БАКАЛАВРСЬКА РОБОТА

РБ. ІП - 19.00.00.000 ПЗ

Група ІП-22-1

Микитій Юрій

2026

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Микитій Юрій Михайлович

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Розробка адаптивної веб-платформи бронювання спортивних майданчиків з інтегрованою системою підтримки рішень

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121– Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня _____Микитій Ю.М._____
(підпис, ініціали та прізвище здобувача)

Науковий керівник _____Мельник Віталій Дмитрович, к.т.н., доцент_____
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

_доц._____Бандура В.В.__
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доцент. В.В. Бандура
“ ____ ” _____2026 р.

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Микитій Юрій Михайлович
(прізвище, ім’я, по-батькові)

1. Тема проекту (роботи) " Розробка адаптивної веб-платформи бронювання спортивних майданчиків з інтегрованою системою підтримки рішень "
керівник проекту (роботи) Мельник Віталій Дмитрович, к.т.н., доцент
затверджені наказом вищого навчального закладу від **“** 21 ” квітня 2026 р. № 179/7

2. Строк подання студентом проекту (роботи) 10 червня 2026р.

3. Вихідні дані до проекту (роботи) Результати і матеріали отримані під час проходження переддипломної практики

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

1 Вступ до проблеми

2 Огляд існуючих концепцій , рішень та сервісів в даній області

3 Побудова моделі або алгоритму власного рішення

4 Документування реалізації власного оригінального рішення вибраними засобами

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

1 Sequence-діаграма сценарію гостьового бронювання (рис. 2.1, ст. 26)

2 Схема робочого сценарію менеджера у CRM (рис. 2.3, ст. 28)

3 Sequence-діаграма «Створення бронювання» (рис. 3.4, ст. 41)

4 ER-діаграма бази даних (основні сутності) (рис. 3.7, ст. 48)

5 Послідовність server-side обробки бронювання (рис. 4.1, ст. 55)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

2. Дата видачі завдання **2026 р.**

Керівник __Мельник В.Д._ _____
(підпис)

Завдання прийняв до виконання__Микитій Ю.М.__ _____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту	Примітка
1	Визначення та обгрунтування теми роботи	15.01.2026	виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	29.01.2026	виконано
3	Проєктування архітектури програмної системи та структури бази даних	24.02.2026	виконано
4	Розробка серверної частини системи на ASP.NET Core	15.03.2026	виконано
5	Розробка клієнтського SPA-застосунку на React	05.04.2026	виконано
6	Реалізація CRM-модуля для менеджерів та власників майданчиків	29.04.2026	виконано
7	Оформлення пояснювальної записки бакалаврської роботи	09.06.2026	виконано

Студент _____ Микитій Ю.М.
(підпис)

Керівник роботи _____ Мельник В.Д.
(підпис)

АНОТАЦІЯ

Бакалаврська робота містить 80 сторінок, 25 рисунків, 16 таблиць, список використаних джерел із 40 найменувань, 3 додатки.

Метою роботи є розробка адаптивної веб-платформи бронювання спортивних майданчиків з інтегрованою системою підтримки рішень.

Об'єкт дослідження: процеси бронювання та оперативного управління спортивними майданчиками в цифровому середовищі.

Предмет дослідження: методи, моделі та програмні засоби проєктування і реалізації веб-системи бронювання спортивних майданчиків.

У першому розділі проведено аналіз проблематики бронювання спортивних об'єктів в Україні, розглянуто основні поняття предметної області та порівняно існуючі рішення.

У другому розділі здійснено аналіз вимог користувачів через сценарії використання, сформульовано функціональні та нефункціональні вимоги.

У третьому розділі розроблено архітектуру системи, виконано моделювання бізнес-процесів, спроєктовано базу даних та обґрунтовано вибір технологій.

У четвертому розділі реалізовано серверну частину на ASP.NET Core, клієнтський SPA на React та CRM-модуль.

У п'ятому розділі представлено результати тестування (unit, інтеграційне, сценарне) та експериментальну перевірку системи.

Висновки роботи містять основні результати дослідження та їх аналіз.

Результат дослідження: повністю функціональний веб-додаток, який автоматизує бронювання, забезпечує надійний server-side контроль, рольовий доступ, інстансну модель ресурсу та інтегровану аналітику.

КЛЮЧОВІ СЛОВА: ВЕБ-СИСТЕМА БРОНЮВАННЯ, СПОРТИВНІ МАЙДАНЧИКИ, ASP.NET CORE, REACT, POSTGRESQL, JWT, CRM, ІНСТАНСНА МОДЕЛЬ, АНАЛІТИКА, ТЕСТУВАННЯ ПЗ.

ANNOTATION

The bachelor's thesis contains 80 pages, 25 figures, 16 tables, a list of 40 references, and 3 appendices.

The purpose of the thesis is to develop an adaptive web platform for booking sports facilities with an integrated decision support system.

Object of the study: the processes of booking and operational management of sports facilities in the digital environment.

Subject of the study: methods, models and software tools for the design and implementation of a web-based sports facilities booking system.

The first section analyzes the current state of sports facilities booking problems in Ukraine, examines the main concepts of the subject area, and compares existing solutions.

The second section presents the collection and analysis of user requirements through usage scenarios, formulates functional and non-functional requirements.

The third section describes the development of the software architecture, business process modeling, database design, and justification of the chosen technologies.

The fourth section covers the implementation of the backend using ASP.NET Core, the client-side SPA on React, and the CRM module.

The fifth section presents the results of testing (unit, integration, and scenario testing) and experimental validation of the system.

The conclusions of the work contain the main results of the research and their analysis.

Result of the study: a fully functional web application that automates the booking process, provides reliable server-side control, role-based access, an instance-based resource model, and integrated analytics.

KEYWORDS: BOOKING WEB SYSTEM, SPORTS FACILITIES, ASP.NET CORE, REACT, POSTGRESQL, JWT, CRM, INSTANCE MODEL, ANALYTICS, SOFTWARE TESTING.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API — Application Programming Interface

SPA — Single Page Application

CRM — Customer Relationship Management

JWT — JSON Web Token

REST — Representational State Transfer

ORM — Object-Relational Mapping

DTO — Data Transfer Object

KPI — Key Performance Indicator

UI — User Interface

UX — User Experience

HTTP — HyperText Transfer Protocol

CRUD — Create, Read, Update, Delete

SQL — Structured Query Language

DBMS — Database Management System

UTC — Coordinated Universal Time

ER-діаграма — Entity-Relationship Diagram

Sports Field Instance (інстанс майданчика) — окремий фізичний ресурс або одиниця спортивного майданчика, доступна для бронювання

ВСТУП

У наш час інформаційні технології активно переходять від автоматизації окремих операцій до комплексної цифровізації цілих сервісних процесів. Програмний продукт сьогодні вже не просто інструмент, а повноцінний посередник між компанією та клієнтом. Якщо раніше достатньо було мати сайт чи сторінку в месенджері, то зараз користувач очікує зручної, швидкої та передбачуваної взаємодії.

Незважаючи на високий попит на цифрові рішення, більшість спортивних об’єктів в Україні досі використовують застарілі методи ведення записів: телефонні дзвінки, переписку в месенджерах та таблиці в Excel. Такі підходи створюють багато проблем — дублювання бронювань, затримки з підтвердженням, відсутність актуальної інформації про вільні слоти та складнощі з масштабуванням [6, с. 54]. Власники майданчиків часто не мають інструментів для нормального контролю завантаженості та прибутковості.

