір технологічного стеку на базі JavaScript/TypeScript-екосистеми. Спроектовано клієнт-серверну архітектуру системи, яка включає серверну частину на Node.js та Express, клієнтську частину на React і TypeScript, реляційну базу даних PostgreSQL, Prisma ORM для доступу до даних, Redis для кешування та rate limiting, Socket.IO для комунікації в реальному часі, а також Docker Compose для контейнеризованого розгортання. Реалізовано підсистеми автентифікації та двофакторного захисту, REST API, рольову модель доступу, каталог коворкінгів, управління зонами, модуль бронювання, систему рекомендацій за інтересами, команди, запрошення, чат,

сповіщення, waitlist, recurring bookings, відгуки, обране, push-підписки та аудит дій користувачів.

Окрему увагу приділено структурі бази даних. У роботі спроектовано реляційну модель, яка охоплює сутності користувачів, профілів, коворкінгів, зон, слотів доступності, бронювань, команд, учасників команд, повідомлень, сповіщень, платежів, правил ціноутворення, waitlist, повторюваних бронювань, категорій інтересів, журналу аудиту та інших допоміжних сутностей. Така структура забезпечує цілісність даних, можливість масштабування та підтримку складних користувацьких сценаріїв.

У процесі розробки реалізовано кабінети для різних ролей системи: звичайного користувача, власника коворкінгу, менеджера, працівника рецепції та адміністратора. Платформа підтримує автоматизований цикл бронювання, оновлення статусів у реальному часі, командний чат, email- та внутрішні сповіщення, а також рекомендаційний механізм на основі інтересів користувача та тегів коворкінг-просторів. Для перевірки працездатності розробленого програмного забезпечення проведено unit- та E2E-тестування, а також оцінку відповідності системи функціональним вимогам. Результати роботи апробовані у вигляді тез доповідей на двох науково-технічних конференціях.

КЛЮЧОВІ СЛОВА: ВЕБ-ПЛАТФОРМА, КОВОРКІНГ, БРОНЮВАННЯ, КОМАНДНА РОБОТА, TYPESCRIPT, JAVASCRIPT, NODE.JS, EXPRESS, REACT, POSTGRESQL, PRISMA ORM, SOCKET.IO, REDIS, JWT.

ЗМІСТ

ВСТУП	10
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ	13
1.1 Характеристика предметної галузі	13
1.2 Аналіз існуючих аналогів.....	14
1.2.1 Аналіз платформи Creative States	15
1.2.2 Аналіз платформи Platforma.....	16
1.2.3 Аналіз платформи Kooperativ.....	18
1.2.4 Аналіз платформи KONTORA Lviv	19
1.3 Визначення проблем предметної області.....	21
1.4 Постановка завдання на розробку системи.....	23
1.5 Визначення вимог до застосунку	24
2 АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ ТА АРХІТЕКТУРИ СИСТЕМИ	29
2.1 Вибір середовища розробки та організація проєкту	29
2.2 Вибір мови програмування та обґрунтування технологічного стеку.....	35
2.3 Архітектура клієнт-серверної взаємодії та організація API.....	39
2.4 Реалізація серверної частини та вибір фреймворку	42
2.5 Реалізація клієнтської частини та вибір бібліотек інтерфейсу	46
2.6 Реалізація безпеки системи.....	49
3 ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ПЛАТФОРМИ	52
3.1 Загальна архітектура системи та структура бази даних	52
3.2 Реалізація модуля автентифікації та двофакторного захисту	55
3.3 Реалізація каталогу, пошуку та алгоритму рекомендацій за інтересами	61
3.4 Реалізація модуля бронювання зон та управління доступністю.....	66
3.5 Реалізація командної взаємодії, кабінетів ролей та системи сповіщень.....	72
4 ТЕСТУВАННЯ ТА ОЦІНКА ЯКОСТІ РОЗРОБЛЕНОЇ СИСТЕМИ	77
4.1 Результати unit-тестування та E2E-тестування	77
4.2 Оцінка відповідності функціональним вимогам	79
4.3 Порівняльний аналіз розробленої платформи з існуючими аналогами.....	81
ВИСНОВКИ.....	85
ПЕРЕЛІК ПОСИЛАНЬ	88
ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ	91
ДОДАТОК Б ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ.....	98

ВСТУП

Актуальність дослідження визначається тим, що в умовах активного поширення гібридної та дистанційної роботи коворкінг-простори стають дедалі більш затребуваними серед фахівців, команд і компаній.

За оцінками аналітиків, світовий ринок коворкінгів щорічно збільшується більш ніж на 15%, а кількість людей, що обирають гнучкі формати зайнятості, до 2025 року перевищила 80 мільйонів осіб [1]. В Україні ця тенденція також набирає обертів: у великих містах активно з'являються нові коворкінг-простори, орієнтовані на різні аудиторії – від фрілансерів та стартапів до корпоративних команд. Проте більшість існуючих рішень для бронювання таких просторів мають суттєві обмеження: відсутність інтелектуальних механізмів підбору місця за інтересами користувача, слабка підтримка командної роботи в режимі реального часу, відсутність єдиного інструменту для управління різними ролями учасників (власник, менеджер, адміністратор, відвідувач).

Глобальна тенденція до цифровізації робочих просторів демонструє ефективність впровадження спеціалізованих платформ бронювання, що інтегрують функції командної взаємодії та персоналізованих рекомендацій [3; 4]. Дослідження показують, що автоматизація процесів бронювання скорочує адміністративне навантаження на 30–50% та підвищує завантаженість приміщень на 20–35% [1]. Наявні комерційні рішення (Creative States, Platforma, Kooperativ, KONTORA Lviv) орієнтовані переважно на зарубіжні ринки, не адаптовані до специфіки українського користувача, мають закритий вихідний код та значну вартість ліцензій, що унеможливорює їхнє використання для малого бізнесу [2].

Об'єкт дослідження – процес організації роботи користувачів у коворкінг-просторах та пов'язані з ним процеси пошуку, бронювання робочих місць і командної взаємодії в онлайн-середовищі.

Предмет дослідження – моделі, методи, алгоритми та програмні засоби побудови веб-платформи бронювання коворкінг-просторів за інтересами з підтримкою командної роботи на основі екосистеми JavaScript.

Мета роботи – підвищення ефективності процесів пошуку, бронювання коворкінг-просторів та організації командної взаємодії шляхом розробки веб-платформи з персоналізованим підбором просторів за інтересами користувачів на основі JavaScript-екосистеми.

Для досягнення поставленої мети визначено наступні задачі:

1. Провести аналіз предметної галузі та існуючих платформ бронювання коворкінг-просторів;
2. Обґрунтувати вибір технологічного стеку та спроектувати архітектуру системи;
3. Реалізувати серверну частину з JWT-автентифікацією, RBAC та REST API;
4. Розробити веб-інтерфейс з персональними кабінетами для чотирьох ролей: власника, менеджера, адміністратора та користувача;
5. Впровадити модуль рекомендацій за інтересами та функціонал командної взаємодії в режимі реального часу;
6. Провести тестування та оцінити ефективність розробленого рішення.

Методи дослідження: аналіз наукових джерел у галузі розробки веб-застосунків та систем бронювання [1–5]; порівняльний аналіз існуючих платформ управління коворкінгами; методи системного аналізу та об'єктно-орієнтованого проєктування; практичні методи розробки та тестування клієнт-серверних веб-застосунків (unit-тестування, e2e-тестування).

Наукова новизна полягає у розробці єдиної JavaScript/TypeScript-екосистеми для веб-платформи коворкінгу з інтегрованим модулем підбору робочих місць за інтересами користувача та командної взаємодії в реальному часі, що відсутній у відкритих рішеннях на українському ринку.

Практична значущість – self-hosted рішення на базі Docker Compose з відкритим вихідним кодом, що скорочує час оформлення бронювання та підвищує завантаженість коворкінг-простору за рахунок інтелектуальних рекомендацій і зручного командного функціоналу [5].

Апробація результатів та публікації. Основні положення кваліфікаційної роботи пройшли апробацію на двох конференціях, організованих ДУІКТ у Києві в 2026 році — Всеукраїнській науково-технічній конференції «Технології цифрового розвитку: програмні системи та децентралізація» та VII Всеукраїнській науково-технічній конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Матеріали досліджень опубліковано у вигляді двох тез доповідей [31, 32].

1. Колпаков Д. О., Жебка В. В. Цифрова платформа бронювання коворкінг-просторів з урахуванням інтересів користувачів та підтримкою командної взаємодії на основі JavaScript // Матеріали Всеукраїнської науково-технічної конференції «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. К. : ДУІКТ, 2026.
2. Колпаков Д. О., Жебка В. В. Цифрова платформа бронювання коворкінг-просторів з урахуванням інтересів користувачів та підтримкою командної взаємодії на основі JavaScript // Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. К. : ДУІКТ, 2026.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Характеристика предметної галузі

Ринок коворкінгів в Україні та світі протягом останнього десятиліття демонструє стаке зростання. Якщо до 2015 року коворкінг-простори сприймалися як нішевий формат для фрілансерів та стартапів, то станом на 2026 рік вони перетворилися на повноцінну альтернативу традиційному офісу для компаній будь-якого розміру [1]. Після пандемії COVID-19 попит на гнучкі робочі простори різко зріс: глобальний ринок коворкінгів оцінюється у понад 13 мільярдів доларів і продовжує зростати [2].

Типовий процес взаємодії між користувачем та коворкінгом складається з кількох етапів. Спочатку людина або команда шукає відповідний простір – враховуючи локацію, ціну, доступну інфраструктуру, атмосферу. Далі здійснюється бронювання конкретної зони (відкритий простір, переговорна кімната, приватний офіс) на певний час або тривалий період. Після бронювання – check-in при прибутті, використання простору, check-out. У разі командної роботи до процесу залучаються кілька учасників, які мають координувати спільне бронювання та комунікувати в рамках однієї задачі.

Кожен із цих етапів потенційно містить проблеми. Пошук відповідного простору є трудомістким – пропозиції розпорошені між різними платформами та сайтами. Підбір простору «за інтересами» (тобто відповідно до сфери діяльності команди чи індивідуальних потреб – ІТ, дизайн, юриспруденція, освіта) на існуючих платформах не підтримується системно. Командне бронювання – ще більш болюча точка: зазвичай одна людина бронює простір, а решта команди не має єдиного інструменту для координації, спільного планування та чату в реальному часі.

Окремою проблемою є відсутність персоналізації. Більшість платформ пропонують лише фільтрацію за базовими параметрами (ціна, локація, тип зони) без урахування специфіки інтересів та роду діяльності користувача. Тим часом

коворкінги, орієнтовані на конкретні спільноти (наприклад, hub для IT-спеціалістів, простір для творчих індустрій або юридичний коворкінг), стають дедалі більш поширеними – і саме для них важлива функціональність підбору «за інтересами» [3].

Предметна галузь даної кваліфікаційної роботи охоплює проєктування та реалізацію веб-платформи, що автоматизує повний цикл бронювання коворкінг-просторів: від пошуку та фільтрації за інтересами до командного бронювання, чату в реальному часі та управління ролями учасників команди.

1.2 Аналіз існуючих аналогів

Для виявлення недоліків існуючих рішень та обґрунтування необхідності розробки власної платформи проаналізовано чотири найбільш показові аналоги українського ринку коворкінгів: Creative States, Platforma, Kooperativ та KONTORA Lviv. Усі чотири платформи є діючими комерційними рішеннями, орієнтованими на різні сегменти аудиторії та географічні ринки в межах України. Аналіз проводився на основі практичного дослідження веб-сайтів, доступного функціоналу онлайн-бронювання, наявності особистих кабінетів, інструментів для командної роботи та можливостей персоналізації відповідно до інтересів користувача.

Creative States – найбільша мережа преміальних бізнес-просторів в Україні, що об'єднує шість київських локацій та простори у Дніпрі. Platforma – мережа київських коворкінгів із трьома локаціями, що робить акцент на спільноті, освітніх заходах та творчому розвитку резидентів. Kooperativ – київський коворкінг, що позиціонує себе як творчу майстерню та каталізатор руху, подій і знайомств, з амфітеатром, терасою та активною спільнотою. KONTORA Lviv – коворкінг у самому центрі Львова з цілодобовим доступом, кімнатами для команд та переговорів, орієнтований на підприємців і спеціалістів різних галузей.

Попри відмінності у позиціонуванні та географії, усі чотири аналоги об'єднує спільна характеристика з точки зору цифрового функціоналу: жоден із них не

реалізує повноцінного підбору просторів за інтересами користувача або команди, жоден не надає інструментів командної роботи на рівні платформи – спільних акаунтів, розподілу ролей, системи запрошень та вбудованого чату в реальному часі. Бронювання у більшості випадків здійснюється через форму заявки або телефон без миттєвого підтвердження, а особисті кабінети або відсутні, або мають обмежений функціонал без управління статусами замовлень.

1.2.1 Аналіз платформи Creative States

Creative States позиціонує себе як найбільшу мережу преміальних бізнес-просторів в Україні, з локаціями у Києві та Дніпрі. Мережа охоплює шість київських локацій із різними форматами: мітинг-руми, скайп-руми, hot desk зони, приватні офіси та event-зали. Для резидентів доступні гнучкі тарифи: від 10 візитів на місяць до повноцінного членства з доступом до всіх сервісів, а також оренда переговорних кімнат для разових зустрічей. Платформа має особистий кабінет учасника та можливість онлайн-бронювання через сайт, проте функціонал суттєво обмежений з точки зору командної роботи та персоналізації.

З точки зору цифрового сервісу підбір простору за інтересами відсутній – пошук обмежений вибором локації та типу зони. Командна робота не реалізована на рівні платформи: немає спільних акаунтів команди, ролей, системи запрошень або чату. Бронювання здійснюється через заявку або телефон, без миттєвого підтвердження в реальному часі для зовнішніх користувачів. Профіль резидента не містить полів для вказання інтересів або сфери діяльності, що унеможливорює будь-яку персоналізацію пропозицій.

Серед переваг Creative States варто виділити найбільшу мережу в Україні, розвинену інфраструктуру преміум-класу, власну кухню на кожній локації, наявність укриттів та безперебійного живлення. Серед недоліків – відсутність підбору за інтересами, командного функціоналу, чату між учасниками та персоналізації пошуку.

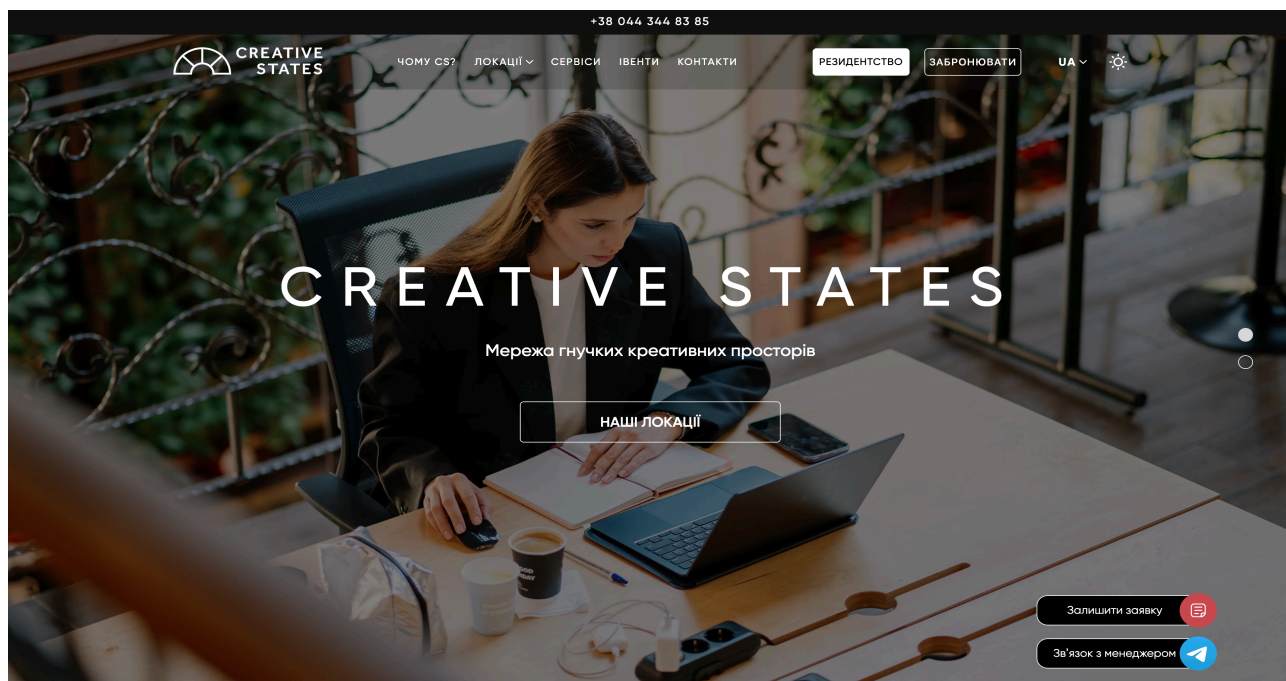


Рис. 1.1 Головна сторінка сайту Creative States

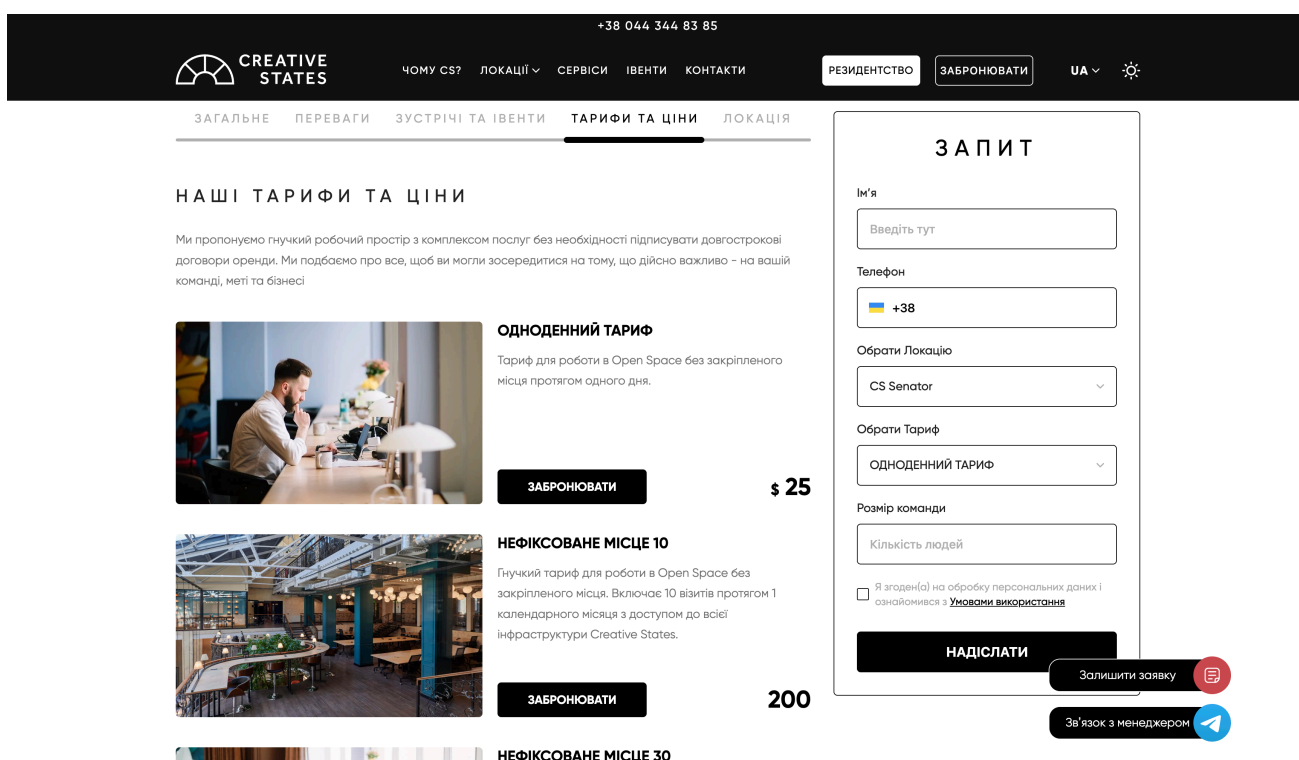


Рис. 1.2 Сторінка вибору локації та тарифів Creative States

1.2.2 Аналіз платформи Platforma

Platforma – київська мережа коворкінгів, що позиціонує себе як простір для роботи, ідей, розвитку та спільноти, де регулярно проводяться лекції, заходи та освітні програми. Мережа налічує три локації у Києві: Leonardo на вулиці Богдана

Хмельницького, Art-Zavod на лівому березі та Korolenkivska поблизу станції метро «Олімпійська». Платформа пропонує сучасні робочі місця, приватні офіси для команд різного розміру та lounge-зони для неформальних зустрічей.

Сайт платформи містить інформацію про тарифи та форму для зв'язку з менеджером, однак повноцінне онлайн-бронювання в реальному часі відсутнє: бронювання здійснюється через заявку з подальшим зворотним зв'язком від адміністратора. Акцент зроблено на спільноту та атмосферу, проте цифрові інструменти для командної взаємодії не реалізовані. Підбір за інтересами, спільне бронювання, розподіл ролей у команді та чат між учасниками відсутні. Особистий кабінет з управління бронюваннями у публічному доступі не представлений.

Серед переваг Platforma – сильна спільнота, розвинений освітній напрям, кілька зручних локацій у різних районах Києва, активне проведення заходів для резидентів. Серед недоліків – відсутність онлайн-бронювання в реальному часі, командного функціоналу, особистого кабінету з повноцінним управлінням та персоналізації за інтересами.

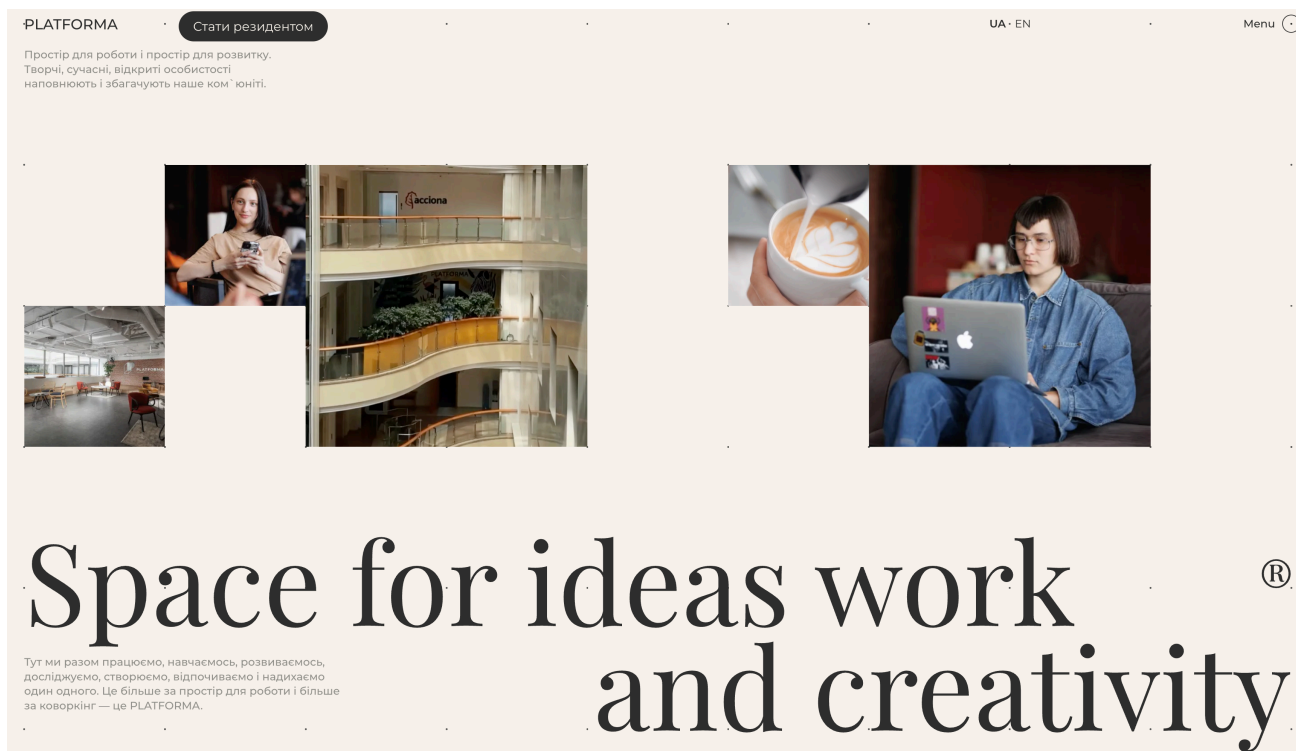


Рис. 1.3 Головна сторінка сайту coworkingplatforma

PLATFORMA

Стати резидентом

UA EN

Menu

Ім'я та прізвище

Чим займаєтесь

Email

Телефон

☐ Ознайомлений з договором оферти та політикою конфіденційності

ВІДПРАВИТИ

music

Рис. 1.4 Форма звернення для бронювання Platforma

1.2.3 Аналіз платформи Kooperativ

Kooperativ – київський коворкінг, що позиціонує себе не лише як робочий простір, а і як творча майстерня, місце подій, знайомств, професійної взаємодії та формування спільноти. Такий підхід свідчить про орієнтацію простору на поєднання функцій класичного коворкінгу з елементами культурного й комунікаційного середовища. Простір пропонує різні формати членства: від цілодобового доступу до спільних зон – амфітеатру, лаунжу, кухні та тераси – до оренди індивідуальних офісів для команд. Регулярно проводяться власні заходи: форуми в амфітеатрі, вечірки на терасі, подкасти та клубні зустрічі. Дизайн простору виконано в дусі берлінської функціональності та принципів Bauhaus – мінімум декору, максимум світла та відкритого простору. У результаті формується середовище, яке сприяє не лише виконанню робочих завдань, а й розвитку нетворкінгу, креативної взаємодії та обміну ідеями між резидентами.

Попри сильний акцент на спільноту, цифровий інструментарій залишається мінімальним. Бронювання здійснюється через сайт або телефон, але без особистого кабінету з повноцінним управлінням замовленнями та їхніми статусами. Це

свідчить про те, що цифрова складова сервісу виконує переважно інформаційну, а не операційну функцію. Командної моделі з ролями та системою запрошень, підбору за інтересами та вбудованого чату між учасниками не передбачено. Платформа фактично функціонує як сайт-вітрина з контактними даними та описом тарифів, а не як повноцінний інструмент для управління бронюваннями. Отже, сервіс орієнтований передусім на презентацію простору та залучення потенційних клієнтів, а не на автоматизацію користувацьких сценаріїв взаємодії.

Серед переваг Kooperativ – унікальна творча атмосфера, активна спільнота, цілодобовий доступ, безперебійне живлення та генератор, амфітеатр і тераса як нестандартні робочі зони. Додатковою перевагою є поєднання робочої, подієвої та соціальної функцій у межах одного простору, що робить його привабливим для представників креативних індустрій та командних проєктів. Серед недоліків – відсутність онлайн-системи бронювання з особистим кабінетом, командного функціоналу та підбору за інтересами.

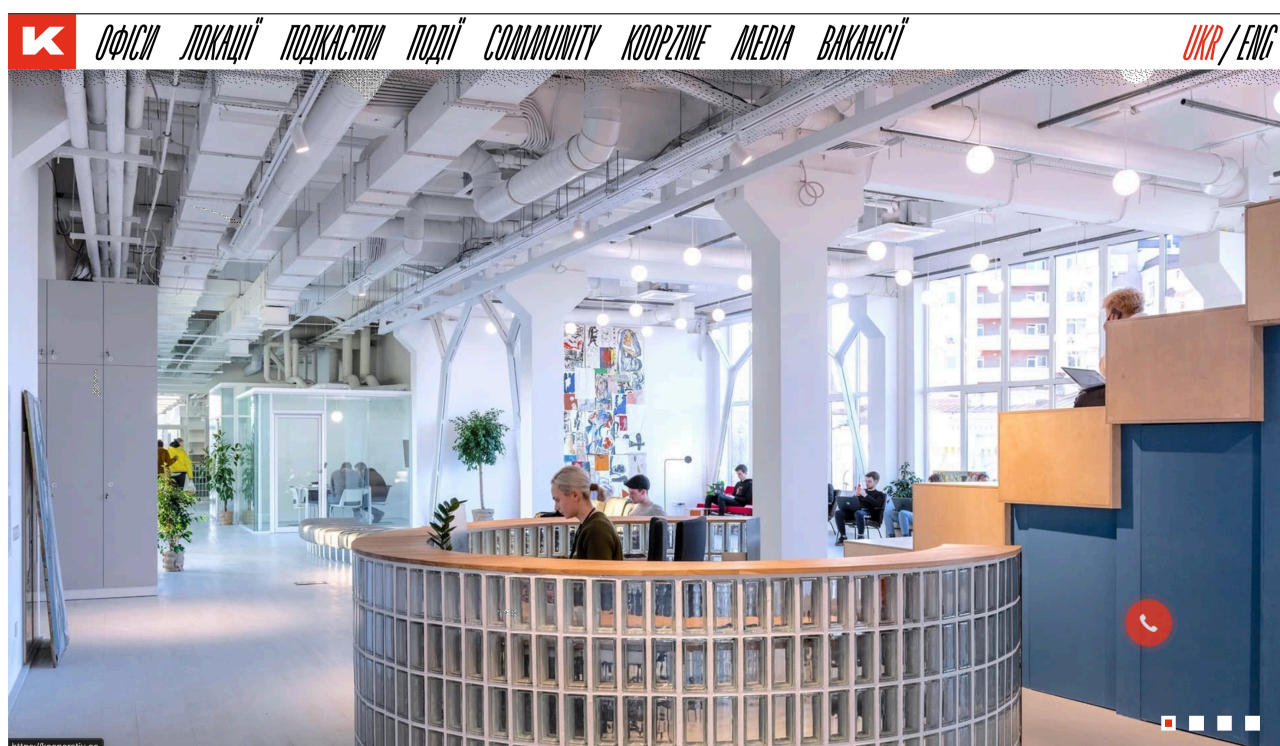


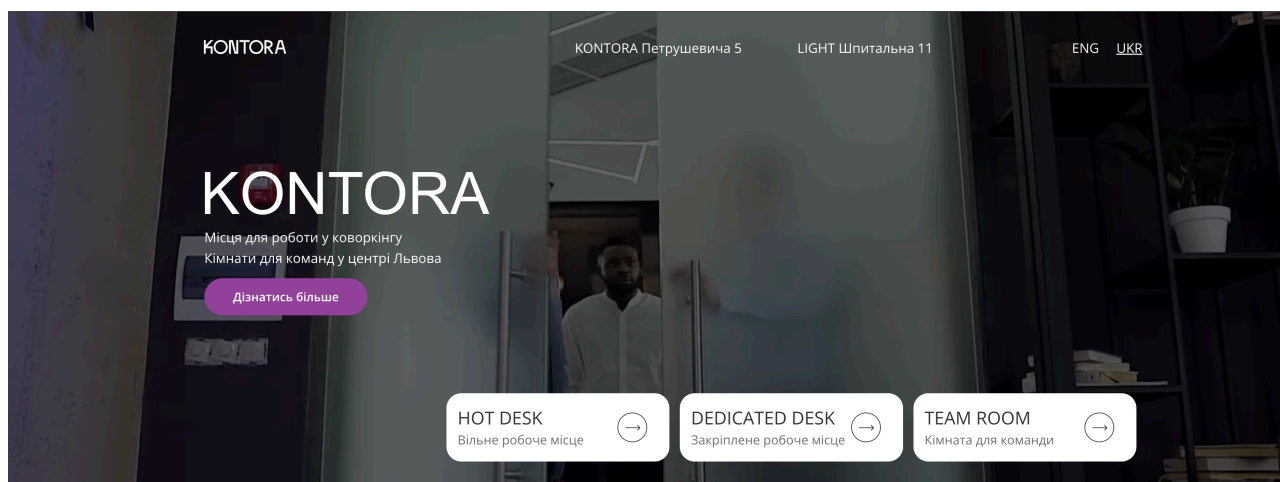
Рис. 1.5 Головна сторінка сайту kooperativ

1.2.4 Аналіз платформи KONTORA Lviv

KONTORA Lviv – коворкінг у центрі Львова з цілодобовим доступом, що пропонує офіси преміум-класу, кімнати для команд та переговорів, open-space та

Sky-лаунж із видом на місто. Мережа має дві локації: флагманську на площі Євгена Петрушевича та компактну Light на вулиці Шпитальній. Простір орієнтований на підприємців, фрілансерів та спеціалістів різних галузей, яких об'єднує прагнення до продуктивної роботи та розвитку.

