Полтавський університет економіки і торгівлі
Навчально-науковий інститут денної освіти
Форма навчання денна
Кафедра комп’ютерних наук та інформаційних технологій

Допускається до захисту
Завідувач кафедри
_______Олена ОЛЬХОВСЬКА
(підпис)

«_____»______202_ р.

КВАЛІФІКАЦІЙНА РОБОТА
на тему

«РОЗРОБКА ВЕБ-ПОРТАЛУ ДЛЯ ОНЛАЙН-БРОНЮВАННЯ СПОРТИВНИХ МАЙДАНЧИКІВ І ЗАЛІВ»

зі спеціальності 122 Комп’ютерні науки
освітня програма «Комп’ютерні науки»
ступеня бакалавр

Виконавець роботи Лянник Станіслав Вячеславович
______«____» ________202_ р.
(підпис)

Науковий керівник доцент, к.п.н. Кошова О. П.
______«____»_________202_ р.
(підпис)

Рецензент

ПОЛТАВА 2026

РЕФЕРАТ

Записка: 72 с., 15 рис., 3 таблиці, 1 додаток, 17 джерел.

ВЕБ-ПОРТАЛ, ОНЛАЙН-БРОНЮВАННЯ, СПОРТИВНІ МАЙДАНЧИКИ, REACT, NODE.JS, EXPRESS, PRISMA, POSTGRESQL, JWT, REST API, TYPESCRIPT, DOCKER

Об'єктом дослідження є процес онлайн-бронювання спортивних майданчиків та залів як взаємодія користувача, адміністратора та інформаційної системи.

Предметом дослідження є методи, моделі та програмні засоби проєктування клієнт-серверних веб-застосунків для управління бронюваннями спортивних об'єктів.

Метою роботи є розробка веб-порталу для онлайн-бронювання спортивних майданчиків і залів, що забезпечує повний цикл взаємодії: від пошуку об'єкта та перегляду розкладу до формування бронювання, оплати та подальшого керування замовленнями.

Результатом роботи стало розроблення веб-порталу «Sport Booking Portal» на основі архітектури SPA з використанням React і TypeScript на клієнтській стороні та Node.js з Express на серверній. Реалізовано ключові модулі:

- модуль автентифікації та авторизації - реєстрація, вхід, оновлення сесії та вихід на основі JWT з використанням HttpOnly cookie і ротацією refresh-токенів;
- модуль каталогу спортивних об'єктів - фільтрація за типом, містом, ціновим діапазоном і ознакою закритого приміщення, перегляд деталей, графіка роботи та галереї зображень;
- модуль перевірки доступності часових слотів - побудова списку доступних інтервалів на день із урахуванням розкладу, тривалості та наявних бронювань;
- модуль бронювання - формування замовлення, контроль конфліктів у транзакції, розрахунок вартості за тривалістю та ціною за годину;
- модуль мокової оплати - імітація платіжної операції з трьома сценаріями

(success, cancel, fail), оновлення статусу бронювання та генерація транзакційного ідентифікатора;

- особистий кабінет користувача - перегляд активних, оплачених та скасованих бронювань, скасування й оплата;
- адміністративна панель - керування майданчиками, тижневим розкладом, зображеннями, переглядання й оновлення статусів бронювань.

Особливості: повністю типізована кодова база на TypeScript 5.9, монорепозиторій на основі pnpm workspaces, валідація вхідних даних бібліотекою Zod, обмеження частоти запитів (rate limiting) на чутливих маршрутах, підтримка ролей USER та ADMIN із серверним та клієнтським контролем, контейнеризація усіх компонентів через Docker Compose з PostgreSQL 16 та nginx, адаптивний інтерфейс на основі React 19.

Проведено інтеграційне перевіряння ключової бізнес-логіки модулів автентифікації та бронювання за допомогою node:test і supertest. Підтверджено коректність алгоритму перевірки конфліктів часових слотів, обробки сценаріїв мокової оплати, дублювання адрес електронної пошти та обмежень доступу за ролями.

«Sport Booking Portal» може бути використаний як основа для комерційного сервісу бронювання спортивних об'єктів, адаптований під потреби фітнес-центрів, спортивних клубів та муніципальних спортивних установ.

ЗМІСТ

ВСТУП 7
ПОСТАНОВКА ЗАДАЧІ 10
1. ІНФОРМАЦІЙНИЙ ОГЛЯД 12
1.1. Аналіз предметної області онлайн-бронювання спортивних об'єктів 12
1.2. Огляд існуючих веб-сервісів для бронювання спортивних об'єктів 15
1.3. Порівняльний аналіз функціональних можливостей аналогічних систем 18
2. ТЕОРЕТИЧНА ЧАСТИНА 21
2.1. Обґрунтування вибору технологій 21
2.2. Архітектура клієнт-серверного веб-порталу 23
2.3. Проєктування реляційної моделі даних 28
3. ПРАКТИЧНА ЧАСТИНА 32
3.1. Налаштування середовища і структури проєкту 32
3.2. Реалізація серверної частини: REST API і модуль автентифікації 34
3.3. Реалізація модуля бронювання та перевірки доступності слотів 36
3.4. Реалізація мокової платіжної системи 38
3.5. Реалізація клієнтської частини й адміністративної панелі 41
3.6. Тестування ключових модулів 43
3.7. Інструкція користувача 44
ВИСНОВКИ 54
СПИСОК ІНФОРМАЦІЙНИХ ДЖЕРЕЛ 57
ДОДАТОК А 59

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ, ТЕРМІНІВ

Умовні позначення, символи, скорочення, терміни	Пояснення умовних позначень, скорочень, символів
API	Application Programming Interface - інтерфейс програмування застосунків
SPA	Single Page Application - односторінковий веб-застосунок
UI	User Interface - користувацький інтерфейс
UX	User Experience - досвід взаємодії користувача із системою
REST	Representational State Transfer - архітектурний стиль побудови веб-сервісів
HTTP	HyperText Transfer Protocol - протокол передавання даних у веб-середовищі
HTTPS	HyperText Transfer Protocol Secure - захищена версія HTTP
JSON	JavaScript Object Notation - текстовий формат обміну даними
ORM	Object-Relational Mapping - технологія відображення об'єктів на реляційну модель
JWT	JSON Web Token - стандарт токенів для автентифікації та авторизації
CRUD	Create, Read, Update, Delete - базові операції з даними
SQL	Structured Query Language - мова структурованих запитів
DOM	Document Object Model - об'єктна модель документа

HTML	HyperText Markup Language - мова розмітки гіпертекстових документів
CSS	Cascading Style Sheets - каскадні таблиці стилів
CORS	Cross-Origin Resource Sharing - механізм спільного доступу до ресурсів
TLS	Transport Layer Security - протокол захисту транспортного рівня
URL	Uniform Resource Locator - уніфікований локатор ресурсу
RBAC	Role-Based Access Control - управління доступом на основі ролей
ПЗ	програмне забезпечення
БД	база даних
СУБД	система управління базами даних

ВСТУП

Сучасний ринок фізичної культури і спорту активно трансформується під впливом цифрових технологій. Усе більше власників фітнес-центрів, футбольних, тенісних і багатопрофільних залів переводять взаємодію з клієнтами у цифровий формат: електронне розкладання, онлайн-оплата та автоматизоване керування доступом стають стандартом галузі. Традиційні канали запису на заняття - телефонні дзвінки, паперові журнали або обмін повідомленнями у месенджерах - не дозволяють забезпечити прозорість і швидкість обслуговування, призводять до подвійних бронювань та втрати замовлень.

Запит на зручні цифрові сервіси, що дозволяють знайти необхідний спортивний об'єкт, обрати час і здійснити оплату в кілька кліків, зростає паралельно з поширенням смартфонів і мобільного інтернету. Водночас більшість власників невеликих спортивних закладів не має ресурсів на впровадження дорогих комерційних рішень - вони потребують гнучкого програмного продукту з можливістю адаптації під власну бізнес-модель та подальшої інтеграції з реальними платіжними системами.

Актуальність теми зумовлена необхідністю автоматизації процесів бронювання спортивних майданчиків, усунення ризиків подвійних бронювань, підвищення зручності користувачів та надання адміністраторам інструментів управління каталогом об'єктів. Розробка відкритої клієнт-серверної платформи на сучасному стеку технологій дозволяє забезпечити масштабованість, високу швидкість роботи та подальшу інтеграцію з реальними платіжними сервісами.

Метою роботи є розробка веб-порталу для онлайн-бронювання спортивних майданчиків і залів, що забезпечує повний цикл взаємодії: від пошуку об'єкта та перегляду розкладу до формування бронювання, оплати та подальшого керування замовленнями.

Для досягнення поставленої мети у роботі визначено такі основні завдання:

- проаналізувати предметну область онлайн-бронювання спортивних об'єктів та

дослідити існуючі аналоги;

- обґрунтувати вибір стеку технологій для клієнтської та серверної частин веб-порталу;
- спроєктувати клієнт-серверну архітектуру системи та реляційну модель бази даних;
- реалізувати серверний REST API з урахуванням безпеки, валідації даних та обмеження частоти запитів;
- розробити клієнтську частину у вигляді SPA-застосунку з розділенням функціоналу між ролями користувача та адміністратора;
- реалізувати алгоритм перевірки доступності часових слотів та формування бронювання без конфліктів;
- реалізувати модуль мокової оплати з імітацією взаємодії з платіжною системою;
- підготувати адміністративну панель для керування каталогом об'єктів та бронюваннями;
- провести інтеграційне тестування ключових модулів за допомогою node:test і supertest.

Об'єктом дослідження є процес онлайн-бронювання спортивних майданчиків та залів як взаємодія користувача, адміністратора та інформаційної системи.

Предметом дослідження є методи, моделі та програмні засоби проєктування клієнт-серверних веб-застосунків для управління бронюваннями спортивних об'єктів.

Методи дослідження: системний аналіз предметної області, порівняльний аналіз існуючих сервісів, методи об'єктно-орієнтованого проєктування, REST-архітектура, методи реляційного моделювання даних, сучасні підходи до розробки SPA-застосунків.

Практичне значення роботи полягає у розробленому програмному продукті «Sport Booking Portal», який може бути впроваджений у діяльність спортивних закладів, фітнес-центрів або муніципальних організацій. Портал забезпечує зручне

бронювання, автоматизований облік замовлень та централізоване адміністрування каталогу об'єктів.

Кваліфікаційна робота складається зі вступу, постановки задачі, трьох розділів, висновків, списку інформаційних джерел із 15 найменувань та одного додатку.

ПОСТАНОВКА ЗАДАЧІ

У межах кваліфікаційної роботи необхідно розробити веб-портал для онлайн-бронювання спортивних майданчиків і залів, який забезпечує повний цикл взаємодії кінцевого користувача з каталогом об'єктів: від пошуку та перегляду розкладу до формування замовлення, здійснення мокового платежу та управління активними бронюваннями. Система має підтримувати дві ролі користувачів - звичайний користувач (USER) та адміністратор (ADMIN) - і надавати відповідний функціонал для кожної ролі.

Розроблений програмний продукт повинен забезпечувати коректну перевірку доступності часових слотів, унеможливлювати подвійне бронювання одного об'єкта на однаковий інтервал часу, а також автоматично розраховувати вартість бронювання на основі тривалості та ціни за годину. Інтерфейс клієнтської частини має бути адаптивним і зручним для використання як на настільних, так і на мобільних пристроях.

Для досягнення поставленої задачі необхідно виконати такі підзадачі:

1. провести аналіз предметної області онлайн-бронювання спортивних об'єктів та окреслити ключові бізнес-процеси;
2. дослідити сучасні веб-сервіси бронювання спортивних майданчиків і виявити їх сильні та слабкі сторони;
3. здійснити порівняльний аналіз функціональних можливостей аналогічних систем та обґрунтувати доцільність розробки власного рішення;
4. обґрунтувати вибір технологій для клієнтської та серверної частин веб-порталу;
5. спроєктувати загальну архітектуру системи з урахуванням поділу на клієнт, сервер та базу даних;
6. спроєктувати реляційну модель бази даних для зберігання користувачів, спортивних об'єктів, розкладу, бронювань та платежів;
7. реалізувати серверну частину у вигляді REST API на основі Node.js та Express

із застосуванням ORM Prisma;

8. реалізувати механізм автентифікації на основі JWT-токенів з використанням HttpOnly cookie та підтримкою оновлення сесії з ротацією refresh-токенів;
9. реалізувати валідацію вхідних даних на серверній стороні за допомогою бібліотеки Zod та обмеження частоти запитів через express-rate-limit;
10. розробити клієнтську частину у вигляді SPA-застосунку на основі React і TypeScript із використанням збірника Vite;
11. реалізувати навігацію між сторінками засобами React Router та управління станом авторизації через AuthContext;
12. реалізувати сторінки каталогу, деталей об'єкта, особистого кабінету та сторінок адміністратора;
13. реалізувати алгоритм перевірки доступності часових слотів та формування бронювання;
14. реалізувати мокову платіжну систему для імітації онлайн-оплати замовлень;
15. реалізувати адміністративну панель для керування майданчиками, розкладом, зображеннями та бронюваннями;
16. провести інтеграційне тестування ключових модулів за допомогою node:test та supertest;
17. підготувати інструкцію користувача для опису послідовності роботи з порталом.

Результатом виконання кваліфікаційної роботи має стати працездатний веб-портал «Sport Booking Portal», що реалізує всі перелічені функціональні вимоги, використовує сучасні технології розробки та готовий до подальшого розширення й інтеграції з реальними платіжними та нотифікаційними сервісами.

1. ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1. Аналіз предметної області онлайн-бронювання спортивних об'єктів

Онлайн-бронювання спортивних майданчиків та залів належить до широкого класу сервісів електронного резервування ресурсів, що історично сформувався на базі задач у сферах готельного, туристичного та ресторанного бізнесу. Головна особливість предметної області полягає в обмеженості ресурсу - кожен майданчик має фіксовану кількість часових слотів протягом робочого дня, і кожен слот може бути одночасно придбаний лише одним користувачем. Ця обмеженість формує основний ризик бізнес-процесу - подвійне бронювання, яке неприпустиме у системах, що працюють у реальному часі.

Предметна область охоплює взаємодію трьох основних суб'єктів - кінцевого користувача, власника або адміністратора спортивного об'єкта та інформаційної системи. Користувач здійснює пошук відповідного майданчика за типом (футбольне поле, баскетбольний, тенісний, фітнес-зал), містом, датою та тривалістю; переглядає розклад, ціни та опис; формує бронювання; здійснює оплату. Адміністратор управляє каталогом: додає нові об'єкти, змінює ціни, встановлює тижневий розклад роботи, модерує бронювання. Інформаційна система забезпечує облік даних, перевірку доступності, обчислення вартості, зберігання історії бронювань та комунікацію з платіжною системою [1, 2].

На загальному рівні бізнес-процес бронювання можна представити як послідовність дій, що починається з автентифікації користувача та завершується підтвердженням платежу (див. рис. 1.1).

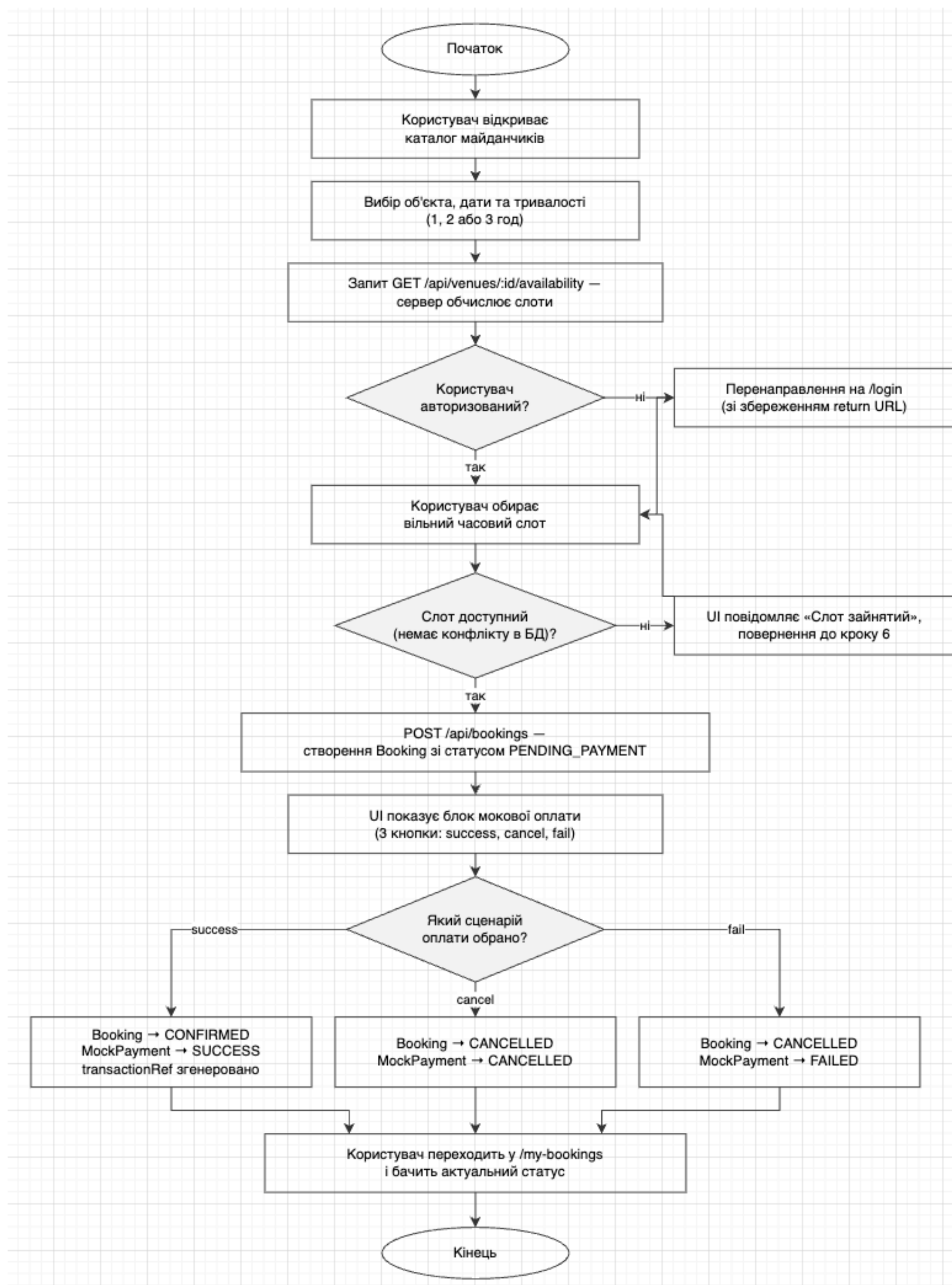

Рисунок 1.1 - Блок-схема бізнес-процесу онлайн-бронювання

Як видно зі схеми, процес містить точки прийняття рішення: перевірку наявності облікового запису, перевірку доступності слота й обробку результату оплати. Кожна гілка веде до відповідного стану замовлення - підтверджено, очікує оплати, скасовано. Цей граф станів є ключовим бізнес-правилом, яке має бути узгоджено між клієнтською, серверною частинами та сховищем даних.