Особливість спортивної сфери полягає в тому, що об’єктом бронювання є не абстрактна послуга, а конкретний ресурс (корт, поле, зал) у чітко визначений час. При цьому один майданчик може мати кілька інстансів (наприклад, кілька кортів). Це вимагає особливої архітектури, яка забезпечує контроль доступності в реальному часі та запобігає конфліктам розкладу.

Актуальність теми зумовлена необхідністю створення сучасної системи, яка поєднує три важливих контури: клієнтський (пошук і бронювання), операційний (керування в CRM) та аналітичний (KPI для прийняття рішень). Без такого комплексного підходу система або залишається просто «вітриною», або стає незручним внутрішнім інструментом.

Об’єктом дослідження є процеси бронювання та оперативного управління спортивними майданчиками в цифровому середовищі.

Предметом дослідження виступають методи, моделі та програмні засоби проєктування і реалізації веб-системи, яка забезпечує контроль доступності, коректність статусних переходів і інтегровану аналітичну підтримку.

Метою роботи є розроблення та валідація практично придатної адаптивної веб-платформи бронювання спортивних майданчиків з інтегрованою системою підтримки рішень, здатної працювати в реальних умовах.

Для досягнення поставленої мети були визначені такі завдання: провести аналіз предметної області та існуючих рішень, сформувати функціональні та нефункціональні вимоги, обґрунтувати архітектуру системи, реалізувати серверну та клієнтську частини, розробити CRM-модуль, виконати тестування та проаналізувати результати.

Методологічну основу роботи склали системний аналіз, принципи об’єктно-орієнтованого проєктування, структурне моделювання процесів, сучасні методи веб-розробки та підходи до забезпечення якості програмного забезпечення [11, с. 39].

Наукова новизна одержаних результатів полягає в практично орієнтованому поєднанні інстансної моделі ресурсу, сервероцентричного контролю доступу та формалізованого життєвого циклу бронювання в межах єдиного програмного рішення.

Практичне значення роботи полягає в тому, що розроблена система може бути безпосередньо використана спортивними об’єктами, а запропонована архітектура легко адаптується під подальше розширення (онлайн-оплата, сповіщення, розширена аналітика тощо).

Бакалаврська робота містить 80 сторінок, 25 рисунків, 5 розділів, список використаних джерел із 40 найменувань, 3 додатки.

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ТА ТЕОРЕТИЧНИХ ОСНОВ ПРОЄКТУВАННЯ ВЕБ-СИСТЕМ БРОНЮВАННЯ

1.1 Аналіз сучасного стану проблематики

У сфері надання послуг інформаційні системи поступово перестали бути лише допоміжним інструментом для ведення обліку. Сьогодні програмний продукт часто виступає основним каналом взаємодії між клієнтом і організацією. Це добре видно на прикладі сервісів доставки, онлайн-запису, електронної комерції та систем бронювання. Користувач уже не хоче витрачати час на телефонні дзвінки або очікування відповіді в месенджері. Він очікує, що зможе самостійно відкрити веб-застосунок, побачити актуальну інформацію, вибрати потрібну послугу та отримати підтвердження без зайвих дій.

Для спортивних об’єктів така зміна поведінки клієнтів також є актуальною. Якщо раніше адміністратор міг вести записи у зошиті або таблиці, то зараз такий підхід уже погано відповідає реальним потребам користувачів. Спортивний майданчик є ресурсом, доступність якого залежить від конкретного часу, виду спорту, кількості фізичних інстансів та стану вже створених бронювань. Тому проста таблиця або чат у месенджері не можуть повністю забезпечити контроль усіх цих умов.

Цифровізація в цій сфері дає переваги не тільки клієнтам, але й адміністраторам та власникам об’єктів. Автоматизована система дозволяє зменшити кількість ручних операцій, пришвидшити обробку заявок, знизити ризик людських помилок і накопичувати дані для подальшого аналізу. Якщо всі бронювання зберігаються в єдиній базі даних, то власник може бачити реальну завантаженість майданчика, оцінювати популярні години, аналізувати скасування та приймати рішення не на основі припущень, а на основі фактичних показників.

Незважаючи на це, значна частина спортивних об’єктів в Україні досі використовує прості ручні способи ведення записів: телефонні дзвінки, переписку в WhatsApp чи Telegram, Excel або Google-таблиці. На перший погляд такі інструменти здаються зручними, бо вони не потребують складного впровадження.

Але в реальній роботі вони швидко створюють проблеми: дублювання бронювань, затримки з підтвердженням, відсутність єдиної актуальної інформації про вільні слоти та складність контролю прибутковості [6, с. 54].

Проблема посилюється тим, що якість роботи такого “ручного” процесу дуже залежить від конкретного адміністратора. Якщо працівник уважний і має досвід, певний час система може працювати відносно нормально. Але при зміні персоналу, високому навантаженні або одночасній роботі кількох менеджерів кількість помилок зростає. Один працівник може підтвердити бронювання в чаті, інший — записати іншу людину в таблиці, а третій — не побачити оновлення. У результаті клієнт отримує неправильну інформацію, а бізнес втрачає довіру.

Окремо потрібно виділити проблему фрагментації даних. У ручній схемі частина інформації зберігається в месенджерах, частина — у таблицях, частина — у паперовому журналі або в пам’яті адміністратора. З інженерної точки зору це означає відсутність єдиного джерела істини. Якщо дані дублюються в різних місцях, то неможливо гарантувати їхню узгодженість. Через це власник не може швидко отримати достовірну картину роботи об’єкта: скільки було бронювань, які години найпопулярніші, скільки заявок скасовано і які ресурси простоюють.

Специфіка спортивної сфери полягає в тому, що об’єктом бронювання є не абстрактна послуга, а конкретний фізичний ресурс у конкретний часовий інтервал. Наприклад, якщо спортивний комплекс має чотири тенісні корти, то кожен корт повинен розглядатися як окрема одиниця доступності. Не можна просто сказати, що “майданчик вільний”, якщо насправді один корт зайнятий, другий неактивний, а третій доступний тільки для певного виду спорту. Саме тому для таких систем потрібна інстансна модель ресурсу, де загальна інформація про спортивний об’єкт відділяється від конкретних фізичних одиниць бронювання.

На практиці відсутність такої моделі призводить до реальних конфліктів. Я сам стикався з ситуацією, коли намагався забронювати майданчик для міні-футболу через Telegram. Адміністратор підтвердив час на 19:00, але після приїзду виявилося, що майданчик уже зайнятий іншою командою, яка бронювала по телефону. Тобто інформація про бронювання існувала, але вона не була

синхронізована між каналами комунікації. Для клієнта це виглядає як ненадійність сервісу, а для бізнесу — як пряма репутаційна втрата.

Під час переддипломної практики я детальніше побачив, як подібні ситуації виникають у роботі спортивних об’єктів. В одному з випадків адміністратори паралельно використовували паперовий журнал, Google-таблицю та Telegram-чат. Така схема не мала жодного формального механізму блокування слоту або перевірки конфліктів. Один менеджер міг внести бронювання в таблицю, але не повідомити іншого. Інший міг підтвердити заявку в чаті, не перевіривши паперовий журнал. У результаті клієнти отримували суперечливі відповіді, а менеджери витрачали час на розв’язання конфліктів замість нормальної роботи з заявками.