Сайт kontoralviv.com надає базову інформацію про простори та тарифи. Бронювання кімнат для команд технічно можливе, проте реалізоване через форму заявки без підтвердження в реальному часі. Особистого кабінету з управлінням бронюваннями, командної моделі з ролями та запрошеннями, чату та підбору за інтересами не передбачено. Як і у випадку з іншими аналогами, платформа залишається сайтом-вітриною окремого коворкінгу, а не повноцінним цифровим інструментом для управління бронюваннями та командною взаємодією.



KONTORA

Vertical City Coworking -

від офісів преміум класу на 2 поверсі, до кімнат для команд та переговорів на третьому, а також серця KONTORA - просторого open-space офісу на 4-му та затишного Sky

LIGHT

Flat one-level coworking -

однорівневий коворкінг на вершині нового торгово-офісного центру в самому серці міста. Усе поруч - світлий лаунж та open-space офіс, кімнати для команд та переговорів, простора тераса та дах із мальовничим видом на місто.

Рис. 1.6 Головна сторінка сайту KONTORA Lviv

Серед переваг KONTORA Lviv – зручна локація в центрі Львова в пішій доступності від площі Ринок, цілодобовий доступ, кімнати для команд та переговорних зустрічей, розвинена інфраструктура. Серед недоліків – відсутність

повноцінної онлайн-системи бронювання з особистим кабінетом, командного функціоналу, системи сповіщень та будь-якої персоналізації за інтересами.

The screenshot displays the KONTORA website interface. At the top, the KONTORA logo is on the left, and the phone number +380 67 200 77 02 with language options (ENG, UKR) is on the right. The main heading reads 'Забронуйте місце у KONTORA за акційними умовами' (Book a place at KONTORA at promotional rates). Below this, a subtext states: 'Залишіть заявку і наш представник відповість якомога швидше, організує екскурсію та підготує пропозицію.' (Leave a request and our representative will respond as soon as possible, organize a tour and prepare a proposal). The form contains four input fields: 'Ім'я*' (Name*), 'Телефон*' (Phone*), 'Коментар' (Comment), and 'Зателефонуйте мені' (Call me) with a dropdown arrow. A 'Надіслати' (Send) button is positioned below the form. To the right of the form is a large, faint 'KONTORA' watermark. At the bottom, a map shows the location of KONTORA, Coworking & Offices at Yevhenia Petrusheycha Square, 5, Lviv, with a 4.5-star rating and a 'Citadel Inn' logo nearby.

Рис. 1.7 Сторінка з описом зон та умовами бронювання KONTORA Lviv

1.3 Визначення проблем предметної області

Ринок коворкінгів в Україні розвивається швидко, але цифрова складова цього ринку помітно відстає від реальних потреб користувачів. Проведений аналіз чотирьох платформ – Creative States, Platforma, Kooperativ та KONTORA Lviv – показав, що всі вони вирішують одну й ту саму задачу: надати людині фізичний простір для роботи. Але жодна з них не намагається вирішити питання того, як саме людина або команда знаходить потрібний простір, як координує спільну роботу та як взаємодіє з платформою протягом усього циклу бронювання.

Перша проблема стосується пошуку. Сьогодні вибір коворкінгу нагадує вибір готелю за кількістю зірок – є фільтри за ціною, локацією та переліком зручностей, але немає жодного інструменту, який враховував би, хто саме шукає простір і навіщо. ІТ-команда, студія дизайну та юридична фірма ставлять до простору принципово різні вимоги щодо атмосфери, спільноти та інфраструктури. Проте жодна з платформ не питає про сферу діяльності чи інтереси й не

використовує ці дані для підбору. В результаті пошук залишається ручним процесом методом спроб і помилок.

Друга проблема виникає тоді, коли простір шукає не одна людина, а команда. Типова ситуація виглядає так: хтось один бере на себе роль «організатора», самотійно обирає простір, бронює його та потім інформує решту учасників через Telegram або електронну пошту. Платформа при цьому взагалі не бачить, що йдеться про команду – вона обробляє запит однієї людини. Немає спільного простору для команди всередині платформи, немає можливості додати учасників, призначити ролі або делегувати управління бронюванням.

Третя проблема є прямим наслідком другої. Оскільки платформа не підтримує командну взаємодію, уся комунікація між учасниками виноситься за її межі. Обговорення варіантів, голосування за дату, уточнення деталей – все це відбувається в сторонніх месенджерах. Платформа стає лише фінальним кроком у процесі, а не його центром. Для продукту, що претендує на управління робочим простором команди, це принципова функціональна прогалина.

Четверта проблема пов'язана з тим, що жодна з платформ не знає нічого про свого користувача після реєстрації. Профіль, якщо він взагалі існує, містить ім'я та контактні дані – і не більше. Ніхто не питає, чим займається людина, які спільноти їй цікаві, які простори вона відвідувала раніше. Відповідно, жодної персоналізації немає: два абсолютно різних користувачі бачать однакову сторінку каталогу в однаковому порядку.

П'ята проблема стосується самого процесу бронювання. На більшості платформ він фактично зводиться до заявки: користувач заповнює форму та чекає, доки хтось з адміністраторів зв'яжеться з ним і підтвердить бронювання вручну. Автоматизованого циклу не існує. Немає миттєвого підтвердження, немає статусів, немає нагадувань, немає можливості зареєструвати прибуття або завершення роботи через інтерфейс. Для користувача це означає невизначеність; для власника простору – зайве адміністративне навантаження.

Сукупність цих п'яти проблем формує чіткий запит на продукт, який не просто дозволяє забронювати стіл чи переговорну кімнату, а й враховує інтереси

користувача при пошуку, підтримує повноцінну командну модель із розподілом ролей, забезпечує внутрішню комунікацію між учасниками та автоматизує весь цикл бронювання від заявки до check-out.

1.4 Постановка завдання на розробку системи

На основі виявлених проблем та проведеного аналізу предметної галузі сформульовано завдання на розробку веб-платформи бронювання коворкінг-просторів за інтересами з підтримкою командної роботи. Розроблювана система має стати єдиним цифровим середовищем, що охоплює повний цикл взаємодії між користувачем та коворкінг-простором – від першого пошуку до завершення робочого сеансу – і при цьому враховує специфіку командної роботи, яку існуючі рішення повністю ігнорують.

Система складається з двох основних компонентів: серверної частини на базі Node.js з Express та реляційною базою даних PostgreSQL, і клієнтської веб-частини на базі React з використанням TypeScript як єдиної мови програмування для обох рівнів. Вся кодова база організована у монорепозиторій, що забезпечує спільне використання типів, схем валідації та бізнес-логіки між сервером і клієнтом без дублювання.

Ключовою функціональною вимогою є реалізація алгоритму підбору просторів за інтересами. Користувач при реєстрації вказує сферу діяльності та теги інтересів, які надалі використовуються для ранжування каталогу та формування персоналізованих рекомендацій. Власники просторів, у свою чергу, також позначають свій простір відповідними тегами спільноти, що забезпечує двосторонній механізм відповідності між пропозицією та попитом.

Командна модель реалізується через повноцінну систему команд із розподілом ролей: адміністратор команди має право створювати бронювання від імені всієї команди, запрошувати учасників через посилання або електронну пошту та керувати складом. Учасники бачать актуальний стан бронювань команди та можуть брати участь у внутрішньому чаті, реалізованому через WebSocket

з'єднання на базі Socket.IO. Це усуває потребу виносити командну комунікацію за межі платформи.

Автентифікація побудована на основі JWT із парою access та refresh токенів і підтримкою двофакторної автентифікації. Рольова модель доступу охоплює п'ять ролей: звичайний користувач, адміністратор команди, власник простору, менеджер простору та адміністратор платформи. Кожна роль отримує відповідний кабінет із власним набором функцій та обмежень доступу до ресурсів системи.

Цикл бронювання є повністю автоматизованим: від вибору зони та часового слоту до миттєвого підтвердження, сповіщень через WebSocket та email, реєстрації прибуття через check-in та завершення сеансу через check-out. Статуси бронювання оновлюються в реальному часі для всіх учасників команди одночасно. Календар бронювань реалізований на базі FullCalendar, карта розташування просторів – на базі Leaflet.

Розгортання системи передбачено через Docker Compose, що дозволяє будь-якому власнику коворкінгу розгорнути платформу на власному сервері без залежності від зовнішніх сервісів. Документація API формується автоматично через Swagger та OpenAPI специфікацію.

1.5 Визначення вимог до застосунку

На основі аналізу предметної галузі, виявлених проблем та сформульованого завдання на розробку визначено вимоги до веб-платформи бронювання коворкінг-просторів. Вимоги поділено на функціональні та нефункціональні.

Функціональні вимоги описують поведінку системи з точки зору користувача – що саме платформа має вміти робити залежно від ролі та сценарію взаємодії.

Реєстрація та автентифікація. Користувач має можливість зареєструватися через веб-форму, вказавши ім'я, електронну пошту та пароль. Після реєстрації користувач одразу отримує доступ до платформи. Вхід до системи здійснюється через пару access та refresh токенів. Передбачено підтримку двофакторної автентифікації та механізм відновлення паролю через електронну пошту.

Реєстрація нових власників просторів можлива лише після підтвердження адміністратором платформи.

Профіль та інтереси. При заповненні профілю користувач вказує сферу діяльності, перелік професійних інтересів у вигляді тегів та додаткову інформацію про себе або свою команду. Ці дані використовуються алгоритмом рекомендацій та відображаються у картці члена команди. Профіль можна редагувати у будь-який момент, при цьому рекомендації оновлюються автоматично.

Каталог та пошук просторів. Платформа надає каталог коворкінг-просторів із можливістю пошуку за назвою, містом та адресою. Фільтрація здійснюється за типом зони, а також за тегами інтересів спільноти простору. Кожен простір має сторінку з детальним описом, фотографіями, переліком зон, тарифами, розкладом роботи та картою розташування на базі Leaflet. Результати пошуку ранжуються з урахуванням збігу між інтересами користувача та тегами простору.

Бронювання. Користувач обирає конкретну зону простору, дату та часовий слот із календаря доступності на базі FullCalendar. Система миттєво підтверджує бронювання та змінює його статус. Про підтвердження бронювання користувач отримує сповіщення на електронну пошту та в особистому кабінеті. Передбачено можливість скасування бронювання з боку користувача до визначеного дедлайну, а також скасування з боку менеджера простору з обов'язковим повідомленням.

Командна робота. Користувач може створити команду, вказати її назву та запросити учасників через електронну пошту або унікальне посилання. Адміністратор команди призначає ролі учасникам та може здійснювати бронювання від імені всієї команди. Усі учасники бачать актуальний стан бронювань команди в спільному календарі. Зміна складу команди та передача прав адміністратора реалізована через відповідний інтерфейс управління.

Чат у реальному часі. Кожна команда має внутрішній чат, реалізований через WebSocket з'єднання на базі Socket.IO. Повідомлення доставляються миттєво всім учасникам, що перебувають онлайн. Для учасників, що були офлайн, повідомлення зберігаються та відображаються при наступному вході. Передбачено відображення статусу прочитання повідомлень.

Check-in та check-out. У день бронювання користувач реєструє своє прибуття через інтерфейс платформи. Після завершення роботи виконується check-out. Обидві дії фіксуються у системі та відображаються в кабінеті менеджера простору. На основі цих даних формується статистика завантаженості зон для власника простору.

Кабінет власника та менеджера. Власник простору має доступ до управління профілем простору, переліком зон, тарифами, тегамі спільноти та розкладом роботи. Менеджер бачить актуальний список бронювань, може підтверджувати або скасовувати їх, а також переглядати статистику завантаженості за обраний період. Адміністратор ресепшн фіксує check-in та check-out відвідувачів через окремий інтерфейс.

Сповіщення. Система надсилає автоматичні сповіщення на електронну пошту при підтвердженні бронювання, його скасуванні, та при зміні статусу з боку менеджера. Внутрішні сповіщення відображаються в особистому кабінеті з лічильником непрочитаних.

Нефункціональні вимоги визначають якісні характеристики системи незалежно від конкретного сценарію використання.

Таблиця 1.1

Функціональні та нефункціональні вимоги до системи

Вимога	Тип	Пріоритет
Реєстрація та автентифікація з 2FA	Функціональна	Високий
Профіль із тегамі інтересів	Функціональна	Високий
Каталог та пошук за інтересами	Функціональна	Високий
Алгоритм рекомендацій просторів	Функціональна	Високий
Бронювання зон із підтвердженням	Функціональна	Високий
Командна модель із ролями та запрошеннями	Функціональна	Високий
Чат у реальному часі (Socket.IO)	Функціональна	Високий

Вимога	Тип	Пріоритет
Check-in та check-out	Функціональна	Середній
Кабінети OWNER / MANAGER / RECEPTION / ADMIN	Функціональна	Високий
Сповіщення на пошту та в кабінеті	Функціональна	Середній
Карта просторів (Leaflet)	Функціональна	Середній
Календар бронювань (FullCalendar)	Функціональна	Середній
Час відповіді API до 500 мс	Нефункціональна	Високий
JWT автентифікація та RBAC	Нефункціональна	Високий
Валідація через Zod на кожному ендпоінті	Нефункціональна	Високий
Кросбраузерна сумісність	Нефункціональна	Середній
Документація API через Swagger	Нефункціональна	Середній
Розгортання через Docker Compose	Нефункціональна	Середній

Безпека. Паролі зберігаються виключно у хешованому вигляді з використанням алгоритму bcrypt. Усі запити до захищених ресурсів перевіряються через JWT middleware. Вхідні дані валідуються на кожному ендпоінті через Zod схеми до потрапляння у бізнес-логіку. Безпечні HTTP заголовки встановлюються через Helmet.js.

Продуктивність. Час відповіді API на типові запити не має перевищувати 500 мс при стандартному навантаженні. Сторінки каталогу та пошуку мають завантажуватися без помітних затримок завдяки серверній пагінації та кешуванню часто запитуваних даних через Redis.

Кросбраузерність. Веб-застосунок має коректно відображатися та функціонувати у Google Chrome, Mozilla Firefox, Microsoft Edge та Safari актуальних версій. Інтерфейс має бути адаптивним для екранів від 375 пікселів у ширину до повноекранного десктопного режиму.

Документація. API платформи документується автоматично через Swagger та OpenAPI специфікацію, що забезпечує зручну інтеграцію для сторонніх розробників та спрощує підтримку системи.

Розгортання. Система має підтримувати розгортання через Docker Compose з мінімальною конфігурацією, що дозволяє власнику будь-якого коворкінгу запустити платформу на власному сервері без залежності від зовнішніх хмарних провайдерів.

2 АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ ТА АРХІТЕКТУРИ СИСТЕМИ

2.1 Вибір середовища розробки та організація проєкту

Вибір інструментів розробки та організація структури проєкту є фундаментальними рішеннями, що визначають зручність підтримки кодової бази, швидкість розробки та якість кінцевого продукту. Для даного проєкту ці рішення приймалися з урахуванням того, що система складається з кількох взаємопов'язаних підпроєктів, які мають спільно використовувати типи, схеми валідації та конфігурацію інструментів.

Як основне середовище розробки обрано Visual Studio Code – редактор коду з відкритим вихідним кодом від Microsoft, що є де-факто стандартом для розробки на TypeScript та JavaScript. VS Code забезпечує глибоку нативну інтеграцію з TypeScript: автодоповнення на рівні типів, навігацію між інтерфейсами та їхніми реалізаціями, вбудоване виявлення помилок типізації безпосередньо у редакторі без запуску компілятора вручну. Вбудований термінал, інтеграція з Git та підтримка розширень для роботи з Docker, Prisma та ESLint дозволяють вести весь процес розробки в єдиному середовищі без перемикання між інструментами.

Розглядалася альтернатива у вигляді WebStorm від JetBrains – повноцінного IDE з більш розвиненим рефакторингом та глибшою підтримкою фреймворків. Однак WebStorm є платним продуктом, тоді як VS Code надає сумірний рівень підтримки TypeScript безкоштовно завдяки вбудованому TypeScript Language Server, розробленому тією самою командою Microsoft. Для цілей даного проєкту функціональності VS Code є цілком достатньою.

Структура проєкту організована як монорепозіторій – єдиний Git-репозіторій, що містить усі підпроєкти системи. Монорепозіторій є доцільним вибором для даної системи, оскільки серверна та клієнтська частини мають суттєву кількість спільного коду: схеми валідації запитів, типи відповідей API та бізнес-константи мають бути однаковими на обох рівнях, щоб уникнути розбіжностей між тим, що очікує сервер, і тим, що надсилає клієнт.

Монорепозиторій складається з чотирьох пакетів із чітко визначеними зонами відповідальності:

`server/` – серверна частина на Node.js та Express. Містить усю бізнес-логіку, 24 файли маршрутів, сервіси, `middleware`, конфігурацію Prisma та підключення до Redis. Є точкою входу для обробки HTTP-запитів та WebSocket-з'єднань.

`client/` – клієнтська веб-частина на React та Vite. Містить компоненти інтерфейсу, логіку роботи з TanStack Query, конфігурацію маршрутизації, інтеграцію з FullCalendar та Leaflet. Збирається у статичні файли для розгортання.

`packages/shared/` – спільний пакет, що містить Zod-схеми валідації, TypeScript-типи та бізнес-константи, які використовуються одночасно сервером і клієнтом. Це ключовий елемент архітектури монорепозиторію: будь-яка зміна у схемі запиту або відповіді API одразу відображається як помилка компіляції в обох підпроектах, що унеможливорює ситуацію, коли сервер і клієнт мають розбіжні уявлення про формат даних.

`packages/eslint-config/` – централізований пакет конфігурації ESLint, що застосовується до всіх підпроектів монорепозиторію. Це гарантує єдині правила статичного аналізу коду незалежно від того, в якому підпроекті ведеться розробка.

Для управління залежностями використовується `pnpm` версії 9.15.4 із `workspace`-конфігурацією у файлі `pnpm-workspace.yaml`. `pnpm` обрано замість `npm` та `yarn` з кількох причин. По-перше, `pnpm` використовує `content-addressable storage`: кожен пакет зберігається в єдиному глобальному кеші на диску, а в `node_modules` підпроектів створюються лише символічні посилання. Це означає, що один і той самий пакет, використовуваний у кількох підпроектах монорепозиторію, зберігається фізично лише один раз незалежно від кількості місць його використання. По-друге, `pnpm` суворо дотримується принципу ізоляції залежностей: підпроект може використовувати лише ті пакети, які явно вказані у його `package.json`, що унеможливорює випадкове використання транзитивних залежностей і робить структуру залежностей прозорою та передбачуваною. По-третє, завдяки механізму `workspaces` локальні пакети монорепозиторію посилаються один на одного як звичайні `npm`-пакети: `server/` та `client/` підключають

packages/shared/ через workspace-протокол, і при зміні shared-пакету обидва підпроєкти миттєво бачать оновлену версію без необхідності публікації чи додаткових команд. Для забезпечення якості коду налаштовано ESLint із власним пакетом конфігурації packages/eslint-config/. ESLint виконує статичний аналіз коду та виявляє потенційні помилки, небезпечні патерни та порушення стилю ще до запуску програми. Централізована конфігурація гарантує, що правила аналізу є однаковими для server/, client/ та shared-пакетів, і розробник не може обійти їх, змінивши конфігурацію лише в одному підпроєкті.

Автоматичне форматування коду забезпечується через Prettier з єдиною конфігурацією для всього монорепозиторію. Prettier вирішує питання стилю коду раз і назавжди: відступи, лапки, крапки з комою, довжина рядка – усі ці параметри визначені один раз і застосовуються однаково до TypeScript, JSX, JSON та CSS файлів у всіх підпроєктах. Це усуває дискусії щодо стилю коду в команді та забезпечує консистентність кодової бази незалежно від того, хто і коли вносив зміни.

Для автоматизації перевірки якості коду перед кожним комітом налаштовано pre-commit хуки через Husky та lint-staged. При спробі зробити коміт lint-staged автоматично запускає ESLint та Prettier лише для файлів, що входять до коміту, а не для всієї кодової бази. Це суттєво скорочує час перевірки та гарантує, що до репозиторію потрапляє лише код, що відповідає встановленим стандартам якості.

Контроль версій здійснюється через Git із хостингом репозиторію на GitHub. Організація роботи з гілками побудована на основі підходу feature-branch: кожна нова функціональність розробляється в окремій гілці, що відгалужується від основної, та вливається через pull request після перевірки. Така організація дозволяє вести паралельну розробку кількох функціональностей без взаємних конфліктів та забезпечує повну історію змін з можливістю відкату до будь-якого попереднього стану.

Кожен підпроєкт має власний файл tsconfig.json, що наслідує спільну базову конфігурацію. Увімкнено суворий режим strict, що активує всі додаткові перевірки типів компілятора: заборону неявного типу any, перевірку на null та undefined,

суворий контроль типів параметрів функцій. Це забезпечує максимальний рівень типобезпеки в усіх підпроєктах та виявляє найширший клас потенційних помилок на етапі компіляції.

Локальне середовище розробки розгортається через Docker Compose, який піднімає усі необхідні сервіси в ізольованих контейнерах: PostgreSQL 16 як основна база даних [11], Redis 7 для кешування та rate-limiting, Mailpit як локальний SMTP-сервер для перехоплення та перегляду листів без їхнього реального надсилання, MinIO як S3-сумісне об'єктне сховище для файлів. Такий підхід гарантує ідентичність середовища розробки [21] між усіма учасниками команди та усуває залежність від зовнішніх сервісів під час локальної розробки. Розробник отримує повністю функціональне середовище однією командою `docker compose up`, без необхідності встановлювати PostgreSQL, Redis чи інші сервіси безпосередньо на операційну систему.

Для забезпечення автоматичної перевірки якості коду при кожному `push` до репозиторію налаштовано конвеєр безперервної інтеграції (CI) на базі GitHub Actions. Конфігурація конвеєру описана у файлі `ci.yml` та запускається автоматично при кожному `push` у гілку `main`. Це гарантує, що жоден коміт, що порушує типізацію, правила лінтера або логіку застосунку, не може залишитися непоміченим.

Конвеєр складається з п'яти незалежних задач, організованих у два паралельних блоки виконання. Перший блок запускається одразу при `push` і містить дві задачі: `Lint & Type-check` та `Prisma Validate & Generate`. Задача `Lint & Type-check` запускає ESLint для статичного аналізу коду та компілятор TypeScript у режимі перевірки типів без генерації вихідних файлів, що дозволяє виявити помилки типізації на рівні CI ще до того, як код потрапить до продакшн-середовища. Задача `Prisma Validate & Generate` перевіряє коректність схеми бази даних та генерує типізований Prisma Client, що підтверджує відповідність схеми та міграцій. Другий паралельний блок містить `Unit Tests` та `Docker Build`, де перевіряється коректність роботи окремих модулів та успішність збірки Docker-образу. Після успішного завершення обох блоків запускається фінальна задача

Playwright E2E, що виконує наскрізне тестування користувацьких сценаріїв у реальному браузері.

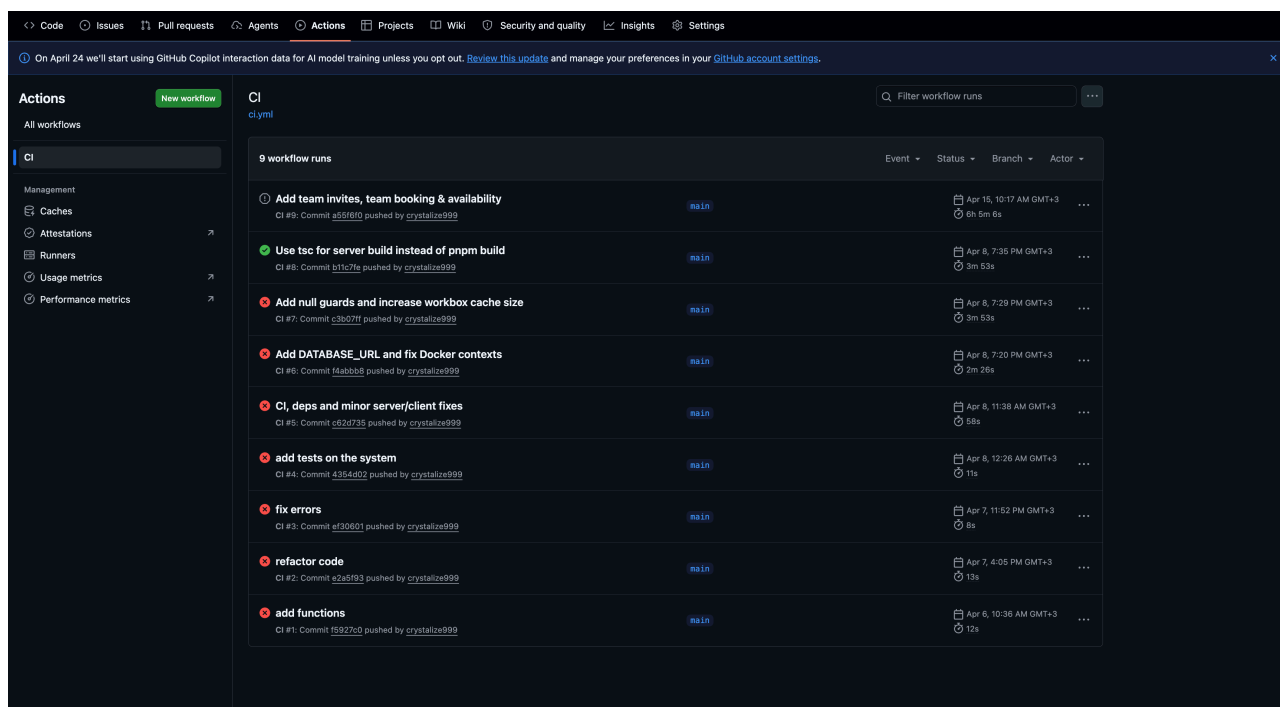


Рис. 2.1 Список workflow-запусків CI у репозиторії проєкту на GitHub Actions

Загальний час виконання повного конвеєру складає близько 3 хвилин 53 секунд, що є прийнятним показником для проєкту даного масштабу. У разі успішного проходження всіх задач конвеєр отримує статус Success, що підтверджує коректність внесених змін.

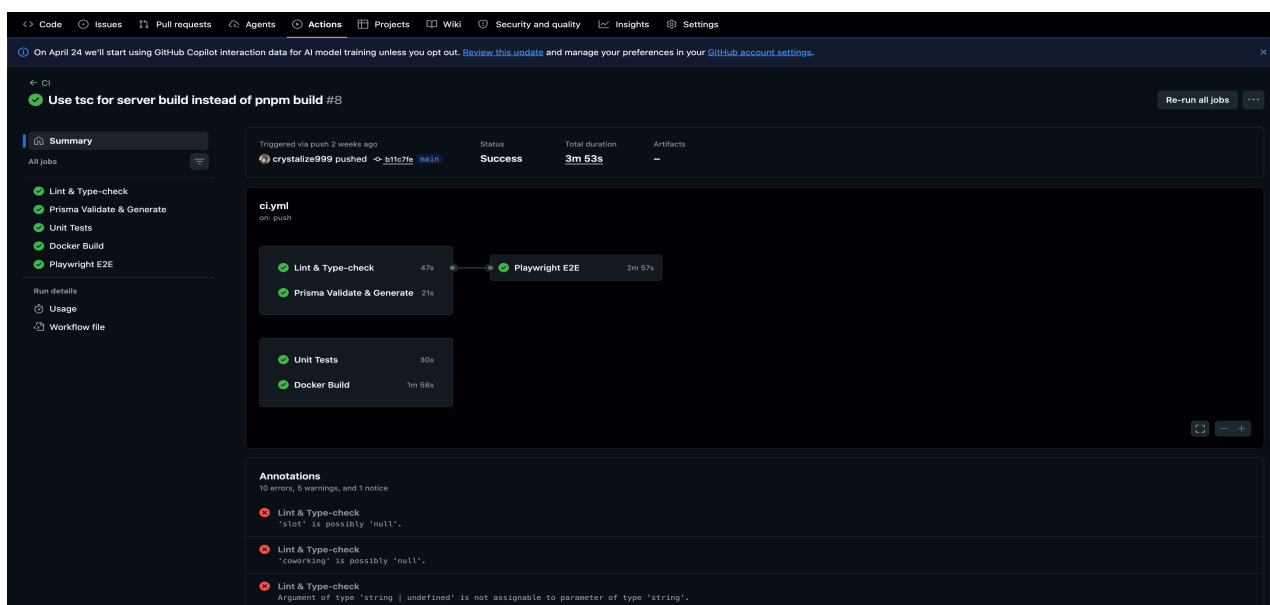


Рис. 2.2 Деталі успішного запуску CI: задачі Lint & Type-check, Prisma Validate & Generate, Unit Tests, Docker Build та Playwright E2E

Наявність окремого `docker-compose.test.yml` із використанням ephemeral бази даних на `tmpfs` забезпечує повну ізоляцію E2E-тестів від локального середовища розробки. У межах такого підходу кожен запуск тестового набору відбувається в окремому тимчасовому середовищі, де створюється чиста порожня база даних у пам'яті без залежності від уже наявних локальних даних. Після завершення тестового сеансу ця база даних автоматично видаляється, що виключає накопичення службової інформації та вплив результатів попередніх запусків на наступні перевірки. Така організація тестового контуру гарантує відтворюваність результатів тестування, підвищує надійність перевірки функціональності системи та зменшує ризик хибних спрацювань, пов'язаних із залишковими даними або відмінностями конфігурації середовища розробника.