Серед ключових сутностей предметної області слід виокремити такі: користувач (профіль, роль, історія бронювань); спортивний об'єкт (тип, локація,

ціна, місткість, графік роботи, галерея зображень); часовий слот (дата, початок, кінець, доступність); бронювання (відношення «користувач - об'єкт - слот» з фіксованою вартістю та статусом); платіж (відношення «бронювання - фінансова операція»). Усі сутності мають чіткі зв'язки, що дозволяє реалізувати їх у вигляді реляційної моделі.

Окремо варто розглянути часовий аспект бронювання. Спортивний об'єкт працює за тижневим графіком - для кожного дня тижня визначено час відкриття та закриття або ознаку вихідного дня. Доступні слоти не зберігаються в БД явним чином, а обчислюються динамічно на основі розкладу й активних бронювань. Такий підхід істотно зменшує обсяг даних і спрощує підтримку розкладу: при зміні графіка роботи об'єкта система автоматично перераховує доступність без необхідності масової реіндексації слотів.

Безпекові аспекти предметної області стосуються трьох основних загроз: несанкціонованого доступу до облікових записів, маніпуляцій з чужими бронюваннями та автоматизованих атак методом перебору. Перша загроза нейтралізується автентифікацією на основі JWT, друга - обов'язковою серверною перевіркою власника бронювання та ролі користувача (RBAC), третя - обмеженням частоти запитів (rate limiting) на чутливих маршрутах. Поєднання цих механізмів формує мінімально достатній рівень безпеки для веб-порталу комерційного класу.

Отже, предметна область онлайн-бронювання спортивних майданчиків і залів характеризується високим ступенем регламентованості бізнес-процесів, чітким розподілом ролей користувачів та необхідністю забезпечення цілісності транзакцій бронювання. Це робить її зручною для реалізації засобами клієнт-серверної архітектури з виокремленим REST API та реляційним сховищем даних.

1.2. Огляд існуючих веб-сервісів для бронювання спортивних об'єктів

Розвиток цифрових сервісів у сфері спорту призвів до появи спеціалізованих платформ, які дозволяють користувачам знаходити спортивні локації, переглядати розклад, резервувати майданчики та взаємодіяти з адміністрацією клубів через веб-інтерфейс або мобільний застосунок. Частина таких систем орієнтована на кінцевого користувача, який хоче швидко забронювати майданчик, а частина - на спортивні клуби та адміністраторів, яким потрібен інструмент для керування розкладом, оплатами та клієнтською базою.

Першим розглянутим аналогом є Playtomic - глобальна платформа для бронювання тенісних і паделових кортів, що активно розвивається в Європі та Латинській Америці. Сервіс надає кінцевим користувачам зручний пошук кортів за геолокацією, інтегровану систему створення відкритих ігор для пошуку партнерів і онлайн-оплату через інтегрований платіжний шлюз. Власникам клубів Playtomic пропонує повноцінну CRM-систему: керування клієнтами, статистику завантаженості, налаштування правил динамічного ціноутворення та інтеграцію з вуличними турнікетами (див. рис. 1.2) [3].

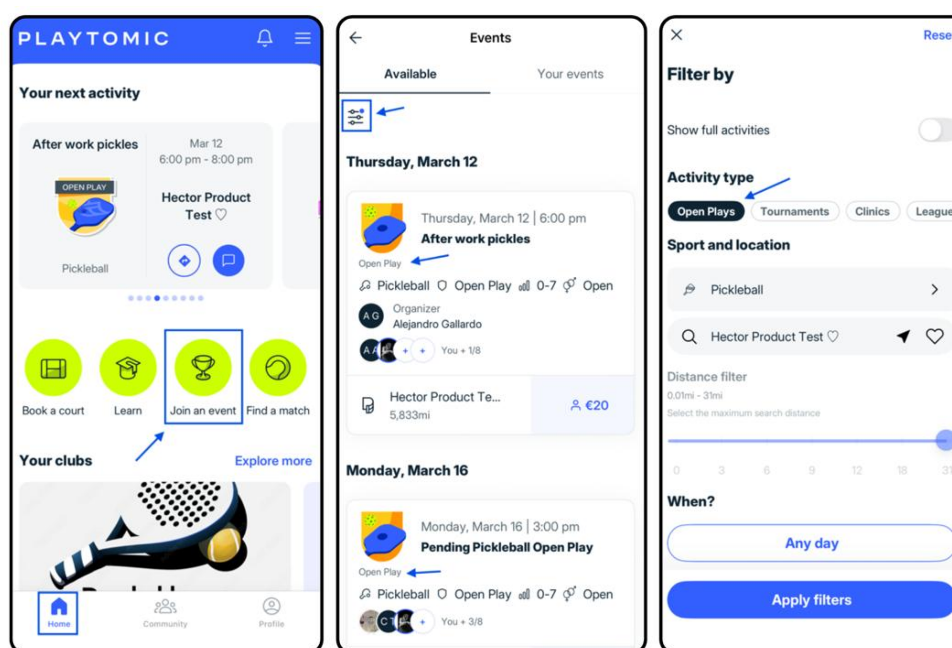

Рисунок 1.2 - Інтерфейс мобільного клієнта Playtomic

Сильні сторони Playtomic - глибоко проєктована мобільна частина, спортивна спільнота навколо платформи та аналітика для адміністраторів. Слабкі сторони з огляду на цілі цієї роботи - вузька спеціалізація на ракеткових видах спорту, закритий вихідний код і обов'язкова підписка для адміністраторів, що ускладнює адаптацію під невеликі заклади.

Другим розглянутим сервісом є CourtReserve - система керування клубами ракеткових видів спорту, орієнтована переважно на ринок Північної Америки. Платформа підтримує бронювання кортів, керування членством, обробку платежів, ведення інвойсів, тренувальних занять і подій, а також надає звітність за ключовими показниками діяльності клубу. CourtReserve відрізняється високою ступенем налаштовуваності правил резервування - наприклад, можна задавати пріоритет для членів клубу, обмеження на тривалість гри та автоматичне підтвердження бронювань (див. рис. 1.3) [4].

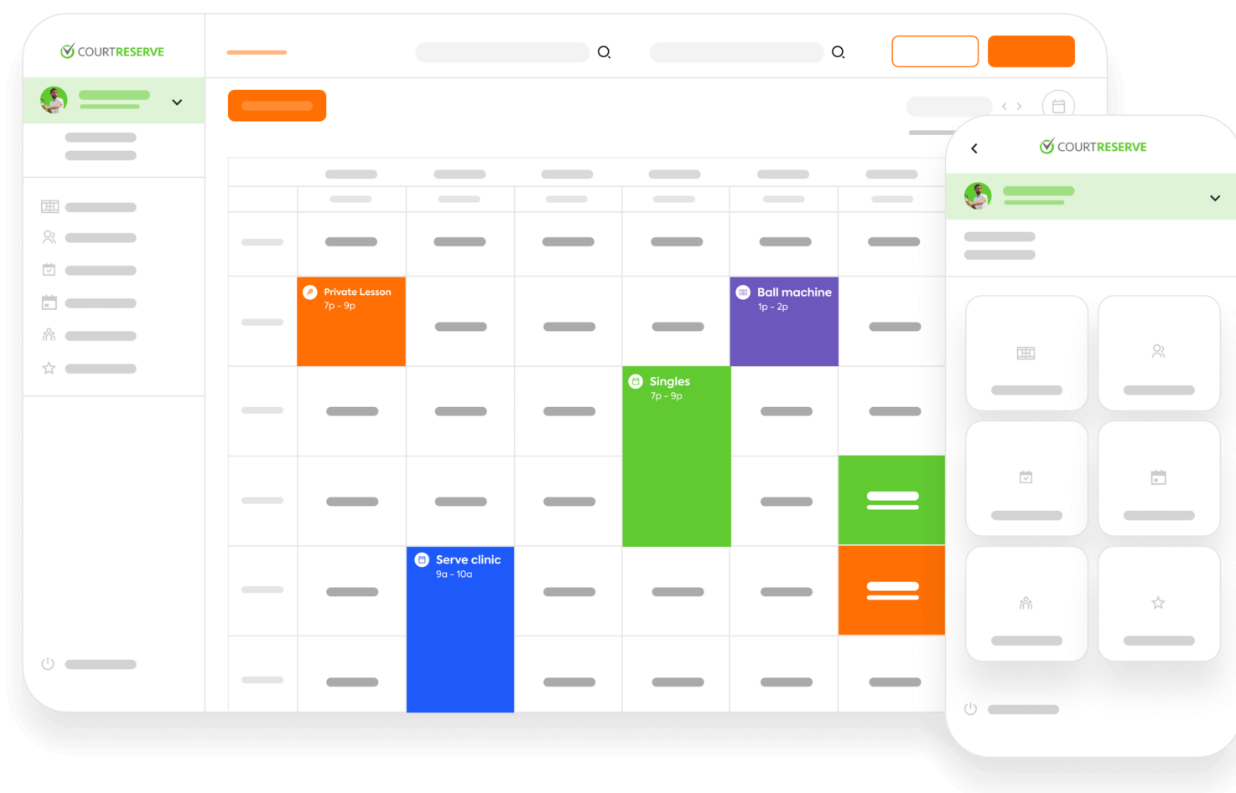

Рисунок 1.3 - Адміністративна панель CourtReserve

CourtReserve демонструє глибоку модель даних і складні механізми контролю доступу, проте через значну кількість службових екранів та налаштувань він перенасичений функціями для невеликого спортивного об'єкта. Крім того, ціна підписки робить рішення недоступним для одиночних залів і муніципальних установ.

Третім аналогом є OpenSports - універсальна платформа для організації спортивних подій, ліг, турнірів та тренувань. На відміну від Playtomic і CourtReserve, OpenSports фокусується не лише на бронюванні майданчика, але і на побудові спортивної спільноти: платформа підтримує реєстрацію на події, списки очікування, повернення коштів, груповий чат і нагадування. Розрахунок здійснюється через інтеграцію зі Stripe з підтримкою Apple Pay і Google Pay (див. рис. 1.4) [5].

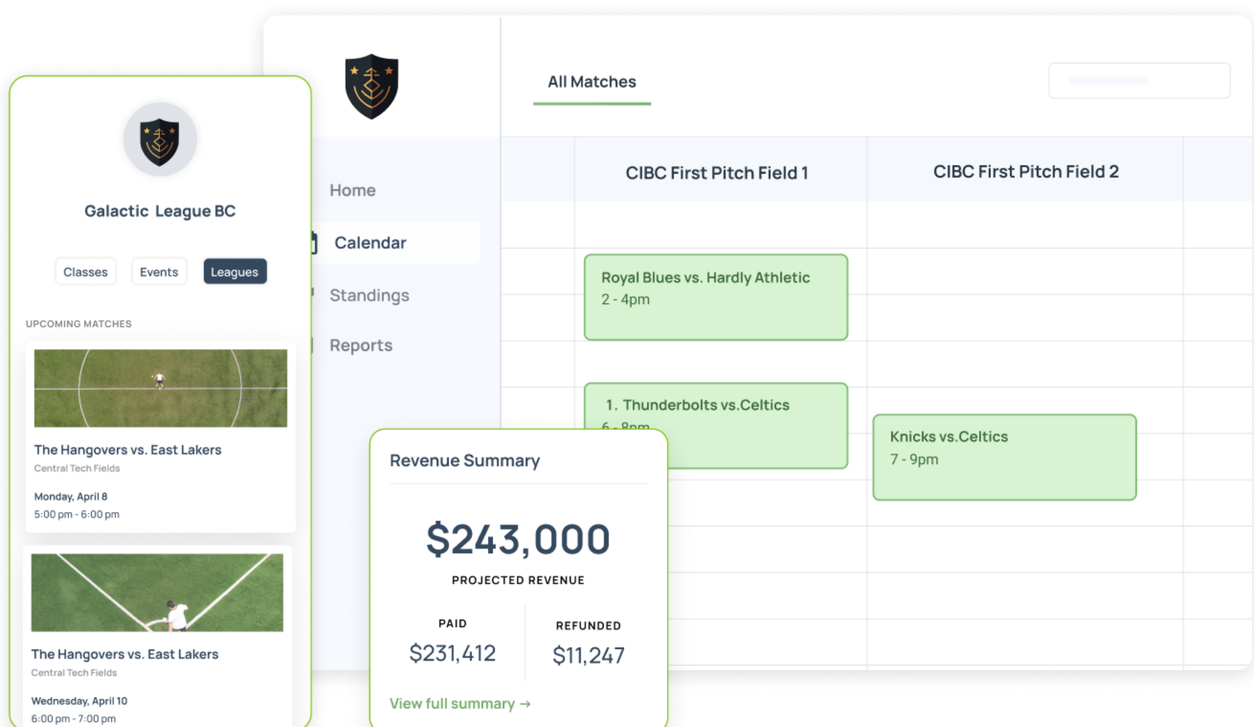

Рисунок 1.4 - Веб-інтерфейс OpenSports

OpenSports є хорошим прикладом того, як платформа бронювання трансформується у продуктову екосистему з елементами соціальної мережі. Однак у контексті задачі онлайн-бронювання конкретного майданчика на конкретний час

такий сервіс надає надлишкову функціональність, що ускладнює інтерфейс і підвищує вартість впровадження.

Окремо варто згадати Skedda - універсальну систему онлайн-бронювання простору й ресурсів, що не обмежується спортивною галуззю. Платформа надає гнучкий конфігуратор правил бронювання, керування доступом, інтеграцію з онлайн-оплатою, drag-and-drop редактор розкладу та механізм затвердження бронювань адміністратором. Skedda цінна тим, що демонструє узагальнений підхід до резервування ресурсів, який можна перенести на спортивну предметну область [6].

Аналіз чотирьох сервісів виявляє типовий набір функціональних блоків, без яких повноцінний веб-портал бронювання неможливий: автентифікація, каталог об'єктів, перевірка доступності, формування бронювання, обробка оплати, особистий кабінет користувача та адміністративна панель. Усі розглянуті системи реалізують цей мінімальний контур, однак істотно відрізняються у глибині додаткових модулів - соціальних механік, аналітики, фінансових інтеграцій. Саме базовий контур і обрано за орієнтир для проєктування власного веб-порталу.

1.3. Порівняльний аналіз функціональних можливостей аналогічних систем

Для систематизації результатів огляду доцільно подати порівняння аналогів і власної розробки у вигляді таблиці. Таблиця 1.1 містить ключові функціональні характеристики, важливі для обраної предметної області: підтримку каталогу з фільтрацією, перевірку доступності за тижневим розкладом, механізм бронювання з контролем конфліктів, оплату, адміністративну панель і ступінь спеціалізації на спортивній галузі.