За моїми спостереженнями, рівень завантаженості окремих майданчиків у будні дні часто залишався невисоким, хоча попит на гру в окремі години існував. Частина проблеми була не у відсутності клієнтів, а в тому, що вони не могли швидко знайти актуальну інформацію. Якщо людина не отримує відповідь протягом кількох хвилин, вона часто шукає інший варіант або просто відмовляється від бронювання. Таким чином, неефективна організація запису прямо впливає на дохід спортивного об’єкта.

Для постійних клієнтів також відсутній зручний сервісний контур. У ручних схемах зазвичай немає особистого кабінету, історії бронювань, можливості повторити попередній запис або залишити відгук після відвідування. Через це система взаємодії з клієнтом кожного разу починається майже з нуля. Менеджер знову уточнює ім’я, телефон, час, вид спорту, а клієнт знову пояснює ті самі дані. Це збільшує навантаження на персонал і знижує якість користувацького досвіду.

Ще одна важлива проблема — відсутність прозорості для клієнта. Якщо користувач хоче пограти в п’ятницю ввечері, йому потрібно писати або телефонувати в кілька місць, окремо уточнювати ціну, наявність вільного часу, умови користування, душ, роздягальні чи інші додаткові послуги. У сучасному веб-сервісі така інформація має бути доступною одразу. Якщо вона прихована в чаті з адміністратором, то процес бронювання стає повільним і непередбачуваним.

З технічної точки зору система бронювання спортивних майданчиків повинна вирішувати не тільки задачу створення запису в базі даних. Вона має забезпечувати перевірку доступності слоту, контроль перетину часових інтервалів, облік статусів бронювання, авторизацію користувачів і захист від одночасного створення конфліктних записів. Особливо важливим є server-side контроль, оскільки перевірка лише на frontend не гарантує коректності даних. Користувацький інтерфейс може показати попередню доступність, але остаточне рішення про створення бронювання повинен приймати backend.

Таким чином, актуальність теми визначається не просто бажанням зробити ще один веб-сайт для запису на майданчик. Основна проблема полягає у створенні цілісної інформаційної системи, яка поєднує клієнтський контур, CRM-контур і аналітичний контур. Клієнтський контур відповідає за пошук і бронювання, CRM — за управління заявками, статусами та ресурсами, а аналітика — за оцінку ефективності роботи об’єкта. Без такого поділу система або перетворюється лише на красиву “вітрину”, або стає внутрішньою таблицею, незручною для клієнтів.

1.2 Основні поняття та визначення в предметній області

Для коректного проєктування системи необхідно формалізувати основні поняття предметної області. Якщо на початку роботи не визначити, що саме вважається бронюванням, ресурсом, інстансом, статусом або роллю користувача, то на етапі реалізації бізнес-логіка швидко стане неузгодженою. У системах бронювання це особливо критично, бо помилка в одному правилі може призвести до подвійного запису або некоректного відображення доступності.

Одним із ключових понять є життєвий цикл бронювання. У межах розроблюваної системи бронювання проходить через набір станів: Pending (очікує підтвердження), Confirmed (підтверджено), Completed (завершено) або Cancelled (скасовано). Така модель дозволяє не просто зберігати факт бронювання, а контролювати його стан у часі. Наприклад, заявка після створення не повинна

автоматично вважатися завершеною або підтвердженою без відповідної дії менеджера чи системного процесу.

Кожен статус має власні правила. Бронювання у стані Pending може бути підтверджене менеджером або скасоване. Confirmed означає, що заявка прийнята і слот має вважатися зайнятим. Completed використовується для позначення фактично завершеної послуги. Cancelled означає, що бронювання більше не повинно блокувати ресурс. Якщо такі правила не визначити наперед, то в коді починають з'являтися неформальні перевірки в різних місцях, а це ускладнює підтримку системи.

Формалізація життєвого циклу також важлива для аналітики. Наприклад, скасовані бронювання не повинні враховуватися так само, як завершені, а заявки у статусі Pending не завжди можна трактувати як гарантовану зайнятість майданчика. Тому статус бронювання впливає не тільки на інтерфейс користувача, а й на розрахунок завантаженості, прибутковості та інших KPI.

Друге важливе поняття — server-side source of truth, тобто сервер як єдине джерело істини. У такій архітектурі саме backend остаточно визначає, чи можна створити бронювання, чи має користувач право виконати дію та чи допустимий перехід між статусами. Frontend може виконувати попередні перевірки для зручності користувача, але він не повинен бути основним місцем прийняття критичних рішень. Це пов'язано з тим, що дані на клієнті можуть бути застарілими або навмисно зміненими.

Для систем бронювання це правило має практичне значення. Якщо два користувачі одночасно відкрили сторінку з вільним слотом, frontend в обох може показати однакову доступність. Але коли обидва натиснуть кнопку підтвердження, backend повинен повторно перевірити стан ресурсу та дозволити створення лише одного коректного запису. Саме тому перевірка доступності повинна виконуватися на сервері безпосередньо перед збереженням бронювання.

Також у системі необхідно визначити рольову модель доступу. У межах проєкту передбачено кілька основних ролей користувачів:

- Гість — може переглядати майданчики, доступні слоти та створювати

гостьове бронювання;

- Клієнт — після реєстрації може бронювати майданчики, переглядати історію власних бронювань і залишати відгуки;
- Менеджер — працює із заявками, підтверджує або скасовує бронювання відповідно до правил;
- Власник — має доступ до налаштувань майданчика, розкладу, цін, інстансів і аналітики.

Рольова модель потрібна не тільки для безпеки, але й для розділення відповідальності між користувачами. Клієнт не повинен мати доступ до редагування розкладу або підтвердження заявок. Менеджер повинен працювати лише з тими майданчиками, до яких має відношення. Власник повинен бачити управлінську інформацію, але це не означає, що всі користувачі системи мають доступ до фінансових або аналітичних даних.

У сучасних веб-застосунках така логіка зазвичай реалізується через механізми автентифікації та авторизації. Автентифікація відповідає на питання, хто саме виконує запит, а авторизація — чи має цей користувач право на конкретну дію. У моєму проєкті ці поняття стали основою для подальшого використання ролей, claims та перевірок доступу на рівні backend.

Окремим поняттям є інстанс майданчика. Це конкретна фізична одиниця ресурсу, яку можна забронювати: окремий корт, поле або зал. Відокремлення інстансу від загального майданчика дає змогу правильно моделювати реальні спортивні комплекси. Наприклад, один об’єкт може мати кілька футбольних полів або кілька тенісних кортів, і кожен із них повинен мати власну доступність у часі.

Ще одним важливим елементом є розклад. Розклад визначає, у які дні та години доступний певний ресурс або тип активності. Без формалізованого розкладу система не зможе коректно відрізнити вільний слот від часу, коли майданчик взагалі не працює. Саме тому розклад у системі розглядається як окрема сутність, а не просто як текстовий опис у картці майданчика.

У таблиці 1.1 наведено основні сутності предметної області та їх роль у процесі.