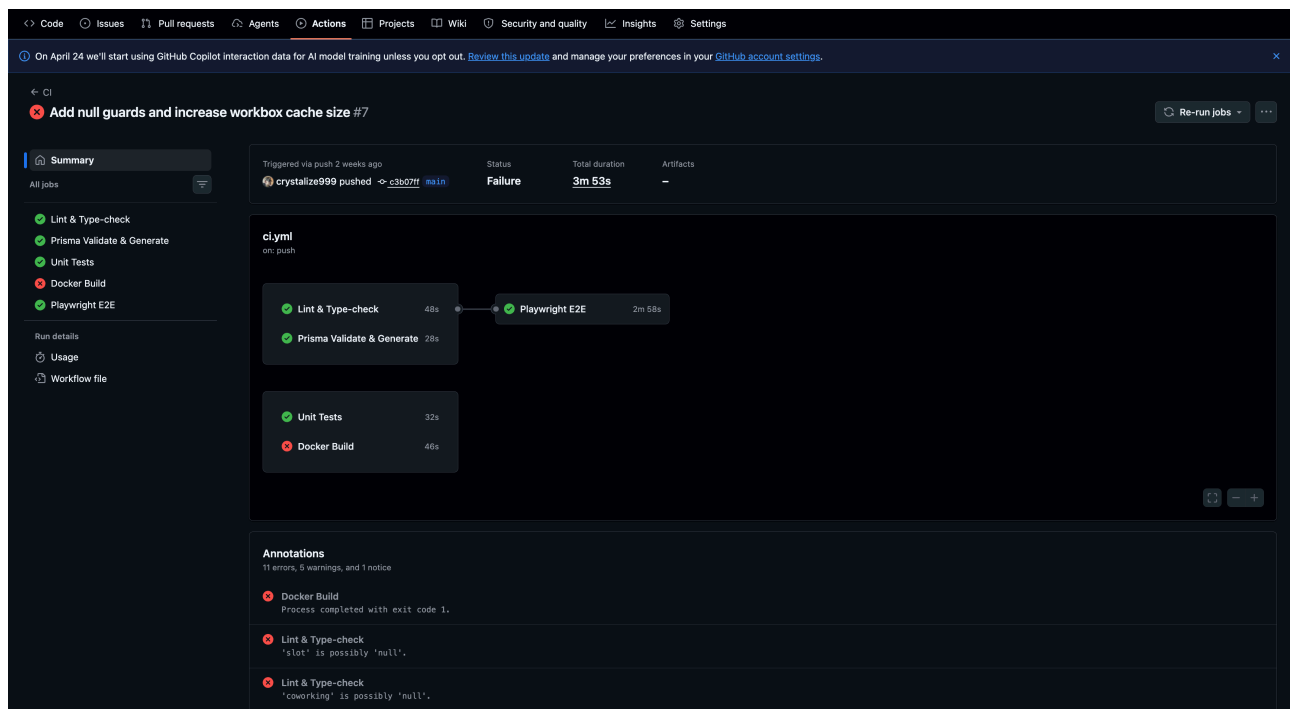


Рис. 2.3 Приклад запуску з помилкою: задача Docker Build завершилась з exit code 1 при збірці образу

Практика роботи з CI під час розробки підтвердила цінність автоматизованої перевірки: кілька ранніх запусків завершилися з помилками на етапі Docker Build та Lint & Type-check, що дозволило виявити та виправити проблеми з типізацією та конфігурацією збірки до того, як вони могли вплинути на стабільність системи.

2.2 Вибір мови програмування та обґрунтування технологічного стеку

Вибір технологічного стеку для платформи бронювання коворкінг-просторів визначався кількома ключовими вимогами одночасно: система має обробляти HTTP-запити від клієнта, підтримувати двонаправлений зв'язок у реальному часі для командного чату, забезпечувати спільну валідацію даних між сервером і клієнтом та розгортатися у контейнерах без складної інфраструктури. Жодна з цих вимог не є унікальною сама по собі, однак їхня сукупність суттєво звужує простір технологічних рішень, що можуть задовольнити всі умови одночасно.

TypeScript обрано як єдину мову програмування для всіх рівнів системи – серверної частини, клієнтської частини та спільного пакету `packages/shared/`. Це рішення продиктоване не модою чи особистими вподобаннями, а конкретною технічною необхідністю: Zod-схеми валідації, описані один раз у спільному пакеті, використовуються одночасно сервером для перевірки вхідних HTTP-запитів та клієнтом для валідації форм [6]. При зміні схеми – наприклад, якщо поле змінює тип або стає обов'язковим – компілятор TypeScript виявляє всі невідповідності у кодовій базі до запуску програми, незалежно від того, у якому підпроекті вони знаходяться. Це усуває цілий клас помилок, характерних для систем, де сервер і клієнт розроблені різними мовами та синхронізуються вручну.

Окремим архітектурним рішенням став вибір підходу до організації серверного API. Розглядалися три основні варіанти: REST, GraphQL та gRPC. REST API з документацією через Swagger та OpenAPI обрано як основний підхід з кількох причин. По-перше, REST має нативну підтримку у браузері без додаткових бібліотек на клієнті. По-друге, екосистема інструментів навколо REST є найзрілішою: автоматична генерація документації, тестування через Postman, middleware для автентифікації та rate-limiting – все це є стандартними рішеннями для Express. По-третє, WebSocket через Socket.IO реалізується як окремий канал поряд із REST, що є природним розподілом: REST для CRUD-операцій, WebSocket для подій реального часу у командному чаті.

GraphQL розглядався як альтернатива через його гнучкість у формуванні запитів – клієнт отримує рівно ті поля, які йому потрібні. Однак для даного проєкту ця гнучкість є надлишковою: структура відповідей API є фіксованою та передбачуваною, а складність впровадження GraphQL – кешування, N+1 проблема, налаштування підписок – суттєво перевищує практичну користь. gRPC відхилено через обмежену підтримку у браузері: використання gRPC-Web потребує додаткового проксі-шару, що ускладнює інфраструктуру без виправданої потреби. Порівняння підходів наведено у табл. 2.1.

Таблиця 2.1

Порівняння мов програмування для серверної частини

Критерій	REST API	GraphQL	gRPC
Складність впровадження	Низька	Середня	Висока
Підтримка браузером	Нативна	Через бібліотеку	Обмежена
Гнучкість запитів	Фіксована структура	Довільна вибірка	Фіксована структура
Генерація документації	Swagger/OpenAPI	Схема типів	Protobuf
WebSocket підтримка	Окремо	Subscriptions	Streaming
Зрілість екосистеми	Дуже висока	Висока	Середня
Підходить для даного проєкту	Так	Частково	Ні

Для серверної частини обрано Node.js – середовище виконання JavaScript поза браузером на базі рушія V8 від Google – у поєднанні з фреймворком Express [7]. Node.js реалізує асинхронну однопотокову модель на основі event loop, що є ефективним підходом для API-сервера, де переважна більшість операцій є I/O-інтенсивними: запити до PostgreSQL, операції з Redis, відправка листів. Процесор при цьому не блокується в очікуванні відповіді від бази даних, а обробляє наступний запит, що забезпечує високу пропускну здатність при відносно малому споживанні ресурсів.

Express обрано як HTTP-фреймворк через його мінімалістичність та повний контроль над архітектурою проєкту [8]. На відміну від NestJS, який нав'язує певну структуру через декоратори та модульну систему, Express дозволяє організувати код саме так, як цього вимагає конкретна предметна галузь. Це є перевагою для проєкту з нестандартними вимогами, де жорстка структура фреймворку може стати обмеженням.

Для роботи з базою даних використовується Prisma ORM – інструмент, що генерує типобезпечний клієнт на основі декларативної схеми [12]. Prisma дозволяє описувати моделі та зв'язки між ними у файлі `schema.prisma`, після чого автоматично генерує TypeScript-клієнт із повною типізацією всіх запитів. Це означає, що звернення до неіснуючого поля таблиці або передача некоректного типу у фільтр запиту є помилкою компіляції, а не SQL-помилкою під час виконання. Міграції схеми бази даних також керуються через Prisma Migrate, що забезпечує версіонування структури бази даних разом із кодом.

Redis використовується як сховище даних у пам'яті через клієнтську бібліотеку `ioredis` версії 5.5.0 [13]. У системі Redis виконує три конкретні функції: зберігання OTP-кодів для двофакторної автентифікації з автоматичним TTL 10 хвилин, що забезпечує анулювання прострочених кодів без додаткової логіки; `rate-limiting` для захисту ендпоінтів автентифікації від автоматизованих атак перебору; зберігання даних сесій для прискорення повторних запитів без звернення до основної бази даних.

Клієнтська частина побудована на React 19 зі збирачем Vite. React реалізує декларативний компонентний підхід до побудови інтерфейсу [9]: розробник описує, як має виглядати інтерфейс залежно від поточного стану, а React самостійно визначає мінімально необхідний набір змін у DOM для відображення цього стану. Vite забезпечує миттєвий старт dev-сервера та оновлення модулів у браузері за мілісекунди завдяки використанню нативних ES-модулів під час розробки.

Управління серверним станом реалізовано через TanStack Query версії 5.90.21, що автоматично кешує відповіді сервера, інвалідує кеш при мутаціях та

повторює запити при мережевих помилках [15]. Для стилізації використовується Tailwind CSS версії 3.4.17 – утилітарний підхід, де стилі описуються безпосередньо у JSX через набір класів [16], що усуває необхідність ведення окремих CSS-файлів. Framer Motion забезпечує анімації інтерфейсу. Локалізація реалізована через i18next. Застосунок зареєстровано як PWA через vite-plugin-pwa, що дозволяє користувачам встановити платформу безпосередньо з браузера на мобільний пристрій.

Таблиця 2.2

Технологічний стек системи

Рівень	Технологія	Версія	Призначення
Мова	TypeScript	5.9.3	Єдина мова для всіх рівнів системи
Сервер	Node.js	22 LTS	Середовище виконання серверної частини
Сервер	Express	4.22.1	HTTP-фреймворк та маршрутизація
Сервер	Prisma ORM	5.22.0	Типобезпечний доступ до бази даних
Сервер	Socket.IO	4.8.3	WebSocket для командного чату
Сервер	Zod	3.25.76	Валідація вхідних даних
Сервер	JWT	9.0.3	Автентифікація access/refresh токени
Сервер	bcrypt / argon2	–	Хешування паролів та backup-кодів
Сервер	Helmet.js	8.1.0	Безпечні HTTP-заголовки
Сервер	speakeasy	2.0.0	TOTP для двофакторної автентифікації
База даних	PostgreSQL	16	Основна реляційна СУБД
Кеш	Redis	7	OTP, rate-limiting, сесії
Клієнт	React	19.2.4	Бібліотека побудови інтерфейсу
Клієнт	Vite	6.4.1	Збирач та dev-сервер
Клієнт	TanStack Query	5.90.21	Кешування серверного стану
Клієнт	Tailwind CSS	3.4.19	Утилітарна стилізація
Клієнт	FullCalendar	6.1.20	Календар бронювань
Клієнт	Leaflet	–	Карта коворкінг-просторів
Клієнт	Framer Motion	–	Анімації інтерфейсу
Клієнт	i18next	–	Локалізація

Рівень	Технологія	Версія	Призначення
Спільне	pnpm workspaces	9.15.4	Монорепозиторій
Спільне	Zod (shared)	3.25.76	Спільні схеми валідації
Інфра	Docker Compose	—	Локальне середовище та тестування
Інфра	GitHub Actions	—	CI/CD конвеєр
Інфра	MinIO	—	Об'єктне сховище файлів
Інфра	Mailpit	—	Локальний SMTP для розробки

Специфічні функціональні вимоги платформи вирішуються через спеціалізовані бібліотеки. Календар бронювань реалізований на базі FullCalendar версії 6.1.20 із плагінами daygrid, timegrid та interaction, що забезпечує відображення місячного та тижневого виду з підтримкою drag-and-drop [17]. Карта коворкінг-просторів побудована через Leaflet у поєднанні з react-leaflet та кластеризацією маркерів для коректного відображення великої кількості точок на одній карті [18].

Локальне середовище розробки розгортається через Docker Compose, що піднімає PostgreSQL 16, Redis 7, Mailpit як локальний SMTP-сервер та MinIO як S3-сумісне об'єктне сховище. Продакшн-розгортання реалізоване через окремі multi-stage Dockerfile для серверної та клієнтської частин без використання Compose, що забезпечує мінімальний розмір фінальних образів. Повний технологічний стек наведено у табл. 2.2. Для тестування використовується окремий docker-compose.test.yml із ephemeral базою даних на tmpfs, що гарантує повну ізоляцію тестів та відтворюваність результатів.

2.3 Архітектура клієнт-серверної взаємодії та організація API

Взаємодія між клієнтською та серверною частинами системи побудована на двох незалежних каналах зв'язку, кожен з яких вирішує свою задачу. Основний канал – REST API через HTTP для всіх операцій створення, читання, оновлення та видалення даних. Допоміжний канал – WebSocket через Socket.IO для командного чату в реальному часі. Такий розподіл є осмисленим архітектурним рішенням:

REST добре підходить для операцій з чітко визначеним результатом та стандартною семантикою HTTP-методів, тоді як WebSocket природно вирішує задачу двонаправленого обміну подіями між учасниками команди без необхідності постійного polling-у з боку клієнта.

Серверна частина організована без класичного шару контролерів як окремої папки. Натомість логіка розподілена між двома рівнями: маршрути у папці `routes/` відповідають за прийом HTTP-запиту, застосування `middleware` та виклик відповідного сервісу, а сервіси у папці `services/` інкапсулюють усю бізнес-логіку та взаємодіють з базою даних через `Prisma Client`. Такий дворівневий підхід є доцільним для проєкту даного масштабу: додатковий шар контролерів між маршрутами та сервісами створював би надлишкову структуру без практичної користі.

Точкою входу для HTTP-запиту є файл маршруту, що визначає шлях, HTTP-метод та ланцюжок `middleware`. Типовий ланцюжок для захищеного ендпоінту виглядає так: `requireAuth` перевіряє JWT-токен з заголовку `Authorization` та записує дані користувача в об'єкт запиту, `validateBody` застосовує `Zod`-схему до тіла запиту і повертає структуровану помилку 400 при невідповідності, `requireRole` звіряє роль користувача з переліком дозволених для даного маршруту, і лише після проходження всіх перевірок `asyncHandler` передає запит до функції сервісу. Обгортка `asyncHandler` у `server/src/utils/async-handler.ts` автоматично перехоплює помилки асинхронних функцій через `.catch(next)` та передає їх до централізованого `error middleware`, що усуває необхідність обгортати кожен обробник у `try-catch` блок вручну. Глобальний `middleware` ланцюжок в `server/src/app.ts` застосовується до всіх запитів у фіксованому порядку: `pinoHttp` забезпечує структуроване логування кожного запиту та відповіді у форматі JSON через бібліотеку `Pino`, `helmet` встановлює безпечні HTTP-заголовки включно з `Content-Security-Policy` та `HSTS`, `cors` налаштовує дозволені джерела запитів, `express.json` та `cookieParser` обробляють тіло запиту та `cookie` відповідно. `Rate limiting` не застосовується глобально, а підключається точково до конкретних маршрутів через `Redis`, що дозволяє гнучко налаштовувати обмеження для різних груп ендпоінтів незалежно.

REST API реалізовано через 24 файли маршрутів, що охоплюють усі функціональні домени системи: `auth`, `bookings`, `coworkings`, `teams`, `users`, `notifications`, `payments`, `pricing`, `reviews`, `zones`, `staff`, `stats`, `availability`, `recommendations`, `waitlist`, `recurring-bookings`, `favorites`, `push`, `audit`, `data-io`, `experiments` та `health`. Кожен файл маршрутів відповідає за один функціональний домен і підключається до головного роутера в `app.ts` з відповідним префіксом шляху. Загальна кількість ендпоінтів становить близько 140, що є показником повноти покриття всіх сценаріїв використання платформи.

Змінні середовища перевіряються при старті сервера через `Zod`-схему у `server/src/config/env.ts`. Якщо будь-яка обов'язкова змінна відсутня або має неправильний формат [20] – сервер не запускається і видає детальне повідомлення про помилку конфігурації. Це усуває ситуацію, коли сервер стартує успішно, але падає при першому зверненні до бази даних через відсутній `DATABASE_URL`.

Централізований обробник помилок у `server/src/middlewares/error.middleware.ts` є стандартним чотириаргументним `Express middleware`, що стоїть останнім у ланцюжку та перехоплює всі помилки незалежно від того, на якому рівні вони виникли. Відповідь клієнту формується залежно від типу помилки. Екземпляр `HttpError` повертає `statusCode` та `message` з об'єкта помилки. `ZodError` формує відповідь зі статусом 400 та масивом `issues` з деталями кожного порушення валідації. `PrismaClientKnownRequestError` обробляється окремо і повертає 400 з кодом помилки `Prisma`, що дозволяє клієнту розрізняти, наприклад, порушення унікальності від порушення зовнішнього ключа. Усі інші помилки повертають 500 з загальним повідомленням без розкриття внутрішніх деталей. Кастомні хелпери у `server/src/utils/http-error.ts` – `badRequest`, `unauthorized`, `forbidden`, `notFound`, `conflict` – дозволяють створювати типізовані HTTP-помилки з одного рядка коду в будь-якому сервісі.

Канал реального часу реалізований через `Socket.IO` у файлі `server/src/socket/team.socket.ts` і обслуговує командний чат [14]. При встановленні `WebSocket`-з'єднання виконується авторизація: сервер витягує `JWT`-токен або з поля `socket.handshake.auth.token`, або з заголовку `Authorization` у форматі `Bearer`.

Токен верифікується через `jwt.verify` з секретом `JWT_ACCESS_SECRET`, і якщо токен відсутній, прострочений або не є access-токеном – з'єднання розривається. Це гарантує, що до WebSocket-каналу не може отримати доступ неавторизований користувач, навіть якщо він знає адресу сокет-сервера. Після успішної авторизації клієнт може надсилати події `join` для входу до кімнати команди, `leave` для виходу, `message` для відправки повідомлення та `presence` для оновлення статусу присутності. Всі учасники кімнати отримують події в реальному часі без будь-якого polling-у.

Сповіщення про зміну статусу бронювань реалізовані через модель `Notification` у базі даних. При настанні події – підтвердження, скасування або потрапляння до `waitlist` – сервер зберігає запис у таблиці сповіщень з відповідним типом: `BOOKING_CREATED`, `BOOKING_CONFIRMED`, `BOOKING_REJECTED`, `BOOKING_CANCELLED` або `WAITLIST_AVAILABLE`. Клієнт отримує сповіщення через `polling` при наступному запиті, що є свідомим компромісом між складністю реалізації та актуальністю даних для даного типу подій.

Документація API реалізована через Swagger у файлі `server/src/docs/openapi.ts` у форматі OpenAPI 3.0.3 та доступна за шляхом `/docs`. Документ написаний вручну як статичний TypeScript-об'єкт без кодогенерації або JSDoc-анотацій. Поточне покриття охоплює ключові групи ендпоінтів: `health`, `auth`, двофакторна автентифікація, `bookings` та `team messages`. Ручний підхід до ведення документації є осмисленим рішенням для проєктів, де якість та точність опису конкретних ендпоінтів важливіша за повноту автоматично згенерованої документації.

2.4 Реалізація серверної частини та вибір фреймворку

Серверна частина платформи побудована на Node.js у поєднанні з фреймворком Express. Вибір цього поєднання обумовлений кількома факторами. Node.js реалізує асинхронну однопотокову модель виконання на основі `event loop`, що є природним підходом для API-сервера з великою кількістю паралельних I/O-операцій: запити до PostgreSQL, читання та запис у Redis, відправка електронних

листів через SMTP. Процесор при цьому не простоює в очікуванні відповіді від зовнішніх сервісів, а продовжує обробляти наступні запити з черги. Express як HTTP-фреймворк обрано через мінімалістичність та відсутність нав'язаної архітектури: він надає базовий механізм маршрутизації та middleware-ланцюжка, залишаючи розробнику повну свободу у структуруванні коду відповідно до потреб предметної галузі.

Кодова база серверної частини організована у папці `server/src/` і складається з кількох функціональних груп. Точкою входу є `main.ts`, що ініціалізує HTTP-сервер та WebSocket-з'єднання. Конфігурація застосунку зосереджена в `app.ts`, де підключаються глобальні middleware та реєструються маршрути. Папка `config/` містить єдиний файл `env.ts` із Zod-схемою для валідації 14 змінних середовища при старті сервера. Папка `lib/` містить три ініціалізовані клієнти: `prisma.ts` – підключення до PostgreSQL, `redis.ts` – підключення до Redis через `ioredis`, `logger.ts` – налаштований екземпляр Pino. Папка `middlewares/` містить п'ять middleware-функцій: `auth.middleware.ts`, `roles.middleware.ts`, `validate.middleware.ts`, `coworking-access.middleware.ts` та `error.middleware.ts`. Папка `utils/` містить дві службові утиліти: `async-handler.ts` та `http-error.ts`.

Окремої папки `controllers/` у проєкті немає. Це свідоме рішення: логіка обробки запитів розподілена між файлами маршрутів у `routes/` та сервісами у `services/`. Маршрут відповідає за прийом запиту, застосування middleware та виклик сервісу. Сервіс містить бізнес-логіку та взаємодіє з базою даних. Такий підхід є достатнім для проєкту даного масштабу і не створює зайвих абстракцій.

Бізнес-логіка системи розподілена між 14 сервісами у папці `services/`. `auth.service.ts` відповідає за реєстрацію, вхід, верифікацію email, управління JWT-токенами та три методи двофакторної автентифікації. `bookings.service.ts` реалізує повний цикл бронювання: перевірку доступності слоту, створення запису, зміну статусів та скасування. `teams.service.ts` керує командами, їхнім складом, запрошеннями та правами учасників. `notifications.service.ts` зберігає та відмічає прочитаними сповіщення користувачів. `email.service.ts` відправляє листи через SMTP з логуванням через Pino. `payment.service.ts` та `pricing.service.ts` відповідають

за обробку платежів та розрахунків вартості відповідно до правил тарифікації. `recommendations.service.ts` реалізує алгоритм підбору просторів на основі інтересів користувача. `recurring-bookings.service.ts` обробляє повторювані бронювання за розкладом. `waitlist.service.ts` керує чергою очікування на зайняті слоти. `stats.service.ts` агрегує статистику для кабінету власника простору. `data-io.service.ts` забезпечує імпорт та експорт даних. `experiments.service.ts` реалізує A/B тестування через моделі `Experiment`, `ExperimentAssignment` та `ExperimentEvent`. `audit.service.ts` фіксує критичні дії в системі через модель `AuditLog`.

Доступ до PostgreSQL реалізований через Prisma ORM. Схема бази даних містить 25 моделей та 6 enum-ів, організованих у сім функціональних груп. Група користувачів включає моделі `User` та `Profile`. Група коворкінгів – `Coworking`, `Zone` та `AvailabilitySlot`. Група бронювань – `Booking`, `RecurringBookingRule` та `Waitlist`. Група команд – `Team`, `TeamMember`, `TeamInvite` та `Message`. Група фінансів – `Payment` та `PricingRule`. Група контенту – `Review`, `Favorite` та `Notification`. Група інтересів – `InterestCategory`, `UserInterest` та `CoworkingInterest`, що є ключовими моделями для алгоритму підбору просторів за інтересами. Окремо виділені моделі аудиту `AuditLog`, push-підписок `PushSubscription` та A/B тестів `Experiment`, `ExperimentAssignment` та `ExperimentEvent`. Шість enum-ів визначають допустимі стани сутностей: `Role`, `BookingStatus`, `TeamRole`, `InviteStatus`, `NotificationType` та `PaymentStatus`.

Prisma генерує типізований клієнт на основі цієї схеми, що робить будь-яке звернення до бази даних перевіреним компілятором TypeScript. Спроба звернутися до неіснуючого поля таблиці або передати некоректний тип у фільтр запиту є помилкою компіляції, а не помилкою виконання. Міграції схеми керуються через Prisma Migrate і зберігаються у репозиторії разом із кодом, що забезпечує відтворюваність структури бази даних у будь-якому середовищі.

Усі змінні середовища визначені та валідуються через Zod-схему у `server/src/config/env.ts` при старті сервера. Схема охоплює 14 змінних, згрупованих за призначенням: `NODE_ENV` та `SERVER_PORT` для конфігурації сервера, `DATABASE_URL` для підключення до PostgreSQL, `REDIS_URL` для підключення

до Redis, `JWT_ACCESS_SECRET`, `JWT_REFRESH_SECRET`, `JWT_ACCESS_TTL` зі значенням 15 хвилин та `JWT_REFRESH_TTL` зі значенням 24 години для управління токенами, `CORS_ORIGIN` та `PUBLIC_WEB_ORIGIN` для налаштування дозволених джерел запитів, `SMTP_HOST`, `SMTP_PORT`, `SMTP_USER`, `SMTP_PASS` та `SMTP_FROM` для конфігурації поштового сервісу. Якщо будь-яка обов'язкова змінна відсутня або не відповідає очікуваному формату – сервер завершує роботу з детальним повідомленням про помилку до того, як обробить перший запит.

Rate limiting реалізований точково через бібліотеку `rate-limiter-flexible` у поєднанні з Redis виключно в `auth.service.ts` для двох критичних операцій. Для входу в систему `loginLimiter` дозволяє не більше п'яти спроб протягом 60 секунд, після чого блокує IP-адресу на 120 секунд. У режимі розробки ліміт підвищений до 100 спроб з блокуванням на 5 секунд для зручності тестування. Для підтвердження двофакторної автентифікації `twoFactorLimiter` також дозволяє п'ять спроб за 60 секунд, однак блокування є тривалішим – 300 секунд, що суттєво ускладнює перебір OTP-кодів. Зберігання лічильників у Redis забезпечує коректну роботу rate limiting навіть при горизонтальному масштабуванні серверної частини: лічильники є спільними між усіма екземплярами сервера, а не локальними для кожного процесу.

Логування реалізовано через `Pino` — високопродуктивну бібліотеку структурованого логування у форматі JSON, яка забезпечує швидку обробку журналів подій без значного впливу на продуктивність сервера. `Pino` задіяний у чотирьох основних місцях: глобальний `middleware pinoHttp` в `app.ts` автоматично логує кожен HTTP-запит та відповідь із зазначенням методу, шляху, статус-коду та часу виконання; `error.middleware.ts` фіксує необроблені помилки через `logger.error` із повним об'єктом помилки; `email.service.ts` логує події, пов'язані з відправкою листів; `main.ts` записує інформацію про успішний старт сервера. Використання структурованого JSON-формату спрощує агрегацію, фільтрацію та пошук у логах у продакшн-середовищі, що є зручнішим і надійнішим порівняно зі звичайним текстовим форматуванням.

2.5 Реалізація клієнтської частини та вибір бібліотек інтерфейсу

Клієнтська частина платформи реалізована як односторінковий застосунок на базі React 19 зі збирачем Vite. Вибір React обумовлений насамперед тим, що компонентна модель бібліотеки природно відображає ієрархічну структуру інтерфейсу платформи: сторінка каталогу складається з фільтрів, карток просторів та пагінації, кабінет власника – з таблиць бронювань, графіків завантаженості та форм управління тарифами, і кожен із цих елементів є ізольованим компонентом із власною логікою та станом. Vite обрано замість традиційного Create React App через принципово іншу модель розробки: замість повного бандлінгу перед стартом Vite використовує нативні ES-модулі браузера, що забезпечує миттєвий старт dev-сервера та оновлення конкретного модуля у браузері за мілісекунди незалежно від розміру проєкту.

Кодова база організована у папці `client/src/` без окремих папок `hooks/`, `stores/` або `types/` – вся інфраструктурна логіка зосереджена у папці `lib/`. Там знаходиться Axios-екземпляр з `interceptors`, контексти автентифікації та теми, конфігурація TanStack Query, Socket.IO клієнт та окремі файли `per-resource` для API-запитів. Сторінки у папці `pages/` згруповані за функціональними доменами: `auth` містить п'ять сторінок автентифікації, `bookings` – список, деталі та створення бронювань, `coworkings` – список, карту та детальну сторінку простору, `teams` – кімнату команди та прийняття запрошень. Окремі секції виділені для кожної ролі: `owner`, `manager`, `reception` та `admin`. Папка `components/` містить бібліотеку UI-примітивів у підпапці `ui/`, а також компонент `ProtectedRoute` для захисту маршрутів, `Layout` для загальної розмітки та `AppErrorBoundary` для перехоплення критичних помилок рендерингу.

Маршрутизація реалізована через React Router версії 7.2.0. Захист маршрутів побудований через компонент `ProtectedRoute` [30], що отримує масив дозволених ролей як пропс та перевіряє поточного користувача через хук `useAuth`. При відсутності автентифікації відбувається редирект на `/auth/login`, при невідповідній ролі – на `/landing`. Такий підхід централізує логіку доступу в одному місці та дозволяє декларативно описувати права прямо у визначенні маршрутів.

Глобального стейт-менеджера на зразок Zustand або Redux у проєкті немає – це свідоме рішення. Серверний стан керується TanStack Query версії 5.67.2, що автоматично кешує відповіді, інвалідує кеш при мутаціях та повторює запити при мережевих збоях. Стан автентифікації зберігається у власному authStore через plain-object з методами над localStorage, обгорнутий у React Context для доступу з будь-якого компонента. UI-стан – відкрита модалка, активний фільтр, поточна вкладка – зберігається локально у компонентах через useState та useReducer.

Усі HTTP-запити виконуються через єдиний Axios-екземпляр у файлі lib/api.ts з двома налаштованими interceptors. Request interceptor автоматично додає заголовок Authorization: Bearer з поточним access-токеном до кожного запиту. Response interceptor перехоплює відповіді зі статусом 401 та виконує автоматичне оновлення токена через POST /auth/refresh з дедуплікацією паралельних запитів через refreshPromise: якщо кілька запитів одночасно отримали 401, лише один виконує рефреш, а решта очікують його завершення та повторюються з новим токеном. Якщо рефреш провалився – authStore.clearAll() видаляє дані сесії. Форми реалізовані через React Hook Form у поєднанні з Zod через zodResolver – той самий патерн для всіх форм застосунку, де Zod-схема з packages/shared/ використовується одночасно сервером для валідації HTTP-запиту та клієнтом для валідації форми.

Для специфічних вимог платформи підібрані окремі бібліотеки. Календар бронювань реалізований через FullCalendar версії 6.1.20 з плагінами daygrid, timegrid та interaction – місячний і тижневий вид з підтримкою кліку на слот для ініціювання бронювання. Карта просторів побудована через Leaflet з react-leaflet та react-leaflet-cluster з опцією chunkedLoading, що розбиває рендеринг кластерів на частини щоб не блокувати головний потік при одночасному відображенні великої кількості точок. Анімації реалізовані через Framer Motion [29], локалізація – через i18next [28], стилізація – через Tailwind CSS 3.4.17. Застосунок зареєстровано як PWA через vite-plugin-pwa з трирівневою стратегією кешування на базі Workbox: precaching статички назавжди [10], NetworkFirst для /api/* з TTL 5 хвилин та максимум 100 записів, CacheFirst для зображень з TTL 7 днів та максимум 200 записів. При недоступності мережі застосунок очікує з'єднання до 10 секунд, після

чого повертає закешовану відповідь, що забезпечує базовий офлайн-режим для користувачів із нестабільним з'єднанням.