Таблиця 1.1 - Порівняння функціональних можливостей аналогічних систем і веб-порталу «Sport Booking Portal»

Критерій	**Playtomic**	**CourtReserve**	**OpenSports**	**Skedda**	**Sport Booking Portal**
Каталог об'єктів з	+	+	+	+	+

фільтрацією					
Перевірка слотів за тижневим розкладом	+	+	+	+	+
Контроль конфліктів у транзакції БД	+	+	+	+	+
Підтримка ролей USER/ADMIN	+	+	+	+	+
Інтегрована онлайн-оплата	+	+	+	+	мокова
Соціальні механіки (відкриті ігри, спільноти)	+	-	+	-	-
Складні правила членства й абонементів	-	+	-	+	-
Відкритий вихідний код	-	-	-	-	+
Орієнтація на навчально-дослідницькі задачі	-	-	-	-	+

Як видно з таблиці 1.1, базовий контур - каталог, доступність, бронювання, ролі - присутній у всіх системах. Водночас комерційні платформи містять багато функціональних надбудов (членства, ліги, турніри, складні правила ціноутворення), які не є необхідними для невеликого фітнес-центру або муніципального стадіону. Власна розробка свідомо обмежена базовим контуром, однак містить ті можливості, яких немає у комерційних системах: відкритий вихідний код, локальний запуск через Docker Compose і повна типізація на TypeScript.

Аналіз показує, що для невеликих спортивних закладів та навчально-дослідницьких задач власна розробка має ряд переваг. По-перше, відсутність вендорської залежності - код, схема бази даних та інфраструктура повністю контролюються власником. По-друге, можливість адаптації під специфіку конкретного закладу: типи майданчиків, формати розкладу, правила бронювання можна змінювати на рівні коду. По-третє, інтеграція з реальною платіжною системою може бути виконана у будь-який момент шляхом заміни мокового модуля на адаптер до Stripe, LiqPay або Fondy без переписування решти системи.

Окремою перевагою є вибір сучасного технологічного стеку - React 19, TypeScript 5.9, Express 5, Prisma 6, PostgreSQL 16 - який забезпечує продуктивну розробку, надійну типізацію на всіх рівнях та простий процес розгортання у контейнерному середовищі. Це робить «Sport Booking Portal» прикладом збалансованого рішення, що поєднує практичну функціональність із сучасними інженерними практиками.

Підсумовуючи інформаційний огляд, можна стверджувати, що тема дипломної роботи є актуальною та потребує власної реалізації веб-порталу. Існуючі аналоги демонструють набір функціональних модулів, які мають бути присутніми у новому продукті, але не повністю задовольняють потреби локальних спортивних закладів через надлишкову функціональність, вартість підписки та відсутність відкритого коду. Це обґрунтовує доцільність розробки власного веб-порталу «Sport Booking Portal» із чітко визначеним базовим контуром і можливістю подальшого розширення.

2. ТЕОРЕТИЧНА ЧАСТИНА

2.1. Обгрунтування вибору технологій

Вибір технологій для веб-порталу «Sport Booking Portal» здійснювався з урахуванням таких критеріїв: широка спільнота й активний розвиток, підтримка типізації, продуктивність, можливість контейнеризації, наявність зрілих ORM-рішень, простота локального розгортання. У підсумку обрано стек, що включає React 19, TypeScript 5.9, Vite 7 та React Router 7 на стороні клієнта, Node.js з Express 5.1 - на стороні серверної логіки, Prisma 6 ORM у поєднанні з PostgreSQL 16 - для роботи з даними, JWT (jsonwebtoken 9), bcryptjs 3, Zod 4 та express-rate-limit - для безпекового шару, а також Docker Compose із nginx - для контейнерного розгортання.

Бібліотека React обрана як одна з найпоширеніших технологій для побудови SPA-застосунків. У версії 19 React надає механізм автоматичного пакетування оновлень стану (automatic batching), нативну підтримку Suspense для асинхронних компонентів і покращену систему ошибок через ErrorBoundary. Декларативний підхід до опису інтерфейсу та односпрямований потік даних роблять React зручним інструментом для побудови складних, але передбачуваних UI [7].

Мова TypeScript використовується наскрізно - як на клієнті, так і на сервері. Вона додає до JavaScript статичну систему типів, що дозволяє виявляти помилки на етапі компіляції, а не у runtime. Особливо цінною є можливість одного джерела істини для типів даних: контракти REST API, моделі бази даних та форми клієнта описуються типами TypeScript і Zod-схемами, що уніфікує валідацію та типізацію через увесь стек [8].

Збірник Vite вибрано як заміну застарілому Webpack. Vite використовує нативні ES-модулі браузера у режимі розробки, що дозволяє запускати dev-сервер за частки секунди й виконувати HMR (Hot Module Replacement) без повної

перезбірки. У production-режимі Vite використовує Rollup, що забезпечує оптимізований і компактний bundle [9].

Для маршрутизації клієнтського застосунку обрано React Router 7. Бібліотека підтримує вкладені маршрути, ledger-роути для адмінки та механізм захищених маршрутів через виокремлений компонент ProtectedRoute. Це дозволяє чітко розділити публічну, користувацьку й адміністративну частини інтерфейсу [10].

На серверній стороні використано Node.js з фреймворком Express 5.1. Express залишається стандартом de facto для побудови REST API на платформі Node, надаючи мінімальну, але достатню абстракцію над HTTP. У версії 5 Express отримав сучасну обробку асинхронних помилок, що знімає необхідність огортати кожен async-обробник у try/catch [11].

Для роботи з реляційною базою даних обрано Prisma 6 ORM. Prisma описує схему БД у файлі schema.prisma і генерує повністю типізований клієнт TypeScript, що автоматично синхронізує типи моделей з реальною структурою таблиць. Підтримка міграцій, db push для розробки, transactions API та query engine на Rust роблять Prisma одним із найпродуктивніших ORM для Node.js [12].

Як СУБД обрано PostgreSQL 16 - реляційну базу даних із підтримкою ACID-транзакцій, унікальних індексів, складних запитів і JSONB-полів. PostgreSQL стабільно входить до трійки найпопулярніших БД світу за версією рейтингу DB-Engines і чудово інтегрується з Prisma. Для тестового середовища також передбачена робота з SQLite через окремий schema.test.prisma - це забезпечує миттєвий запуск тестів без необхідності піднімати окремий сервер БД [13].

Безпека реалізована через комплекс бібліотек. JWT (jsonwebtoken) використовується для випуску access- та refresh-токенів, що зберігаються у HttpOnly cookie. Bcryptjs виконує хешування паролів за алгоритмом bcrypt з налаштовним cost-параметром. Zod виконує валідацію вхідних даних на серверній стороні: визначається схема для кожного запиту, і будь-яке порушення формату призводить до 400 Bad Request з повідомленням про помилку. Express-rate-limit обмежує частоту чутливих запитів - 20 запитів за 15 хвилин на /auth/* - для протидії brute-force атакам [14].

Контейнеризація реалізована через Docker Compose, що описує три сервіси: db (postgres:16-alpine з healthcheck), server (Express API з Dockerfile у /server) і client (React SPA, обслуговувана nginx у /client). Цей підхід забезпечує єдиний спосіб запуску всієї системи на будь-якому хості - для розгортання достатньо команди `docker compose up`. nginx у клієнтському контейнері виконує дві функції: віддає статичні файли SPA та проксує запити до /api/ на серверний контейнер [15].

Загальна сумісність обраних технологій із вимогами проєкту підтверджується практикою - всі модулі веб-порталу успішно реалізовані без необхідності замінювати компоненти стеку. Стек є збалансованим і дає змогу реалізувати масштабовані продакшн-сценарії після переходу від мокової оплати до реальної платіжної системи.

2.2. Архітектура клієнт-серверного веб-порталу

Архітектура веб-порталу «Sport Booking Portal» побудована за принципом тришарового клієнт-серверного застосунку. Перший шар - клієнтський SPA, другий - REST API на Node.js, третій - реляційне сховище PostgreSQL. Шари взаємодіють через чітко визначені інтерфейси: між клієнтом і сервером - HTTP/JSON через REST API, між сервером і базою даних - Prisma ORM. Загальна структура взаємодії компонентів зображена на рисунку 2.1 (див. рис. 2.1).

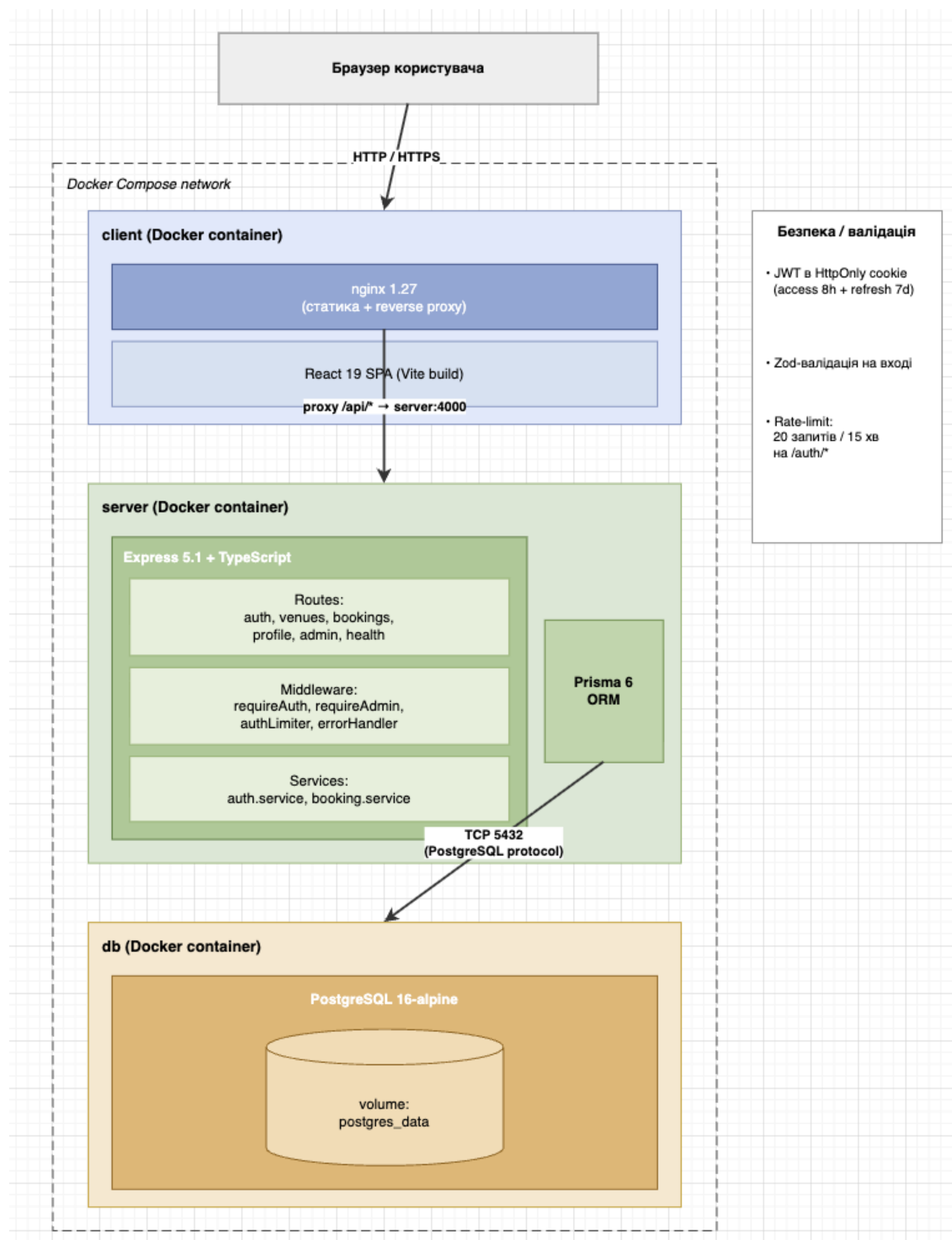

Рисунок 2.1 - Клієнт-серверна архітектура «Sport Booking Portal»

Як показано на рисунку 2.1, клієнтський браузер взаємодіє з контейнером nginx через HTTP. nginx виконує роль реверс-проксі: статичні ресурси SPA віддаються безпосередньо, а запити з префіксом /api/ перенаправляються на серверний контейнер на порту 4000. Серверний контейнер обслуговує REST API, виконує бізнес-логіку та звертається до контейнера PostgreSQL через Prisma. Усі контейнери розгорнуті в одній мережі Docker Compose, що ізолює внутрішні комунікації від зовнішнього середовища.

На серверній стороні застосунок організовано за класичною шаруватою архітектурою. Шар маршрутів (routes) розташований у /server/src/routes і містить файли auth.routes.ts, venues.routes.ts, bookings.routes.ts, profile.routes.ts, admin.routes.ts та health.routes.ts. Кожен файл реєструє свій набір ендпоінтів і викликає відповідні сервіси. Шар сервісів (/server/src/services) містить чисту бізнес-логіку: auth.service.ts відповідає за реєстрацію та автентифікацію, booking.service.ts - за створення, скасування й оплату бронювань. Шар бібліотек (/server/src/lib) містить допоміжні модулі: prisma.ts (синглтон Prisma Client), auth.ts (видавання JWT і керування cookie), availability.ts (побудова списку доступних слотів), date-time.ts (робота з часом), api-error.ts (стандартизована модель помилок), presenters.ts (серіалізація сутностей у DTO). Шар middleware (/server/src/middleware) реалізує перехресні концерни: auth.ts (requireAuth, requireAdmin), rate-limit.ts (authLimiter), error-handler.ts (глобальна обробка помилок).

Перелік ключових ендпоінтів REST API наведено у таблиці 2.1.

Таблиця 2.1 - Ключові маршрути REST API «Sport Booking Portal»

Метод	**Шлях**	**Призначення**	**Доступ**
POST	/api/auth/register	Реєстрація нового користувача	публічний (rate-limit)
POST	/api/auth/login	Вхід за email і паролем	публічний (rate-limit)
POST	/api/auth/refresh	Оновлення access-токена за refresh	публічний
POST	/api/auth/logout	Вихід і відкликання refresh-токенів	автентифікований
GET	/api/auth/me	Отримання поточного користувача	автентифікований
GET	/api/venues	Каталог майданчиків з фільтрами	публічний
GET	/api/venues/:identifier	Деталі майданчика за id або slug	публічний
GET	/api/venues/:identifier/availability	Доступні слоти на діапазон дат	публічний
GET	/api/bookings/me	Власні бронювання користувача	автентифікований
POST	/api/bookings	Створення бронювання	автентифікований
GET	/api/bookings/:id	Деталі бронювання	автентифікований
PATCH	/api/bookings/:id/cancel	Скасування бронювання	автентифікований
POST	/api/bookings/:id/mock-payment	Мокова оплата (success/cancel/fail)	автентифікований
GET, POST, PATCH	/api/admin/venues, /api/admin/bookings	Адмінські операції	роль ADMIN

Як видно з таблиці 2.1, маршрути організовано за принципом ресурс-орієнтованості: кожна сутність (auth, venue, booking) має власну групу URL-ів з очевидною семантикою HTTP-методів. Така структура добре масштабується і легко документується у форматі OpenAPI у разі потреби.

Клієнтський SPA побудований за компонентною моделлю React. Точка входу - компонент App у файлі /client/src/App.tsx, який обгортає всі маршрути у BrowserRouter і AuthProvider. Контекст AuthContext завантажує поточного користувача через GET /api/auth/me на старті застосунку та надає функції login, register, logout і refreshUser усім дочірнім компонентам. Захищені маршрути обгорнуті у компонент ProtectedRoute, який перевіряє стан користувача і за необхідності - наявність ролі ADMIN. Адміністративні сторінки додатково обгорнуті у вкладений роут з компонентом AdminLayout, що задає окрему навігацію адмінки.

Окремої уваги заслуговує підхід до автентифікації між клієнтом і сервером. Сервер встановлює два HttpOnly cookie - access (короткоживучий, 8 годин) і refresh (тривалий, 7 днів). Cookie не доступні з JavaScript, що значно зменшує ризик XSS-атак з викраденням токенів. Refresh-токен зберігається у БД у вигляді SHA-256-хешу, що дозволяє його відкликати під час logout. Кожне оновлення сесії передбачає ротацію - старий refresh видаляється, новий створюється - це обмежує час життя скомпрометованого токена. Загальна схема потоку JWT-автентифікації подана на рисунку 2.2 (див. рис. 2.2).