Таблиця 1.1

Сутності предметної області та їх роль

Сутність	**Ключові атрибути**	**Роль у процесі**
Користувач	Id, Email, Role, FullName	Ініціює дії, авторизація та контроль доступу
Майданчик	Id, Name, OwnerId, IsDeleted	Об’єкт надання послуги
Інстанс майданчика	Id, SportsFieldId, DisplayName, IsActive	Фізична одиниця доступності
Розклад	DayOfWeek, AvailableFrom, AvailableTo	Правила часової доступності
Бронювання	Id, StartTimeUtc, EndTimeUtc, Status, TotalPrice	Операційна подія циклу бронювання
Відгук	Id, Rating, Comment, CreatedAt	Оцінка якості після завершеної послуги

З наведеної таблиці видно, що центральною сутністю є бронювання, оскільки саме воно зв’язує користувача, майданчик, інстанс і часовий інтервал. При цьому бронювання не може існувати повністю ізольовано. Воно залежить від доступності ресурсу, статусу, прав користувача та правил розкладу. Тому під час проєктування бази даних необхідно було забезпечити зв’язки між сутностями через зовнішні ключі та уникнути дублювання критичних даних.

Для кращого розуміння взаємодії користувачів із системою нижче наведено контекстну діаграму на рисунку 1.1

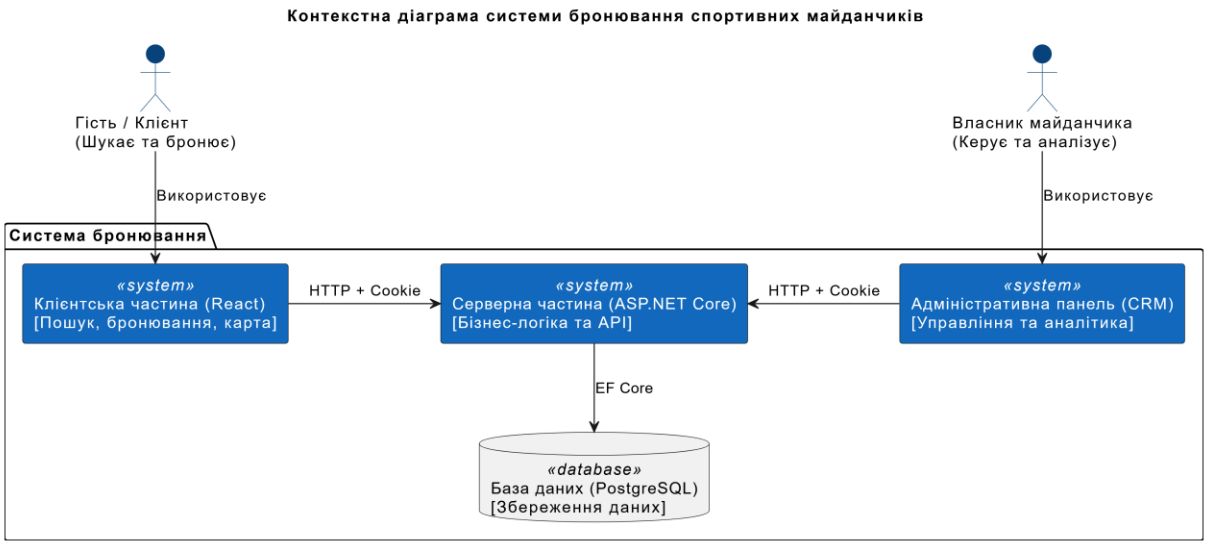

Рисунок 1.1 — Контекстна діаграма системи

1.3 Огляд існуючих рішень та порівняння

Перед розробкою власної системи мною було проаналізовано кілька класів готових рішень, які можуть використовуватися для запису або бронювання послуг. Такий аналіз потрібен для того, щоб визначити, які підходи вже існують на ринку, які функції реалізовані добре, а які не повністю підходять для задачі бронювання спортивних майданчиків.

Умовно розглянуті рішення можна поділити на три групи. Перша група — універсальні календарні та booking-сервіси, наприклад Calendly, Google Calendar або Bookly. Вони зручні для планування консультацій, зустрічей або прийомів, де один виконавець надає одну послугу в певний час. Але для спортивного майданчика така модель є занадто спрощеною. Вона не враховує наявність кількох фізичних ресурсів в одному об’єкті, різних типів спорту, різних цін і складних правил доступності.

Друга група — більші платформні рішення для фітнесу та сервісного бізнесу, наприклад Mindbody, Glofox, PerfectGym або ClubManager. Такі системи мають широкий набір функцій: роботу з клієнтами, абонементами, груповими заняттями, тренерами та оплатами. Але ця функціональність часто надмірна для невеликого спортивного майданчика, який здає корти або поля погодинно. У таких випадках складність системи починає заважати, бо адміністратор повинен налаштовувати багато непотрібних модулів.

Третя група — вузькоспеціалізовані рішення для спортивних шкіл, клубів або академій. Вони ближчі до предметної області, але часто орієнтовані на тренувальний процес, абонементи або довгострокову роботу з учнями. Для простої погодинної оренди майданчика такі системи також не завжди зручні. Крім того, частина таких рішень має застарілий інтерфейс, слабку адаптивність під мобільні пристрої або недостатню локалізацію для українського ринку.

Під час аналізу я звертав увагу не тільки на наявність окремих функцій, а й на те, як вони поєднані між собою. Наприклад, система може мати календар, але не мати інстансної моделі ресурсу. Вона може мати особистий кабінет, але не

мати чіткого статусного циклу бронювання. Або може підтримувати CRM, але не забезпечувати достатньої аналітики для власника. Саме такі розриви між модулями часто роблять готові рішення незручними для реального використання.

Окремою проблемою є конкурентний доступ. У системах бронювання спортивних майданчиків потрібно гарантувати, що один і той самий слот не буде зайнятий двічі. Якщо готова платформа не виконує повторну server-side перевірку перед створенням запису або не використовує транзакційну логіку, то при одночасних запитах можуть виникати конфлікти. Під час тестового ознайомлення з кількома сервісами я помітив, що не всі системи однаково добре обробляють такі ситуації.

Для українських спортивних об’єктів також важлива вартість впровадження. Багато платформних рішень орієнтовані на західний ринок і мають підписку у валюті. Для невеликого приватного майданчика така модель може бути економічно невигідною. Крім того, частина систем не враховує сценарій роботи без обов’язкової онлайн-оплати, хоча в українських умовах багато об’єктів спочатку хочуть працювати з оплатою на місці або після підтвердження менеджером.

Для порівняння було обрано критерії, які найбільше впливають на щоденну роботу спортивного об’єкта: підтримка інстансної моделі, рольовий контроль доступу, статусний цикл бронювання, CRM-контур, KPI-аналітика та можливість адаптації під локальний контекст. Результати порівняння наведено в таблиці 1.2.

Таблиця 1.2

Порівняння класів рішень

Критерій	**Універсальні системи**	**Платформні рішення**	**Цільове рішення**
Інстансна модель ресурсу	Частково	Частково	Повноцінно
Рольовий контроль доступу	Базово	Середньо	Policies + claims
Статусний цикл	Базовий	Середній	Формалізований автомат
CRM-контур	Мінімальний	Середній	Повноцінний
KPI-аналітика	Обмежена	Часткова	Розширена

З таблиці видно, що універсальні системи можуть бути корисними для простого запису, але вони недостатньо враховують специфіку спортивних ресурсів. Платформні рішення мають ширший функціонал, проте не завжди зручні для невеликих спортивних об’єктів через складність, вартість і надмірність налаштувань. Найбільш придатним підходом для поставленої задачі є цільове рішення, яке відразу проєктується під інстансну модель майданчиків, статусний цикл і CRM-обробку заявок.