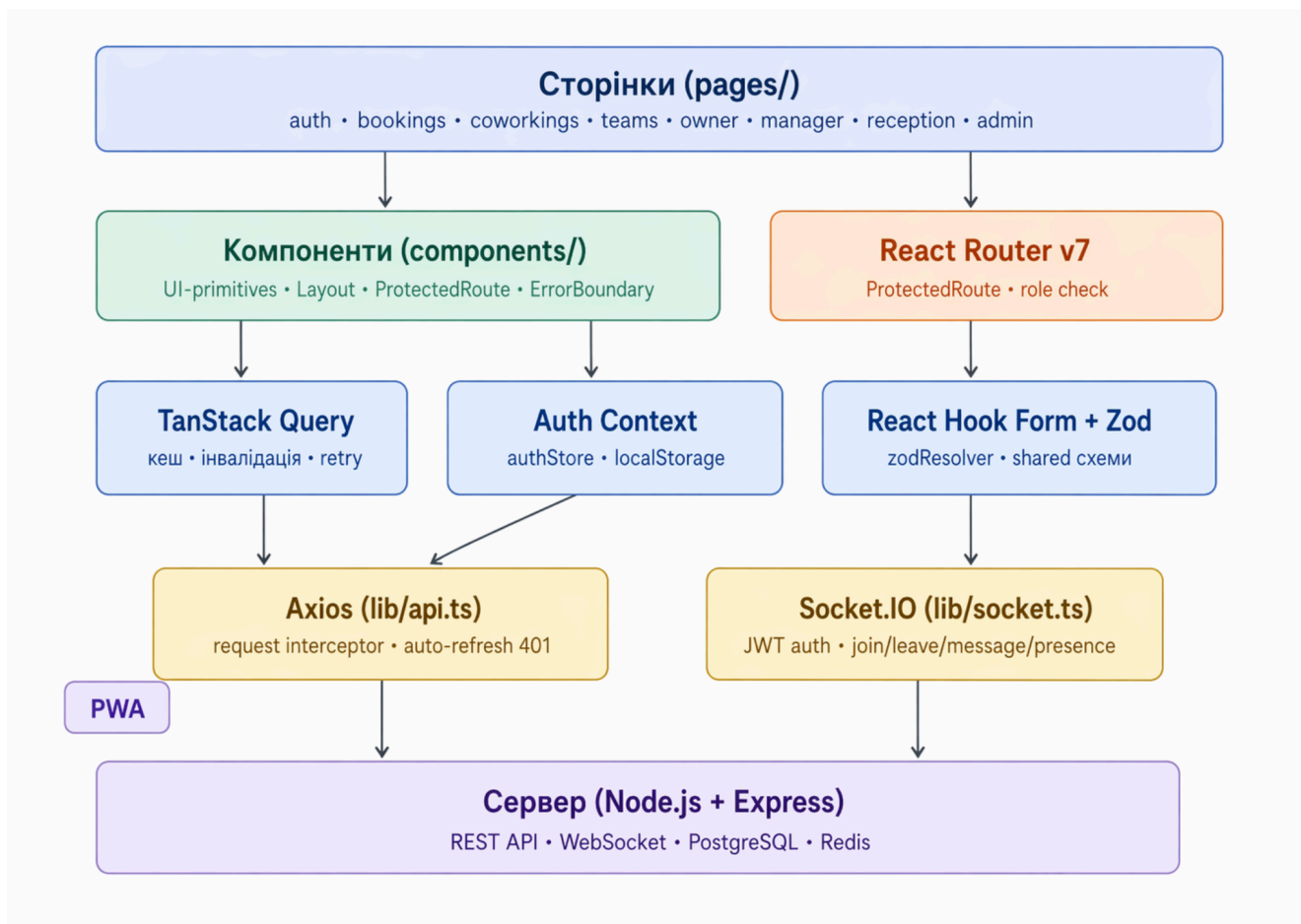


Рис. 2.4 Схема потоку даних клієнтської частини платформи

Наведена схема ілюструє однонаправлений потік даних у клієнтській частині платформи: кожен шар взаємодіє лише з суміжним і не має прямого доступу до шарів нижче через рівень. Сторінки звертаються до компонентів та інфраструктурних утиліт, ті у свою чергу передають запити до транспортного шару, і лише Axios та Socket.IO безпосередньо комунікують із сервером. Така архітектура спрощує відлагодження: помилка у відповіді API локалізується на рівні TanStack Query або Axios interceptor і не поширюється вгору по дереву компонентів неконтрольовано. Розділення каналів REST та WebSocket також є принциповим – зміна стану бронювання через API та отримання повідомлення у командному чаті є незалежними потоками, що не конкурують за ресурси і не блокують один одного при деградації мережі.

2.6 Реалізація безпеки системи

Безпека є наскрізною вимогою для платформи, що зберігає персональні дані користувачів, інформацію про команди та фінансові транзакції бронювань. У системі реалізовано багаторівневий підхід, що охоплює автентифікацію з підтримкою трьох методів двофакторного захисту, рольову модель доступу, захист HTTP-заголовків та валідацію всіх вхідних даних.

Автентифікація побудована на JWT з двома типами токенів [19]. Access-токен має TTL 15 хвилин і передається у заголовок Authorization кожного захищеного запиту. Refresh-токен має TTL 24 години і зберігається не лише на клієнті у localStorage, а й у базі даних у вигляді argon2-хешу у полі User.refreshTokenHash. При оновленні токена старий хеш перезаписується новим – реалізується ротація без накопичення застарілих записів. При logout або скиданні пароля поле refreshTokenHash обнуляється, що миттєво інвалідує refresh-токен на рівні бази даних незалежно від того, чи клієнт видалив його локально. Це унеможливорює використання вкраденого refresh-токену після виходу з системи.

Двофакторна автентифікація реалізована трьома незалежними методами через два послідовних ендпоінти. POST /auth/2fa/setup викликає setupTwoFactor(), що генерує секрет через speakeasy.generateSecret(), зберігає secret.base32 у базі даних та повертає клієнту qrDataUrl [27] у форматі base64 PNG і otpauthUrl для підключення застосунку-автентифікатора. На цьому етапі isTwoFactorEnabled залишається false – двофакторна автентифікація ще не активна. Активація відбувається лише після підтвердження через POST /auth/2fa/verify, де користувач вводить перший згенерований код. Перевірка виконується з допуском window: 1, що дозволяє похибку ± 30 секунд для компенсації розсинхронізації годинника. Тільки після успішної перевірки isTwoFactorEnabled встановлюється у true, генеруються вісім backup-кодів формату XXXX-XXXX, хешуються через argon2 та зберігаються у базі даних, а plaintext-версія повертається користувачу єдиний раз і більше не відтворюється системою. При використанні backup-коду функція

`consumeBackupCode()` фізично видаляє його з масиву через `filter`, а не позначає як використаний – після витрачання всіх восьми кодів масив стає порожнім і `backup-автентифікація` припиняє працювати. Email OTP як третій метод реалізований через шестизначний код, що зберігається у Redis з TTL 10 хвилин та автоматично анулюється після закінчення терміну без додаткової логіки очищення.

Паролі зберігаються виключно у хешованому вигляді через `bcrypt`. Авторизація реалізована за моделлю RBAC з п'ятьма системними ролями: `USER`, `OWNER`, `MANAGER`, `RECEPTION` та `ADMIN`. При реєстрації через публічну форму користувач завжди отримує роль `USER` – це дефолтне значення у `Prisma`-схемі, яке не передається явно і не може бути змінене самим користувачем. Роль змінює виключно адміністратор через захищений ендпоінт `PATCH /admin/users/:id/role` з валідацією через `z.nativeEnum(Role)`. Передбачено захист від самозміни: спроба адміністратора змінити власну роль повертає `badRequest` з відповідним повідомленням. Кожна зміна ролі фіксується в `AuditLog` з метаданими про нову роль. Механізму самостійної подачі заявки на роль `OWNER` не існує – призначення відбувається виключно через пряму дію адміністратора. Додатково у межах команди діє рольова модель `TeamRole` з двома значеннями: `MEMBER` та `MANAGER`, що дозволяє делегувати управління командними бронюваннями без надання системних прав.

Захист HTTP-заголовків реалізований через `Helmet` з кастомною конфігурацією замість стандартного набору за замовчуванням [26]. Заголовок `Strict-Transport-Security` встановлює `max-age` на 31536000 секунд з директивами `includeSubDomains` та `preload`, що примушує браузер використовувати виключно HTTPS протягом року навіть при прямому введенні HTTP-адреси. `Cross-Origin-Resource-Policy` зі значенням `cross-origin` дозволяє завантаження ресурсів з інших походжень. `Content-Security-Policy` налаштовано вручну: `default-src` та `script-src` обмежені значенням `self` з додатковим `unsafe-inline` для підтримки Vite та React без механізму `nonce`, `img-src` дозволяє `data URI` та зовнішні HTTPS та HTTP джерела для зображень коворкінгів, `object-src` та `frame-ancestors` заблоковані через `none`, що унеможливорює вбудовування сторінок у `iframe` та запобігає `clickjacking`. Заголовок

X-Powered-By вимкнено вручну через `app.disable`, що приховує інформацію про серверний стек від потенційних зловмисників.

Валідація вхідних даних реалізована через Zod-схеми на кожному ендпоінті через `middleware validateBody` до потрапляння запиту у бізнес-логіку. Схеми зберігаються у папці `validators/` сервера та частково у спільному пакеті `packages/shared/`, що забезпечує однакові правила валідації між сервером і клієнтом. При невідповідності даних схемі `middleware` повертає структуровану відповідь зі статусом 400 та масивом `issues` з деталями кожного порушення, що дозволяє клієнту відображати конкретні повідомлення про помилки без додаткових запитів. Захист від автоматизованих атак перебору на ендпоінти автентифікації реалізований через `rate-limiter-flexible` у поєднанні з Redis: п'ять невдалих спроб входу протягом 60 секунд блокують IP-адресу на 120 секунд, п'ять невдалих спроб підтвердження 2FA – на 300 секунд.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-ПЛАТФОРМИ

3.1 Загальна архітектура системи та структура бази даних

Розроблена платформа функціонує як єдина цифрова екосистема, що об'єднує п'ять типів учасників із принципово різними сценаріями взаємодії. Звичайний користувач із роллю USER шукає коворкінг через каталог або карту, отримує персоналізовані рекомендації на основі своїх інтересів, бронює конкретну зону, приєднується до команди або створює власну, спілкується з учасниками у командному чаті та отримує QR-квиток для check-in. Власник простору з роллю OWNER керує профілем свого коворкінгу, визначає перелік зон та їхні тарифи, налаштовує теги спільноти для алгоритму рекомендацій та переглядає статистику завантаженості. Менеджер із роллю MANAGER підтверджує або відхиляє бронювання, керує розкладом доступності зон та може скасовувати записи з повідомленням користувача. Працівник ресепшн із роллю RECEPTION фіксує фактичний прихід відвідувача через check-in та завершення сеансу через check-out, що замикає повний операційний цикл бронювання. Адміністратор платформи з роллю ADMIN має розширений доступ до всіх ресурсів системи, призначає ролі користувачам та переглядає аудит-лог критичних дій.

Взаємодія між усіма учасниками відбувається через два незалежних канали. Основний – REST API, що обробляє всі CRUD-операції: реєстрацію, пошук, бронювання, управління командою, платежі та сповіщення. Допоміжний – WebSocket через Socket.IO, що забезпечує командний чат у реальному часі з підтримкою подій join, leave, message та presence. Такий розподіл є принциповим архітектурним рішенням: зміна статусу бронювання та повідомлення у чаті є незалежними потоками, що не конкурують за ресурси навіть при одночасному навантаженні.



Рис. 3.1 Загальна архітектура системи

База даних системи містить 25 моделей, організованих навколо кількох ключових бізнес-сутностей та зв'язків між ними. Центром усієї структури є модель User, від якої відходять зв'язки до Profile з персональними даними та інтересами, до Booking як ініціатора бронювань, до Team через TeamMember як учасника командної взаємодії та до Payment як платника. Модель Coworking пов'язана з User через поле ownerId і є контейнером для моделей Zone, що описують конкретні приміщення, та AvailabilitySlot, що визначають розклад роботи кожної зони по днях тижня. Саме перевірка AvailabilitySlot є першим кроком при створенні будь-якого бронювання – система звіряє запитований часовий проміжок із наявним розкладом до того, як зберегти запис у таблиці Booking.

Модель Booking є центральним вузлом операційного циклу платформи. Вона зв'язує User, Coworking та Zone в одному записі і проходить через сім статусів протягом свого життєвого циклу: **AWAITING_PAYMENT** після створення, **CONFIRMED** після оплати, **CHECKED_IN** після реєстрації прибуття через

ресепшн, COMPLETED після check-out, або CANCELLED чи REJECTED при скасуванні з будь-якого боку. Поле qrToken генерується при підтвердженні і використовується для ідентифікації відвідувача при check-in. При командному бронюванні Booking додатково пов'язується з Team, а кожен учасник отримує власний запис у TeamBookingMember зі своїм унікальним QR-токеном, що дозволяє кожному члену команди незалежно зареєструвати прихід.

Алгоритм рекомендацій за інтересами спирається на три моделі: InterestCategory як довідник категорій, UserInterest як зв'язок між користувачем і його інтересами та CoworkingInterest як зв'язок між коворкінгом і його тегами спільноти. При відображенні каталогу recommendations.service.ts підраховує кількість збігів між інтересами поточного користувача та тегами кожного коворкінгу, після чого ранжує результати за рівнем відповідності. Чим більше спільних тегів – тим вище позиція простору у персоналізованій видачі.

Командна взаємодія описується чотирма пов'язаними моделями. Team зберігає команду та її власника. TeamMember фіксує всіх учасників із їхніми ролями MEMBER або MANAGER у межах команди. TeamInvite реалізує запрошення через email та одноразовий токен, що анулюється після прийняття або закінчення терміну дії. Message зберігає повідомлення командного чату з прив'язкою до команди та автора і є персистентним сховищем для відображення історії листування при повторному вході в кімнату команди.

ER-діаграма вебсистеми для керування коворкінг-просторами

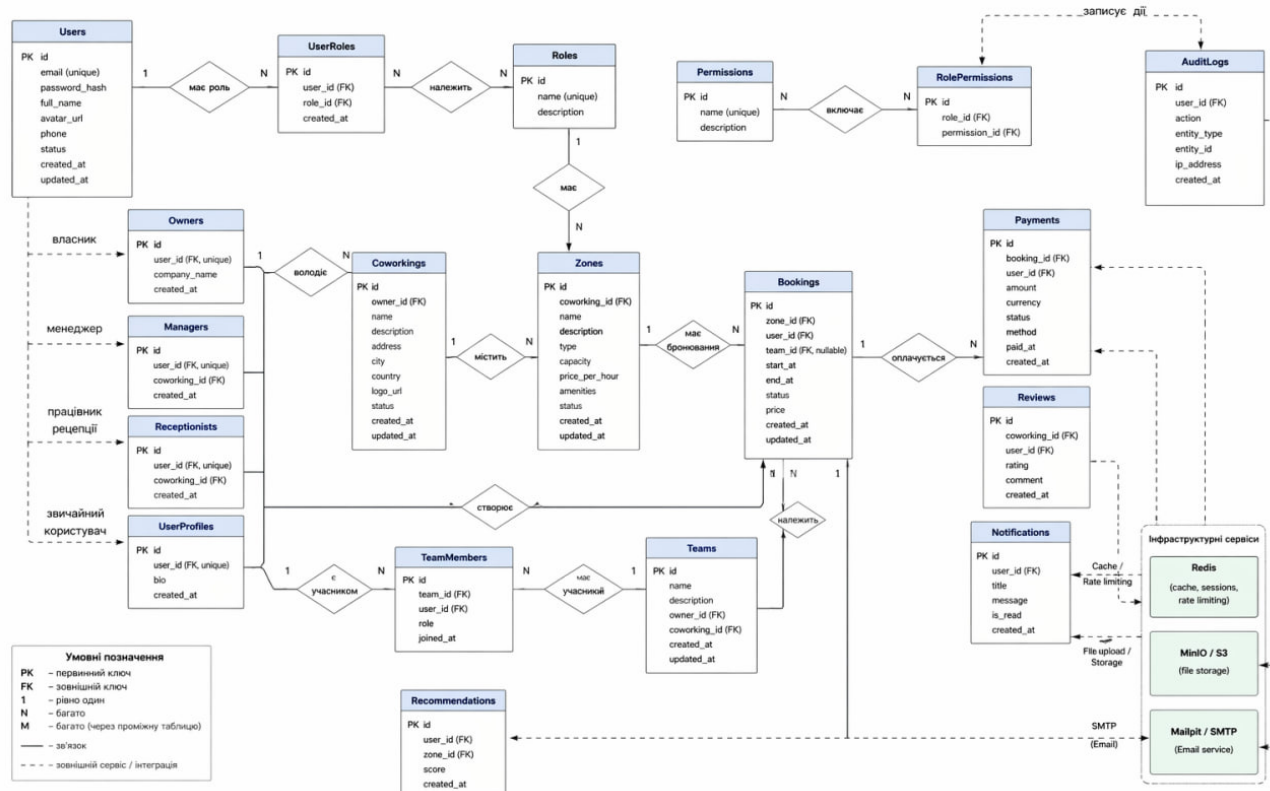


Рис. 3.2 Структура бази даних (ER-діаграма)

Для збереження файлів – фотографій коворкінгів та згенерованих матеріалів – використовується MinIO як S3-сумісне об'єктне сховище [25]. Сервер взаємодіє з MinIO через офіційний SDK, зберігаючи бінарні файли окремо від реляційної бази даних і записуючи лише URL до відповідного поля моделі. Це забезпечує масштабованість файлового сховища незалежно від основної БД та спрощує міграцію на хмарний S3 при необхідності без зміни серверного коду.

3.2 Реалізація модуля автентифікації та двофакторного захисту

Модуль автентифікації є першою точкою взаємодії будь-якого користувача з платформою і реалізований як послідовність чітко розмежованих сценаріїв: реєстрація, вхід, вхід із двофакторним підтвердженням та відновлення доступу. Кожен із цих сценаріїв є самостійним потоком з власними ендпоінтами, валідацією та обробкою помилок, що дозволяє тестувати та розвивати їх незалежно.

Реєстрація нового користувача відбувається через публічну форму на сторінці /auth/register. Користувач вводить ім'я, електронну пошту та пароль, після

чого дані валідуються на клієнті через React Hook Form із Zod-схемою зі спільного пакету `packages/shared/` ще до надсилання запиту на сервер. При успішній клієнтській валідації виконується `POST /auth/register`, де сервер повторно перевіряє ті самі Zod-схеми, хешує пароль через `bcrypt` та створює запис у таблиці `User` із роллю `USER` за замовчуванням [5]. Після реєстрації користувач одразу отримує `access` та `refresh` токени і перенаправляється до особистого кабінету без додаткових кроків верифікації. Обліковий запис стає активним одразу після успішного `POST`-запиту.

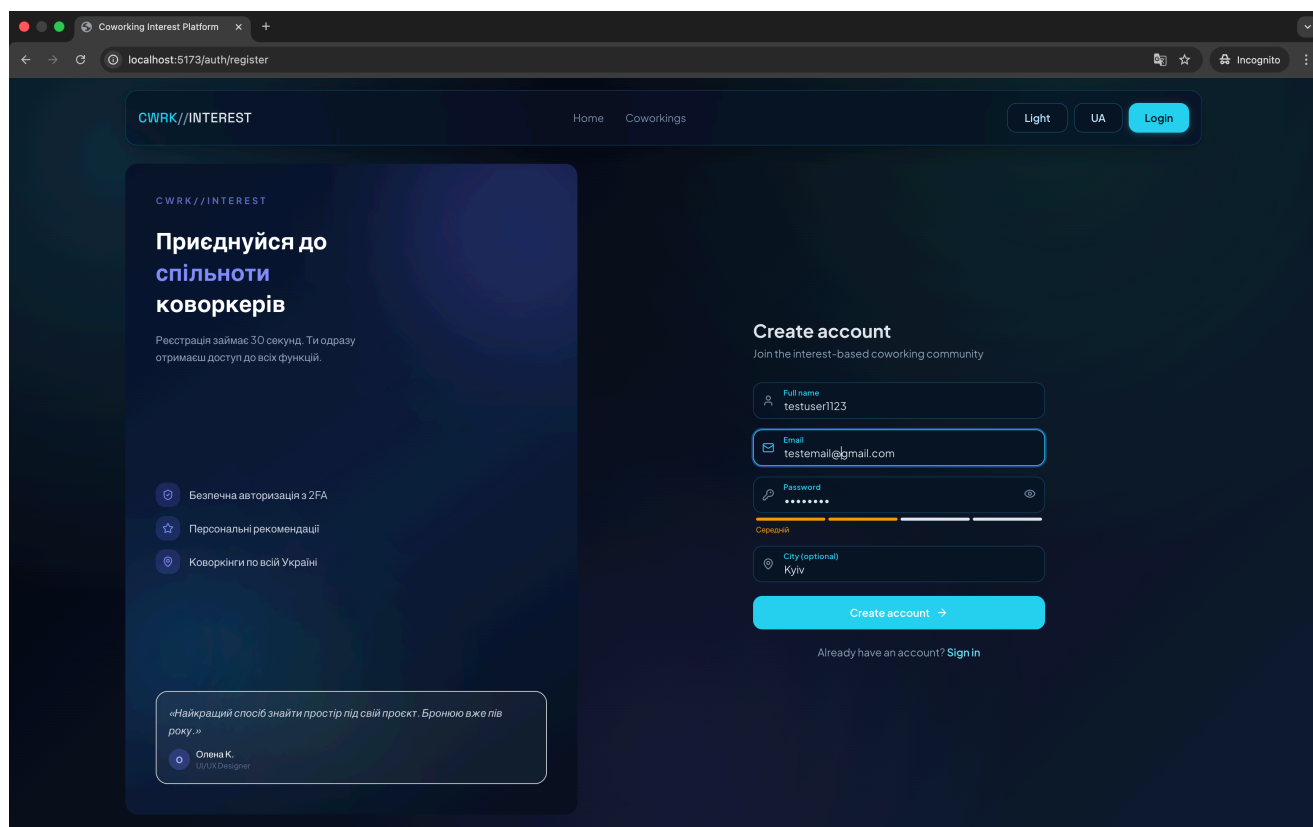


Рис. 3.3 Сторінка реєстрації CWRK//INTEREST

Вхід до системи реалізований через `POST /auth/login`. При отриманні запиту сервер спочатку перевіряє наявність користувача з вказаним email, потім порівнює введений пароль із збереженим `bcrypt`-хешем. До виконання цих перевірок спрацьовує `loginLimiter` – якщо IP-адреса вичерпала ліміт спроб, запит відхиляється на рівні `rate-limiting` ще до звернення до бази даних. При успішній перевірці пароля сервер перевіряє поле `isTwoFactorEnabled`. Якщо двофакторна автентифікація вимкнена – одразу повертається пара `access` та `refresh` токенів, і користувач потрапляє до кабінету. Якщо ввімкнена – повертається проміжна відповідь з

ознакою необхідності підтвердження, клієнт перенаправляє на сторінку введення коду, і повноцінні токени видаються лише після успішного підтвердження через відповідний метод 2FA.

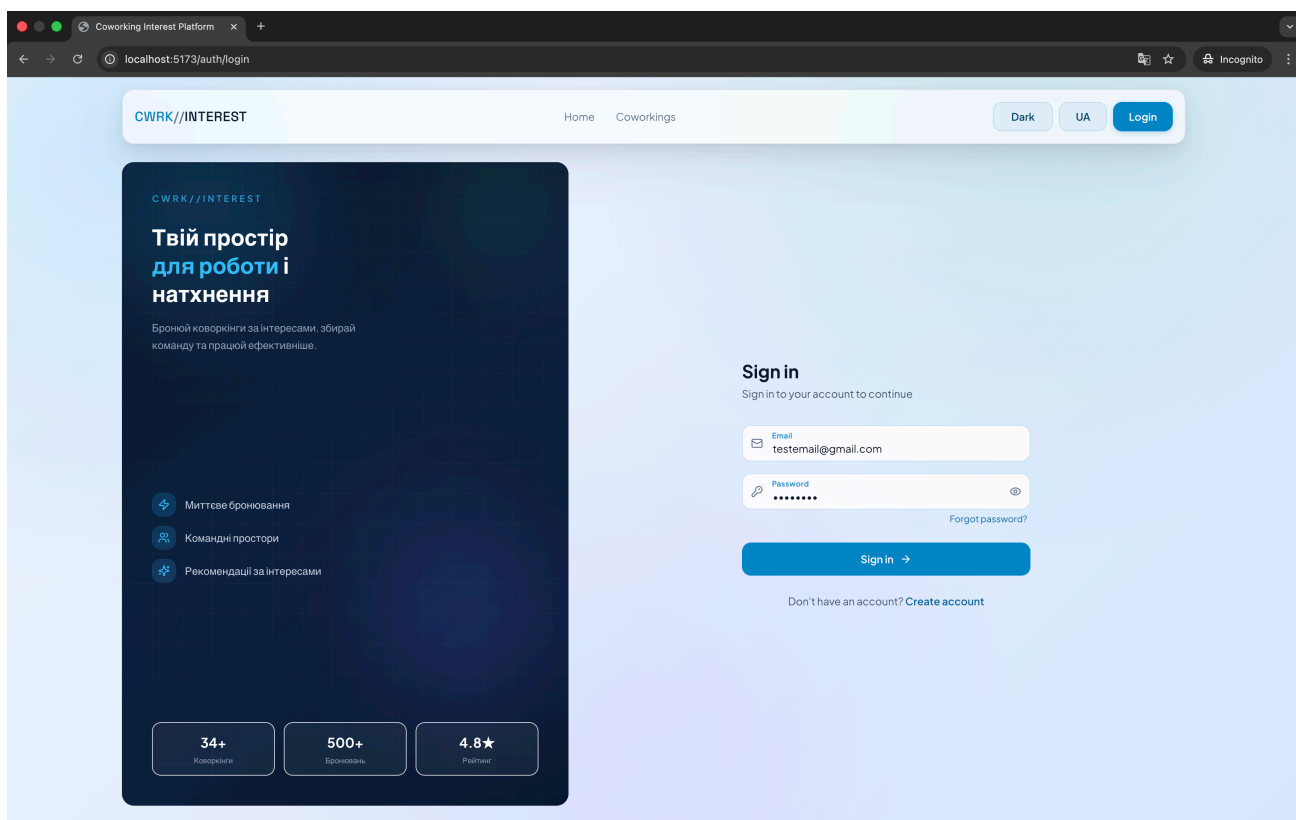


Рис. 3.4 Сторінка входу CWRK//INTEREST

Налаштування двофакторної автентифікації доступне в особистому кабінеті користувача через окрему секцію безпеки. Процес складається з двох обов'язкових кроків. На першому кроці користувач натискає кнопку активації, після чого клієнт виконує `POST /auth/2fa/setup`. Сервер генерує TOTP-секрет через `speakeasy.generateSecret()`, зберігає `secret.base32` у базі даних при цьому залишаючи `isTwoFactorEnabled` у значенні `false`, та повертає `qrDataUrl` – зображення QR-коду у форматі `base64 PNG`. Клієнт відображає цей QR-код, який користувач сканує у застосунку Google Authenticator або Authy. На другому кроці користувач вводить перший згенерований шестизначний код для підтвердження що сканування пройшло успішно. `POST /auth/2fa/verify` перевіряє код з допуском ± 30 секунд для компенсації розсинхронізації годинника, і лише після цього встановлює `isTwoFactorEnabled` у `true`, генерує вісім backup-кодів формату `XXXX-XXXX`,

хешує їх через argon2 та повертає у plaintext єдиний раз. Користувач має зберегти ці коди самостійно – система більше ніколи їх не відображає.

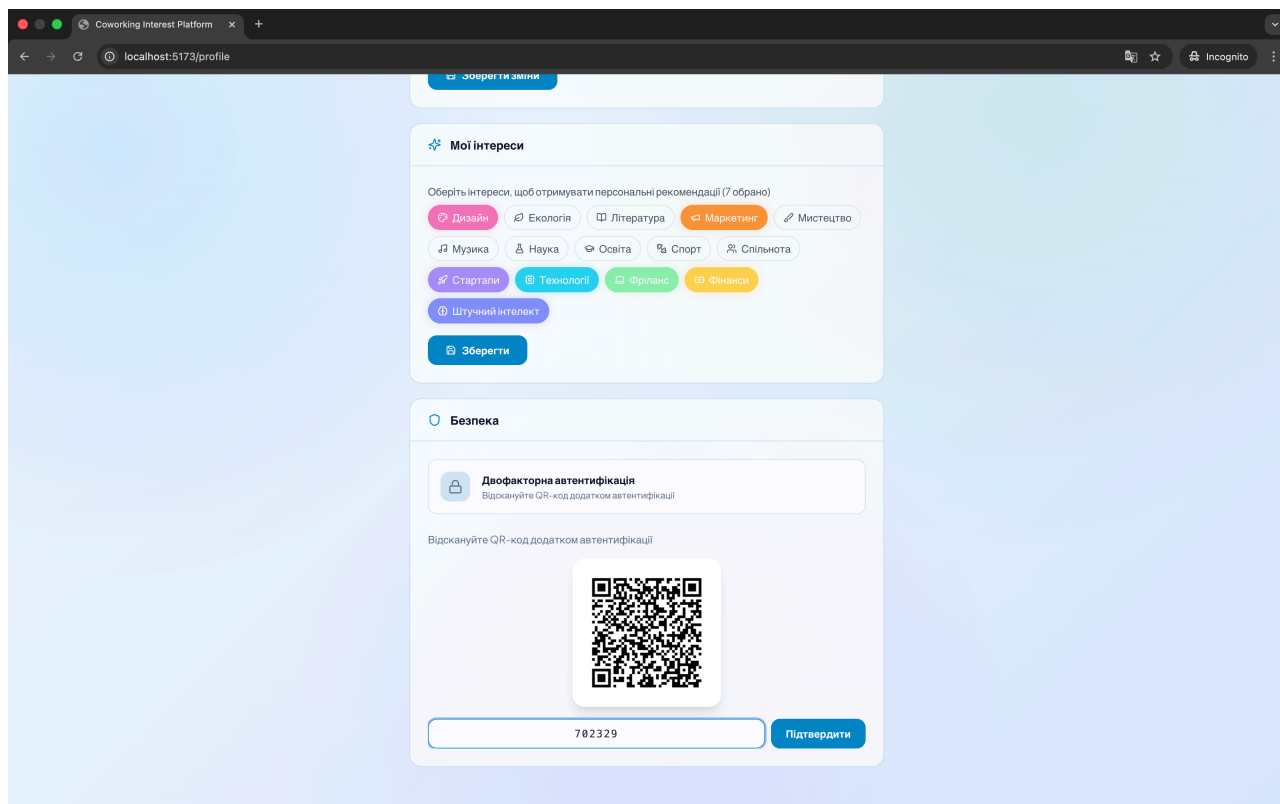


Рис. 3.5 Сторінка налаштування 2FA з QR-кодом

При вході з увімкненою 2FA користувач після введення правильного пароля бачить форму підтвердження, де обирає один із трьох методів. TOTP – введення поточного шестизначного коду з застосунку-автентифікатора, що змінюється кожні 30 секунд. Email OTP – запит на надсилання шестизначного коду на електронну пошту, де код зберігається у Redis з TTL 10 хвилин та анулюється після використання. Вакір-код – введення одного з восьми зарезервованих кодів формату XXXX-XXXX, що фізично видаляється з масиву після використання. Незалежно від обраного методу, twoFactorLimiter обмежує кількість спроб підтвердження п'ятьма за 60 секунд з блокуванням на 300 секунд при перевищенні, що суттєво ускладнює автоматизований перебір кодів.