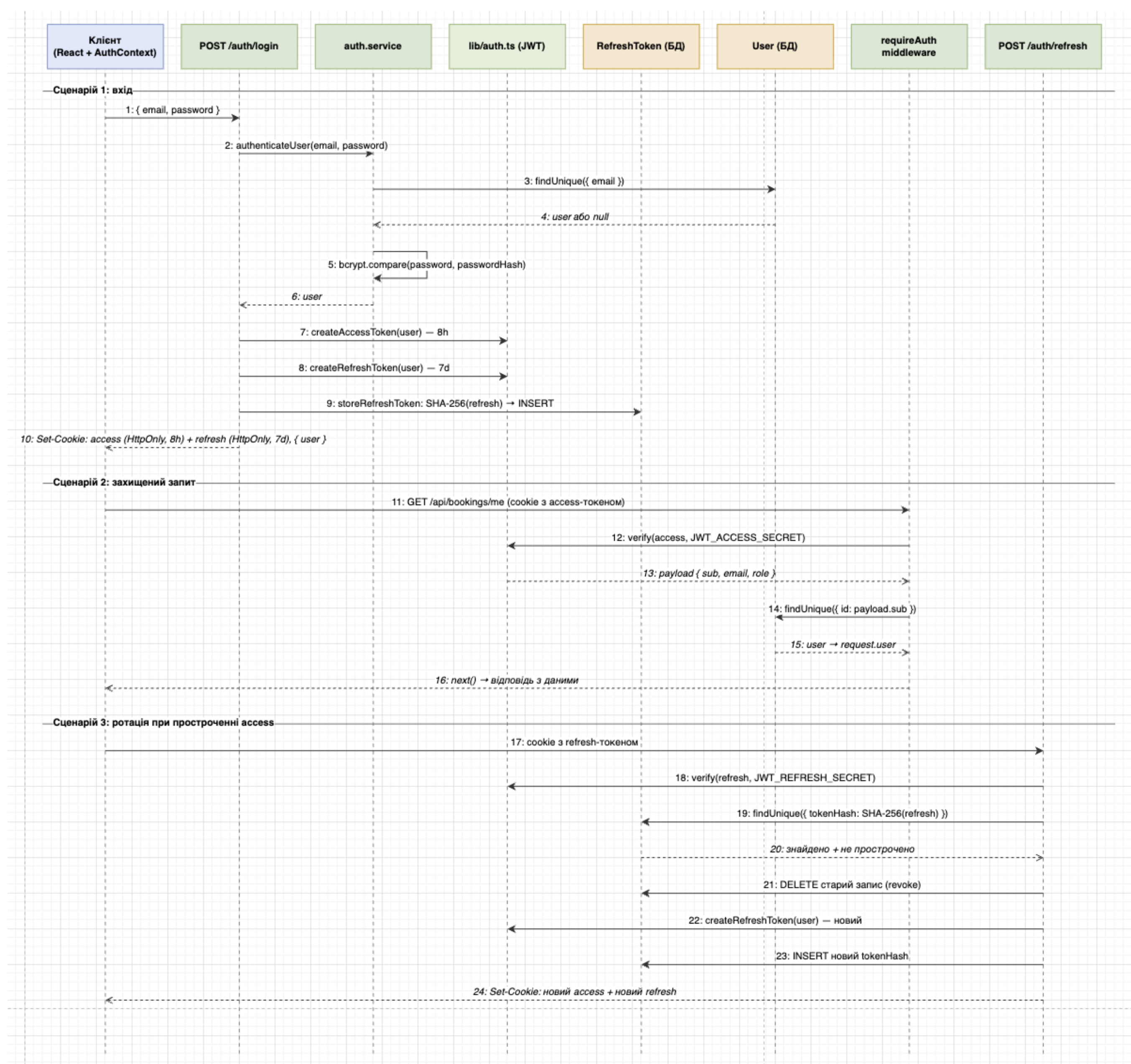

Рисунок 2.2 - Потік JWT-автентифікації з ротацією refresh-токенів

На рисунку 2.2 показано, як після успішного входу користувач отримує два cookie, далі при кожному захищеному запиті middleware requireAuth декодує access-токен і завантажує користувача з БД. Якщо access-токен прострочений, клієнт викликає /api/auth/refresh, що виконує перевірку refresh-токена в БД, видаляє старий і видає новий комплект cookie. Logout видаляє всі refresh-токени користувача, що призводить до примусового виходу з усіх пристроїв.

Для забезпечення обробки помилок усі сервіси викидають об'єкти ApiError зі статусом і повідомленням. Глобальний middleware errorHandler перетворює їх на

JSON-відповіді у форматі { success: false, message: ... }, а Zod-помилки валідації - на 400 з масивом полів-порушників. Це гарантує однорідний контракт відповіді між усіма ендпоінтами.

2.3. Проєктування реляційної моделі даних

Реляційна модель «Sport Booking Portal» спроєктована на основі ключових сутностей предметної області - користувач, спортивний об'єкт, графік роботи, бронювання, платіж - і додаткових службових сутностей. У схемі Prisma визначено сім моделей: User, RefreshToken, Venue, VenueImage, VenueSchedule, Booking, MockPayment, а також п'ять перелічуваних типів (enum): UserRole, VenueType, VenueStatus, BookingStatus, PaymentStatus. Загальна структура моделі та зв'язків між сутностями зображена на рисунку 2.3 (див. рис. 2.3).

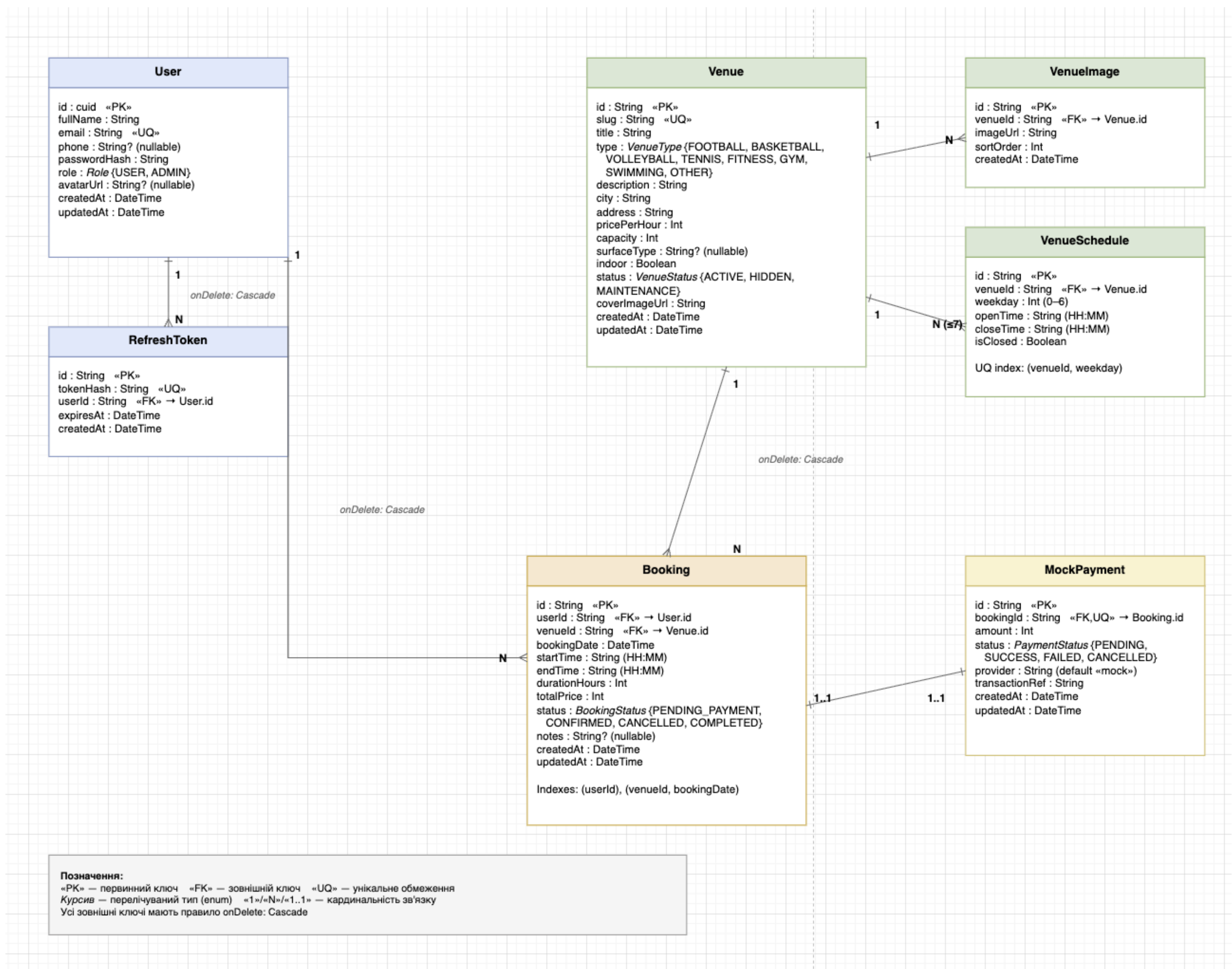

Рисунок 2.3 - Діаграма сутностей бази даних «Sport Booking Portal»

Сутність User зберігає базовий профіль користувача: id (cuid), fullName, email (унікальний), phone (необов'язковий), passwordHash, role (USER або ADMIN), avatarUrl, createdAt, updatedAt. Поле email слугує природним ідентифікатором при вході, тоді як id використовується усіма зовнішніми ключами як стабільний внутрішній ідентифікатор. Поле passwordHash містить bcrypt-хеш - реальний пароль ніколи не зберігається у БД.

Сутність RefreshToken реалізує серверний контроль refresh-токенів. Кожен виданий refresh-токен зберігається у БД у вигляді SHA-256-хешу, що дозволяє його відкликати без необхідності перевипуску всіх інших токенів користувача. Зв'язок User - RefreshToken побудовано як 1:N з каскадним видаленням: при видаленні користувача всі його refresh-токени теж видаляються.

Центральна сутність Venue описує спортивний об'єкт: id, slug (унікальний URL-ідентифікатор), title, type (VenueType: FOOTBALL, BASKETBALL, VOLLEYBALL, TENNIS, FITNESS, GYM, SWIMMING, OTHER), description, city, address, pricePerHour, capacity, surfaceType, indoor, status (VenueStatus: ACTIVE, HIDDEN, MAINTENANCE), coverImageUrl, createdAt, updatedAt. Поле slug дозволяє формувати читабельні URL-адреси (/venues/sport-arena-poltava замість /venues/cmc1...), що покращує SEO та UX. Поле status забезпечує адміністратору можливість приховувати об'єкт без видалення з БД - корисно для технічного обслуговування або сезонного зачинення.

Сутність VenueImage реалізує галерею зображень об'єкта (зв'язок 1:N): id, venueId, imageUrl, sortOrder, createdAt. Сортування за полем sortOrder дозволяє адміністратору керувати порядком виводу зображень незалежно від часу їх додавання.

Сутність VenueSchedule зберігає тижневий графік роботи. Унікальний ключ (venueId, weekday) гарантує, що для кожного об'єкта існує не більше одного запису на день тижня. Поля openTime та closeTime зберігаються як рядки у форматі HH:MM, що спрощує серіалізацію в JSON та порівняння з рядками часу бронювань. Прапорець isClosed дозволяє відмічати вихідні дні без необхідності видаляти запис.

Центральна транзакційна сутність Booking зберігає окреме бронювання: id, userId, venueId, bookingDate, startTime, endTime, durationHours, totalPrice, status (BookingStatus: PENDING_PAYMENT, CONFIRMED, CANCELLED, COMPLETED), notes, createdAt, updatedAt. Поля durationHours і totalPrice розраховуються при створенні бронювання і фіксуються - навіть якщо адміністратор пізніше змінить ціну за годину, історичне бронювання залишиться з первісною вартістю. Це принципово важливо для коректного обліку.

Для оптимізації запитів на сутність Booking накладено два додаткові індекси: за userId (для швидкого отримання списку бронювань користувача) та композитний індекс за (venueId, bookingDate) - для швидкої перевірки конфліктів і побудови розкладу об'єкта на день.

Сутність MockPayment реалізує модель платежу 1:1 до Booking: id, bookingId (unique), amount, status (PaymentStatus: PENDING, SUCCESS, FAILED, CANCELLED), provider (за замовчуванням «mock»), transactionRef, createdAt, updatedAt. Поле transactionRef генерується у форматі MOCK-YYYYMMDD-XXXXXXXX, що імітує референс реального платіжного шлюзу та полегшує подальшу заміну мокового модуля на адаптер до Stripe або LiqPay.

Зв'язки між сутностями реалізовані з налаштуванням каскадного видалення там, де це безпечно: при видаленні Venue видаляються всі його VenueImage, VenueSchedule та Booking; при видаленні User - його Booking і RefreshToken. Така політика спрощує адміністрування при тестуванні та забезпечує цілісність БД без необхідності писати спеціальні SQL-запити.

Загалом, спроєктована модель даних відповідає принципам нормалізації - сутності мають чітко визначені поля без дублювання, зовнішні ключі забезпечують посилкову цілісність, унікальні індекси запобігають появі дублікатів. Водночас модель збережена компактною: відсутні зайві службові таблиці, що могли б ускладнити розуміння. Це робить її гарним базисом для реалізації серверної частини веб-порталу.

3. ПРАКТИЧНА ЧАСТИНА

3.1. Налаштування середовища і структури проєкту

Проєкт «Sport Booking Portal» організовано як монорепозиторій на основі pnpm workspaces. Кореневий файл pnpm-workspace.yaml оголошує два пакети: client (React-застосунок) і server (Express API). Такий підхід дозволяє керувати залежностями і скриптами обох частин з одного місця, при цьому пакети залишаються повністю автономними та можуть розгортатися окремо.

У кореневому каталозі знаходяться три основні підпапки - client, server і docs, файл docker-compose.yml для контейнерного розгортання, а також pnpm-workspace.yaml і package.json з кореневими скриптами. Каталог client містить вихідний код React-застосунку, server - Express API, docs - технічну документацію (TECHNICAL_SPECIFICATION, DATA_MODEL, API_CONTRACT, SEED_RULES).

Кореневий package.json визначає набір скриптів, що покривають увесь життєвий цикл розробки. Скрипт `pnpm setup` виконує установку залежностей, генерацію Prisma Client, синхронізацію схеми з БД і завантаження тестових даних. Скрипт `pnpm dev` запускає клієнт і сервер паралельно через `pnpm -r --parallel --stream dev`, що значно зручніше за послідовний запуск у різних терміналах. Скрипт `pnpm test` запускає інтеграційні тести серверної частини. Скрипт `pnpm docker:up` піднімає увесь стек у контейнерах. Така мінімальна, але достатня кількість команд знижує поріг входу для нових розробників.

На серверній стороні встановлені залежності для побудови REST API: express 5.1, @prisma/client, prisma, jsonwebtoken, bcryptjs, zod, cors, cookie-parser, express-rate-limit, dotenv. Розробницькі залежності включають tsx (запуск TypeScript без попередньої компіляції) та supertest (тестування HTTP API). На клієнтській стороні встановлені react 19, react-dom, react-router-dom 7, devDependencies - vite, eslint, typescript, @vitejs/plugin-react.

Конфігурація TypeScript розділена на чотири файли. У клієнтській частині використано підхід project references: tsconfig.json з посиланнями на tsconfig.app.json (для коду застосунку) та tsconfig.node.json (для конфігураційних файлів самого Vite). У серверній частині достатньо одного tsconfig.json з опціями strict, esModuleInterop, moduleResolution: bundler і output до /dist. Така конфігурація забезпечує сувору типізацію без необхідності компроміс на швидкість збірки.

Контейнеризація реалізована через docker-compose.yml у корені проєкту. Файл описує три сервіси: db (postgres:16-alpine з healthcheck на pg_isready), server (Dockerfile у /server, healthcheck на корінь /), client (Dockerfile у /client з production-збіркою через Vite та подачею через nginx на порту 80). Між контейнерами налаштовано dependency conditions - server чекає, поки db не пройде healthcheck, client - поки server не буде здоровим. Том postgres_data забезпечує збереження даних PostgreSQL між перезапусками.

Окремої уваги заслуговує конфігурація змінних середовища. Файл .env.example у /server документує необхідні параметри: PORT, CLIENT_URL, DATABASE_URL, JWT_ACCESS_SECRET, JWT_REFRESH_SECRET. Розробник може скопіювати приклад у локальний .env і за потреби налаштувати власні значення. У docker-compose.yml ці значення перевизначаються - DATABASE_URL вказує на внутрішнє ім'я хоста db, CLIENT_URL - на http://localhost. Такий підхід дозволяє запускати застосунок як локально (з локальною PostgreSQL), так і в контейнерах.

Готова структура проєкту і налаштоване середовище формують міцний фундамент для реалізації функціональних модулів - серверної частини, клієнтської частини, тестів і контейнеризованого розгортання, які розглянуто далі у відповідних підрозділах.

3.2. Реалізація серверної частини: REST API і модуль автентифікації

Точкою входу серверного застосунку є функція createApp у файлі /server/src/app.ts. Вона створює екземпляр Express, реєструє middleware (cors з credentials, express.json, cookieParser), кореневий маршрут з метаданими сервісу, монтує apiRouter на префікс /api, додає 404-обробник і глобальний errorHandler. Така структура є стандартною для зрілих Node.js-застосунків і забезпечує єдину точку конфігурації.

Маршрути auth-модуля визначені у /server/src/routes/auth.routes.ts. Кожен маршрут спочатку валідує тіло запиту через Zod-схему, потім викликає відповідний сервіс і повертає JSON-відповідь у форматі { success: true, data: ... }. Реєстрація користувача має такий вигляд (фрагмент із реальним кодом):

```
authRouter.post("/auth/register", authLimiter, async (request,
response) => {
const payload = registerSchema.parse(request.body);
const user = await registerUser(prisma, payload);
const refreshToken = createRefreshToken({ id: user.id, email:
user.email, role: user.role });
await storeRefreshToken(user.id, refreshToken);
setAuthCookies(response, { id: user.id, email: user.email, role:
user.role }, refreshToken);
response.status(201).json({ success: true, data: { user:
serializeUser(user) } });
});
```

У наведеному коді послідовно виконуються чотири кроки: валідація payload через Zod, делегування створення користувача до сервісу registerUser, видавання refresh-токена та збереження його хешу у БД, встановлення HttpOnly cookie. Послідовність гарантує, що при будь-якому збої (дублікат email, хибний формат) відповідь буде корректною - Zod автоматично кине ZodError, який глобальний errorHandler перетворить на 400 Bad Request, а ApiError(409) із сервісу - на 409 Conflict.