У результаті аналізу я дійшов висновку, що розробка власної веб-системи є доцільною. Вона дозволяє врахувати ті сценарії, які не завжди якісно реалізовані в готових продуктах: гостьове бронювання, server-side контроль доступності, підтвердження менеджером, роботу з кількома інстансами одного майданчика та аналітику для власника. При цьому система може залишатися простішою за великі платформи, але точніше відповідати реальним процесам спортивного об’єкта.

Також перевагою власної розробки є можливість поступового розширення. На першому етапі можна реалізувати базовий цикл бронювання, CRM і аналітику, а в майбутньому додати онлайн-оплату, SMS або push-сповіщення, інтеграцію з календарями чи систему абонементів. Такий підхід дозволяє не перевантажувати початкову версію системи, але не закривати можливість подальшого розвитку.

1.4 Висновки по розділу

У першому розділі було розглянуто сучасний стан проблеми бронювання спортивних майданчиків, основні поняття предметної області та класи існуючих програмних рішень. Під час аналізу встановлено, що ручні способи ведення записів не забезпечують достатньої надійності, оскільки не мають централізованого джерела даних, формалізованого статусного циклу та механізмів захисту від конфліктів бронювання.

Було визначено, що для спортивної сфери критичним є поняття інстансу ресурсу. Один спортивний об’єкт може містити кілька фізичних одиниць бронювання, тому спрощена модель “один майданчик — один ресурс” не

підходить для реальної експлуатації. Також було обґрунтовано необхідність server-side контролю, оскільки саме сервер повинен остаточно перевіряти доступність слоту, права користувача та коректність статусного переходу.

Порівняння готових рішень показало, що універсальні системи мають обмежену доменну адаптацію, а платформні сервіси часто є надмірними для невеликих спортивних об’єктів. Жоден із розглянутих класів рішень не поєднує в достатній мірі інстансну модель ресурсу, CRM-контур, формалізований життєвий цикл бронювання та аналітику для власника.

На основі проведеного аналізу було підтверджено доцільність розробки власної адаптивної веб-платформи бронювання спортивних майданчиків. Така система повинна поєднувати клієнтський інтерфейс для пошуку й бронювання, CRM-модуль для менеджерів та аналітичний блок для підтримки управлінських рішень. Отримані результати стали основою для подальшого формування вимог, постановки задачі та проєктування архітектури системи.

РОЗДІЛ 2. АНАЛІЗ ВИМОГ ТА ПОСТАНОВКА ЗАДАЧІ

2.1 Збір та аналіз вимог користувачів (сценарії використання)

Аналіз вимог я проводив на основі реальних проблем, які виникають під час бронювання спортивних майданчиків. У багатьох спортивних об’єктах процес запису досі організований через телефонні дзвінки, повідомлення в месенджерах або таблиці Excel. Такий підхід може працювати при невеликій кількості клієнтів, але він погано масштабується, коли одночасно з’являється кілька заявок на один і той самий часовий проміжок. Через це виникають подвійні бронювання, затримки з підтвердженням, помилки при зміні розкладу та складнощі з контролем прибутковості [6, с. 54].

На етапі збору вимог я намагався розглядати систему не лише як веб-форму для створення бронювання, а як повноцінний програмний продукт із кількома групами користувачів. У такій системі є клієнт, який хоче швидко знайти вільний слот; менеджер, який обробляє заявки; власник, який контролює завантаженість і фінансові показники; а також гостьовий користувач, який може створити бронювання без повної реєстрації. Кожна з цих ролей має власні очікування, тому вимоги не можна було формувати лише з позиції одного користувача.

Для клієнта найбільш важливими є швидкість роботи, простий інтерфейс і актуальна інформація про доступність майданчика. Якщо користувач бачить вільний слот на сторінці, але після заповнення форми отримує повідомлення про помилку, довіра до системи знижується. Тому вже на етапі аналізу було зрозуміло, що frontend може лише показувати попередній стан, а остаточна перевірка повинна виконуватися на сервері безпосередньо перед створенням бронювання.

Для постійних клієнтів важливим є не тільки створення нового запису, але й перегляд історії попередніх бронювань. У реальних умовах багато людей відвідують один і той самий майданчик у схожий час: наприклад, щосуботи зранку або після роботи у будні. Через це функція особистого кабінету та історії бронювань має практичне значення, оскільки дозволяє користувачу швидше

повторити типову дію без повторного введення всіх даних.

Під час спілкування з адміністраторами і менеджерами спортивних об’єктів я звернув увагу, що значна частина їхньої роботи складається з повторюваних операцій. Менеджер перевіряє, чи вільний час, підтверджує або скасовує заявку, уточнює контактні дані, оновлює розклад і відповідає клієнтам на однакові запитання. Якщо ці дії виконуються вручну, то помилка може виникнути навіть при звичайному перенесенні даних з одного джерела в інше. Саме тому CRM-модуль був розглянутий як окрема важлива частина системи, а не як додаткова функція.

Для власника майданчика головним є не сам факт створення бронювання, а контроль ефективності використання ресурсів. Власник повинен бачити, які години найбільш завантажені, скільки бронювань було скасовано, які майданчики використовуються частіше, а які простоюють. Без таких даних управління відбувається інтуїтивно. Тому в межах вимог було передбачено аналітичний контур, який дозволяє формувати показники завантаженості, скасувань, рейтингу та пікових годин.

Окремо я враховував зручність інтерфейсу для користувачів без технічної підготовки. У невеликих спортивних об’єктах менеджери часто не мають досвіду роботи зі складними CRM-системами. Через це надмірно складний інтерфейс може стати такою ж проблемою, як і відсутність системи. Вимоги до інтерфейсу формувалися так, щоб основні дії виконувалися за мінімальну кількість кроків: перегляд заявки, підтвердження, скасування, перегляд аналітики та редагування параметрів майданчика.

Щоб краще зрозуміти поведінку різних користувачів, я виділив основні сценарії використання системи. Такий підхід дозволив перейти від загальних побажань до конкретних функціональних вимог. У процесі аналізу я розглядав не тільки позитивні сценарії, коли користувач робить усе правильно, але й помилкові випадки: введення некоректних контактних даних, спробу забронювати зайнятий слот, скасування без прав доступу, підтвердження заявки в неправильному статусі.

Найважливішими виявилися такі кейси:

Гостьове бронювання — сценарій, у якому користувач без реєстрації хоче швидко створити заявку. У цьому випадку необхідно зберегти простоту процесу, але не послабити перевірки. Користувач повинен ввести мінімальні контактні дані, а сервер має перевірити доступність слоту, валідність введеної інформації та можливість створення бронювання для конкретного інстансу майданчика.

Авторизоване бронювання — сценарій для користувача, який уже має обліковий запис. У такому випадку частина даних може підтягуватися з профілю, а користувач отримує доступ до історії бронювань та можливості залишити відгук після завершення послуги. З точки зору backend-логіки важливо, щоб система не довіряла ідентифікатору користувача, переданому з клієнта, а брала його з автентифікованого контексту.

Робота менеджера в CRM-модулі включає обробку заявок у статусі Pending, підтвердження або скасування бронювань, перегляд розкладу та контроль активних записів. У цьому сценарії критичною є формалізація статусних переходів. Менеджер не повинен мати можливості виконати дію, яка суперечить бізнес-логіці системи, наприклад повторно підтвердити вже завершене бронювання або скасувати заявку без відповідних прав.