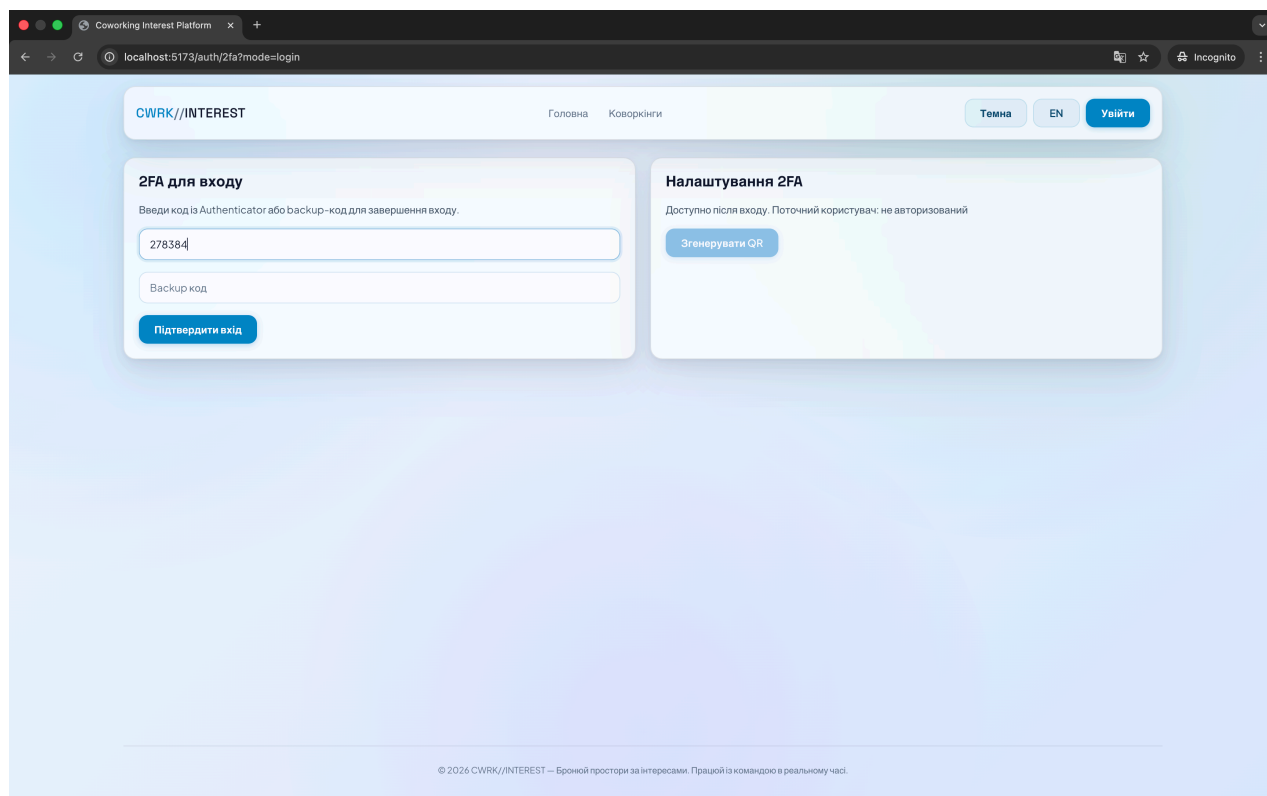


Рис. 3.6 Сторінка введення 2FA-коду при вході

Відновлення пароля реалізоване через двокроковий flow з одноразовим токеном. Користувач вводить свій email на сторінці `/auth/forgot-password`, після чого на вказану адресу надходить лист від платформи CWRK//INTEREST із кнопкою «Скинути пароль» та резервним посиланням у текстовому форматі. Посилання містить унікальний UUID-токен і є дійсним протягом однієї години з моменту надсилання. При переході за посиланням користувач потрапляє на сторінку `/auth/reset-password`, де вводить новий пароль та його підтвердження з мінімальною довжиною вісім символів. Після успішного скидання сервер хешує новий пароль через `bcrypt` та обнуляє поле `refreshTokenHash`, що миттєво інвалідує всі активні сесії – навіть якщо зломисник мав доступ до старого refresh-токену, він стає недійсним відразу після зміни пароля.

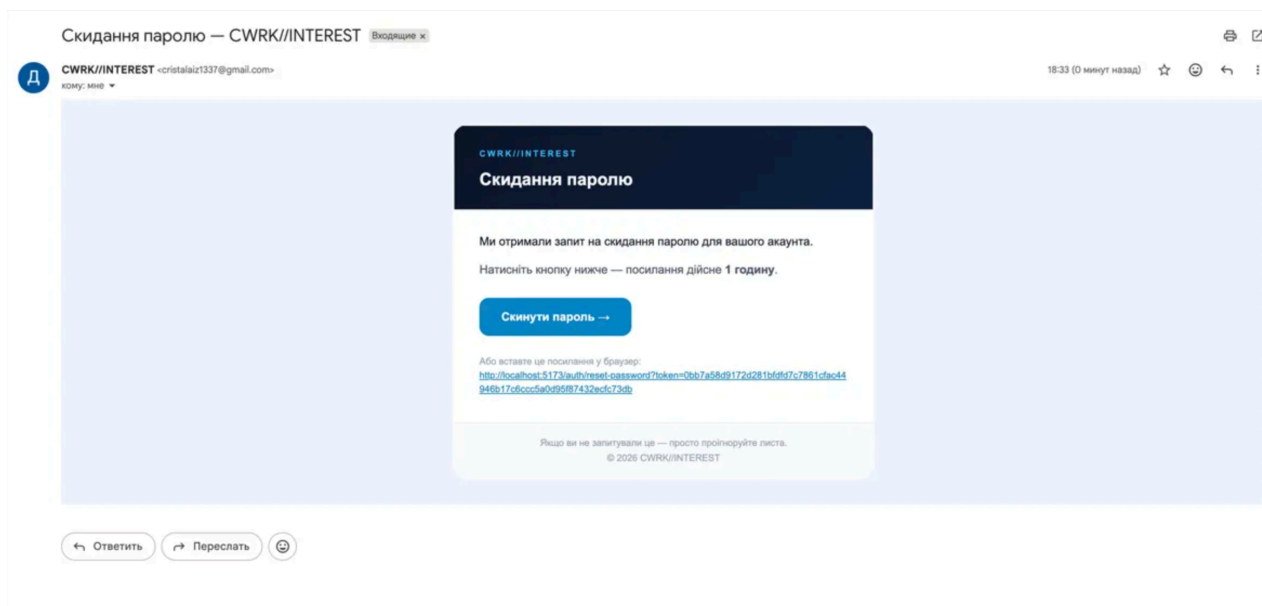


Рис. 3.7 Email зі скиданням паролю від CWRK//INTEREST

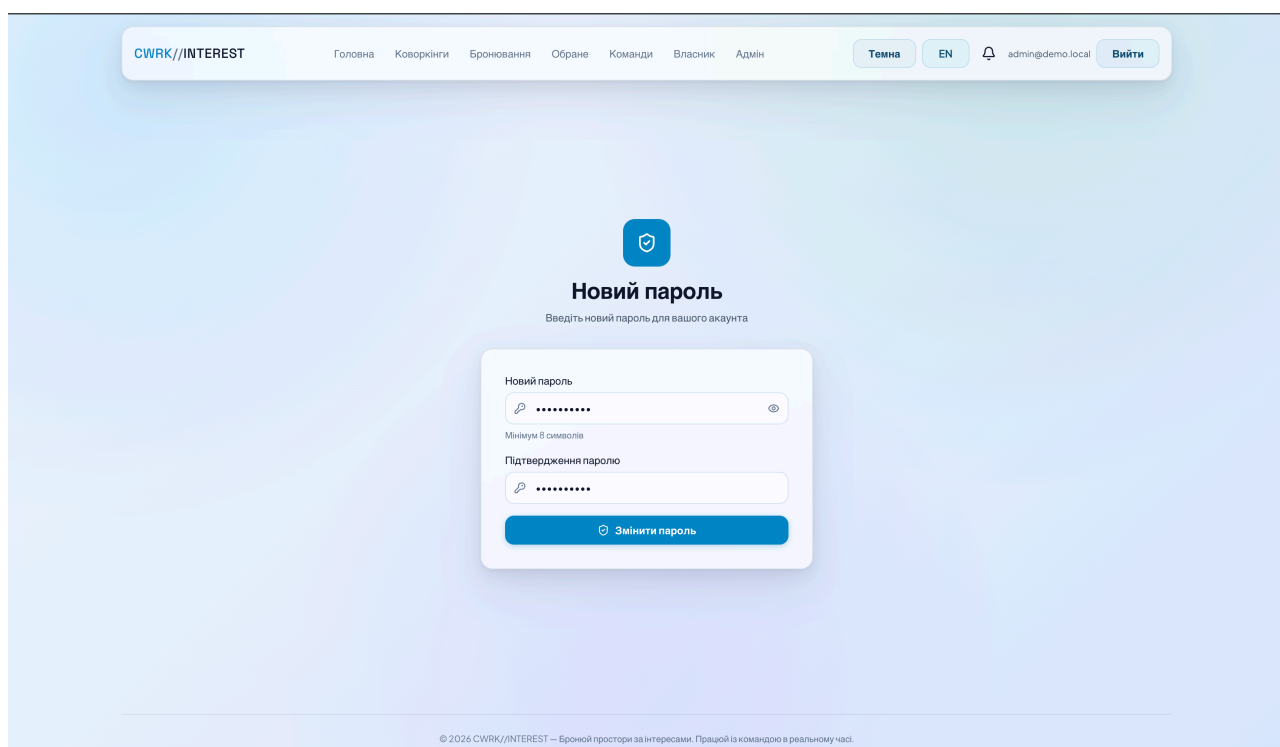


Рис. 3.8 Сторінка введення нового паролю

Наведені екранні форми демонструють послідовність взаємодії користувача з модулем автентифікації на всіх ключових етапах: від первинної реєстрації до відновлення доступу через захищений email-токен. Реалізований підхід забезпечує баланс між зручністю використання та рівнем захисту: реєстрація без обов'язкової верифікації знижує поріг входу для нових користувачів, тоді як двофакторна

автентифікація та серверна інвалідація токенів при зміні пароля захищають акаунт від несанкціонованого доступу навіть у разі компрометації облікових даних.

3.3 Реалізація каталогу, пошуку та алгоритму рекомендацій за інтересами

Каталог коворкінг-просторів є центральним функціональним модулем платформи з точки зору користувача – саме з нього починається шлях від першого відкриття сайту до бронювання конкретної зони. Сторінка каталогу реалізована як два незалежних режими перегляду через систему вкладок: загальний список All та персоналізована вибірка For you. Перемикання між вкладками не перезавантажує сторінку – TanStack Query кешує обидва запити окремо та інвалідує їх лише при зміні фільтрів або оновленні профілю інтересів.

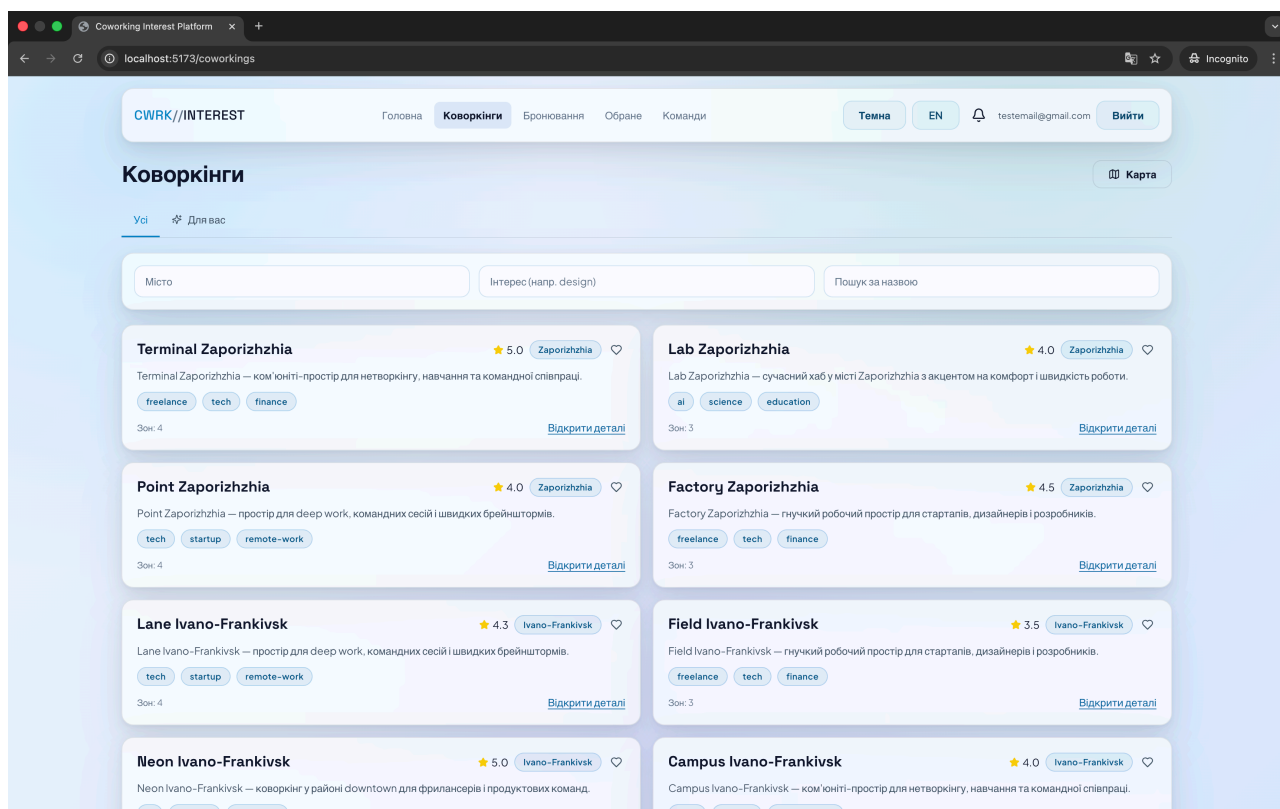


Рис. 3.9 Каталог коворкінгів, вкладка All

Пошук та фільтрація у загальному каталозі реалізовані через три незалежних параметри. Фільтр city звужує вибірку за містом без урахування регістру. Фільтр interest перевіряє наявність обраної категорії в масиві тегів коворкінгу. Текстовий пошук через параметр query використовує розширення pg_trgm PostgreSQL для

нечіткого повнотекстового пошуку по полях name, description та city одночасно, що дозволяє знаходити результати навіть при неточному написанні або опечатках. Всі три фільтри передаються як query-параметри до ендпоінту GET /coworkings і обробляються на сервері через Prisma з динамічно формованим where-об'єктом залежно від наявності кожного параметру.

Карта просторів реалізована як окрема сторінка з принципово іншим набором фільтрів, орієнтованих на географічну прив'язку. Замість текстового пошуку та тегів інтересів тут доступні: вибір міста через chips-елементи, кнопка Near Me для автоматичного визначення геолокації через Geolocation API браузера та вибір радіуса пошуку серед значень 1, 3, 5, 10 та 25 кілометрів. Маркери на карті кластеризуються через react-leaflet-cluster з опцією chunkedLoading, що розбиває рендеринг на частини і не блокує головний потік браузера при одночасному відображенні великої кількості точок.

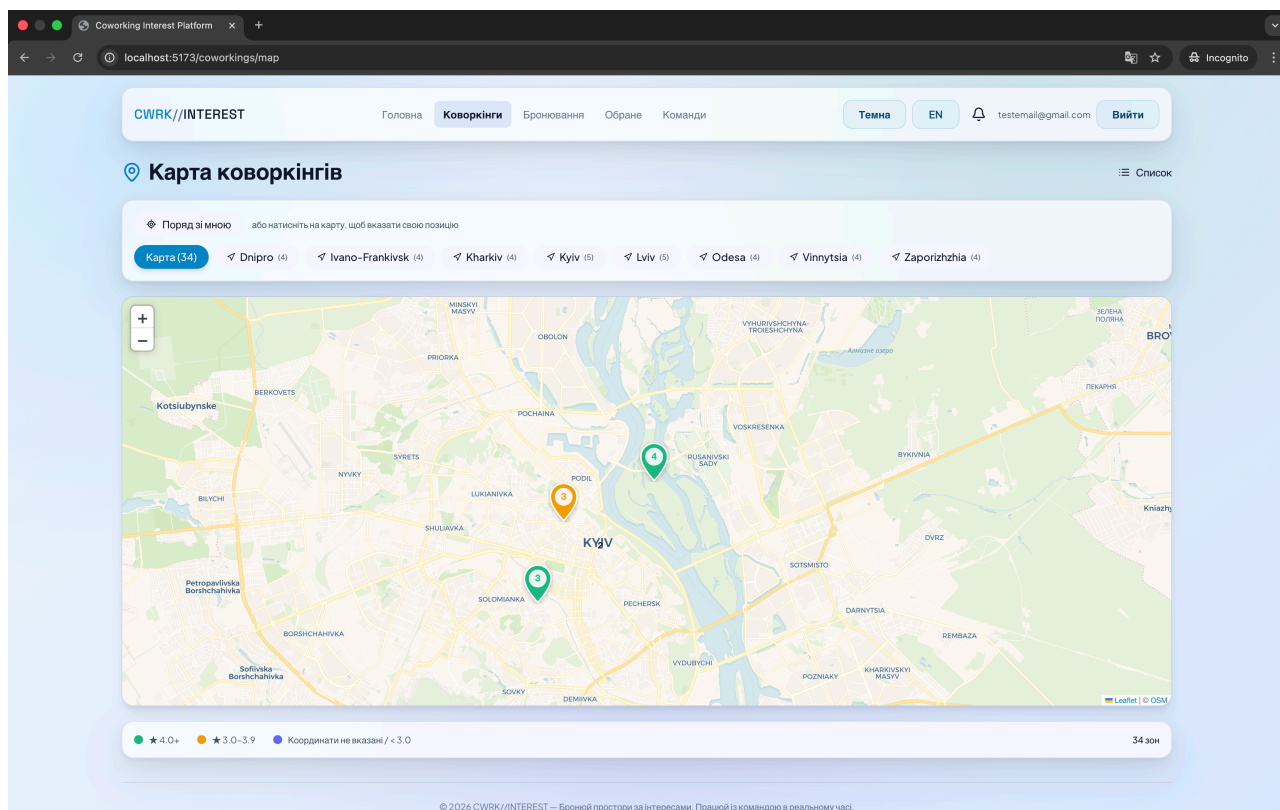


Рис. 3.10 Карта коворкінгів із кластеризацією маркерів

Детальна сторінка коворкінгу відкривається при кліку на картку в каталозі або маркер на карті. Вона містить повну інформацію про простір: назву, місто, адресу, опис, середній рейтинг, теги спільноти, перелік зручностей amenities та

блок Who's Going з відображенням учасників команди що вже забронювали цей простір. Карта локації будується через Leaflet на основі полів latitude та longitude моделі Coworking і відображається лише якщо координати заповнені. Нижче розташований список зон із детальною інформацією по кожній: назва, опис, ціна за годину, місткість та обладнання. Саме тут користувач обирає конкретну зону та ініціює бронювання – для себе або від імені команди. Відгуки з рейтингом відображаються окремим блоком внизу сторінки, і там же знаходиться форма додавання нового відгуку та відповідь власника якщо вона є.

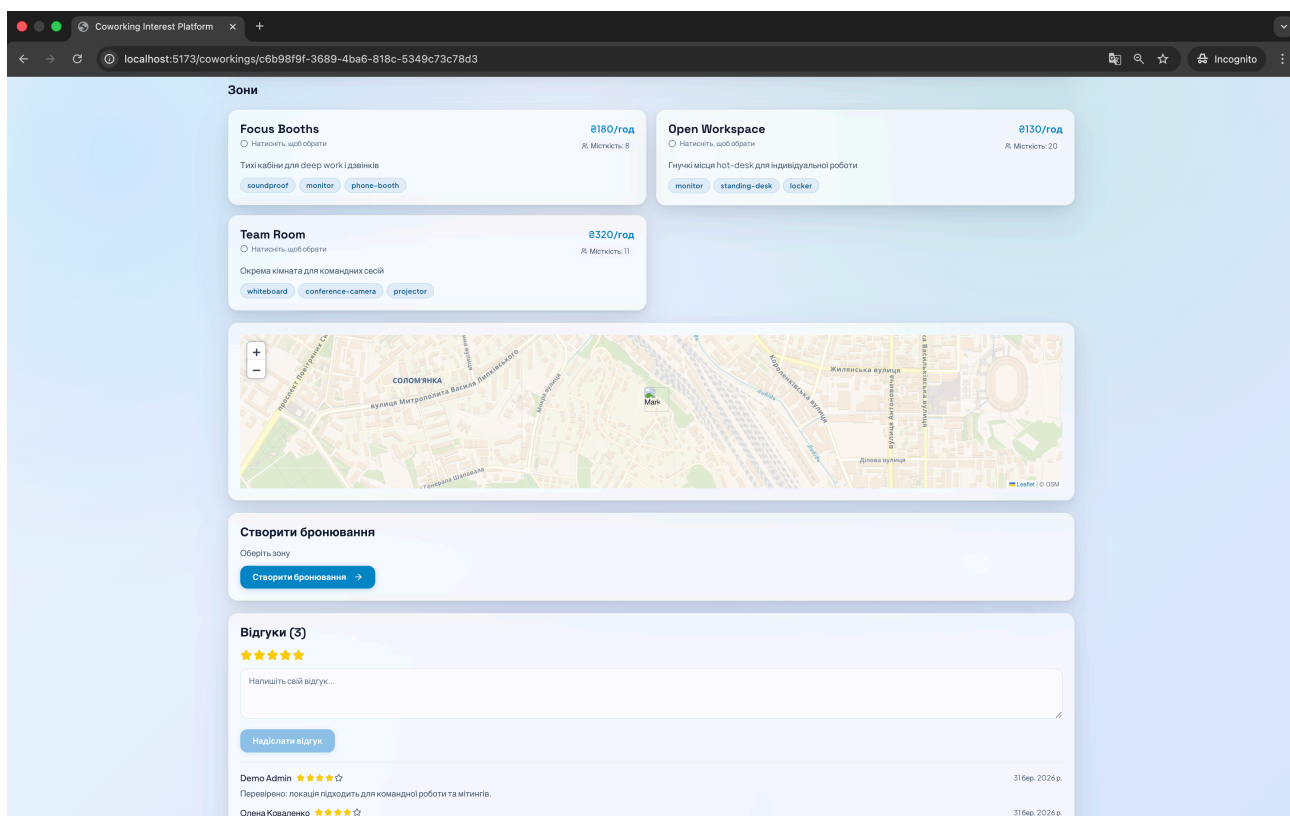


Рис. 3.11 Детальна сторінка коворкінгу із зонами та відгуками

Ключовою унікальною функцією платформи є персоналізована вкладка For you, що реалізує алгоритм підбору просторів на основі інтересів користувача. При першому відкритті вкладки клієнт виконує запит до GET /coworkings/recommendations, який обробляється у recommendations.service.ts. Алгоритм розраховує для кожного коворкінгу числовий показник відповідності – score – за формулою що поєднує три незалежних складових із різними вагами.

Перша складова – jaccard – розраховує збіг між набором інтересів поточного користувача та набором тегів коворкінгу як відношення кількості спільних

категорій до кількості всіх унікальних категорій в обох наборах. Це класичний коефіцієнт Жаккара, що дорівнює нулю при повній відсутності збігів та одиниці при повному збігу. Ця складова має вагу 0.5 у фінальній формулі і є домінуючим сигналом при ранжуванні. Друга складова – collaborative – відображає популярність коворкінгу серед інших користувачів зі схожим профілем інтересів, реалізуючи принцип collaborative filtering: якщо люди з аналогічними інтересами часто бронюють певний простір, він отримує вищий рейтинг відповідності навіть при неповному збігу тегів. Ця складова має вагу 0.2. Третя складова – history – враховує власну історію бронювань поточного користувача з часовим затуханням через функцію $\exp(-\text{daysDiff} / 90)$, де daysDiff – кількість днів від останнього бронювання. Це означає що простори, які користувач відвідував нещодавно, отримують вищий бонус ніж ті що він відвідував рік тому, а з плином часу вплив старих бронювань поступово знижується до нуля. Ця складова має вагу 0.3.

Фінальний score обчислюється за формулою: $\text{score} = \text{round}((0.5 \times \text{jaccard} + 0.2 \times \text{collaborative} + 0.3 \times \text{history}) \times 100)$ і є цілим числом від 0 до 100. Коворкінги сортуються за спаданням score, і перші позиції у вкладці For you займають простори з найвищою сукупною відповідністю профілю користувача. При однаковому score додаткова сортування відбувається за рейтингом відгуків.

Щоб алгоритм рекомендацій міг розрахувати персональний score, користувач має заповнити свій профіль інтересів. Секція інтересів розташована в особистому кабінеті і дозволяє обрати одну або кілька категорій із довідника InterestCategory. Вибрані категорії зберігаються у таблиці UserInterest і стають основою для розрахунку коефіцієнта Жаккара при кожному зверненні до ендпоінту рекомендацій. При зміні інтересів TanStack Query автоматично інвалідує кешований запит рекомендацій, і наступне відкриття вкладки For you показує оновлений ранжований список.

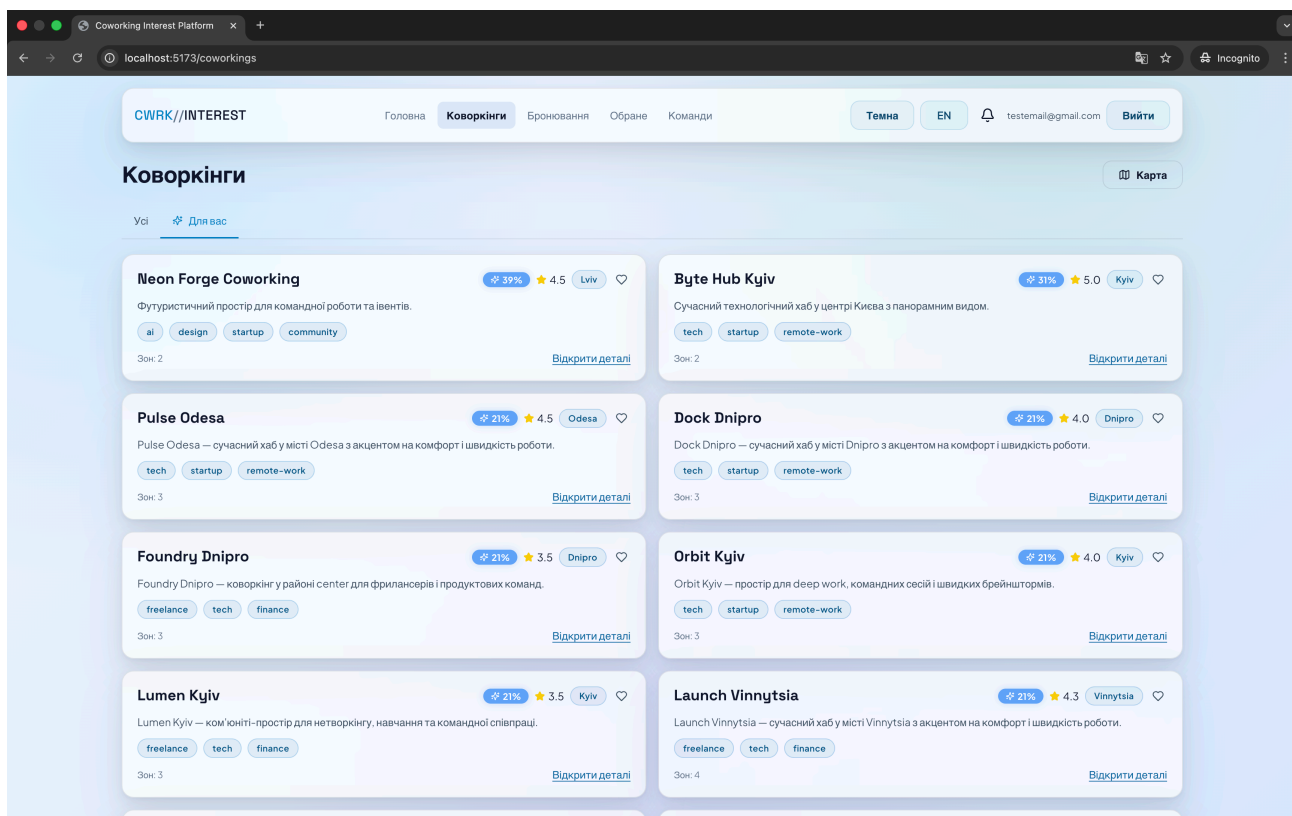


Рис. 3.12 Вкладка For you з персоналізованими рекомендаціями

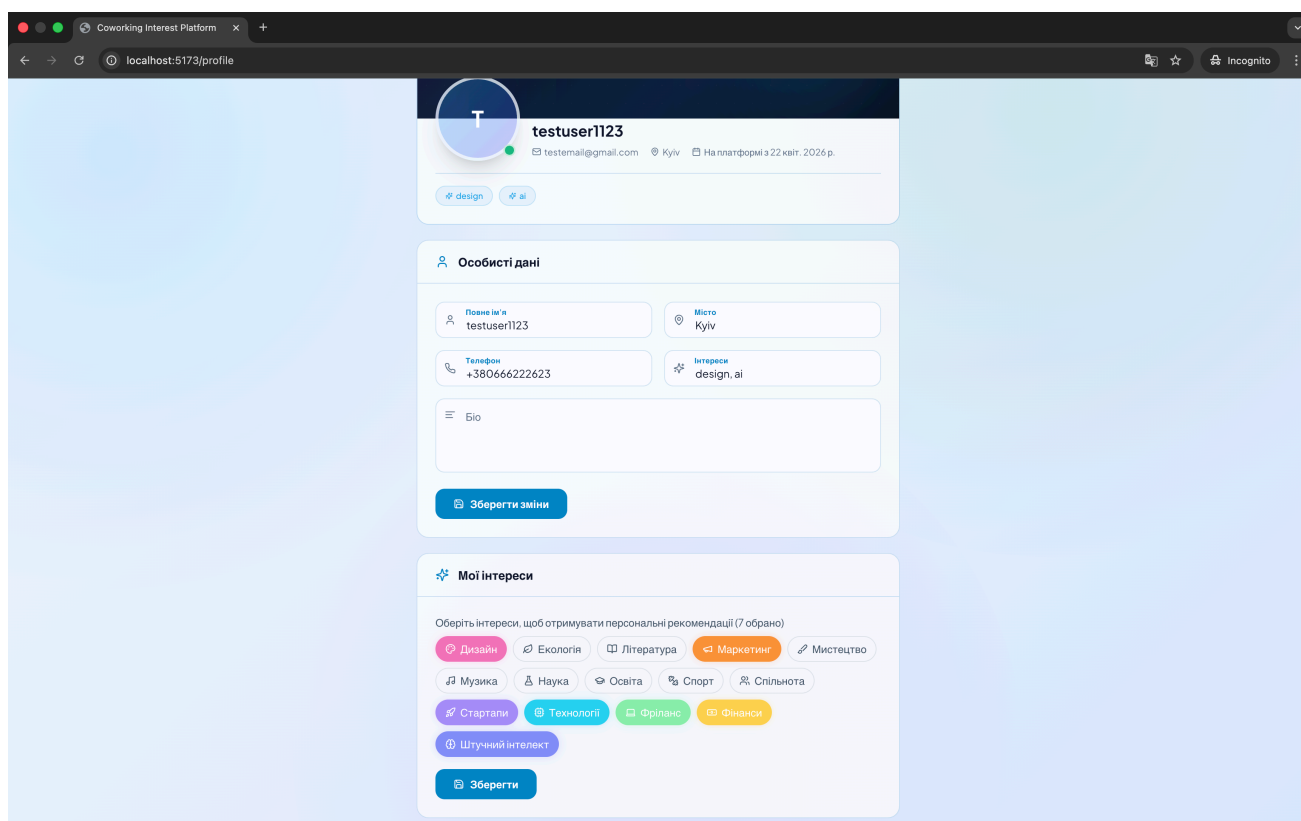


Рис. 3.13 Секція інтересів у профілі користувача

Реалізований підхід до рекомендацій відрізняється від простої фільтрації за тегами тим, що враховує одночасно три сигнали різної природи: декларативні

інтереси користувача, поведінку схожої аудиторії та власну історію взаємодії з платформою. Жодна з проаналізованих у розділі 1 українських платформ не реалізує подібного механізму навіть у спрощеному вигляді, що робить цей модуль ключовою функціональною перевагою розробленої системи.