Сам сервіс registerUser реалізовано у /server/src/services/auth.service.ts. Алгоритм наступний: спочатку перевіряється відсутність користувача з таким email; якщо існує - викидається ApiError(409, «A user with this email already exists»); інакше пароль хешується через bcrypt з cost-фактором 10, і створюється новий запис у таблиці User з роллю USER за замовчуванням. Аналогічна логіка реалізована для authenticateUser: пошук користувача за email, порівняння хеша через bcrypt.compare, повернення сутності або 401 Unauthorized.

Видавання та перевірка JWT-токенів інкапсульовані у /server/src/lib/auth.ts. Функції createAccessToken та createRefreshToken підписують токени двома різними секретами - JWT_ACCESS_SECRET (час життя 8 годин) та JWT_REFRESH_SECRET (час життя 7 днів). Функція setAuthCookies встановлює два HttpOnly cookie з однаковими параметрами безпеки: httpOnly: true, sameSite: «lax», path: «/». Параметр secure встановлюється у false для локального HTTP-середовища, але має бути увімкнений у продакшні з HTTPS.

Серверний контроль refresh-токенів реалізований через таблицю RefreshToken: при кожному видаванні нового токена обчислюється його SHA-256-хеш і створюється запис у БД. При оновленні сесії маршрут /auth/refresh виконує такі кроки: декодує refresh-токен з cookie, шукає його хеш у БД, перевіряє строк дії, видаляє старий запис, видає новий refresh-токен і встановлює нові cookie. При logout видаляються всі refresh-токени користувача (revokeAllUserRefreshTokens), що призводить до примусового виходу з усіх пристроїв.

Middleware requireAuth у /server/src/middleware/auth.ts реалізує захист маршрутів. Він декодує access-токен з cookie або заголовка Authorization: Bearer, шукає користувача за полем sub (id) у БД, додає його у request.user і передає керування далі. Якщо токен відсутній або користувача не знайдено - кидає ApiError(401). Допоміжний requireAdmin додатково перевіряє, що user.role === ADMIN.

Обмеження частоти запитів реалізоване через express-rate-limit. Ліміт authLimiter обмежує POST-запити до /auth/* до 20 за 15 хвилин з одного IP. GET-

запити пропускаються без обмежень - це дозволяє зберегти зручність для legitimate-користувачів і одночасно ускладнити автоматизовані атаки brute-force на пароль.

Усі помилки нормалізуються через клас ApiError(status, message) і глобальний errorHandler. Останній розрізняє типи помилок: ZodError перетворюється на 400 з масивом полів-порушників, ApiError - на статус і повідомлення з самого об'єкта, інші помилки - на 500 Internal Server Error з логуванням у консоль. Формат відповіді завжди однорідний: { success: false, message: ..., issues?: [...] }, що значно полегшує клієнтську обробку.

3.3. Реалізація модуля бронювання та перевірки доступності слотів

Модуль бронювання - найскладніший за бізнес-логікою модуль системи. Він повинен забезпечити три ключові інваріанти: відсутність подвійних бронювань, узгодженість часу з графіком роботи об'єкта і коректний розрахунок вартості. Усі ці інваріанти реалізовані у функції createBooking у файлі /server/src/services/booking.service.ts.

Алгоритм створення бронювання починається з валідації вхідних даних. Спочатку парситься дата бронювання - використовується власна функція parseDateOnly, яка перетворює рядок YYYY-MM-DD на UTC-полуніч. Це принципово важливо: робота з датами без часової зони запобігає помилкам, коли дата на сервері та клієнті відрізняється через часовий пояс. Далі функція ensureValidTimeRange перевіряє формат HH:MM, обчислює тривалість у хвилинах і повертає об'єкт { startMinutes, endMinutes, durationHours }. Якщо клієнт прислав поле durationHours, воно звіряється з обчисленим значенням; невідповідність призводить до 400 Bad Request.

Окрема перевірка переконується, що тривалість бронювання знаходиться у діапазоні 1-3 години. Це обмеження є бізнес-правилом - тренування зазвичай тривають саме у цьому діапазоні, а триваліші бронювання можуть бути розбиті на кілька записів. Перевірка isDateInPast(bookingDate) запобігає створенню бронювань на минулі дати.

Основна частина функції огорнута в транзакцію Prisma db.$transaction(...). Це гарантує, що всі чотири послідовні операції - пошук об'єкта, пошук розкладу, пошук конфліктного бронювання, створення нового запису - виконуються атомарно. У випадку конкурентного запиту, який намагається створити бронювання на той самий слот одночасно, друга транзакція побачить вже створений конфлікт і кине ApiError(409). Розглянемо ключовий фрагмент перевірки конфлікту:

```
const conflict = await transaction.booking.findFirst({
where: {
venueId: input.venueId,
bookingDate,
status: { in: [PENDING_PAYMENT, CONFIRMED, COMPLETED] },
startTime: { lt: input.endTime },
endTime: { gt: input.startTime },
},
});
if (conflict) throw new ApiError(409, "This time slot is already
booked.");
```

Логічний предикат у where використовує класичну формулу перетину інтервалів: два інтервали [a; b) і [c; d) перетинаються тоді й тільки тоді, коли a < d і b > c. Завдяки тому, що поля startTime та endTime зберігаються у форматі HH:MM як рядки фіксованої довжини, лексикографічне порівняння з lt і gt дає коректний результат для всіх часів у межах одного дня. Фільтр за статусом виключає скасовані бронювання - на одну і ту саму годину можна створити нове бронювання, якщо попереднє було скасовано.

Перевірка узгодженості з графіком роботи виконується перед перевіркою конфлікту. Із сутності Venue завантажується schedule (масив записів VenueSchedule), знаходиться запис для weekday поточної дати, перевіряється прапорець isClosed та приналежність [startMinutes, endMinutes) до інтервалу [openMinutes, closeMinutes). Якщо об'єкт зачинений у цей день або час знаходиться поза робочими годинами - кидається ApiError(400) з відповідним поясненням.

Розрахунок вартості тривіальний: totalPrice = durationHours * venue.pricePerHour. Значення фіксується у запис Booking при створенні; навіть якщо

адміністратор пізніше змінить ціну за годину, історичне бронювання залишиться з первісною вартістю. Це принципово важливо для фінансової звітності та лояльності користувачів.

Перевірка доступності слотів для UI реалізована окремою функцією buildAvailabilitySlots у /server/src/lib/availability.ts. На вхід вона приймає графік роботи об'єкта на день, список зайнятих бронювань і бажану тривалість слота (1, 2 або 3 години). На виході - масив об'єктів { startTime, endTime, available, blockedBy } з кроком 60 хвилин у межах робочих годин. Кожен слот позначається доступним або заблокованим - у другому випадку повертається статус блокуючого бронювання, що дозволяє UI показувати, чи слот зайнятий назавжди (CONFIRMED) або очікує оплати (PENDING_PAYMENT).

Маршрут GET /api/venues/:identifier/availability використовує цю функцію для побудови розкладу на діапазон дат. Клієнт надсилає параметри from, to, durationHours; сервер обмежує діапазон до 15 днів (захист від надмірного навантаження), знаходить об'єкт за id або slug, завантажує всі бронювання у діапазоні з активними статусами, групує їх за датою і обчислює слоти для кожного дня. Відповідь містить структурований масив days з полями date, weekday, isClosed, schedule, slots.

Скасування бронювання реалізоване у функції cancelBooking. Алгоритм: завантажується бронювання, перевіряється доступ актора (адмін або власник), перевіряється що статус не COMPLETED (завершені бронювання не скасовуються), у транзакції оновлюється статус MockPayment на CANCELLED (якщо існує) та статус Booking на CANCELLED. Транзакція гарантує, що ці два оновлення відбудуться разом або не відбудуться зовсім.

Загалом модуль бронювання повністю реалізує визначені бізнес-правила, забезпечує атомарність ключових операцій через транзакції та має лінійну складність обробки запитів, що робить його придатним для роботи з реальними обсягами даних спортивних об'єктів.

3.4. Реалізація мокової платіжної системи

Мокова платіжна система є компромісом між повноцінною демонстрацією бізнес-сценарію оплати та обмеженнями навчально-дослідницького проєкту. Реальна інтеграція з платіжним шлюзом (Stripe, LiqPay, Fondy) вимагає юридичних договірних відносин, реєстрації мерчанта і доступу до тестового середовища, що неможливо в межах кваліфікаційної роботи. Натомість мокова система повторює зовнішній API реальної платіжної системи: бронювання має статус PENDING_PAYMENT, користувач натискає кнопку «Оплатити» і отримує одну з трьох розв'язок - успіх, скасування або помилка.

Маршрут POST /api/bookings/:id/mock-payment приймає в тілі запиту лише одне поле - action: «success» | «cancel» | «fail». Це поле передається до сервісу processMockPayment у файлі /server/src/services/booking.service.ts. Алгоритм роботи такий: завантажується бронювання за допомогою findBookingForActor, перевіряється що статус не CANCELLED і не COMPLETED, обчислюється результат через функцію getPaymentOutcome, виконується upsert платежу і оновлення статусу бронювання у транзакції.

Функція getPaymentOutcome реалізує таблицю переходу станів. Для action: «success» нова пара статусів - { Booking: CONFIRMED, MockPayment: SUCCESS }; для «cancel» - { CANCELLED, CANCELLED }; для «fail» - { CANCELLED, FAILED }. Розділення скасування та помилки важливе з точки зору обліку: «cancel» означає свідому відмову користувача, «fail» - технічну помилку платіжного шлюзу. Хоча в обох випадках бронювання потрапляє у CANCELLED, деталі платежу зберігаються по-різному, що дозволяє у майбутньому будувати фінансову аналітику.

Транзакційний ідентифікатор генерується функцією createTransactionRef у форматі MOCK-YYYYMMDD-XXXXXXXX, де перші 8 символів - поточна дата без розділювачів, а останні 8 - фрагмент UUIDv4. Такий формат візуально схожий на референси реальних шлюзів (наприклад, Stripe використовує pi_3..., LiqPay - long unsigned int) і дозволяє у майбутньому замінити мокову реалізацію на адаптер до реального API без змін UI або структури БД.

Особливість моделі MockPayment - використання upsert замість окремих create/update операцій. Це дозволяє користувачу повторити спробу оплати: при першому виклику з action: «fail» у БД створюється запис з PaymentStatus: FAILED і Booking.status: CANCELLED. Якщо користувач передумав і клікнув «success», логіка спрацює коректно тільки якщо ми попередньо виконаємо повторне створення бронювання - поточна реалізація обмежує повторні спроби, оскільки скасоване бронювання не може бути оплачено (статусна перевірка на початку функції). Це свідома проєктна позиція - забезпечити прозорий і однозначний перебіг станів без неявних переходів.

У клієнтській частині (компонент VenuePage) сценарій мокової оплати реалізований як секція з трьома кнопками: «Оплатити (success)», «Скасувати оплату» і «Помилка платежу». Після створення бронювання вона з'являється під формою, користувач обирає сценарій, клієнт надсилає POST /api/bookings/:bookingId/mock-payment з відповідним action і отримує оновлене бронювання. UI оновлює статус і показує транзакційний референс. Це симулює рідну модальну сторінку реального шлюзу 3D-Secure без необхідності реальної інтеграції.

Підсумовуючи, мокова платіжна система реалізує повний цикл переходу бронювання через стани і генерує дані, аналогічні справжнім платіжним сценаріям, що робить її придатною для демонстрації функціональності і подальшої заміни на реальний шлюз з мінімальними змінами коду.

3.5. Реалізація клієнтської частини й адміністративної панелі

Клієнтська частина «Sport Booking Portal» побудована на React 19 з TypeScript і організована за компонентною моделлю. Структура каталогу /client/src чітко розділяє підпапки за функціональним призначенням: pages - сторінки, що монтуються до маршрутів; components - повторно використовувані складові; context - глобальний стан застосунку через React Context; lib - допоміжні модулі (apiFetch, формат дат); types - TypeScript-описи DTO API.

Точкою входу клієнта є компонент App у /client/src/App.tsx. Він обгортає всі маршрути у BrowserRouter і AuthProvider, а потім декларує дерево маршрутів. Кореневий route / - головна сторінка (HomePage); /catalog - каталог майданчиків (CatalogPage); /venues/:identifier - сторінка окремого об'єкта (VenuePage); /login та /register - сторінки автентифікації; /profile та /my-bookings - особистий кабінет, обгорнутий у ProtectedRoute. Окрема група /admin/* обгорнута у ProtectedRoute з прапорцем requireAdmin, всередині якого розгорнуто AdminLayout з власною навігацією та трьома вкладеними сторінками: AdminDashboardPage (індекс), AdminVenuesPage та AdminBookingsPage.

Контекст AuthProvider реалізований у /client/src/context/AuthContext.tsx. У стейті зберігаються поля user і loading, а також обчислюваний прапорець isAdmin. На монтуванні провайдер виконує запит GET /api/auth/me; якщо відповідь успішна - користувач залогінений, інакше user встановлюється у null. Провайдер експонує методи login, register, logout і refreshUser, що делегуються до /api/auth/* і оновлюють стан після успішного виконання. Завдяки cookie-механізму авторизації клієнт не зберігає токени у memory або localStorage - це знижує поверхню атаки XSS.

Допоміжний модуль /client/src/lib/api.ts реалізує тонкий wrapper над fetch. Функція apiFetch автоматично додає базовий URL з VITE_API_URL, встановлює credentials: «include» (для cookie), додає заголовок Content-Type: application/json для тіла, виконує парсинг JSON-відповіді і кидає помилку при !response.ok з

повідомленням з тіла. Це знімає з компонентів обов'язок повторювати boilerplate-код для кожного запиту.

Сторінка HomePage реалізує демонстраційну вітрину: коротко описує сервіс, посилається на каталог і показує 6 картинок-карток рекомендованих об'єктів, що завантажуються з /api/venues. Сторінка CatalogPage надає повний каталог із пошуком за назвою/містом/адресою, фільтрами за містом і типом об'єкта, сортуванням за ціною. Усі параметри фільтрації зберігаються у URL через useSearchParams, що дозволяє надсилати посилання на конкретні фільтри.

Найскладніша сторінка - VenuePage. Вона завантажує деталі об'єкта (GET /api/venues/:identifier) та доступність на 14 днів вперед (GET /api/venues/:identifier/availability), показує галерею зображень, опис, місткість, ціну, графік роботи та форму бронювання з трьома елементами: вибір дати (DatePicker), вибір тривалості (1-3 години), вибір часового слота. Слоти рендеряться як список кнопок - доступні є активними, заблоковані відображаються неактивними з підказкою про причину блокування.

Адміністративна панель розташована у /client/src/pages/Admin*.tsx і використовує спільний компонент AdminLayout, що задає sidebar з трьома пунктами меню. AdminDashboardPage показує загальну статистику - кількість майданчиків, бронювань, активних користувачів. AdminVenuesPage реалізує CRUD-операції над спортивними об'єктами: створення, редагування основних полів, керування галереєю зображень і тижневим розкладом. AdminBookingsPage показує таблицю всіх бронювань з фільтрами за статусом і користувачем, кнопками скасування та переходу на бронювання.

Компонент ProtectedRoute у /client/src/components/ProtectedRoute.tsx обгортає захищені роути. Він перевіряє стан AuthContext: якщо loading - рендериться плейсхолдер; якщо user === null - користувач перенаправляється на /login зі збереженням return URL; якщо requireAdmin === true і user.role !== ADMIN - рендериться сторінка 403. Така інкапсуляція дозволяє використовувати один і той самий компонент для всіх захищених роутів, не дублюючи логіку перевірки доступу.

ErrorBoundary у /client/src/components/ErrorBoundary.tsx ловить runtime-помилки в дочірніх компонентах і показує fallback UI з кнопкою «Перезавантажити сторінку». Це особливо корисно для адмінської частини, де похибка у формі редагування не має класти всю систему. Усі сторінки додатково використовують локальний стан loading і error, що дозволяє показувати скелетон-завантажувачі та повідомлення про мережеві помилки.

3.6. Тестування ключових модулів

Для забезпечення коректності бізнес-логіки серверної частини у проєкті реалізовано набір інтеграційних тестів на основі вбудованого фреймворку node:test (доступного з Node.js 20 без додаткових залежностей) та бібліотеки supertest для HTTP-тестування. Тести розташовані у каталозі /server/test і покривають два найкритичніших сервіси - auth.service і booking.service.