Окремий сценарій пов’язаний зі зміною параметрів майданчика. Власник або менеджер може змінювати ціни, розклад, активність інстансів або опис майданчика. При цьому такі зміни не повинні руйнувати вже створені бронювання. Наприклад, якщо інстанс тимчасово деактивується, система повинна коректно враховувати це для майбутніх бронювань, але не втрачати історію попередніх записів.

Найбільш технічно складним сценарієм є конкурентний доступ. Типова ситуація полягає в тому, що два користувачі майже одночасно намагаються забронювати один і той самий слот. Якщо система спочатку перевіряє доступність, а потім окремою операцією створює запис без транзакційного захисту, між цими двома діями може пройти достатньо часу, щоб інший користувач зайняв той самий слот. Через це вимога атомарності стала однією з основних.

У процесі аналізу було визначено, що перевірка доступності повинна враховувати не лише дату і час, але й конкретний інстанс майданчика, статус існуючих бронювань, розклад роботи та можливі перетини часових інтервалів. Просте порівняння початкового часу недостатнє. Наприклад, бронювання з 18:00 до 19:30 конфліктує не тільки з іншим записом на 18:00, а й із заявкою на 19:00, якщо часові інтервали перетинаються.

Ще одним аспектом стала робота з часовими даними. Спортивні об’єкти можуть мати різний графік роботи залежно від дня тижня, виду спорту або конкретного інстансу. Тому система повинна підтримувати гнучке налаштування доступних часових меж. Крім того, усі розрахунки часу доцільно виконувати на backend, щоб уникнути розбіжностей між клієнтським інтерфейсом і фактичним станом даних у базі.

Під час вивчення роботи спортивних об’єктів я також встановив, що для власників важливим є отримання агрегованих показників. Йдеться не лише про кількість бронювань, а про показники, які допомагають ухвалювати рішення: середня завантаженість, частка скасованих записів, пікові години, рейтинг майданчика та динаміка використання ресурсів. Ці вимоги безпосередньо вплинули на подальше проєктування аналітичного модуля.

Контрольні точки для основних сценаріїв наведено в таблиці 2.1.

Таблиця 2.1

Сценарії використання та контрольні точки

Сценарій	**Контрольна точка**	**Критерій успіху**
Гостьове бронювання	Валідація контактів	Бронювання без помилок даних
Авторизоване бронювання	Дії відповідно до статусу	Відсутність нелегальних дій
CRM-обробка pending	Регламент переходів	Коректний статусний цикл
Конкурентний доступ	Атомарна перевірка	Немає подвійних записів
Зміна параметрів поля	Цілісність активних записів	Немає конфліктів стану
Аналітичний перегляд	Коректність агрегатів	KPI відповідають сирим даним

Аналіз сценаріїв показав, що вони пов’язані між собою. Помилка в перевірці доступності впливає не лише на створення бронювання, а й на роботу CRM та

аналітики. Якщо система допустить подвійне бронювання, менеджер отримає конфліктні заявки, а клієнт зіткнеться з проблемою вже під час відвідування.

Контрольні точки я використав як основу для проєктування бізнес-логіки та подальшого тестування. Для кожного сценарію було визначено очікувану дію користувача і критерій, за яким можна перевірити коректність роботи системи.

Основна складність полягає не у формі бронювання, а в узгодженій роботі frontend, backend, бази даних і CRM. Якщо один із цих компонентів працює неправильно, якість усієї системи знижується.

Послідовність дій при гостьовому бронюванні показана на рисунку 2.1.

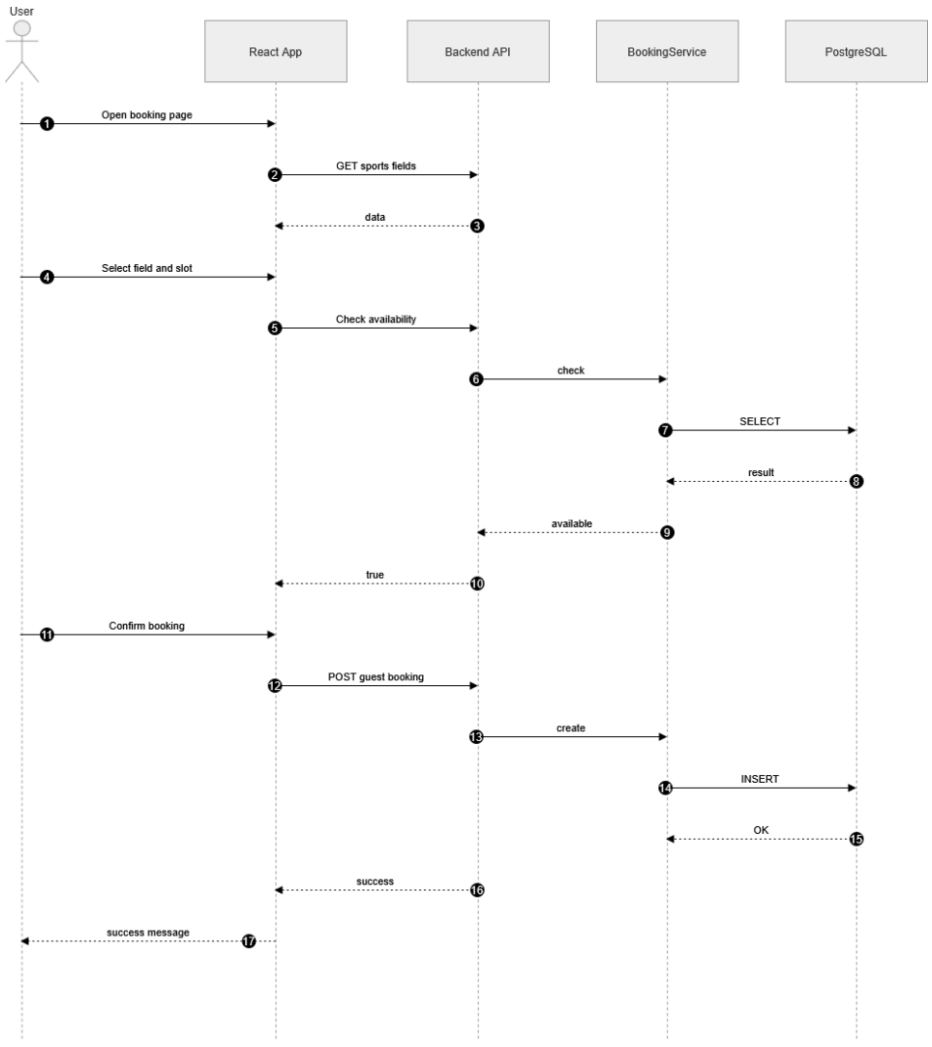

Рисунок 2.1 — Sequence-діаграма сценарію гостьового бронювання

Діаграма відображає двоетапний підхід до створення бронювання. Спочатку

користувач заповнює дані та отримує попередню перевірку доступності, після чого сервер виконує остаточну перевірку перед збереженням запису. Такий підхід дозволяє поєднати зручність інтерфейсу з надійністю серверної логіки.

Сценарій конкурентного доступу, який є критичним для систем бронювання, наведено на рисунку 2.2.