3.4 Реалізація модуля бронювання зон та управління доступністю

Модуль бронювання є операційним ядром платформи – саме через нього реалізується основний комерційний сценарій взаємодії між користувачем та коворкінг-простором. Весь цикл від вибору зони до фізичного завершення сеансу роботи описується послідовністю статусних переходів бронювання та набором ендпоінтів, що обслуговують кожен крок цього переходу.

Створення бронювання реалізоване як багатокрокова форма на сторінці `/bookings/create`. На першому кроці користувач обирає зону конкретного коворкінгу та вказує чи бронює він для себе особисто або від імені команди – при командному бронюванні доступний список команд, учасником яких є поточний користувач. На другому кроці вводяться дата та часовий проміжок через поля `startAt` та `endAt`, додається опціональна примітка, а також доступні дві допоміжні дії: перевірка наявності вільних місць та попередній розрахунок вартості через `pricing.service.ts`. Третій крок є сторінкою підтвердження де відображається повне `summary`: назва коворкінгу, зона, команда, кількість місць, часовий проміжок, розрахована вартість та примітка. Після натискання кнопки `Book Now` виконується `POST /bookings`.

← Назад

Нові бронювання

1 Зона — 2 Дата та час — 3 Підтвердження

Оберіть зону

- Team Room** 11
Окрема кімната для командних сесій
320/г
- Focus Booths** 8
Тихі кабінки для deep work і дзвінків
180/г
- Open Workspace** 20
Гучні місця hot-desk для індивідуальної роботи
130/г

Бронювання для команди (опційно)
Якщо бронюєте для команди — оберіть її. Кількість місць = кількість учасників.

- Тільки я** 1 місце
- Programming** 1 учасник
- Тестова команда** 2 учасників

Далі →

Рис. 3.14 Багатокрокова форма створення бронювання, крок вибору зони

← Назад

Нові бронювання

1 Зона — 2 Дата та час — 3 Підтвердження

Оберіть дату та час

Початок: 23.04.2026, 12:30 Кінець: 23.04.2026, 15:00

Примітка (необов'язково): Тестова примітка

Перевірити наявність місць Розрахувати ціну

Доступно 20 з 20 місць
Місткість: 20 Зайнято: 0 Вільно: 20

← Назад Далі →

Рис. 3.15 Багатокрокова форма створення бронювання, крок вибору дати та часу

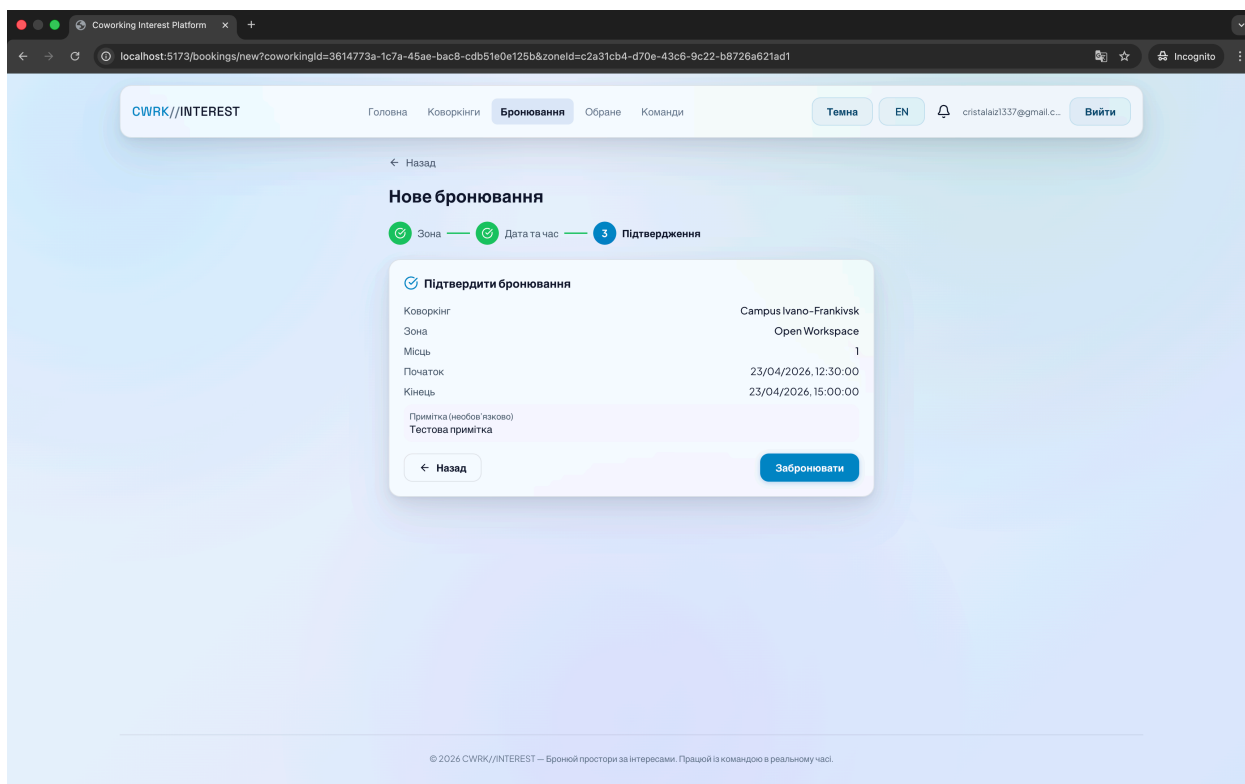


Рис. 3.16 Багатокрокова форма створення бронювання, крок підтвердження

На сервері при отриманні POST /bookings виконується послідовна перевірка кількох умов. Спочатку валідується структура запиту через bookingCreateSchema: обов'язкові поля coworkingId, zoneId, startAt та endAt, опціональні teamId та note. Далі перевіряється що startAt строго менший за endAt. Після цього підтверджується що вказана zoneId дійсно належить вибраному coworkingId, що унеможливорює бронювання зони у чужому коворкінгу навіть при правильному форматі запиту. При командному бронюванні додатково перевіряється що поточний користувач є активним учасником вказаної команди.

Ключовою перевіркою є контроль доступності зони. Сервер виконує пошук усіх існуючих бронювань цієї зони зі статусами PENDING, CONFIRMED або CHECKED_IN та часовим перетином з запитуваним проміжком за умовою $startAt < newEndAt \text{ AND } endAt > newStartAt$. Сума зайнятих місць у знайдених бронюваннях порівнюється з місткістю зони: якщо $occupiedSeats + requestedSeats$ перевищує $zone.capacity$, сервер повертає відповідь зі статусом 409 Conflict. Весь процес перевірки та створення виконується в межах однієї транзакції з рівнем ізоляції Serializable, що мінімізує ризик подвійного бронювання при паралельних запитах від різних користувачів на один і той самий слот.

При успішному проходженні всіх перевірок бронювання створюється одразу зі статусом CONFIRMED, що означає автоматичне підтвердження замовлення без необхідності додаткового ручного погодження з боку адміністратора. Перед створенням запису система перевіряє доступність вибраного робочого місця або кімнати, коректність часових меж бронювання, відсутність конфліктів з іншими активними бронюваннями та відповідність обраного простору заданим параметрам користувача. Після цього сервер виконує кілька додаткових дій, необхідних для подальшої ідентифікації та контролю доступу: генерує унікальний QR-токен для проходження check-in, зберігає службові дані бронювання та прив'язує їх до конкретного користувача.

У разі командного бронювання система додатково створює окремий запис TeamBookingMember для кожного учасника команди. Кожному учаснику призначається власний QR-токен, що дозволяє незалежно проходити ідентифікацію під час відвідування коворкінг-простору. Такий підхід забезпечує зручну організацію групових бронювань, оскільки основне бронювання залишається пов'язаним з ініціатором, а кожен учасник має окремий запис для перевірки доступу. Паралельно сервер інвалідує кеш рекомендацій, щоб наступна видача враховувала оновлену historical-складову, тобто нові дії користувача та зміну історії його бронювань. Завершальним етапом є фіксація події в AuditLog, що дає змогу відстежувати факт створення бронювання, його статус, пов'язаного користувача та службові дії системи. Таким чином, процес підтвердження бронювання охоплює не лише створення основного запису, а й формування QR-ідентифікаторів, оновлення рекомендаційної логіки та журналювання важливих подій для подальшого контролю.

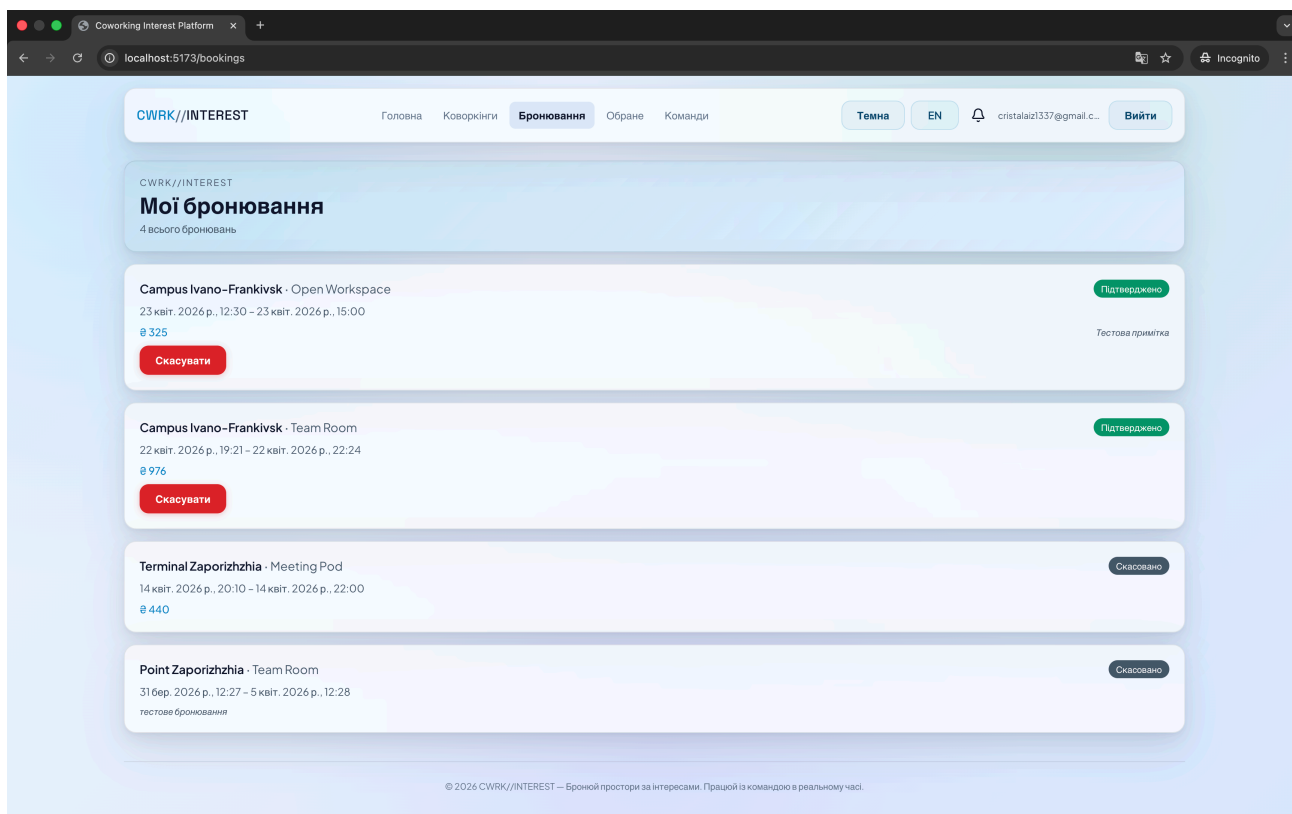


Рис. 3.17 Список бронювань користувача зі статусами

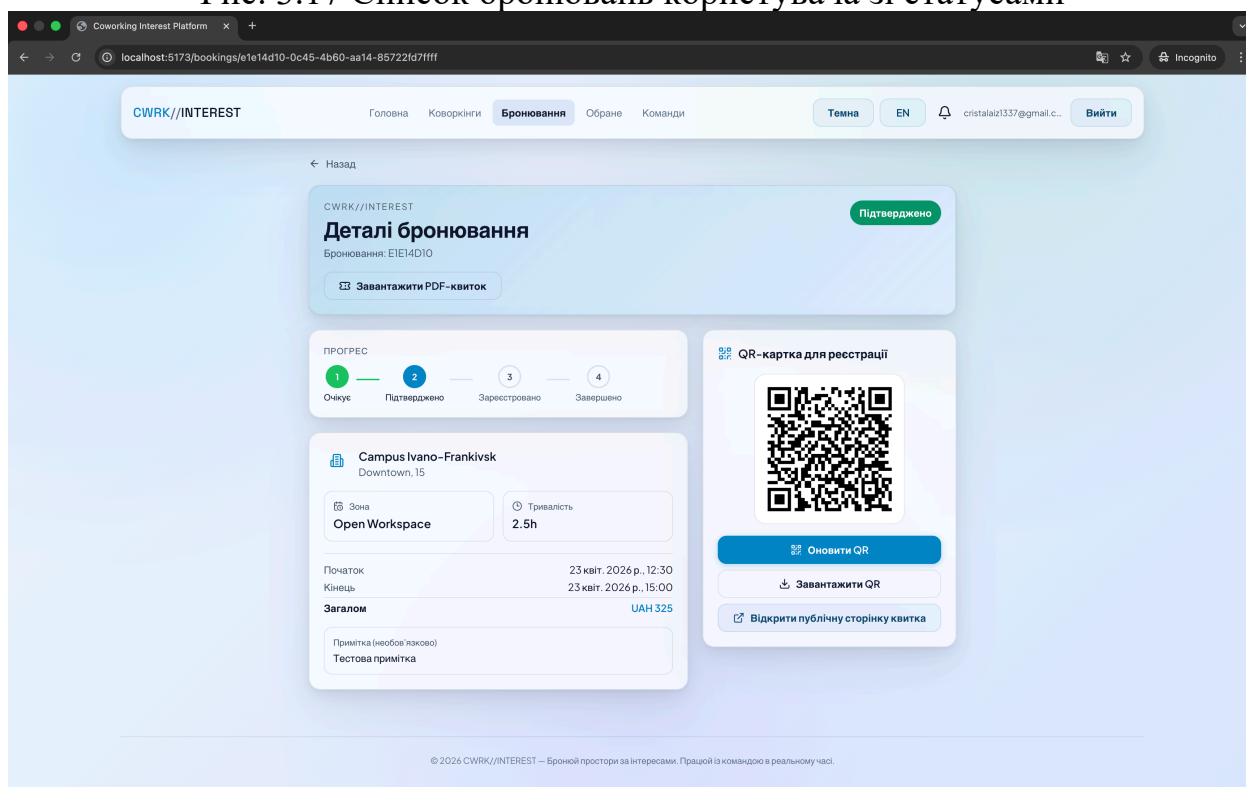


Рис. 3.18 QR-квиток бронювання

Цикл підтвердження фізичного відвідування реалізований через механізм check-in та check-out на сторінці кабінету ресепшн. Сторінка reception-page містить поле введення QR-токену, кнопки Check In та Check Out, а також список поточних

бронювань на сьогоднішній день. Ресепшн вводить або вставляє QR-токен з квитка відвідувача, після чого виконується POST /bookings/check-in або POST /bookings/check-out відповідно. Перехід check-in допускається лише зі статусу CONFIRMED у CHECKED_IN, перехід check-out – лише зі статусу CHECKED_IN у COMPLETED. Ці дії доступні для ролей RECEPTION, MANAGER, OWNER та ADMIN. Камера-сканер у браузері у поточній версії не реалізований – введення токена відбувається вручну або через буфер обміну при копіюванні з мобільного пристрою відвідувача.

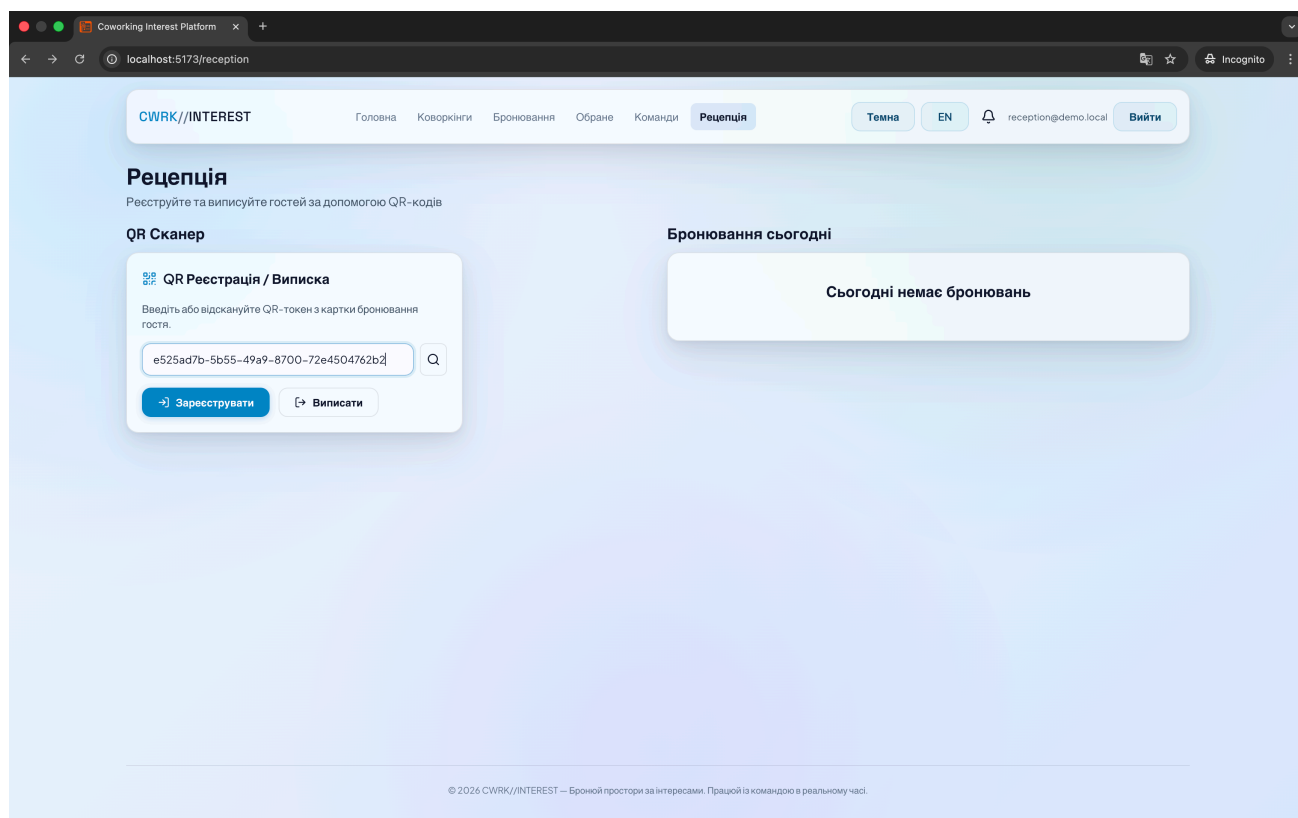


Рис. 3.19 Кабінет ресепшн із полем введення QR-токену

Повний перелік статусних переходів бронювання утворює замкнутий операційний цикл. CONFIRMED є початковим статусом при створенні. Звідси можливі три гілки: перехід у CHECKED_IN через check-in ресепшн, перехід у CANCELLED через скасування з боку користувача, власника або адміністратора, або перехід у REJECTED якщо власник коворкінгу відхиляє запит. Зі статусу CHECKED_IN єдиним можливим переходом є COMPLETED через check-out. Статус PENDING існує у схемі та використовується для сценарію коли власник або адміністратор хоче підтвердити бронювання вручну через confirmBooking, проте у

стандартному flow через публічний ендпоінт бронювання створюється одразу як CONFIRMED, міняючи стадію очікування.

Черга очікування реалізована через модель Waitlist на серверному рівні. При скасуванні бронювання сервер автоматично перевіряє наявність активних записів waitlist для тієї самої зони та часового проміжку і надсилає сповіщення типу WAITLIST_AVAILABLE першому користувачу в черзі за часом реєстрації. Прямого UI для самостійного запису до waitlist у поточній версії клієнтської частини не реалізовано – ця функціональність є присутньою на рівні API та може бути підключена у наступних ітераціях розробки.

Реалізований модуль бронювання забезпечує повний операційний цикл простору від вибору зони до фізичного завершення сеансу, підтримує як індивідуальні так і командні бронювання з окремими QR-квитками для кожного учасника, та захищає від подвійного бронювання через транзакції з рівнем ізоляції Serializable на рівні бази даних.

3.5 Реалізація командної взаємодії, кабінетів ролей та системи сповіщень

Командна взаємодія реалізована через окрему групу сторінок у розділі teams/ клієнтської частини. Користувач може створити команду, вказати її назву та запросити учасників через email або унікальне посилання – кожне запрошення зберігається у моделі TeamInvite як одноразовий токен що анулюється після прийняття. Учасник отримує email із посиланням на сторінку /teams/invite/:token, де підтверджує приєднання. Після входу до команди доступна кімната команди зі списком учасників, спільним календарем бронювань та вбудованим чатом у реальному часі.

Чат реалізований через Socket.IO з персистентним зберіганням повідомлень у моделі Message. При відкритті кімнати клієнт встановлює WebSocket-з'єднання та надсилає подію join з ідентифікатором команди – сервер перевіряє членство користувача і лише після цього підключає його до кімнати. Повідомлення що були

надіслані під час відсутності користувача відображаються при наступному вході завдяки збереженій історії в базі даних. Присутність учасників онлайн відстежується через подію presence.

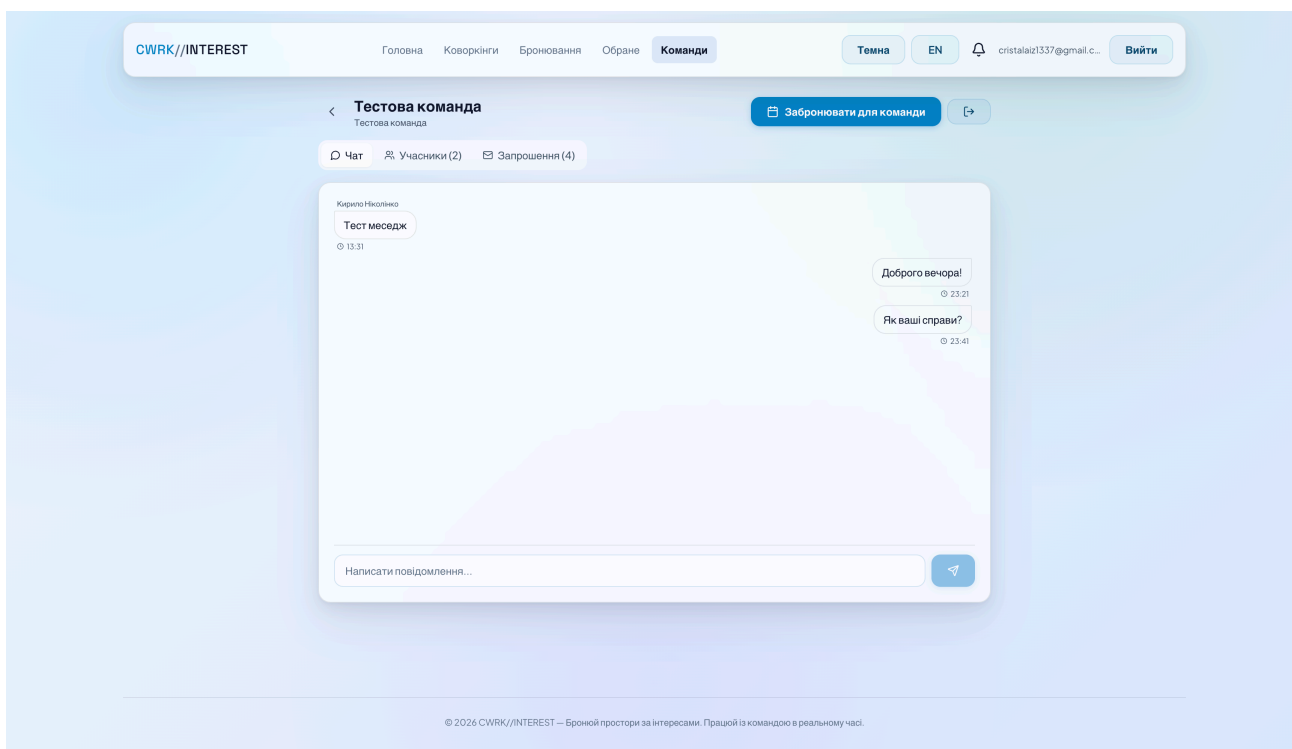


Рис. 3.20 Кімната команди з чатом та списком учасників

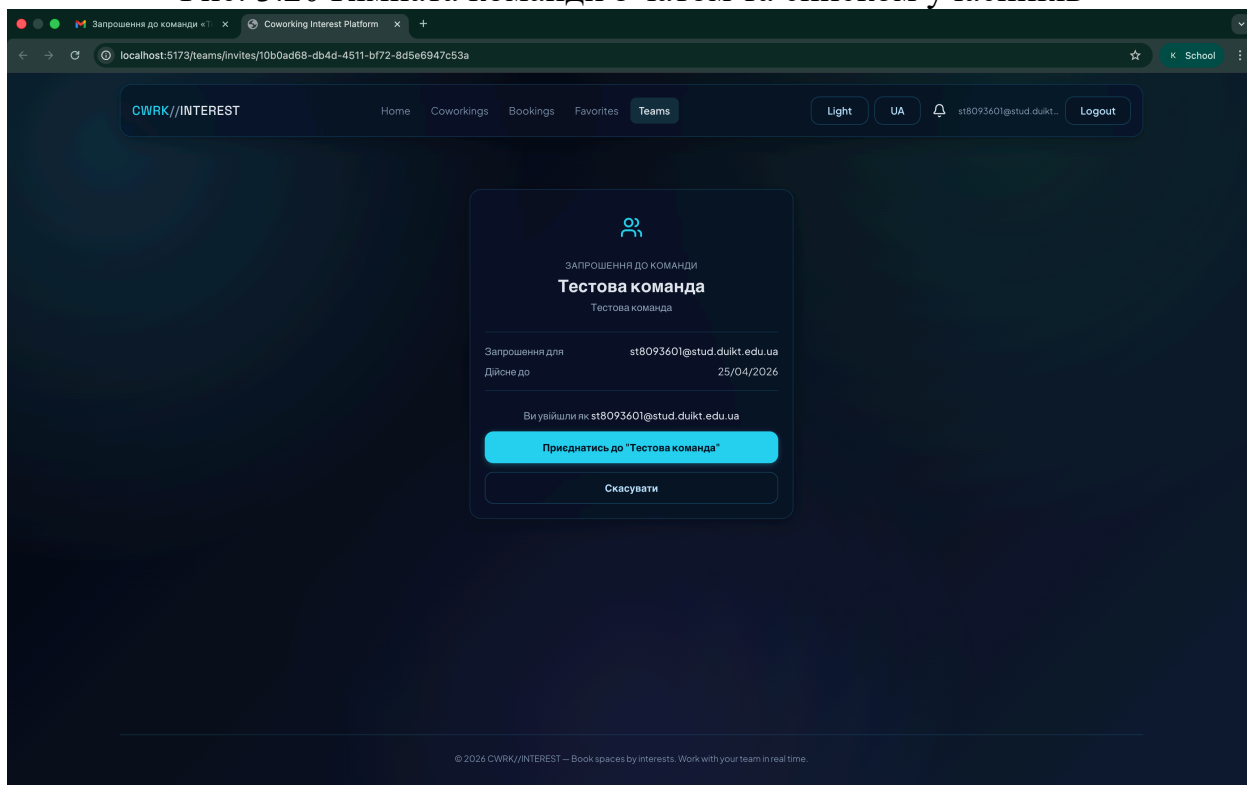


Рис. 3.21 Сторінка прийняття запрошення до команди

Кабінети для різних ролей реалізовані як окремі захищені секції, доступ до яких контролюється через компонент ProtectedRoute з відповідним набором дозволених ролей. Кабінет власника простору OWNER надає доступ до управління профілем коворкінгу, переліком зон із цінами та місткістю, тегами спільноти для алгоритму рекомендацій та статистикою завантаженості за обраний період. Кабінет менеджера MANAGER відображає список поточних бронювань з можливістю їх підтвердження або скасування. Кабінет ресепшн RECEPTION містить поле введення QR-токену та список бронювань на поточний день для реєстрації прибуття та відправлення відвідувачів. Кабінет адміністратора платформи ADMIN надає доступ до управління користувачами, зміни їхніх ролей та перегляду аудиту критичних дій із фільтрацією за типом події та часовим діапазоном.