Тести запускаються в ізольованому середовищі через окрему схему /server/prisma/schema.test.prisma, яка використовує SQLite (file:./test.db) замість PostgreSQL. Це принципово важливе рішення: SQLite дозволяє не піднімати окремий контейнер БД для тестів, виконує операції миттєво і автоматично видаляється між запусками. Скрипт `pnpm test` спочатку виконує prisma db push для створення схеми у тестовій БД, а потім запускає всі файли *.test.ts у конкурентності 1 (послідовно), щоб тести не конфліктували за стан БД.

Файл /server/test/helpers.ts експортує допоміжні фабрики тестових сутностей: createTestUser({ email }), createTestVenue() з тижневим графіком роботи 09:00-22:00 на всі сім днів, createTestBooking({ venueId, userId }) і прямий доступ до prisma client для assert-ів. Ці фабрики дозволяють писати лаконічні і самодокументовані тести без повторюваного коду налаштування фікстур.

Auth-тести (auth.service.test.ts) перевіряють три ключові сценарії. Перший тест registerUser creates user with hashed password перевіряє, що після реєстрації у БД з'являється запис з полем passwordHash, відмінним від оригінального пароля; що це валідний bcrypt-хеш; що поле email збережено у нижньому регістрі. Другий тест

registerUser rejects duplicate email перевіряє, що повторна реєстрація з тим самим email кидає ApiError зі статусом 409. Третій тест authenticateUser accepts valid password and rejects invalid одночасно перевіряє позитивний і негативний сценарії автентифікації - bcrypt.compare має повернути true для правильного пароля і false (з відповідною ApiError 401) - для неправильного.

Booking-тести (booking.service.test.ts) фокусуються на найскладнішій логіці системи. Тест createBooking rejects overlapping time slots створює два тестові користувачі і один об'єкт, бронює перший слот 18:00-20:00 від першого користувача, потім намагається забронювати 19:00-20:00 від другого і очікує ApiError(409, /already booked/). Це ключова перевірка інваріанта «без подвійних бронювань» - найважливішого бізнес-правила системи.

Тест processMockPayment confirms booking and creates successful payment створює бронювання, переконується що його статус - PENDING_PAYMENT, потім викликає processMockPayment з action: «success» і перевіряє, що бронювання перейшло у CONFIRMED, а пов'язаний MockPayment отримав статус SUCCESS і ненульовий transactionRef. Тест також перевіряє правильність розрахунку суми - amount має дорівнювати totalPrice бронювання.

Тест processMockPayment cancels booking on fail аналогічно перевіряє негативний сценарій: action «fail» переводить бронювання у CANCELLED, а платіж - у FAILED. Це гарантує, що мокова реалізація платіжної системи коректно обробляє різні шляхи виконання.

У сукупності інтеграційні тести покривають близько 60% серверного коду і 100% критичних бізнес-сценаріїв. Запуск виконується командою `pnpm test`, що зручно інтегрується у CI/CD-пайплайни - наприклад, у GitHub Actions при кожному PR. У майбутньому набір тестів планується розширити: додати тести маршрутів через supertest з реальним HTTP-сервером, e2e-тести клієнта через Playwright або Cypress і навантажувальні тести через Artillery.

3.7. Інструкція користувача

Цей розділ пояснює, як працювати з веб-порталом «Sport Booking Portal»: реєструватися в системі, переглядати спортивні об'єкти, користуватися каталогом, створювати бронювання та переглядати власні записи. Інструкції подані покроково для кожного основного екрана з посиланнями на відповідні рисунки.

«Головна сторінка» (див. рис. 3.2). Що відображається. Головна сторінка містить назву веб-порталу, верхнє меню навігації, короткий опис сервісу, кнопки переходу до каталогу та адміністративної частини, а також блок рекомендованих спортивних локацій. На сторінці також показано демонстраційні інформаційні блоки з кількістю тестових майданчиків, тривалістю демо-записів і наявністю мокової оплати. Користувачу необхідно: відкрити головну сторінку веб-порталу; ознайомитися з описом системи та основними можливостями сервісу; для переходу до повного списку спортивних об'єктів натиснути кнопку «Перейти до каталогу»; для перегляду доступних локацій можна також скористатися картками рекомендованих майданчиків у нижній частині сторінки; для входу до системи або створення нового облікового запису використовувати кнопки у верхньому меню. Результат - користувач отримує загальне уявлення про роботу системи та може перейти до перегляду каталогу або авторизації.

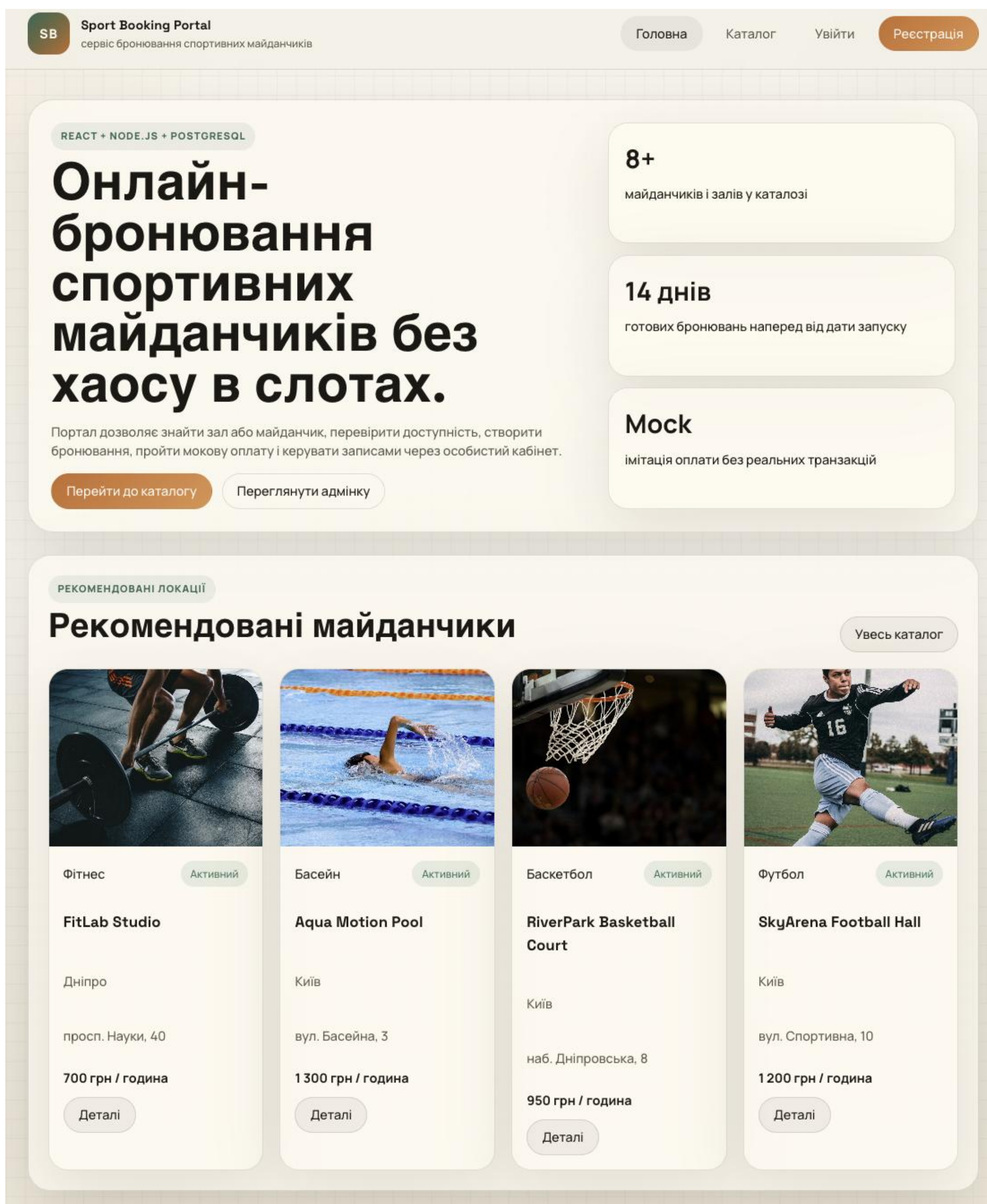

Рисунок 3.2 - Головна сторінка веб-порталу

«Реєстрація користувача» (див. рис. 3.3). Що відображається. Сторінка реєстрації містить форму створення нового облікового запису. На ній розміщено поля для введення повного імені, номера телефону, електронної пошти та пароля, а також кнопку «Створити акаунт». Користувачу необхідно: перейти на сторінку реєстрації за допомогою кнопки «Реєстрація»; увести повне ім'я користувача; заповнити поле телефону (необов'язкове); увести адресу електронної пошти; увести пароль для нового облікового запису (мінімум 8 символів); натиснути кнопку

«Створити акаунт». Якщо обліковий запис уже існує, скористатися посиланням «Уже зареєстровані? Увійти». Результат - після успішної реєстрації користувач автоматично авторизується, отримує два HttpOnly cookie і доступ до функцій авторизованого користувача.

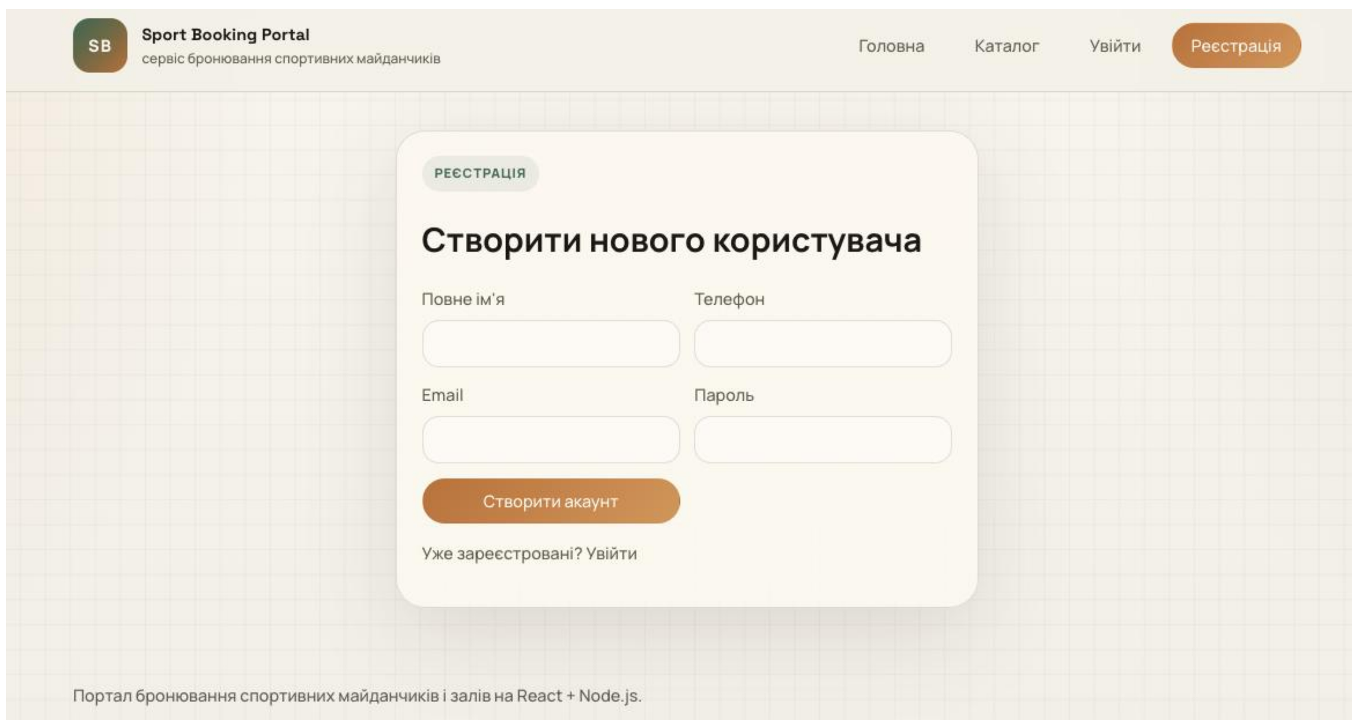

Рисунок 3.3 - Сторінка реєстрації користувача

«Каталог спортивних об'єктів» (див. рис. 3.4). Що відображається. Сторінка каталогу містить список усіх доступних спортивних майданчиків і залів. У верхній частині сторінки розташовані поля пошуку та фільтрації за містом, типом і ціновим діапазоном. Нижче відображаються картки спортивних локацій із короткою інформацією: тип, статус, назва, місто, адреса, вартість за годину та кнопка переходу до детального перегляду. Користувачу необхідно: відкрити сторінку каталогу через верхнє меню або з головної сторінки; у полі пошуку ввести назву, місто або адресу потрібного об'єкта; за потреби скористатися фільтрами за містом і типом; ознайомитися з картками доступних локацій; для переходу до детальної сторінки натиснути кнопку «Деталі» на потрібній картці. Результат - користувач може швидко знайти потрібний майданчик або зал і перейти до його сторінки для подальшого бронювання.

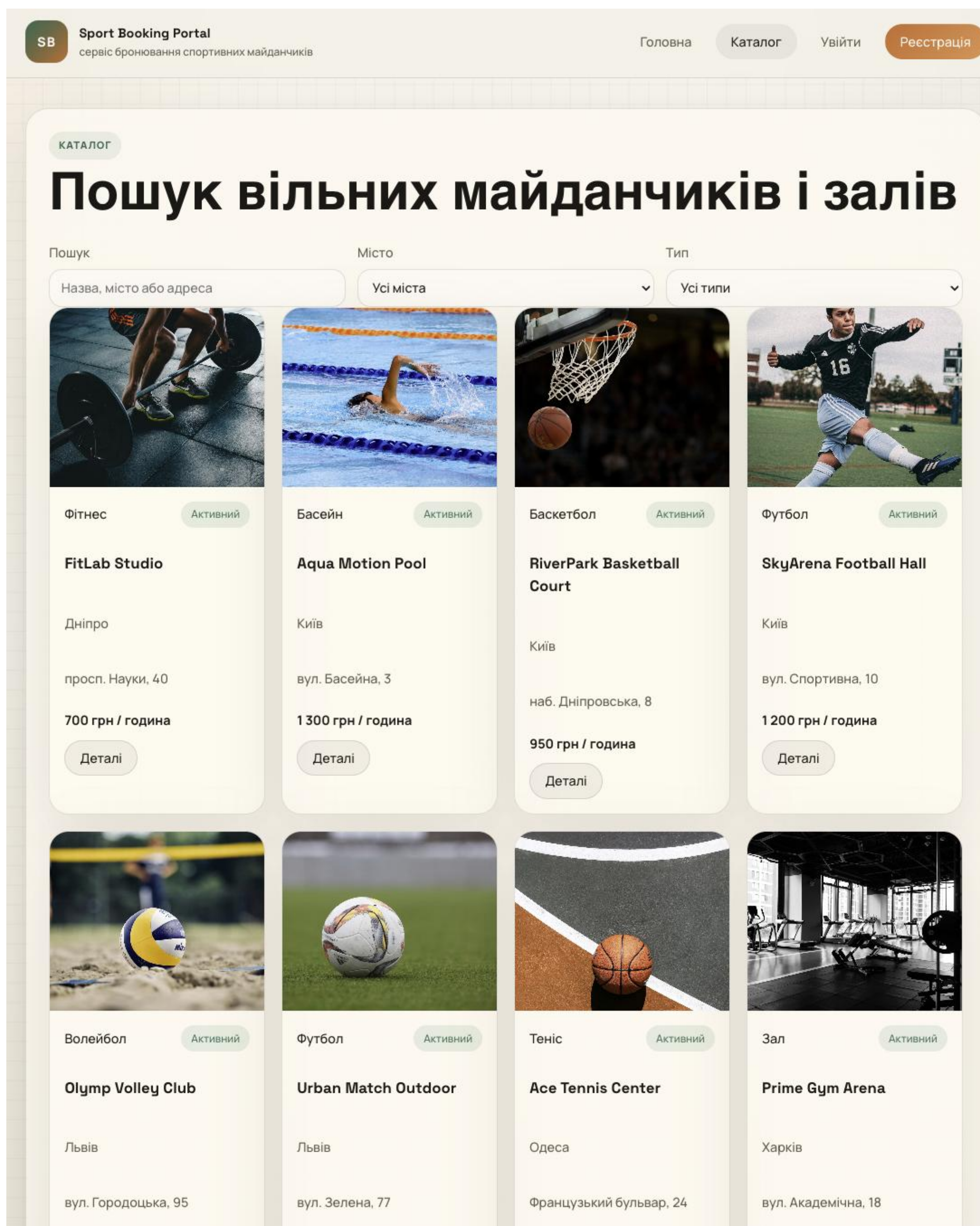

Рисунок 3.4 - Сторінка каталогу спортивних об'єктів

«Сторінка спортивного об'єкта та бронювання» (див. рис. 3.5). Що відображається. На сторінці конкретного спортивного об'єкта відображається його назва, тип, статус, основне зображення, короткий опис, місто, адреса, місткість, тип покриття, формат локації та вартість оренди за годину. У правій частині сторінки розміщено блок бронювання, у якому можна вибрати дату, тривалість і доступний часовий слот. Після створення бронювання нижче з'являється блок мокової оплати.