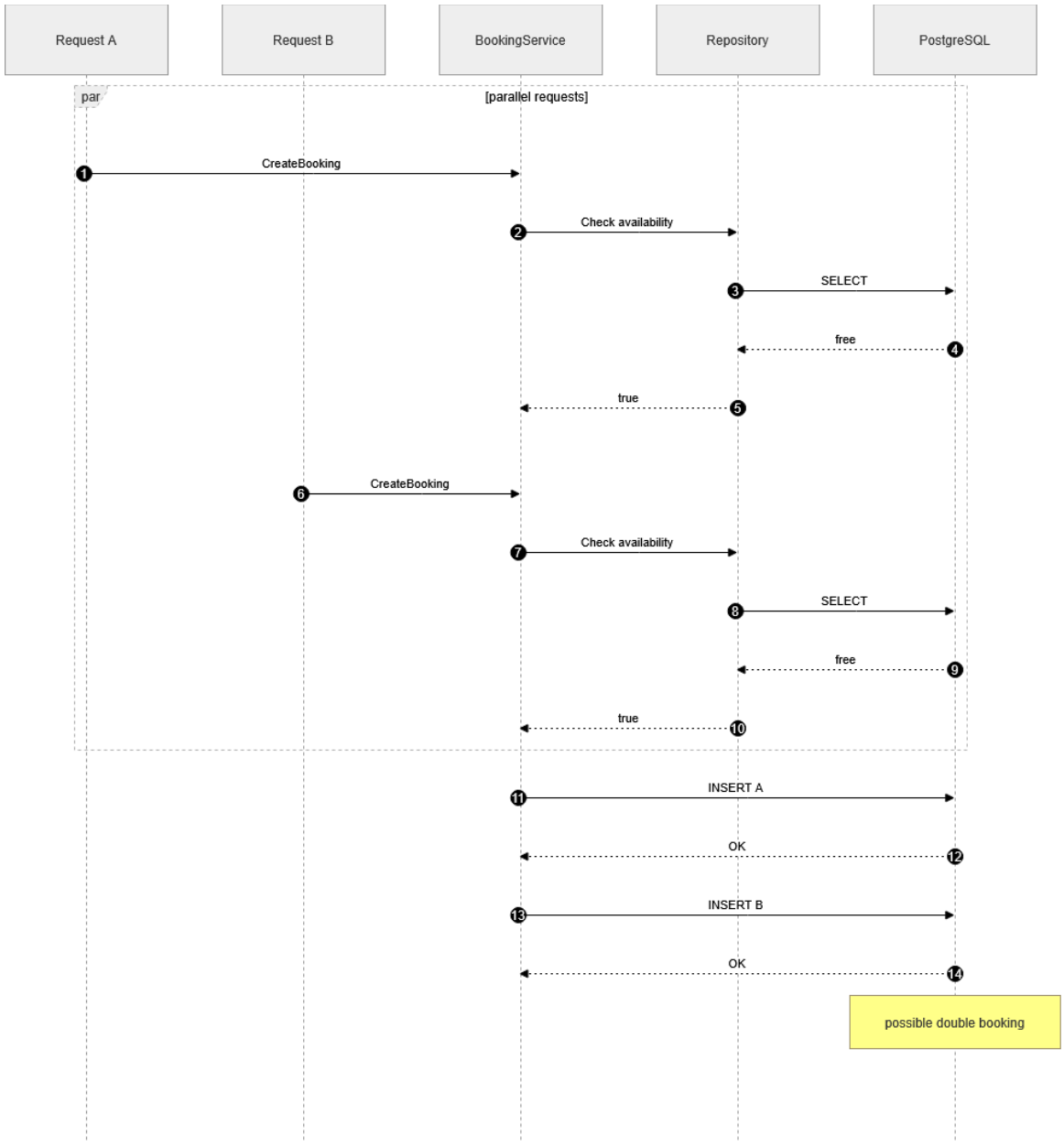

Рисунок 2.2 — Sequence-діаграма сценарію конкурентного доступу

Представлена послідовність підтверджує необхідність атомарної серверної перевірки при фіксації бронювання. У цьому випадку важливо, щоб перевірка

доступності та створення запису виконувалися як єдина логічна операція. Саме такий підхід дозволяє зменшити ризик race condition.

Організацію роботи менеджера в CRM-підсистемі показано на рисунку 2.3.

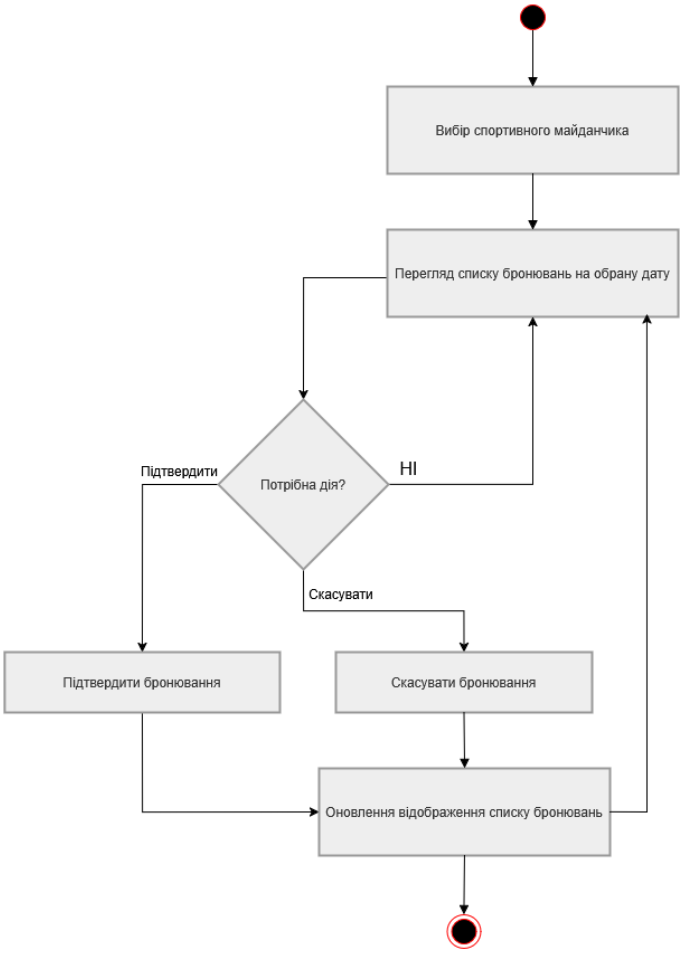

Рисунок 2.3 — Схема робочого сценарію менеджера у CRM

Аналіз цих сценаріїв показав, що для розроблюваної системи ключовими є server-side контроль, формалізовані статуси та чітке розділення ролей. Саме ці вимоги були покладені в основу подальшого формування функціональних і нефункціональних вимог.

2.2 Формулювання функціональних та нефункціональних вимог

На основі сценаріїв використання мною було сформовано перелік функціональних і нефункціональних вимог до системи. Функціональні вимоги

описують, які дії повинна виконувати система, а нефункціональні — якими властивостями вона повинна володіти під час виконання цих дій. Для системи бронювання обидва типи вимог є однаково важливими, оскільки наявність функції створення бронювання не має практичної цінності без гарантії коректної перевірки доступності.

Функціональні вимоги охоплюють пошук майданчиків, перегляд доступних слотів, створення бронювань, гостьове бронювання, роботу з історією, систему відгуків, CRM-керування статусами та аналітичні звіти. При формуванні цих вимог я орієнтувався на сценарії, описані в попередньому підрозділі, щоб кожна вимога відповідала конкретній проблемі предметної області.