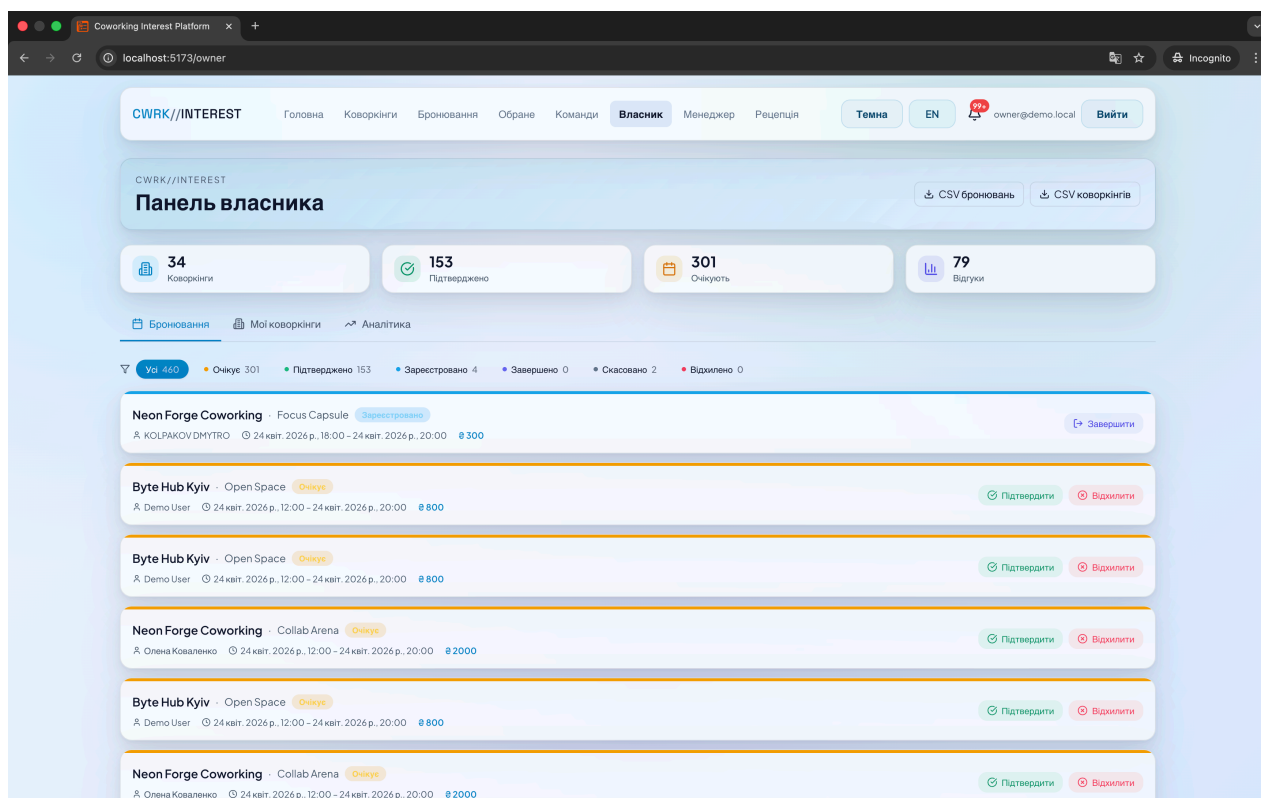


Рис. 3.22 Кабінет власника коворкінгу

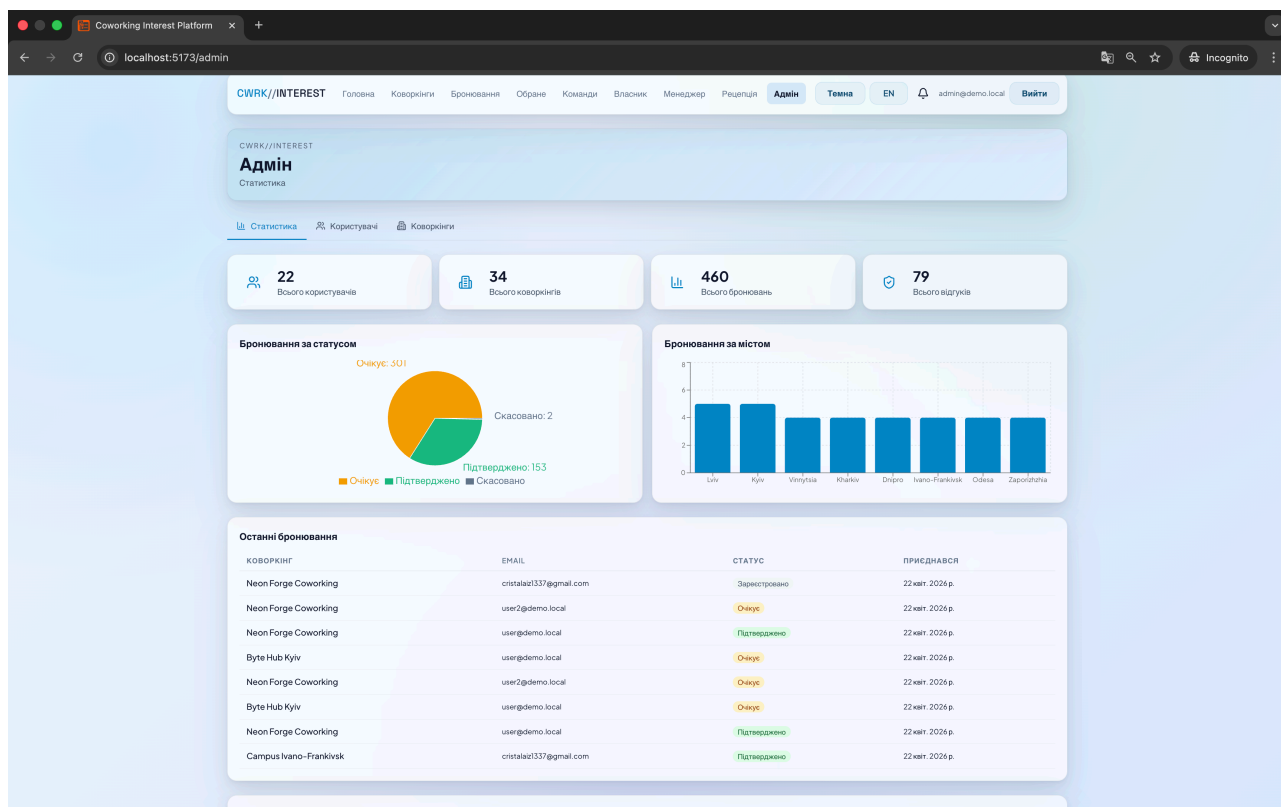


Рис. 3.23 Кабінет адміністратора платформи

Система сповіщень функціонує на двох рівнях. Перший рівень – внутрішні сповіщення в кабінеті користувача: при зміні статусу бронювання, появі нового учасника команди або звільненні місця у waitlist сервер створює запис у моделі Notification, а клієнт під час наступного запиту отримує оновлений лічильник непрочитаних. Другий рівень – email-сповіщення через Nodemailer: підтвердження бронювання, його скасування та лист для скидання пароля надсилаються через SMTP з брендованим шаблоном платформи CWRK//INTEREST. Обидва рівні працюють незалежно: помилка відправки email фіксується через Pino, але не впливає на виконання бізнес-операції.

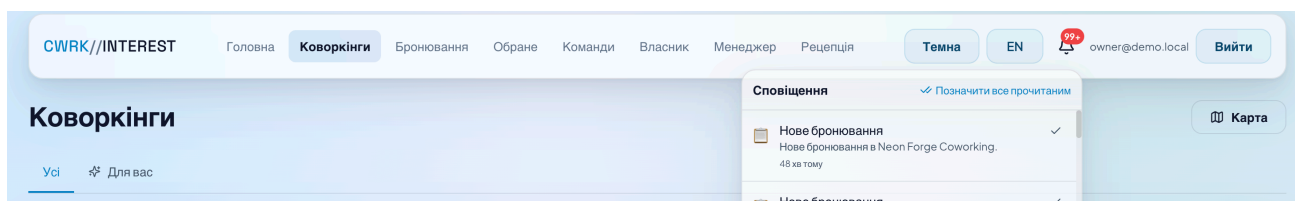


Рис. 3.24 Дропдаун сповіщень із лічильником непрочитаних

Сукупність реалізованих модулів формує цілісне функціональне середовище платформи та забезпечує узгоджену роботу всіх її основних підсистем. Реалізовані

засоби командної взаємодії, внутрішній чат, рольові кабінети та дворівнева система сповіщень дозволяють підтримувати повний цикл роботи користувачів у межах одного сервісу. Завдяки цьому платформа охоплює потреби різних категорій учасників, починаючи від звичайного користувача, який шукає та бронює простір, і завершуючи адміністратором, що контролює загальне функціонування системи. Такий підхід дає змогу не лише забезпечити зручну взаємодію між усіма учасниками, а й реалізувати на практиці функціональні вимоги, визначені у підрозділі 1.5. Наочне представлення інтерфейсу платформи на всіх ключових екранах наведено у Додатку А.

4 ТЕСТУВАННЯ ТА ОЦІНКА ЯКОСТІ РОЗРОБЛЕНОЇ СИСТЕМИ

4.1 Результати unit-тестування та E2E-тестування

Тестування розробленої системи організоване на двох незалежних рівнях що відповідають різним цілям перевірки. Unit-тестування перевіряє ізольовану логіку окремих сервісів без залежності від бази даних або зовнішніх сервісів. E2E-тестування перевіряє повні користувацькі сценарії у реальному браузері з розгорнутою інфраструктурою. Обидва рівні запускаються автоматично при кожному push до репозиторію через GitHub Actions CI конвеєр описаний у підрозділі 2.1 [22].

Unit-тести реалізовані через Jest і зосереджені на двох найкритичніших сервісах системи [24]. Перший тест у файлі `auth.service.spec.ts` перевіряє логіку валідації ТOTP-токену у модулі двофакторної автентифікації: позитивний сценарій із коректним кодом що відповідає поточному 30-секундному вікну та негативний сценарій із невалідним кодом де перевіряється що функція повертає `false` і не надає доступ. Другий тест у файлі `bookings.service.spec.ts` перевіряє механізм захисту від перебронювання: при спробі забронювати зону де вже зайнята вся місткість сервіс має повернути виняток із HTTP-кодом 409 Conflict. Цей тест безпосередньо підтверджує коректність транзакції з рівнем ізоляції `Serializable` описаної у підрозділі 3.4. Додатково у конфігурації `jest-e2e.json` визначені три інтеграційні тести у файлі `app.e2e-spec.ts` що перевіряють базову доступність серверних ендпоінтів та коректність відповідей health-чеку.

```

pnpm --filter @diploma/server test

> @diploma/server@1.0.0 test /Users/dmytrokolpakov/Library/Mobile Documents/com~apple~CloudDocs/Диплом 4 курс/coworking_duikt/server
> jest

PASS src/services/bookings.service.spec.ts
PASS src/services/auth.service.spec.ts
A worker process has failed to exit gracefully and has been force exited. This is likely caused by tests leaking due to improper teardown. Try running with --detectOpenHandles to find leaks. Active timers can also cause this, ensure that .unref() was called on them.

Test Suites: 2 passed, 2 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.966 s, estimated 2 s
Ran all test suites.

pnpm --filter @diploma/server test:e2e

> @diploma/server@1.0.0 test:e2e /Users/dmytrokolpakov/Library/Mobile Documents/com~apple~CloudDocs/Диплом 4 курс/coworking_duikt/server
> jest --config ./test/jest-e2e.json

PASS test/app.e2e-spec.ts
Minimal REST E2E
  ✓ GET /health (22 ms)
  ✓ POST /auth/login (8 ms)
  ✓ POST /bookings (2 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.187 s, estimated 2 s
Ran all test suites.

```

Рис. 4.1 Результати виконання Jest unit-тестів

E2E-тестування реалізоване через Playwright і включає 13 тестів розподілених між чотирма файлами за функціональними доменами [23]. Файл `auth.spec.ts` містить три тести що охоплюють повний цикл автентифікації: сценарій реєстрації нового акаунту з подальшим входом та виходом із системи, позитивний сценарій входу з правильним email та паролем де перевіряється успішне перенаправлення до кабінету, та негативний сценарій входу з невірним паролем де Playwright перевіряє що відображається повідомлення про помилку без отримання доступу до захищених сторінок. Файл `booking.spec.ts` містить чотири тести що перевіряють навігацію по ключових сторінках платформи: каталог коворкінгів, карта з кластеризацією маркерів, детальна сторінка конкретного простору та список бронювань авторизованого користувача. Файл `owner.spec.ts` містить три тести для перевірки доступності та базової функціональності кабінету власника простору. Файл `admin.spec.ts` містить три тести для адміністративного кабінету включно з перевіркою доступності аудит-логу.

Кожен E2E-тест виконується у відокремленому середовищі з чистою базою даних у пам'яті через `docker-compose.test.yml` з ephemeral PostgreSQL на tmpfs. Після завершення тестового сеансу ця база даних автоматично видаляється разом з усіма тестовими даними. Такий підхід гарантує що результати тестів не залежать від стану даних попереднього запуску та є повністю відтворюваними на будь-якій машині з Docker.

```

pnpm exec playwright test
Running 13 tests using 1 worker

✓ 1 [chromium] › e2e/admin.spec.ts:12:7 › Admin panel › admin page renders (1.9s)
✓ 2 [chromium] › e2e/admin.spec.ts:17:7 › Admin panel › admin can see user list or stats (1.7s)
✓ 3 [chromium] › e2e/admin.spec.ts:23:5 › public stats are shown on landing page (587ms)
✓ 4 [chromium] › e2e/auth.spec.ts:4:7 › Auth flows › landing page is visible (630ms)
✓ 5 [chromium] › e2e/auth.spec.ts:10:7 › Auth flows › register → login → logout (2.7s)
✓ 6 [chromium] › e2e/auth.spec.ts:47:7 › Auth flows › wrong password shows error (1.5s)
✓ 7 [chromium] › e2e/booking.spec.ts:12:7 › Booking flows › coworkings list is visible (1.6s)
✓ 8 [chromium] › e2e/booking.spec.ts:18:7 › Booking flows › coworkings map loads (1.7s)
✓ 9 [chromium] › e2e/booking.spec.ts:23:7 › Booking flows › coworking detail page loads (1.9s)
✓ 10 [chromium] › e2e/booking.spec.ts:33:7 › Booking flows › my bookings page loads (1.9s)
✓ 11 [chromium] › e2e/owner.spec.ts:12:7 › Owner dashboard › owner page renders stats (1.7s)
✓ 12 [chromium] › e2e/owner.spec.ts:17:7 › Owner dashboard › owner analytics tab loads charts (2.1s)
✓ 13 [chromium] › e2e/owner.spec.ts:27:7 › Owner dashboard › owner bookings tab shows pending list (1.7s)

```

Рис. 4.2 Результати виконання Playwright E2E-тестів

У процесі розробки СІ конвеєр зафіксував кілька реальних помилок що були виявлені автоматично до злиття коду в основну гілку. Зокрема, кілька ранніх запусків завершилися з помилкою на задачі Docker Build через некоректну конфігурацію багатоетапної збірки, а задача Lint & Type-check виявила кілька випадків де змінна могла мати значення null без відповідної перевірки. Виправлення цих проблем на рівні СІ до злиття коду підтвердило практичну цінність автоматизованого тестування як частини процесу розробки, а не як окремого кроку після її завершення. Скриншоти результатів виконання тестових наборів наведено у Додатку А.

4.2 Оцінка відповідності функціональним вимогам

Після завершення розробки всіх модулів системи проведено перевірку відповідності реалізованого продукту функціональним та нефункціональним вимогам, визначеним у підрозділі 1.5. Перевірка здійснювалась комбінованим методом: автоматизовані сценарії Playwright підтвердили коректність ключових користувацьких потоків, unit-тести Jest підтвердили правильність бізнес-логіки критичних сервісів, а ручне функціональне тестування охопило сценарії що не входять до автоматизованого набору — командний чат, двофакторну автентифікацію, роботу карти та кабінети всіх ролей. Результати зведено у табл. 4.1.

Усі двадцять функціональних вимог підтверджені тестуванням. Окремої уваги заслуговує перевірка алгоритму рекомендацій: ручне тестування проводилось із двома обліковими записами з різними наборами інтересів, і обидва

отримали різний ранжований список у вкладці For you при однаковому каталозі, що підтверджує персоналізований характер видачі. Командний чат тестувався одночасно з двох браузерів у режимі інкогніто під різними акаунтами – повідомлення доставлялись без затримки в обох напрямках, що підтверджує коректність WebSocket-з'єднання через Socket.IO.

Таблиця 4.1

Відповідність реалізованих функцій вимогам системи

Вимога	Статус	Метод перевірки
Реєстрація та автентифікація з JWT	Реалізовано	E2E auth.spec.ts
Двофакторна автентифікація (TOTP, Email OTP, Backup)	Реалізовано	Unit auth.service.spec.ts, ручне
Профіль із тегами інтересів	Реалізовано	Ручне тестування
Каталог із пошуком та фільтрацією	Реалізовано	E2E booking.spec.ts
Карта просторів із кластеризацією	Реалізовано	E2E booking.spec.ts
Алгоритм рекомендацій за інтересами	Реалізовано	Ручне тестування
Бронювання зони з перевіркою місткості	Реалізовано	Unit bookings.service.spec.ts
Командне бронювання з QR-квитками	Реалізовано	Ручне тестування
Запрошення до команди через email та токен	Реалізовано	Ручне тестування
Чат команди в реальному часі (Socket.IO)	Реалізовано	Ручне тестування
Check-in та check-out через QR-токен	Реалізовано	Ручне тестування
Кабінет OWNER з управлінням простором	Реалізовано	E2E owner.spec.ts, ручне
Кабінет MANAGER з підтвердженням бронювань	Реалізовано	Ручне тестування
Кабінет RECEPTION з реєстрацією відвідувачів	Реалізовано	Ручне тестування
Кабінет ADMIN з аудит-логом та управлінням ролями	Реалізовано	E2E admin.spec.ts, ручне
Email-сповіщення через SMTP	Реалізовано	Ручне тестування
In-app сповіщення з лічильником непрочитаних	Реалізовано	Ручне тестування
PWA із трирівневим кешуванням	Реалізовано	Ручне тестування
Локалізація інтерфейсу (i18next)	Реалізовано	Ручне тестування
Захист маршрутів за роллю (ProtectedRoute)	Реалізовано	E2E усі файли

Нефункціональні вимоги також підтверджені в ході тестування. JWT-автентифікація з ротацією refresh-токену та серверною інвалідацією при logout перевірена вручну через Postman – після виходу старий refresh-токен повертає 401 при спробі оновлення. RBAC перевірений спробою доступу до захищених ендпоінтів з токеном із неправильною роллю – у всіх випадках сервер повернув 403 Forbidden. Валідація вхідних даних через Zod перевірена навмисно некоректними запитами до кожної групи ендпоінтів – у всіх випадках middleware повернув 400 з детальним описом порушень у масиві issues. Кросбраузерна сумісність перевірена в Google Chrome, Mozilla Firefox та Microsoft Edge – інтерфейс відображався коректно в усіх браузерах без візуальних артефактів. Розгортання через Docker Compose підтверджене запуском повного середовища командою `docker compose up` на чистій машині без попередньо встановлених залежностей.

4.3 Порівняльний аналіз розробленої платформи з існуючими аналогами

Завершальним етапом оцінки розробленої системи є її зіставлення з чотирма платформами українського ринку що були проаналізовані у підрозділі 1.2: Creative States, Platforma, Kooperativ та KONTORA Lviv. На момент первинного аналізу жодна з них не реалізовувала підбору просторів за інтересами, повноцінної командної моделі або чату в реальному часі. Після завершення розробки та тестування системи CWRK//INTEREST є підстави стверджувати що ці прогалини закриті у повному обсязі, а реалізовані рішення підтверджені функціональним тестуванням. Порівняння наведено у табл. 4.2.

Найбільш принциповою відмінністю розробленої платформи від усіх чотирьох аналогів є наявність алгоритму персональних рекомендацій. Жоден з існуючих українських сервісів не використовує дані про інтереси користувача для ранжування каталогу – пошук скрізь зводиться до базової фільтрації за містом та ціною. У CWRK//INTEREST кожен авторизований користувач бачить

персоналізовану вкладку For you де коворкінги відсортовані за комбінованим показником що враховує збіг інтересів за коефіцієнтом Жаккара, поведінку схожої аудиторії через collaborative filtering та власну історію бронювань із часовим затуханням. Це переводить взаємодію з каталогом з пасивного перегляду в активний підбір що адаптується до конкретного користувача з кожним новим бронюванням.

Таблиця 4.2

Порівняння CWRK//INTEREST з аналогами українського ринку

Критерій	CWRK// INTEREST	Creative States	Platforma	Kooperativ	KONTORA Lviv
Онлайн-бронювання в реальному часі	+	Частково	—	—	—
Підбір просторів за інтересами	+	—	—	—	—
Алгоритм персональних рекомендацій	+	—	—	—	—
Командне бронювання з ролями	+	—	—	—	—
Чат команди в реальному часі	+	—	—	—	—
QR-квиток для кожного учасника	+	—	—	—	—
Кабінет власника простору	+	+	+	—	—
Кабінет менеджера та ресепшн	+	—	—	—	—
Особистий кабінет користувача	+	+	—	—	—
Система сповіщень	+	—	—	—	—
Аудит-лог дій адміністратора	+	—	—	—	—
PWA із офлайн-режимом	+	—	—	—	—

Критерій	CWRK// INTEREST	Creative States	Platforma	Kooperativ	KONTORA Lviv
Відкритий вихідний код	+	—	—	—	—
Self-hosted розгортання	+	—	—	—	—

Другою ключовою відмінністю є повноцінна командна модель. Creative States як найбільша мережа в Україні надає фізичні простори для команд, але не має жодного цифрового інструменту для координації спільного бронювання – один учасник бронює, решта дізнаються про це поза межами платформи. У CWRK//INTEREST команда є повноправним суб'єктом системи: вона має власний простір із чатом у реальному часі, спільний календар бронювань, розподіл ролей між учасниками та окремий QR-квиток для кожного члена при командному бронюванні. Це дозволяє великим командам координувати відвідування коворкінгу без виходу за межі платформи.

Третьою відмінністю є повнота рольової моделі з боку операторів простору. Platforma та Creative States мають певний адміністративний функціонал для управління власними просторами, проте він не є публічно доступним як частина веб-платформи. CWRK//INTEREST реалізує чотири окремих захищених кабінети для OWNER, MANAGER, RECEPTION та ADMIN із чітко розмежованими правами доступу. Особливо важливим є кабінет RECEPTION що реалізує фізичний цикл check-in та check-out через QR-токен – жоден з аналогів не має подібного інструменту в публічному інтерфейсі.

З точки зору технічної доступності розроблена платформа також займає унікальну позицію серед аналогів. Усі чотири розглянуті сервіси є закритими комерційними продуктами із SaaS-архітектурою де власник коворкінгу залежить від провайдера щодо зберігання даних, доступності сервісу та умов ліцензування. CWRK//INTEREST розповсюджується з відкритим вихідним кодом і може бути розгорнута на власному сервері будь-якого розміру командою `docker compose up` без залежності від зовнішніх хмарних провайдерів. Для невеликих коворкінгів та

стартапів у цій сфері це означає відсутність щомісячних ліцензійних платежів та повний контроль над даними своїх відвідувачів.

Підсумовуючи результати порівняльного аналізу, можна констатувати що розроблена система закриває п'ять проблем предметної галузі виявлених у підрозділі 1.3: відсутність підбору за інтересами, відсутність командної моделі на рівні платформи, відсутність комунікації між учасниками всередині сервісу, слабку персоналізацію та залежність від закритих SaaS-рішень. Кожна з цих проблем вирішена конкретним технічним рішенням: алгоритм рекомендацій, модель Team із ролями та запрошеннями, Socket.IO чат, профіль інтересів та self-hosted Docker-розгортання відповідно. Це підтверджує що розроблена платформа не є черговою реалізацією типового CRUD-застосунку для бронювань, а являє собою цілісне рішення зі сформульованою позицією на конкретному ринку та підтвердженими перевагами над існуючими аналог

ВИСНОВКИ

У межах кваліфікаційної роботи бакалавра мету досягнуто — розроблено повнофункціональну цифрову платформу бронювання коворкінг-просторів з урахуванням інтересів користувачів та підтримкою командної взаємодії на основі JavaScript, що забезпечує підвищення ефективності процесів пошуку, бронювання та організації командної роботи шляхом персоналізованого підбору просторів за інтересами.

Проведено аналіз предметної галузі та чотирьох існуючих платформ українського ринку — Creative States, Platforma, Kooperativ та KONTORA Lviv. Встановлено що жодна з них не реалізує підбору просторів за інтересами користувача, повноцінної командної моделі з ролями та чатом або прозорого автоматизованого циклу бронювання з підтвердженням у реальному часі. Виявлені п'ять ключових проблем визначили вектор розробки власного рішення.

Обґрунтовано вибір технологічного стеку та спроектовано архітектуру системи. TypeScript обрано як єдину мову програмування для всіх рівнів — серверного, клієнтського та спільного пакету — що дозволило перевикористовувати Zod-схеми валідації між сервером і клієнтом без дублювання. Серверна частина побудована на Node.js з Express, клієнтська на React 19 з Vite та TanStack Query, база даних реалізована на PostgreSQL 16 з Prisma ORM, кешування та rate-limiting через Redis 7. Вся інфраструктура розгортається через Docker Compose. Вибір REST API з документацією через Swagger та WebSocket через Socket.IO як двох незалежних каналів комунікації обґрунтований порівняльним аналізом альтернатив GraphQL та gRPC.

Реалізована серверна частина із 24 файлами маршрутів, 14 сервісами та близько 140 ендпоінтів. JWT-автентифікація реалізована з парою access та refresh tokenів де refresh зберігається у вигляді argon2-хешу в базі даних та інвалідується на сервері при logout. Двофакторна автентифікація підтримує три незалежних методи: TOTP через speakeasy, Email OTP через Redis з TTL та backup-коди хешовані argon2. RBAC реалізований з п'ятьма системними ролями та захистом

кожного маршруту через відповідний middleware. Безпека HTTP-рівня забезпечена через Helmet з кастомною CSP-конфігурацією.

Розроблено веб-інтерфейс із персональними кабінетами для чотирьох системних ролей. Кабінет власника простору OWNER надає управління профілем коворкінгу, зонами та тарифами. Кабінет менеджера MANAGER – управління бронюваннями. Кабінет ресепшн RECEPTION – реєстрацію відвідувачів через QR-токени. Кабінет адміністратора ADMIN – управління користувачами та аудит-логом критичних дій. Захист маршрутів реалізований через компонент ProtectedRoute з перевіркою ролі. Застосунок зареєстровано як PWA із трирівневою стратегією кешування на базі Workbox.

Впроваджено модуль рекомендацій за інтересами та командну взаємодію в реальному часі. Алгоритм рекомендацій у recommendations.service.ts розраховує показник відповідності за формулою що поєднує коефіцієнт Жаккара з вагою 0.5, collaborative filtering з вагою 0.2 та аналіз власної історії бронювань із часовим затуханням через $\exp(-\text{daysDiff}/90)$ з вагою 0.3. Командна модель реалізована через сутності Team, TeamMember, TeamInvite та Message – учасники команди мають спільний чат через Socket.IO з персистентною історією повідомлень, спільний календар бронювань та окремий QR-квиток для кожного при командному бронюванні.

Проведено тестування на двох рівнях. Unit-тестування через Jest охоплює критичну бізнес-логіку auth.service та bookings.service. E2E-тестування через Playwright включає 13 тестів у чотирьох файлах що покривають автентифікацію, навігацію по каталогу, кабінет власника та адміністратора. Усі тести запускаються автоматично при кожному push через GitHub Actions CI конвеєр з п'ятьма задачами. Порівняльний аналіз із аналогами підтвердив що розроблена платформа є єдиним рішенням серед розглянутих що одночасно реалізує підбір за інтересами, командну взаємодію з чатом, повний операційний цикл для операторів та self-hosted розгортання з відкритим вихідним кодом. Демонстраційні матеріали до захисту, включно з архітектурною схемою, діаграмами та екранними формами платформи, наведено у Додатку А.

Практична цінність роботи полягає у створенні повнофункціональної платформи орієнтованої на конкретний сегмент українського ринку – коворкінги що формують спільноти за інтересами та потребують цифрового інструменту для координації командної роботи. Розроблена система розповсюджується з відкритим вихідним кодом і розгортається на власному сервері через Docker Compose без залежності від зовнішніх хмарних провайдерів, що знижує поріг входу для операторів малих та середніх коворкінгів. Подальший розвиток системи передбачає підключення реальних платіжних провайдерів через наявну абстракцію PaymentService, реалізацію клієнтського інтерфейсу для повторюваних бронювань та черги очікування що наразі існують лише на рівні серверного API, а також розширення покриття автоматизованими E2E-тестами до включення повного сценарію створення бронювання через форму.

ПЕРЕЛІК ПОСИЛАНЬ

1. Gandini A. The Reputation Economy: Understanding Knowledge Work in Digital Society. London : Palgrave Macmillan, 2016. 208 p.
2. Creative States. Коворкінги України. URL: <https://creativestates.com.ua> (дата звернення: 10.04.2026).
3. Garrett J. J. The Elements of User Experience: User-Centered Design for the Web and Beyond. 2nd ed. Berkeley : New Riders, 2011. 192 p.
4. Schafer J. B., Konstan J., Riedl J. Recommender Systems in E-Commerce // Proceedings of the 1st ACM Conference on Electronic Commerce. New York : ACM, 1999. P. 158–166.
5. Stallings W. Cryptography and Network Security: Principles and Practice. 8th ed. London : Pearson, 2022. 816 p.
6. TypeScript Documentation. URL: <https://www.typescriptlang.org/docs/> (дата звернення: 15.04.2026).
7. Node.js Documentation. URL: <https://nodejs.org/en/docs/> (дата звернення: 15.04.2026).
8. Express.js Documentation. URL: <https://expressjs.com/> (дата звернення: 15.04.2026).
9. React Documentation. URL: <https://react.dev/> (дата звернення: 15.04.2026).
10. Vite Documentation. URL: <https://vite.dev/> (дата звернення: 15.04.2026).
11. PostgreSQL 16 Documentation. URL: <https://www.postgresql.org/docs/16/> (дата звернення: 15.04.2026).
12. Prisma ORM Documentation. URL: <https://www.prisma.io/docs> (дата звернення: 15.04.2026).
13. Redis Documentation. URL: <https://redis.io/docs/> (дата звернення: 15.04.2026).
14. Socket.IO Documentation. URL: <https://socket.io/docs/v4/> (дата звернення: 15.04.2026).

15. TanStack Query Documentation. URL: <https://tanstack.com/query/latest> (дата звернення: 15.04.2026).
16. Tailwind CSS Documentation. URL: <https://tailwindcss.com/docs> (дата звернення: 15.04.2026).
17. FullCalendar Documentation. URL: <https://fullcalendar.io/docs> (дата звернення: 15.04.2026).
18. Leaflet Documentation. URL: <https://leafletjs.com/reference.html> (дата звернення: 15.04.2026).
19. JSON Web Token Introduction. URL: <https://jwt.io/introduction> (дата звернення: 15.04.2026).
20. Zod Documentation. URL: <https://zod.dev/> (дата звернення: 15.04.2026).
21. Docker Documentation. URL: <https://docs.docker.com/> (дата звернення: 15.04.2026).
22. GitHub Actions Documentation. URL: <https://docs.github.com/en/actions> (дата звернення: 15.04.2026).
23. Playwright Documentation. URL: <https://playwright.dev/docs/intro> (дата звернення: 15.04.2026).
24. Jest Documentation. URL: <https://jestjs.io/docs/getting-started> (дата звернення: 15.04.2026).
25. MinIO Object Storage Documentation. URL: <https://min.io/docs/minio/linux/index.html> (дата звернення: 15.04.2026).
26. Helmet.js Documentation. URL: <https://helmetjs.github.io/> (дата звернення: 15.04.2026).
27. speakeasy – Two-factor authentication for Node.js. URL: <https://github.com/speakeasyjs/speakeasy> (дата звернення: 15.04.2026).
28. i18next Documentation. URL: <https://www.i18next.com/> (дата звернення: 15.04.2026).
29. Framer Motion Documentation. URL: <https://www.framer.com/motion/> (дата звернення: 15.04.2026).

30. React Router Documentation. URL: <https://reactrouter.com/en/main> (дата звернення: 15.04.2026).

31. Колпаков Д. О., Жебка В. В. Цифрова платформа бронювання коворкінг-просторів з урахуванням інтересів користувачів та підтримкою командної взаємодії на основі JavaScript // Матеріали Всеукраїнської науково-технічної конференції «Технології цифрового розвитку: програмні системи та децентралізація». Збірник тез. К. : ДУІКТ, 2026.