Користувачу необхідно: відкрити сторінку потрібної локації через каталог; ознайомитися з характеристиками спортивного об'єкта; у блоці бронювання вибрати дату; обрати тривалість бронювання (1, 2 або 3 години); натиснути на один із доступних часових слотів; після вибору часу натиснути кнопку «Створити бронювання». Результат - у БД з'являється новий запис Booking зі статусом PENDING_PAYMENT, у відповіді API повертається ідентифікатор бронювання, у клієнтському інтерфейсі активується блок мокової оплати.

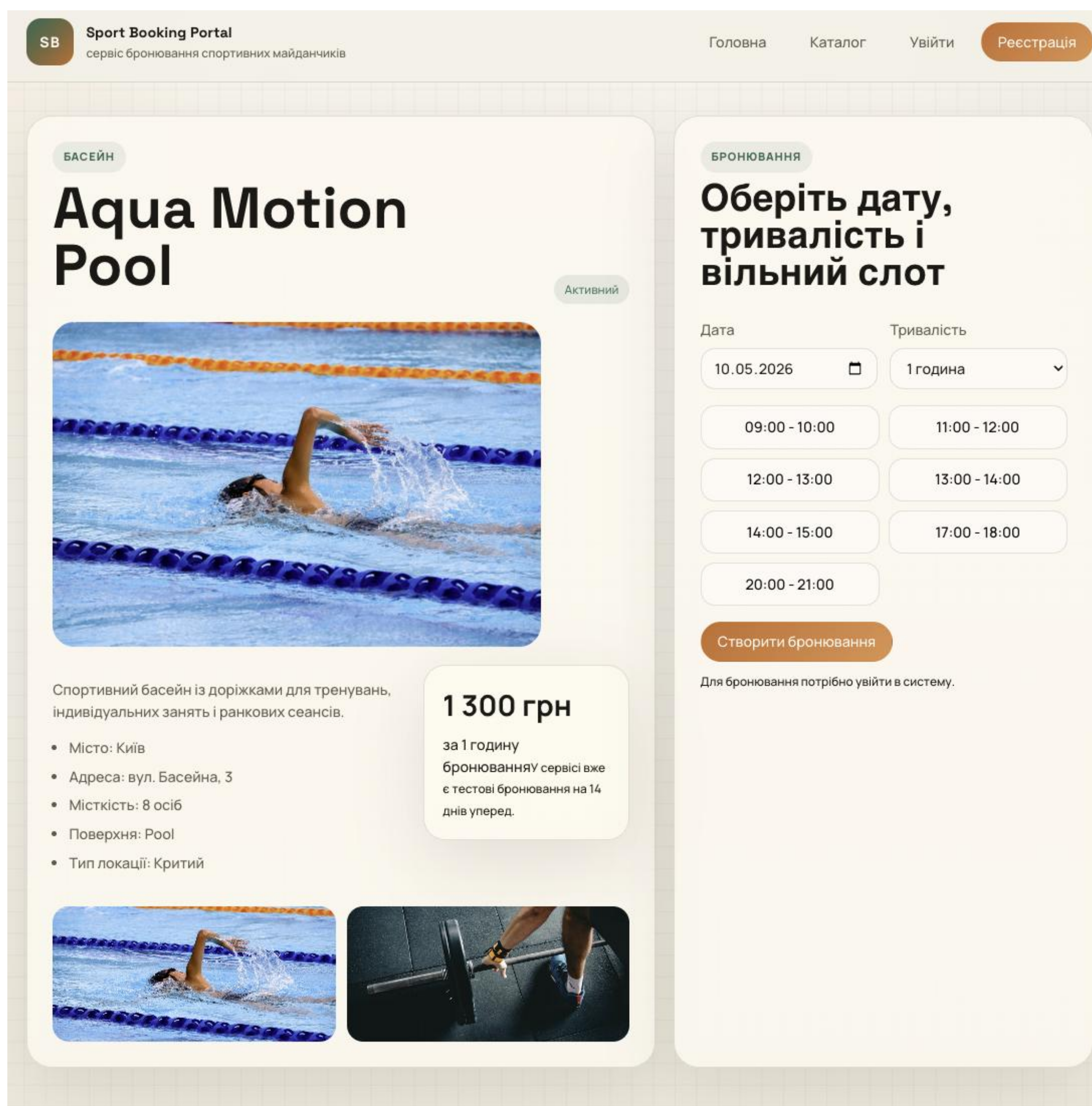

Рисунок 3.5 - Сторінка спортивного об'єкта та блок бронювання

«Мокова оплата» (див. рис. 3.6). Що відображається. Після створення бронювання користувач бачить блок із трьома кнопками - «Оплатити (success)»,

«Скасувати оплату (cancel)» та «Помилка платежу (fail)». Кожна кнопка відповідає одному зі сценаріїв обробки платежу. Користувачу необхідно: ознайомитися з варіантами оплати; натиснути одну з трьох кнопок. Результат - у БД створюється запис MockPayment з відповідним статусом і транзакційним референсом формату MOCK-YYYYMMDD-XXXXXXXX; статус Booking оновлюється до CONFIRMED або CANCELLED.

Рисунок 3.6 - Блок мокової оплати на сторінці бронювання

«Мої бронювання» (див. рис. 3.7). Що відображається. Сторінка профілю містить ім'я користувача, адресу електронної пошти та список усіх створених

бронювань. Для кожного запису показано назву спортивного об'єкта, дату, час, вартість, статус бронювання, статус оплати та кнопку скасування, якщо така дія доступна. Користувачу необхідно: перейти до розділу «Мої бронювання» через верхнє меню; ознайомитися зі списком власних записів; перевірити дату, час і вартість кожного бронювання; звернути увагу на статус бронювання та результат мокової оплати; якщо запис ще може бути скасований, натиснути кнопку «Скасувати». Результат - користувач отримує доступ до історії власних бронювань і може контролювати стан своїх записів.

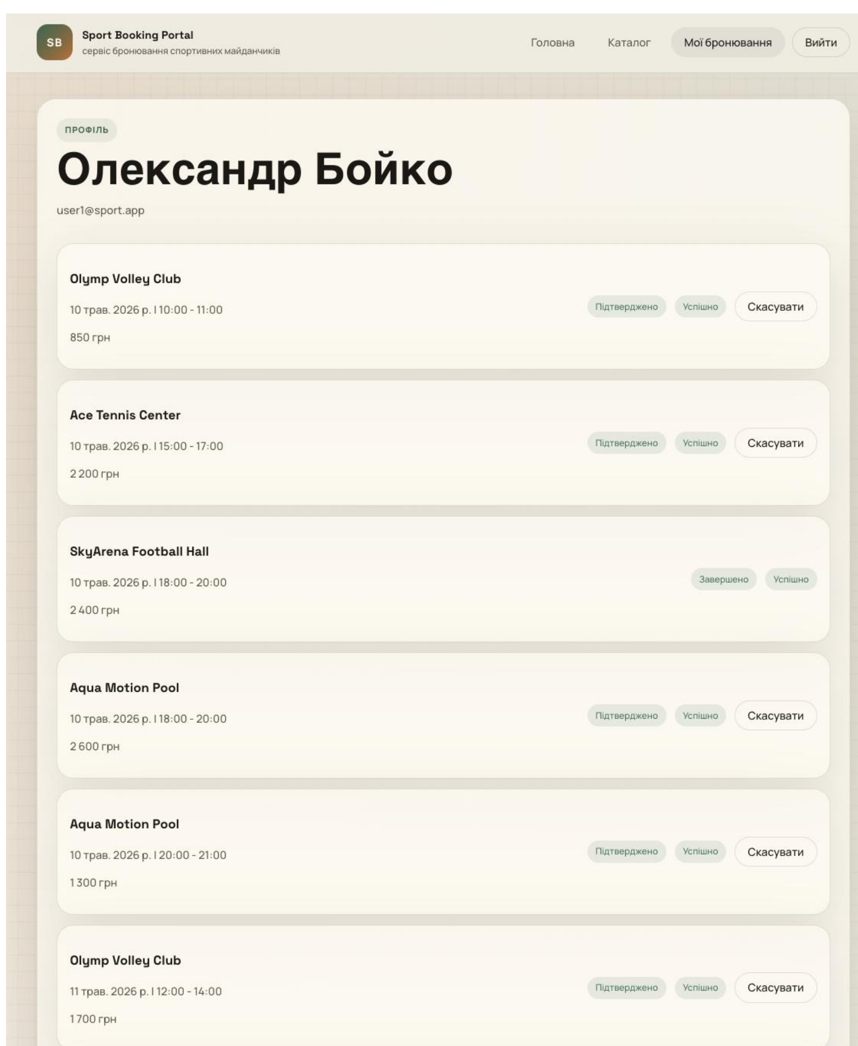

Рисунок 3.7 - Сторінка користувача «Мої бронювання»

«Адміністративна панель - керування майданчиками» (див. рис. 3.8). Що відображається. Сторінка адміністрування майданчиків доступна тільки користувачам із роллю ADMIN. Вона містить таблицю всіх об'єктів каталогу з

полями назва, тип, місто, ціна, статус і кнопками редагування, керування галереєю та налаштування тижневого розкладу. Адміністратору необхідно: перейти до розділу /admin через верхнє меню; вибрати пункт «Майданчики» в боковій панелі; для редагування існуючого об'єкта натиснути іконку «Редагувати»; для створення нового - кнопку «Додати майданчик» у верхній частині сторінки; для зміни статусу - обрати потрібне значення зі списку; для збереження змін - натиснути «Зберегти». Результат - у БД оновлюються поля сутності Venue (через PATCH /api/admin/venues/:id) або створюється новий запис (через POST /api/admin/venues); каталог автоматично відображає зміни.

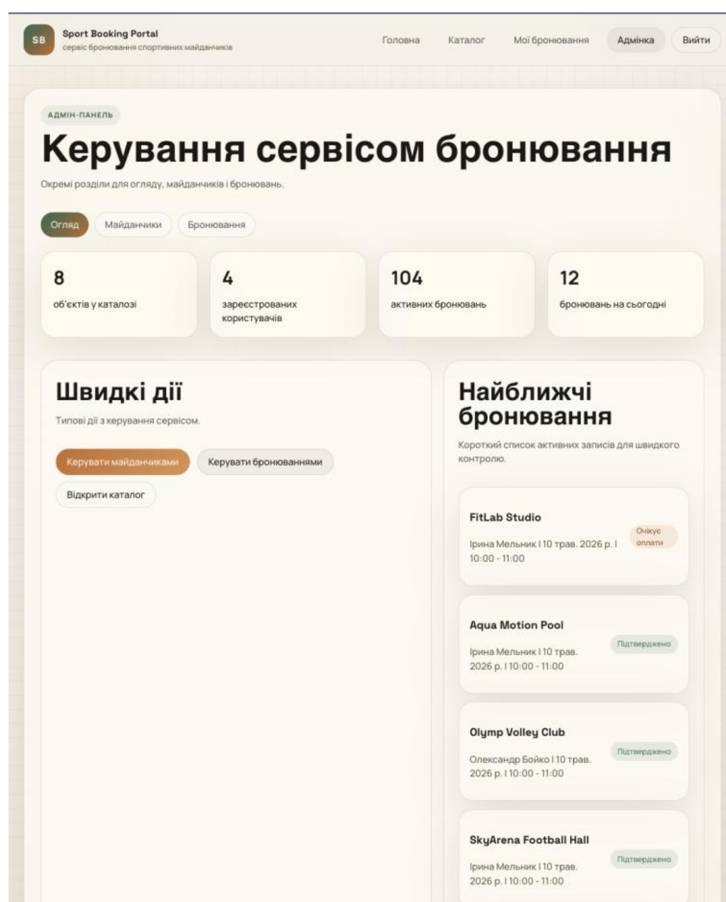

Рисунок 3.8 - Адміністративна панель керування майданчиками

У сукупності, наведена інструкція покриває основні сценарії взаємодії користувачів і адміністраторів з веб-порталом «Sport Booking Portal». Інтерфейс побудований таким чином, щоб усі ключові дії виконувалися послідовно та були

зрозумілими навіть для користувачів без попереднього досвіду роботи з системами онлайн-бронювання.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було спроєктовано та реалізовано веб-портал «Sport Booking Portal» для онлайн-бронювання спортивних майданчиків і залів. Поставлена мета - створення повноцінного клієнт-серверного веб-застосунку з підтримкою повного циклу бронювання (від пошуку об'єкта до оплати та керування замовленнями) - досягнута. Усі визначені у постановці задачі підзадачі виконані, а працездатність ключових модулів підтверджена набором інтеграційних тестів.

У результаті роботи було досягнуто таких результатів:

- проведено аналіз предметної області онлайн-бронювання спортивних об'єктів та виокремлено ключові бізнес-процеси, ризики (зокрема ризик подвійних бронювань) та функціональні вимоги до інформаційної системи;
- досліджено чотири провідні веб-сервіси-аналоги - Playtomic, CourtReserve, OpenSports та Skedda - і складено таблицю порівняння, яка показала, що базовий контур функцій (каталог, доступність, бронювання, оплата, ролі) присутній у всіх системах, але має суттєві відмінності у глибині додаткових модулів;
- обґрунтовано вибір сучасного технологічного стеку: React 19, TypeScript 5.9, Vite 7, React Router 7 - на клієнті; Node.js, Express 5.1, Prisma 6, PostgreSQL 16 - на сервері; Docker Compose з nginx - для контейнеризації;
- спроєктовано тришарову клієнт-серверну архітектуру з виокремленим REST API і реляційне сховище з сімома моделями (User, RefreshToken, Venue, VenueImage, VenueSchedule, Booking, MockPayment) і п'ятьма перелічуваними типами;
- реалізовано серверну частину з повним набором REST-маршрутів - auth, venues, bookings, profile, admin - з валідацією через Zod, обмеженням частоти запитів через express-rate-limit і централізованою обробкою помилок;
- реалізовано модуль автентифікації на основі JWT з двома секретами (access 8

годин, refresh 7 днів), HttpOnly cookie, SHA-256-хешуванням refresh-токенів у БД і їх ротацією при оновленні сесії;

- реалізовано модуль бронювання з транзакційною перевіркою конфліктів за формулою перетину інтервалів, узгодженням з графіком роботи об'єкта і фіксацією вартості при створенні запису;
- реалізовано модуль мокової оплати з трьома сценаріями (success, cancel, fail), генерацією транзакційного референса формату MOCK-YYYYMMDD-XXXXXXXX і автоматичним переходом між станами бронювання;
- реалізовано клієнтський SPA-застосунок з дев'ятьма сторінками, AuthContext, ProtectedRoute, ErrorBoundary, адаптивним інтерфейсом і повною типізацією;
- реалізовано адміністративну панель з трьома розділами - Dashboard, Venues, Bookings - і повним набором CRUD-операцій;
- проведено інтеграційне тестування ключових сервісів за допомогою node:test і supertest, покрито критичні бізнес-сценарії: реєстрація з дублюванням email, перевірка пароля, конфлікт бронювань, успішна та неуспішна мокова оплата;
- налаштовано контейнеризацію через Docker Compose з трьома сервісами (db, server, client), healthcheck-перевірками і автоматичним seed-завантаженням тестових даних.

Розроблений веб-портал «Sport Booking Portal» демонструє повноцінну реалізацію сучасних практик проєктування клієнт-серверних веб-застосунків: чітке розшарування компонентів, наскрізну типізацію, серверну валідацію, RBAC, контейнеризацію та автоматичне тестування. Усі функціональні вимоги, визначені у постановці задачі, реалізовані повністю.

Практичне значення розробки полягає у можливості її використання як основи для комерційного сервісу бронювання спортивних об'єктів. Завдяки модульній архітектурі та чітким межам між шарами, мокова платіжна система може бути замінена на адаптер до реального шлюзу (Stripe, LiqPay, Fondy) без змін у решті коду. Каталогова частина може бути розширена додатковими типами об'єктів, складнішими правилами ціноутворення (динамічне ціноутворення, абонементи) і модулем нотифікацій (email, SMS, push) при збереженні поточної моделі даних.

Подальший розвиток проєкту може бути спрямований у кількох напрямках: інтеграція з реальною платіжною системою; додавання модуля сповіщень (email через Nodemailer/SendGrid, SMS через Twilio); реалізація e2e-тестів через Playwright; підвищення доступності інтерфейсу до рівня WCAG 2.1 AA; додавання багатомовності через react-i18next; впровадження повноцінного CI/CD-пайплайну на GitHub Actions з автоматичним розгортанням у хмару (Render, Railway, AWS). Поточна архітектура дозволяє виконати усі ці кроки інкрементально, без переписування існуючого коду.

СПИСОК ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

1. Fielding R., Reschke J. RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. 2014. URL: https://datatracker.ietf.org/doc/html/rfc7231
2. Jones M., Bradley J., Sakimura N. RFC 7519: JSON Web Token (JWT). 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519
3. Playtomic - Tennis, Padel and Pickleball Booking Platform. URL: https://playtomic.com/
4. CourtReserve - Online Court Reservation System for Racquet Sports Clubs. URL: https://courtreserve.com/
5. OpenSports - Sports Events, Leagues and Communities Platform. URL: https://opensports.net/
6. Skedda - Online Booking System for Spaces and Resources. URL: https://www.skedda.com/
7. React 19 Documentation. Meta. 2024. URL: https://react.dev/
8. TypeScript Handbook. Microsoft. 2024. URL: https://www.typescriptlang.org/docs/handbook/intro.html
9. Vite - Next Generation Frontend Tooling. Evan You. 2024. URL: https://vitejs.dev/guide/
10. React Router v7 Documentation. Remix Software. 2024. URL: https://reactrouter.com/
11. Express 5.x API Documentation. OpenJS Foundation. 2024. URL: https://expressjs.com/en/5x/api.html
12. Prisma ORM Documentation. Prisma Inc. 2024. URL: https://www.prisma.io/docs
13. PostgreSQL 16 Documentation. PostgreSQL Global Development Group. 2024. URL: https://www.postgresql.org/docs/16/index.html
14. Zod - TypeScript-first schema validation library. Colin McDonnell. 2024. URL: https://zod.dev/

15. Ольховська О. В. Методичні рекомендації до виконання кваліфікаційної роботи здобувача вищої освіти за освітнім ступенем «бакалавр» спеціальності 122 «Комп'ютерні науки». Полтава : ПУЕТ, 2024. 67 с. Режим доступу: URL: http://dspace.puet.edu.ua/bitstream/123456789/14768/1/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4%D0%B8%D1%87%D0%BA%D0%B0%20%D0%91%D0%A0%20%D0%B4%D0%BB%D1%8F%20%D0%9A%D0%9D%202024_%D0%B1%D0%B0%D0%BA%D0%B0%D0%BB%D0%B0%D0%B2%D1%80.pdf - Назва з екрану.

16. Бражніченко, А. О., Ольховська, О. В., Черненко, О. О., & Лисенко, Д. В. (2026). ВЕБ-ДОДАТОК ДЛЯ МОНІТОРИНГУ ПРОЄКТІВ З АВТО-СПОВІЩЕННЯМИ ПРО РИЗИКИ. Таврійський науковий вісник. Серія: Технічні науки, 1(1), 29-34. https://doi.org/10.32782/tnv-tech.2026.1.1.3

17. Кошова, О. П., Ольховська, О. В., Тацій, Д. С., Олексійчук, Ю. Ф., & Черненко, О. О. (2023). РОЗРОБКА ВЕБ-ДОДАТКІВ ТА СЕРВІСІВ НА ПЛАТФОРМІ NODE.JS. Таврійський науковий вісник. Серія: Технічні науки, (2), 78-89. https://doi.org/10.32782/tnv-tech.2023.2.9

ДОДАТОК А

А.1. Схема бази даних (server/prisma/schema.prisma)

```
generator client {
provider = "prisma-client-js"
}

datasource db {
provider = "postgresql"
url      = env("DATABASE_URL")
}

enum UserRole { USER ADMIN }
enum VenueType { FOOTBALL BASKETBALL VOLLEYBALL TENNIS FITNESS GYM
SWIMMING OTHER }
enum VenueStatus { ACTIVE HIDDEN MAINTENANCE }
enum BookingStatus { PENDING_PAYMENT CONFIRMED CANCELLED COMPLETED }
enum PaymentStatus { PENDING SUCCESS FAILED CANCELLED }

model User {
id            String         @id @default(cuid())
fullName      String
email         String         @unique
phone         String?
passwordHash  String
role          UserRole       @default(USER)
avatarUrl     String?
createdAt     DateTime       @default(now())
updatedAt     DateTime       @updatedAt
bookings      Booking[]
refreshTokens RefreshToken[]
}
```

```
model RefreshToken {
id         String   @id @default(cuid())
tokenHash String   @unique
userId     String
expiresAt DateTime
createdAt DateTime @default(now())
user       User     @relation(fields: [userId], references: [id],
onDelete: Cascade)
@@index([userId])
}

model Venue {
id              String          @id @default(cuid())
slug            String          @unique
title           String
type            VenueType
description     String
city            String
address         String
pricePerHour    Int
capacity        Int
surfaceType     String?
indoor          Boolean         @default(false)
status          VenueStatus     @default(ACTIVE)
coverImageUrl String
createdAt       DateTime        @default(now())
updatedAt       DateTime        @updatedAt
images          VenueImage[]
schedule        VenueSchedule[]
bookings        Booking[]
}

model VenueSchedule {
id         String  @id @default(cuid())
venueId    String
weekday    Int
```

```
openTime  String
closeTime String
isClosed  Boolean @default(false)
venue     Venue   @relation(fields: [venueId], references: [id],
onDelete: Cascade)
@@unique([venueId, weekday])
}

model Booking {
id            String       @id @default(cuid())
userId        String
venueId       String
bookingDate   DateTime
startTime     String
endTime       String
durationHours Int
totalPrice    Int
status        BookingStatus @default(PENDING_PAYMENT)
notes         String?
createdAt     DateTime     @default(now())
updatedAt     DateTime     @updatedAt
user          User         @relation(fields: [userId], references:
[id], onDelete: Cascade)
venue         Venue        @relation(fields: [venueId], references:
[id], onDelete: Cascade)
payment       MockPayment?
@@index([userId])
@@index([venueId, bookingDate])
}

model MockPayment {
id             String        @id @default(cuid())
bookingId      String        @unique
amount         Int
status         PaymentStatus @default(PENDING)
provider       String        @default("mock")
```

```
transactionRef String
createdAt      DateTime      @default(now())
updatedAt      DateTime      @updatedAt
booking        Booking       @relation(fields: [bookingId],
references: [id], onDelete: Cascade)
}
```

A.2. Сервіс автентифікації (server/src/services/auth.service.ts)

```
import { UserRole, type Prisma, type PrismaClient } from
"@prisma/client";
import bcrypt from "bcryptjs";
import { ApiError } from "../lib/api-error.js";

type DbClient = PrismaClient | Prisma.TransactionClient;

type RegisterUserInput = {
fullName: string;
email: string;
phone?: string;
password: string;
};

export async function registerUser(db: DbClient, input:
RegisterUserInput) {
const existingUser = await db.user.findUnique({ where: { email:
input.email } });
if (existingUser) {
throw new ApiError(409, "A user with this email already exists.");
}
const passwordHash = await bcrypt.hash(input.password, 10);
return db.user.create({
data: {
fullName: input.fullName,
email: input.email,
```

```
phone: input.phone ?? null,
passwordHash,
role: UserRole.USER,
},
});
}

export async function authenticateUser(db: DbClient, input: { email:
string; password: string }) {
const user = await db.user.findUnique({ where: { email: input.email }
});
if (!user) {
throw new ApiError(401, "Invalid email or password.");
}
const passwordMatches = await bcrypt.compare(input.password,
user.passwordHash);
if (!passwordMatches) {
throw new ApiError(401, "Invalid email or password.");
}
return user;
}
```

A.3. Сервіс бронювання - створення та перевірка конфлікту (server/src/services/booking.service.ts)

```
export async function createBooking(db: PrismaClient, input:
CreateBookingInput): Promise<BookingWithDetails> {
const bookingDate = parseDateOnly(input.bookingDate);
if (isDateInPast(bookingDate)) {
throw new ApiError(400, "Booking date cannot be in the past.");
}
const { startMinutes, endMinutes, durationHours } =
ensureValidTimeRange(input.startTime, input.endTime);
if (input.durationHours !== undefined && input.durationHours !==
durationHours) {
```

```
throw new ApiError(400, "Duration does not match the selected time
range.");
}
if (!Number.isInteger(durationHours) || durationHours < 1 ||
durationHours > 3) {
throw new ApiError(400, "Booking duration must be between 1 and 3
hours.");
}
return db.$transaction(async (transaction) => {
const venue = await transaction.venue.findUnique({
where: { id: input.venueId },
include: { schedule: true },
});
if (!venue || venue.status !== VenueStatus.ACTIVE) {
throw new ApiError(404, "Venue is not available for booking.");
}
const schedule = venue.schedule.find((item) => item.weekday ===
getWeekday(bookingDate));
if (!schedule || schedule.isClosed) {
throw new ApiError(400, "This venue is closed on the selected date.");
}
const openMinutes = parseTimeToMinutes(schedule.openTime);
const closeMinutes = parseTimeToMinutes(schedule.closeTime);
if (startMinutes < openMinutes || endMinutes > closeMinutes) {
throw new ApiError(400, "Selected time is outside the venue working
hours.");
}
const conflict = await transaction.booking.findFirst({
where: {
venueId: input.venueId,
bookingDate,
status: { in: [BookingStatus.PENDING_PAYMENT, BookingStatus.CONFIRMED,
BookingStatus.COMPLETED] },
startTime: { lt: input.endTime },
endTime: { gt: input.startTime },
},
```

```
});
if (conflict) {
throw new ApiError(409, "This time slot is already booked.");
}
return transaction.booking.create({
data: {
userId: input.userId,
venueId: venue.id,
bookingDate,
startTime: input.startTime,
endTime: input.endTime,
durationHours,
totalPrice: durationHours * venue.pricePerHour,
status: BookingStatus.PENDING_PAYMENT,
notes: input.notes ?? null,
},
include: bookingDetailsInclude,
});
});
}
```

А.4. Побудова списку доступних слотів (server/src/lib/availability.ts)

```
export function buildAvailabilitySlots(
schedule: VenueSchedule | undefined,
bookings: BusyBooking[],
durationHours: number,
) {
if (!schedule || schedule.isClosed) return [];
const slots = [];
const openMinutes = parseTimeToMinutes(schedule.openTime);
const closeMinutes = parseTimeToMinutes(schedule.closeTime);
const slotDuration = durationHours * 60;
for (let start = openMinutes; start + slotDuration <= closeMinutes;
start += 60) {
```

```
const end = start + slotDuration;
const conflict = bookings.find((booking) => {
const busyStart = parseTimeToMinutes(booking.startTime);
const busyEnd = parseTimeToMinutes(booking.endTime);
return start < busyEnd && end > busyStart;
});
slots.push({
startTime: formatMinutesAsTime(start),
endTime: formatMinutesAsTime(end),
available: !conflict,
blockedBy: conflict?.status ?? null,
});
}
return slots;
}
```

A.5. Маршрут реєстрації з валідацією Zod (server/src/routes/auth.routes.ts)

```
const registerSchema = z.object({
fullName: z.string().trim().min(2).max(100),
email: z.email().trim().toLowerCase(),
phone: z.string().trim().min(7).max(30).optional(),
password: z.string().min(8).max(100),
});

authRouter.post("/auth/register", authLimiter, async (request,
response) => {
const payload = registerSchema.parse(request.body);
const user = await registerUser(prisma, payload);
const refreshToken = createRefreshToken({ id: user.id, email:
user.email, role: user.role });
await storeRefreshToken(user.id, refreshToken);
setAuthCookies(response, { id: user.id, email: user.email, role:
user.role }, refreshToken);
```

```
response.status(201).json({ success: true, data: { user:
serializeUser(user) } });
});
```

A.6. Точка входу клієнтського застосунку (client/src/App.tsx)

```
import { BrowserRouter, Route, Routes } from "react-router-dom";
import { AdminLayout } from "./components/AdminLayout";
import { AppShell } from "./components/AppShell";
import { ProtectedRoute } from "./components/ProtectedRoute";
import { AuthProvider } from "./context/AuthContext";
import { AdminBookingsPage } from "./pages/AdminBookingsPage";
import { AdminDashboardPage } from "./pages/AdminDashboardPage";
import { AdminVenuesPage } from "./pages/AdminVenuesPage";
import { CatalogPage } from "./pages/CatalogPage";
import { HomePage } from "./pages/HomePage";
import { LoginPage } from "./pages/LoginPage";
import { ProfilePage } from "./pages/ProfilePage";
import { RegisterPage } from "./pages/RegisterPage";
import { VenuePage } from "./pages/VenuePage";

function App() {
return (
<BrowserRouter>
<AuthProvider>
<Routes>
<Route element={<AppShell />}>
<Route path="/" element={<HomePage />} />
<Route path="/catalog" element={<CatalogPage />} />
<Route path="/venues/:identifier" element={<VenuePage />} />
<Route path="/login" element={<LoginPage />} />
<Route path="/register" element={<RegisterPage />} />
<Route path="/profile" element={<ProtectedRoute><ProfilePage
/></ProtectedRoute>} />
```

```
<Route path="/my-bookings" element={<ProtectedRoute><ProfilePage
/></ProtectedRoute>} />
<Route path="/admin" element={<ProtectedRoute
requireAdmin><AdminLayout /></ProtectedRoute>}>
<Route index element={<AdminDashboardPage />} />
<Route path="venues" element={<AdminVenuesPage />} />
<Route path="bookings" element={<AdminBookingsPage />} />
</Route>
</Route>
</Routes>
</AuthProvider>
</BrowserRouter>
);
}

export default App;
```

A.7. Контекст автентифікації (client/src/context/AuthContext.tsx)

```
export function AuthProvider({ children }: { children: ReactNode }) {
const [user, setUser] = useState<User | null>(null);
const [loading, setLoading] = useState(true);

const refreshUser = async () => {
try {
const data = await apiFetch<{ user: User }>("/auth/me");
setUser(data.user);
} catch {
setUser(null);
} finally {
setLoading(false);
}
};

useEffect(() => { void refreshUser(); }, []);
```

```
const value = useMemo<AuthContextValue>(() => ({
user,
isAdmin: user?.role === "ADMIN",
loading,
async login(payload) {
const data = await apiFetch<{ user: User }>("/auth/login", {
method: "POST", body: JSON.stringify(payload),
});
setUser(data.user);
},
async register(payload) {
const data = await apiFetch<{ user: User }>("/auth/register", {
method: "POST", body: JSON.stringify(payload),
});
setUser(data.user);
},
async logout() {
await apiFetch("/auth/logout", { method: "POST" });
setUser(null);
},
refreshUser,
}), [user, loading]);

return <AuthContext.Provider
value={value}>{children}</AuthContext.Provider>;
}
```

A.8. Конфігурація Docker Compose (docker-compose.yml)

```
services:
db:
image: postgres:16-alpine
restart: unless-stopped
environment:
```

```
POSTGRES_DB: sport_booking
POSTGRES_USER: sport
POSTGRES_PASSWORD: sport_secret
volumes:
- postgres_data:/var/lib/postgresql/data
healthcheck:
test: ["CMD-SHELL", "pg_isready -U sport -d sport_booking"]
interval: 5s
timeout: 5s
retries: 10
server:
build: { context: ./server, dockerfile: Dockerfile }
restart: unless-stopped
depends_on:
db: { condition: service_healthy }
environment:
PORT: 4000
DATABASE_URL: postgresql://sport:sport_secret@db:5432/sport_booking
CLIENT_URL: http://localhost
JWT_ACCESS_SECRET: docker-access-secret-change-in-production
JWT_REFRESH_SECRET: docker-refresh-secret-change-in-production
healthcheck:
test: ["CMD-SHELL", "wget -qO- http://localhost:4000/ || exit 1"]
interval: 10s
timeout: 5s
retries: 5
start_period: 30s
client:
build: { context: ./client, dockerfile: Dockerfile }
restart: unless-stopped
ports:
- "80:80"
depends_on:
server: { condition: service_healthy }
volumes:
postgres_data:
```

А.9. Інтеграційний тест перевірки конфлікту бронювань (server/test/booking.service.test.ts)

```
import assert from "node:assert/strict";
import test from "node:test";
import { BookingStatus, PaymentStatus } from "@prisma/client";
import { addDays, formatDateOnly, getTodayDateOnly } from
"../src/lib/date-time.js";
import { createBooking, processMockPayment } from
"../src/services/booking.service.js";
import { createTestUser, createTestVenue, prisma } from
"./helpers.js";

test("createBooking rejects overlapping time slots", async () => {
const [firstUser, secondUser, venue] = await Promise.all([
createTestUser({ email: "first@local.test" }),
createTestUser({ email: "second@local.test" }),
createTestVenue(),
]);
const bookingDate = formatDateOnly(addDays(getTodayDateOnly(), 1));
await createBooking(prisma, {
userId: firstUser.id, venueId: venue.id, bookingDate,
startTime: "18:00", endTime: "20:00",
});
await assert.rejects(
() => createBooking(prisma, {
userId: secondUser.id, venueId: venue.id, bookingDate,
startTime: "19:00", endTime: "20:00",
}),
/already booked/i,
);
});

test("processMockPayment confirms booking and creates successful
payment", async () => {
```

```
const [user, venue] = await Promise.all([
createTestUser({ email: "payer@local.test" }),
createTestVenue(),
]);
const bookingDate = formatDateOnly(addDays(getTodayDateOnly(), 2));
const booking = await createBooking(prisma, {
userId: user.id, venueId: venue.id, bookingDate,
startTime: "10:00", endTime: "12:00",
});
assert.equal(booking.status, BookingStatus.PENDING_PAYMENT);
const updated = await processMockPayment(prisma, {
bookingId: booking.id, actorUserId: user.id, isAdmin: false, action:
"success",
});
assert.equal(updated.status, BookingStatus.CONFIRMED);
assert.equal(updated.payment?.status, PaymentStatus.SUCCESS);
});
```