32. Колпаков Д. О., Жебка В. В. Цифрова платформа бронювання коворкінг-просторів з урахуванням інтересів користувачів та підтримкою командної взаємодії на основі JavaScript // Матеріали VII Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. К. : ДУІКТ, 2026.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

CWRK//INTEREST

Веб-платформа бронювання коворкінг-просторів за інтересами з підтримкою командної роботи

JavaScript / TypeScript

Виконав:
Колпаков Дмитро Олегович
група ТЦР-41

Кваліфікаційна робота бакалавра

Спеціальність 121
Інженерія програмного забезпечення

Керівник

Жебка В.В.
д.т.н., професор

Кафедра

Технології цифрового розвитку

Університет

ДУІКТ, 2026

Київ — 2026

Титульний слайд презентації

Мета, об'єкт та предмет дослідження

МЕТА РОБОТИ

Розробка веб-платформи бронювання коворкінг-просторів за інтересами з підтримкою командної роботи з використанням JavaScript/TypeScript-екосистеми.

ОБ'ЄКТ ДОСЛІДЖЕННЯ

Процес організації роботи користувачів у коворкінг-просторах та пов'язані з ним процеси пошуку, бронювання та командної взаємодії.

ПРЕДМЕТ ДОСЛІДЖЕННЯ

Моделі, методи, алгоритми та програмні засоби побудови веб-платформи бронювання коворкінг-просторів за інтересами.

ЗАДАЧІ РОБОТИ

- 1. Аналіз предметної галузі та існуючих платформ-аналогів
- 2. Обґрунтування технологічного стеку та проектування архітектури
- 3. Реалізація серверної частини: JWT, RBAC, REST API, Socket.IO
- 4. Розробка веб-інтерфейсу з кабінетами для чотирьох ролей
- 5. Алгоритм рекомендацій за інтересами та командна взаємодія
- 6. Тестування та оцінка якості розробленого рішення

Мета, об'єкт та предмет дослідження

Аналіз платформ-аналогів українського ринку

Creative States	Platforma	Kooperativ	KONTORA Lviv
Найбільша мережа преміальних просторів	Київська мережа з освітнім напрямом	Творча майстерня з активною спільнотою	Преміум-офіси у центрі Львова
Розвинена інфраструктура	Сильна спільнота	Унікальна атмосфера	Цілодобовий доступ
Немає онлайн-бронювання	Форма звернення замість бронювання	Мінімальний цифровий інструментарій	Базовий сайт без функціоналу

Спільна проблема всіх аналогів: відсутність підбору за інтересами, командної моделі, онлайн-бронювання в реальному часі та відкритого вихідного коду.

Аналіз платформ-аналогів українського ринку

Функціональні та нефункціональні вимоги

ФУНКЦІОНАЛЬНІ ВИМОГИ

- Реєстрація, автентифікація, 2FA (TOTP/OTP/Backup)
- Профіль з тегами інтересів для рекомендацій
- Каталог із пошуком, фільтрами та картою (Leaflet)
- Алгоритм рекомендацій за коефіцієнтом Жаккара
- Бронювання зон: вибір зони, часу, підтвердження
- Командне бронювання з QR-квитком кожному учаснику
- Чат команди в реальному часі (Socket.IO)
- Check-in / Check-out через QR-токен (ресепшн)
- Кабінети: USER, OWNER, MANAGER, RECEPTION, ADMIN
- Сповіщення: email + in-app

НЕФУНКЦІОНАЛЬНІ ВИМОГИ

Безпека
bcrypt, argon2, JWT, Helmet, Zod

Продуктивність
API ≤ 500 мс, Redis кеш

Кросбраузерність
Chrome, Firefox, Edge, Safari

PWA
Офлайн-кешування, Workbox

Локалізація
i18next (UA / EN)

Розгортання
Docker Compose, self-hosted

API документація
Swagger / OpenAPI 3.0.3

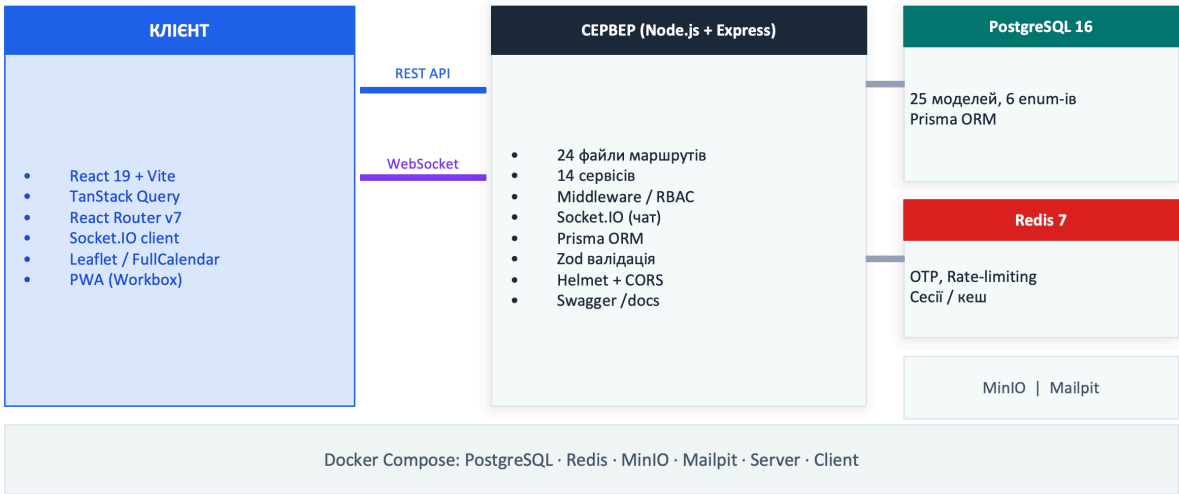
Функціональні та нефункціональні вимоги до системи

Обраний технологічний стек

СЕРВЕРНА ЧАСТИНА	КЛІЄНТСЬКА ЧАСТИНА	ІНФРАСТРУКТУРА
• Node.js 22 LTS • Express 4.22 • TypeScript • Prisma ORM 5.22 • PostgreSQL 16 • Redis 7 • Socket.IO 4.8 • JWT + argon2 + bcrypt	• React 19 • Vite 6 • TanStack Query 5 • React Router v7 • Tailwind CSS 3.4 • FullCalendar 6.1 • Leaflet + react-leaflet • Framer Motion, i18next	• Docker Compose • MinIO (S3-сумісний) • Mailpit (SMTP dev) • GitHub Actions CI • Jest (unit) • Playwright (E2E) • Swagger / OpenAPI • pnpm monorepo

Обраний технологічний стек системи

Архітектура системи



CI/CD: GitHub Actions — Lint · Prisma Validate · Unit Tests · Docker Build · Playwright E2E

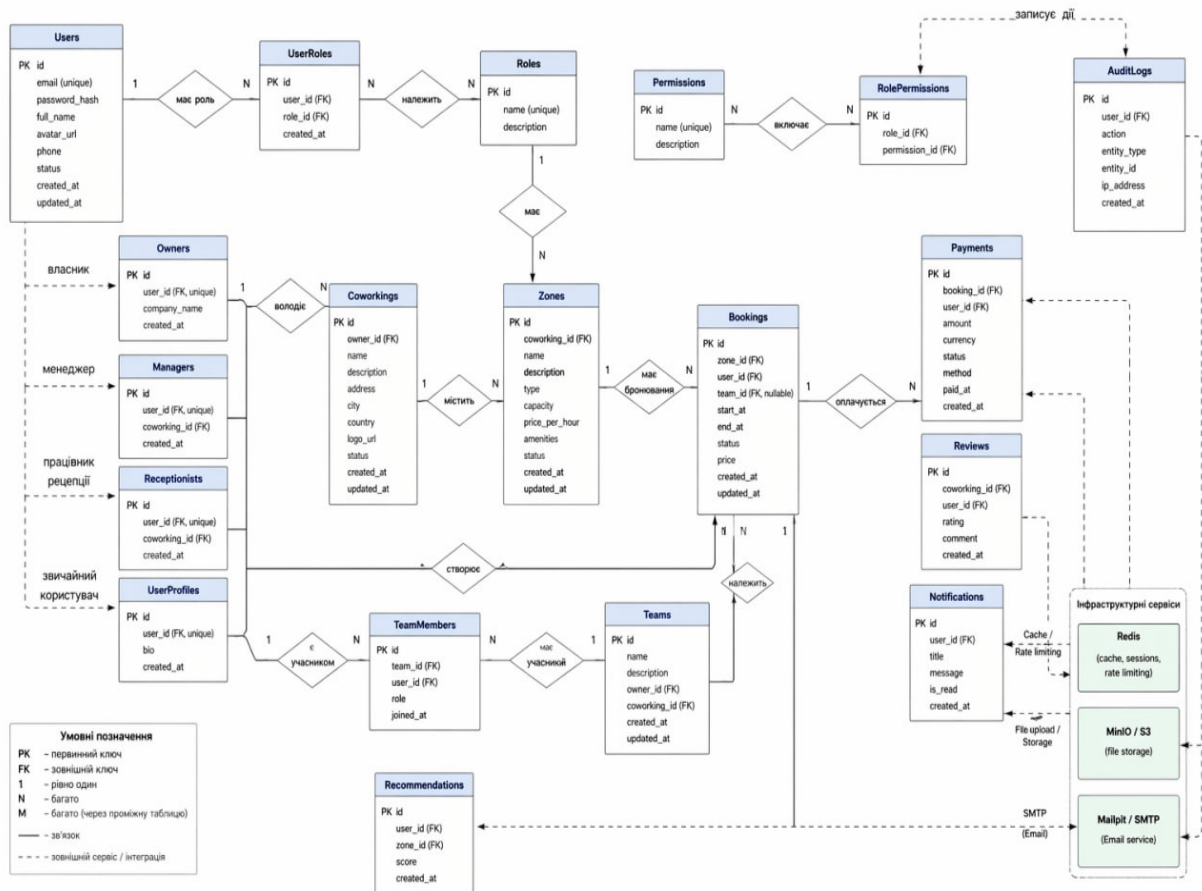
Архітектура системи CWRK//INTEREST

Діаграма варіантів використання — користувач платформи



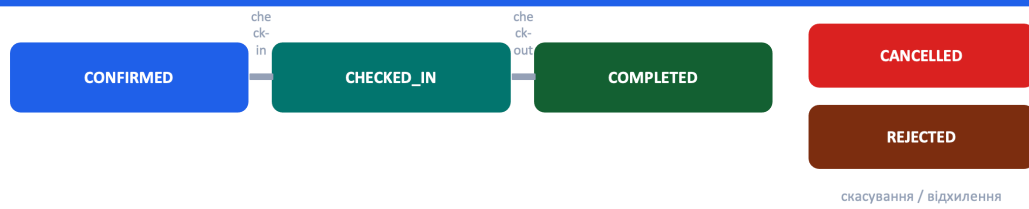
Діаграма варіантів використання та рольова модель

ER-діаграма вебсистеми для керування коворкінг-просторами



Структура бази даних (ER-діаграма)

Діаграма бізнес-процесів: життєвий цикл бронювання



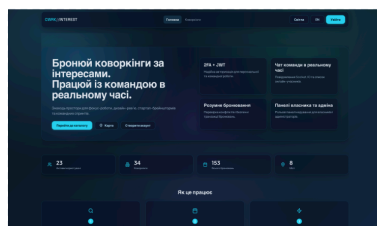
- 1 Вибір зони**
Користувач обирає зону та команду (або особисто)
- 2 Вибір часу**
Введення startAt / endAt, перевірка доступності
- 3 Підтвердження**
Summary + POST /bookings → CONFIRMED + QR-токен
- 4 Check-in**
Ресепшн вводить QR-токен → CHECKED_IN
- 5 Check-out**
Фіксація завершення → COMPLETED

ДОДАТКОВО

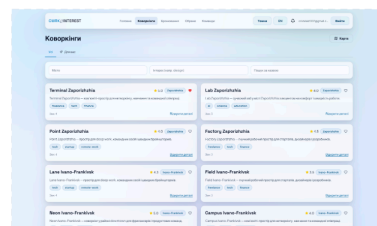
- Serializable транзакція
- → захист від подвійного бронювання
- Waitlist — автосповищення
- при звільненні слоту
- TeamBookingMember —
- окремий QR для кожного учасника
- RecurringBookingRule —
- повторювані бронювання

Діаграма бізнес-процесів: життєвий цикл бронювання

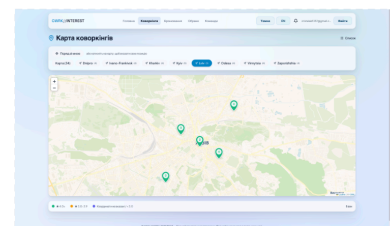
Екранні форми веб-платформи CWRK//INTEREST



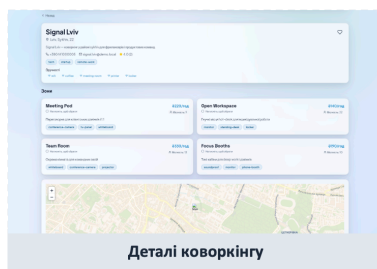
Головна / Landing



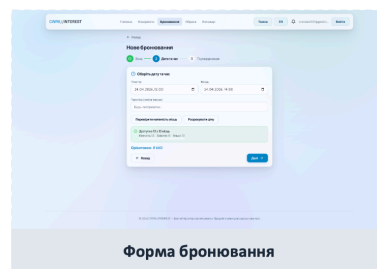
Каталог коворкінгів



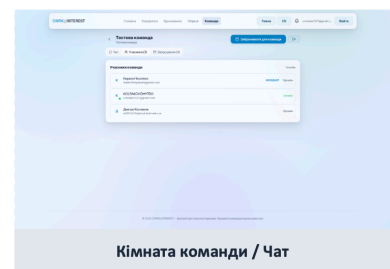
Карта просторів



Деталі коворкінгу



Форма бронювання



Кімната команди / Чат

Екранні форми веб-платформи CWRK//INTEREST

Алгоритм рекомендацій за інтересами

$$\text{score} = \text{round}((0.5 \times \text{jaccard} + 0.2 \times \text{collaborative} + 0.3 \times \text{history}) \times 100)$$

jaccard ×0.5

Коефіцієнт Жаккара

$$|A \cap B| / |A \cup B|$$

Збіг між інтересами користувача та тегами коворкінгу (0..1)

collaborative ×0.2

Collaborative Filtering

Popular among peers

Популярність серед інших користувачів зі схожими інтересами

history ×0.3

History Decay

$$\exp(-\text{daysDiff} / 90)$$

Власна історія бронювань з часовим затуханням (90 днів)

Жоден з проаналізованих аналогів не реалізує подібного алгоритму — це ключова конкурентна перевага платформи.

Алгоритм рекомендацій за інтересами

Результати тестування

2

Unit-тести
(Jest)

3

Jest E2E
(app.e2e-spec)

13

Playwright
E2E тестів

5

GitHub Actions
CI задачі

UNIT-ТЕСТИ (Jest)

- auth.service.spec.ts — валідація TOTP-токену (позитивний + негативний сценарій)
- bookings.service.spec.ts — конфлікт бронювання → 409 Conflict
- app.e2e-spec.ts — 3 інтеграційних тести доступності ендпоінтів

```
npm run test:unit
> @diploma/server@1.0.0 test /Users/daytr0k0/pak0u/Library/Mobile Documents/com-apple-CloudDocs/Диплом 4 kyp/coworking_duikt/server
> jest
PASS src/services/bookings.service.spec.ts
PASS src/services/auth.service.spec.ts
A worker process has failed to exit gracefully and has been force exited. This is likely caused by tests leaking due to improper teardown.
detectOpenHandles to find leaks. Active timers can also cause this; ensure that .unref() was called on them.
Test Suites: 2 passed, 2 total
Tests: 2 passed, 2 total
Snapshots: 0 total
Time: 1.966 s, estimated 2 s
Ran all test suites.
```

E2E-ТЕСТИ (Playwright)

- auth.spec.ts (3): реєстрація, вхід, невірний пароль
- booking.spec.ts (4): каталог, карта, деталі, бронювання
- owner.spec.ts (3): кабінет власника
- admin.spec.ts (3): кабінет адміністратора

```
npm run test:e2e
Running 13 tests using 1 worker
1 [chromium] › e2e/auth.spec.ts:12:7 › Admin panel › admin page renders (1.3s)
2 [chromium] › e2e/admin.spec.ts:12:7 › Admin panel › admin can see user list or state (1.7s)
3 [chromium] › e2e/admin.spec.ts:25:5 › public state are shown on landing page (2.2s)
4 [chromium] › e2e/auth.spec.ts:47:7 › Auth flow › landing page is visible (0.5s)
5 [chromium] › e2e/auth.spec.ts:116:7 › Auth flow › register → login → logout (2.1s)
6 [chromium] › e2e/auth.spec.ts:147:7 › Auth flow › using password shows error (1.2s)
7 [chromium] › e2e/booking.spec.ts:12:7 › Booking flow › coworking list is visible (1.6s)
8 [chromium] › e2e/booking.spec.ts:12:7 › Booking flow › coworking map loads (1.9s)
9 [chromium] › e2e/booking.spec.ts:23:7 › Booking flow › coworking detail page loads (1.9s)
10 [chromium] › e2e/booking.spec.ts:133:7 › Booking flow › no bookings show loads (1.9s)
11 [chromium] › e2e/owner.spec.ts:12:7 › Owner dashboard › owner page renders state (1.7s)
12 [chromium] › e2e/owner.spec.ts:17:7 › Owner dashboard › owner analytics tab loads charts (2.1s)
13 [chromium] › e2e/owner.spec.ts:27:7 › Owner dashboard › owner bookings tab shows pending list (1.7s)
```

Результати тестування (Jest та Playwright)

Результати та порівняння з аналогами

Критерій	CWRK//INTEREST	Creative States	Platforma	Kooperativ
Онлайн-бронювання	✓	Частково	✗	✗
Підбір за інтересами	✓	✗	✗	✗
Алгоритм рекомендацій	✓	✗	✗	✗
Командне бронювання	✓	✗	✗	✗
Чат у реальному часі	✓	✗	✗	✗
QR check-in/check-out	✓	✗	✗	✗
Кабінети ролей	✓	✓	✓	✗
Система сповіщень	✓	✗	✗	✗
Self-hosted / Open source	✓	✗	✗	✗

92 сторінки · 5 таблиць · 37 рисунків · 32 джерела · 2 тези на конференціях

Порівняльний аналіз з аналогами ринку

CWRK//INTEREST

Підсумок та перспективи розвитку

- Спроектовано та реалізовано повнофункціональну веб-платформу на TypeScript-монорепозиторії
- Реалізовано унікальний алгоритм рекомендацій (Жаккар + collaborative + history decay)
- Командна модель: Team, TeamMember, TeamInvite, Socket.IO чат, TeamBookingMember (QR)
- П'ять системних ролей з відокремленими кабінетами та повним циклом бронювання
- 13 Playwright E2E тестів + 2 Jest unit-тести + CI/CD GitHub Actions
- Self-hosted розгортання через Docker Compose з відкритим вихідним кодом

Дякую за увагу! | Колпаков Д.О., ТЦР-41 | ДУІКТ 2026

Підсумок та перспективи розвитку

ДОДАТОК Б ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

1. Схема бази даних (фрагмент schema.prisma)

Нижче наведено ключові моделі та enum-и Prisma-схеми що описують структуру реляційної бази даних платформи CWRK//INTEREST.

```
enum Role {
  USER
  OWNER
  ADMIN
  MANAGER
  RECEPTION
}

enum BookingStatus {
  PENDING
  CONFIRMED
  CANCELLED
  REJECTED
  CHECKED_IN
  COMPLETED
  AWAITING_PAYMENT
}

enum TeamRole {
  MEMBER
  MANAGER
}

model User {
  id          String  @id @default(uuid()) @db.Uuid
  email       String  @unique
  passwordHash String
  role        Role    @default(USER)
  isActive    Boolean @default(true)
  isTwoFactorEnabled Boolean @default(false)
  twoFactorSecret String?
  refreshTokenHash String?
  backupCodes String[] @default([])
  createdAt   DateTime @default(now())
  updatedAt   DateTime @updatedAt

  profile      Profile?
  bookings     Booking[]
  teamMembers  TeamMember[]
  interests    UserInterest[]
  notifications Notification[]
}

model Booking {
  id          String  @id @default(uuid()) @db.Uuid
  userId      String  @db.Uuid
  coworkingId String  @db.Uuid
  zoneId      String  @db.Uuid
  teamId      String? @db.Uuid
  startAt     DateTime
  endAt       DateTime
  status      BookingStatus @default(PENDING)
  totalPrice  Float?
  qrToken     String?  @unique
  checkedInAt DateTime?
```


checkedOutAt DateTime?

```
user      User      @relation(...)
zone      Zone       @relation(...)
team      Team?      @relation(...)
teamMembers TeamBookingMember[]
```

```
@@index([zoneId, startAt, endAt])
@@index([status])
@@index([qrToken])
}
```

```
model TeamBookingMember {
  id      String @id @default(uuid()) @db.Uuid
  bookingId String @db.Uuid
  userId  String @db.Uuid
  qrToken String @unique

  @@unique([bookingId, userId])
}
```

```
model InterestCategory {
  id   String @id @default(uuid()) @db.Uuid
  name String @unique
  slug String @unique

  users      UserInterest[]
  coworkings CoworkingInterest[]
}
```

```
model UserInterest {
  userId      String @db.Uuid
  categoryId  String @db.Uuid

  @@unique([userId, categoryId])
}
```

```
model CoworkingInterest {
  coworkingId String @db.Uuid
  categoryId  String @db.Uuid

  @@unique([coworkingId, categoryId])
}
```

2. Алгоритм рекомендацій (recommendations.service.ts)

Функція jaccardSimilarity реалізує розрахунок коефіцієнта Жаккара між двома наборами ідентифікаторів категорій. Основна функція getRecommendedCoworkings формує ранжований список коворкінгів за трьома складовими – jaccard, collaborative та history.

```
function jaccardSimilarity(setA: string[], setB: string[]): number {
  if (setA.length === 0 && setB.length === 0) return 0;
  const a = new Set(setA);
  const b = new Set(setB);
  const intersection = [...a].filter((x) => b.has(x)).length;
  const union = new Set([...a, ...b]).size;
  return union === 0 ? 0 : intersection / union;
}
```

```
async getRecommendedCoworkings(userId: string) {
  const cacheKey = `rec:coworkings:${userId}`;
  const cached = await redis.get(cacheKey);
  if (cached) return JSON.parse(cached);
}
```

```

// Отримуємо інтереси користувача
const userInterests = await prisma.userInterest.findMany({
  where: { userId },
  select: { categoryId: true },
});
const userCategoryIds = userInterests.map((ui) => ui.categoryId);
if (userCategoryIds.length === 0) return [];

// Часове затухання історії бронювань:  $\exp(-\text{daysDiff} / 90)$ 
const nowMs = Date.now();
const userBookings = await prisma.booking.findMany({
  where: { userId, status: { in: ["CONFIRMED", "COMPLETED"] } },
  select: { coworkingId: true, createdAt: true },
  take: 50,
});
const historyAffinity: Record<string, number> = {};
for (const b of userBookings) {
  const daysDiff = (nowMs - b.createdAt.getTime()) / 86_400_000;
  const decayedScore = Math.exp(-daysDiff / 90);
  historyAffinity[b.coworkingId] =
    (historyAffinity[b.coworkingId] ?? 0) + decayedScore;
}
const maxHistory = Math.max(...Object.values(historyAffinity), 1);
for (const k of Object.keys(historyAffinity)) {
  historyAffinity[k] = (historyAffinity[k] ?? 0) / maxHistory;
}

// Collaborative filtering: популярність серед схожих користувачів
const similarUsers = await prisma.userInterest.findMany({
  where: { categoryId: { in: userCategoryIds }, userId: { not: userId } },
  select: { userId: true },
  distinct: ["userId"],
});
const popularBookings = await prisma.booking.groupBy({
  by: ["coworkingId"],
  where: { userId: { in: similarUsers.map((u) => u.userId) } },
  _count: { coworkingId: true },
});
const collaborativeScore: Record<string, number> = {};
const maxBookings = popularBookings[0]?._count.coworkingId ?? 1;
for (const pb of popularBookings) {
  collaborativeScore[pb.coworkingId] = pb._count.coworkingId / maxBookings;
}

// Фінальна формула:  $\text{score} = 0.5 * \text{jaccard} + 0.2 * \text{collaborative} + 0.3 * \text{history}$ 
const coworkings = await prisma.coworking.findMany({
  where: { isActive: true },
  include: { coworkingInterests: { select: { categoryId: true } } },
});
const scored = coworkings.map((cw) => {
  const cwCategoryIds = cw.coworkingInterests.map((ci) => ci.categoryId);
  const jaccard = jaccardSimilarity(userCategoryIds, cwCategoryIds);
  const collaborative = collaborativeScore[cw.id] ?? 0;
  const history = historyAffinity[cw.id] ?? 0;
  const matchScore = Math.min(1, 0.5 * jaccard + 0.2 * collaborative
    + 0.3 * history);
  return { ...cw, matchScore: Math.round(matchScore * 100) };
});

const result = scored
  .filter((s) => s.matchScore > 0)
  .sort((a, b) => b.matchScore - a.matchScore);

```

```

    await redis.set(cacheKey, JSON.stringify(result), "EX", 900);
    return result;
}

```

3. Модуль автентифікації (auth.service.ts, фрагмент)

Нижче наведено реалізацію налаштування та підтвердження двофакторної автентифікації через TOTP, а також механізм збереження refresh-токену у вигляді argon2-хешу в базі даних.

```

const saveRefreshTokenHash = async (
  userId: string,
  refreshToken: string
): Promise<void> => {
  const refreshTokenHash = await argon2.hash(refreshToken);
  await prisma.user.update({
    where: { id: userId },
    data: { refreshTokenHash },
  });
};

```

```

const consumeBackupCode = async (
  userId: string,
  backupCode: string
): Promise<boolean> => {
  const user = await prisma.user.findUnique({
    where: { id: userId },
    select: { backupCodes: true },
  });
  if (!user || user.backupCodes.length === 0) return false;

  for (const [index, hash] of user.backupCodes.entries()) {
    const isValid = await argon2.verify(hash, backupCode);
    if (!isValid) continue;
    // Фізично видаляємо використаний код з масиву
    const updated = user.backupCodes.filter((_, i) => i !== index);
    await prisma.user.update({
      where: { id: userId },
      data: { backupCodes: updated },
    });
    return true;
  }
  return false;
};

```

```

async setupTwoFactor(userId: string) {
  const user = await prisma.user.findUnique({ where: { id: userId } });

  // Генеруємо TOTP-секрет через speakeasy
  const secret = speakeasy.generateSecret({
    length: 20,
    name: `Coworking Platform (${user.email})`,
    issuer: "Coworking Platform",
  });

  // Зберігаємо секрет але НЕ активуємо 2FA до підтвердження
  await prisma.user.update({
    where: { id: userId },
    data: { twoFactorSecret: secret.base32, isTwoFactorEnabled: false },
  });

  const otpauthUrl = secret.otpauth_url ?? "";
  const qrDataURL = await QRCode.toDataURL(otpauthUrl);
  return { otpauthUrl, qrDataURL };
},

```

```

async verifyTwoFactor(userId: string, code: string, ip?: string) {
  const user = await prisma.user.findUnique({ where: { id: userId } });
  const valid = speakeasy.totp.verify({
    secret: user.twoFactorSecret,
    token: code,
    encoding: "base32",
    window: 1, // допуск ±30 секунд
  });

  if (!valid) {
    if (ip) await consumeTwoFactor(`${ip}:${user.email}`);
    unauthorized("Invalid 2FA code");
  }

  // Генеруємо 8 backup-кодів і хешуємо через argon2
  const backupCodes = Array.from({ length: 8 }, () => {
    const a = randomBytes(2).toString("hex").toUpperCase();
    const b = randomBytes(2).toString("hex").toUpperCase();
    return `${a}-${b}`;
  });
  const backupHashes = await Promise.all(
    backupCodes.map((item) => argon2.hash(item))
  );

  await prisma.user.update({
    where: { id: userId },
    data: { isTwoFactorEnabled: true, backupCodes: backupHashes },
  });

  return { backupCodes }; // plaintext повертається лише один раз
},

```

4. Модуль бронювання (bookings.service.ts, фрагмент)

Функція createBooking реалізує повний цикл створення бронювання з перевіркою місткості зони та захистом від подвійного бронювання через Serializable транзакцію.

```

async createBooking(userId, input, ip?) {
  const startAt = new Date(input.startAt);
  const endAt = new Date(input.endAt);

  if (startAt >= endAt) badRequest("startAt must be lower than endAt");

  const booking = await prisma.$transaction(async (tx) => {

    // Перевіряємо що зона належить вказаному коворкінгу
    const zone = await tx.zone.findUnique({ where: { id: input.zoneId } });
    if (!zone || zone.coworkingId !== input.coworkingId)
      badRequest("Zone does not belong to coworking");

    // Для командного бронювання перевіряємо членство
    if (input.teamId) {
      const member = await tx.teamMember.findUnique({
        where: { teamId_userId: { teamId: input.teamId, userId } },
      });
      if (!member) forbidden("User is not a member of this team");
    }

    // Знаходимо всі перетинаючі бронювання зі статусами що займають місця
    const overlapping = await tx.booking.findMany({
      where: {
        zoneId: input.zoneId,
        status: { in: ["PENDING", "CONFIRMED", "CHECKED_IN"] },
      },
    });
  });
}

```

```

    startAt: { lt: endAt },
    endAt: { gt: startAt },
  },
  include: { team: { include: { members: true } } },
});

const occupiedSeats = overlapping.reduce(
  (sum, b) => sum + (b.team ? b.team.members.length : 1), 0
);

let requestedSeats = 1;
if (input.teamId) {
  const team = await tx.team.findUnique({
    where: { id: input.teamId },
    include: { members: true },
  });
  requestedSeats = team?.members.length ?? 1;
}

// Перевірка місткості зони
if (occupiedSeats + requestedSeats > zone.capacity) {
  conflict(`Not enough seats. Capacity: ${zone.capacity},
    occupied: ${occupiedSeats}, requested: ${requestedSeats}`);
}

const price = await pricingService.calculatePrice(
  input.zoneId, startAt, endAt
);

return tx.booking.create({
  data: {
    userId, coworkingId: input.coworkingId, zoneId: input.zoneId,
    teamId: input.teamId, startAt, endAt,
    note: input.note, totalPrice: price > 0 ? price : undefined,
    status: "CONFIRMED",
  },
  include: { coworking: true, zone: true, team: true },
});

}, { isolationLevel: Prisma.TransactionIsolationLevel.Serializable });

// Створюємо окремий QR-квиток для кожного учасника команди
if (input.teamId && booking.team) {
  const members = await prisma.teamMember.findMany({
    where: { teamId: input.teamId },
    select: { userId: true },
  });
  await prisma.teamBookingMember.createMany({
    data: members.map((m) => ({
      bookingId: booking.id,
      userId: m.userId,
      qrToken: randomUUID(),
    })),
    skipDuplicates: true,
  });
}

// Інвалідуємо кеш рекомендацій після нового бронювання
void redis.del(`rec:cworkings:${userId}`);

return booking;
},

```