Під час визначення пріоритетів я розділив вимоги за їх впливом на працездатність системи. Високий пріоритет отримали ті функції, без яких система не може виконувати своє основне призначення: створення бронювання, перевірка конфліктів, авторизація та керування статусами. Середній пріоритет отримали функції, які покращують сервіс, але не є критичними для базового циклу бронювання. Повний перелік функціональних вимог наведено в таблиці 2.2.

Таблиця 2.2

Функціональні вимоги системи

Код	**Формулювання**	**Пріоритет**
FR-01	Реєстрація/логін користувача	Високий
FR-02	Створення бронювання слоту	Високий
FR-03	Перевірка конфліктів бронювання	Високий
FR-04	Підтвердження/скасування бронювання	Високий
FR-05	Гостьове бронювання	Високий
FR-06	Відгуки/рейтинги після послуги	Середній
FR-07	Аналітика для менеджера	Високий
FR-08	CRM-керування статусами	Високий
FR-09	Фонове завершення прострочених	Середній

Наведені вимоги формують повний цикл взаємодії користувача із системою. Користувач може знайти майданчик, перевірити доступність, створити бронювання, а після завершення послуги залишити відгук. Менеджер при цьому отримує можливість контролювати заявки через CRM, а власник — переглядати

аналітичні показники.

Окремо варто пояснити вимогу FR-03, оскільки вона є однією з найважливіших у системі. Перевірка конфліктів бронювання повинна враховувати перетин часових інтервалів, статуси вже існуючих бронювань, активність інстансу та правила розкладу. Якщо цю логіку реалізувати поверхнево, система може показувати неправильну доступність або дозволяти створення конфліктних записів.

Вимога FR-08 пов'язана з формалізацією роботи менеджера. CRM-модуль не повинен бути просто таблицею із кнопками. Він має контролювати допустимі переходи статусів і не дозволяти виконувати дії, які суперечать бізнес-правилам. Наприклад, підтвердження можливе лише для заявки у відповідному стані, а скасування повинно враховувати роль користувача та поточний статус бронювання.

Нефункціональні вимоги визначають якість роботи системи. Для веб-платформи бронювання ключовими є безпека, продуктивність, надійність, підтримуваність і спостережуваність. Ці вимоги безпосередньо впливають на архітектуру. Наприклад, вимога підтримуваності привела до використання шарової структури Controller → Service → Repository, а вимога безпеки — до використання JWT і рольових перевірок.

Детальніше нефункціональні вимоги наведено в таблиці 2.3.

Таблиця 2.3

Нефункціональні вимоги системи

Код	**Категорія**	**Критерій приймання**
NFR-01	Безпека	JWT + role policies + claims-based context
NFR-02	Продуктивність	Інтерактивний час відповіді API для типових сценаріїв
NFR-03	Надійність	Коректна обробка 400/401/403/404/500 + без подвійних бронювань
NFR-04	Підтримуваність	Шарова архітектура Controller → Service → Repository
NFR-05	Спостережуваність/масштабованість	Structured logging + розширення без зміни ядра

При проєктуванні контуру безпеки потрібно було вирішити, як саме сервер буде визначати користувача та його права. Для цього було обрано підхід з JWT, де після автентифікації користувач отримує токен із необхідними claims. Це дозволяє backend-частині перевіряти роль, ідентифікатор користувача та контекст доступу без довіри до даних, які передаються з frontend.

Вимога продуктивності пов’язана з тим, що користувач очікує швидкої відповіді при пошуку доступних слотів. Якщо перевірка доступності працюватиме повільно, користувач може залишити систему ще до створення бронювання. Тому вже на етапі вимог було зрозуміло, що запити до бази даних мають бути оптимізовані, а найбільш часті операції повинні виконуватися без зайвих обчислень на клієнті.

Надійність системи означає не тільки відсутність помилок у позитивних сценаріях. Система повинна коректно реагувати на невалідні дані, відсутність авторизації, заборонені дії, неіснуючі ресурси та внутрішні помилки. Для цього в API мають використовуватися відповідні HTTP-статуси: 400 для некоректного запиту, 401 для неавторизованого доступу, 403 для забороненої дії, 404 для відсутнього ресурсу і 500 для непередбачених помилок.

Підтримуваність стала окремою вимогою, оскільки система має розвиватися. Якщо вся бізнес-логіка буде розміщена в контролерах, то тестування і подальші зміни стануть складними. Тому логіка бронювання, статусів і доступності повинна бути винесена в сервісний шар, а доступ до даних — у репозиторії. Це дозволяє ізолювати відповідальність компонентів і спростити модульне тестування.

Спостережуваність системи пов’язана з логуванням і можливістю аналізувати роботу програми після розгортання. Для реального програмного продукту недостатньо лише показати користувачу повідомлення про помилку. Потрібно мати журнали подій, за якими можна відстежити, що саме сталося: хто виконав дію, який endpoint був викликаний, який статус повернув сервер і на якому етапі виникла помилка.

Окремо я узагальнив головні недоліки сучасних booking-систем. Вони

наведені в таблиці 2.4.

Таблиця 2.4

Основні проблеми існуючих систем бронювання

Проблема	**Прояв у роботі**	**Наслідок для сервісу**
Слабка server-side валідація	Конфлікти при підтвердженні	Подвійні бронювання
Нечітка рольова модель	Некоректні доступи	Порушення безпеки
Неформалізовані статуси	Різне трактування дій	Операційний хаос
Відсутня інстансна модель	Хибна доступність	Втрата слотів/конфлікти
Слабка аналітика	Немає керованих KPI	Реактивне управління
Фрагментарна логіка	Складність підтримки	Регресії та помилки

З таблиці видно, що більшість проблем виникає не через відсутність однієї конкретної функції, а через слабку формалізацію бізнес-логіки. Наприклад, система може мати календар, але якщо вона не перевіряє конфлікти на сервері, то ризик подвійного бронювання залишається. Так само наявність ролей не гарантує безпеки, якщо backend не перевіряє права при кожній критичній операції.

Причинно-наслідковий зв’язок цих проблем зображено на рисунку 2.4.

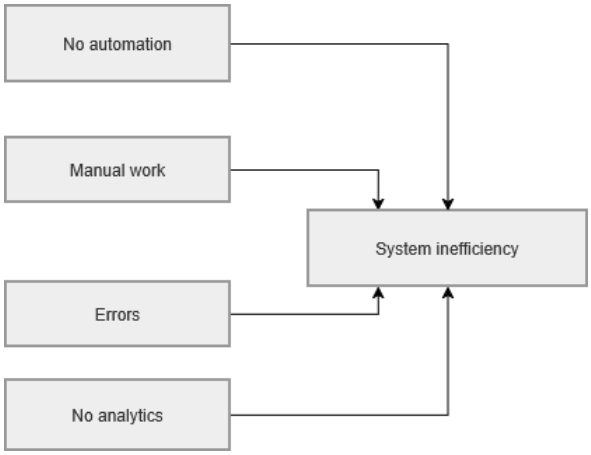

Рисунок 2.4 — Причинно-наслідкова діаграма проблем існуючих систем

2.3 Постановка задачі проєктування системи

Метою бакалаврської роботи є розробка адаптивної веб-платформи бронювання спортивних майданчиків з CRM-модулем та аналітичним контуром.

Система повинна забезпечувати зручний інтерфейс для клієнтів, інструменти управління для менеджерів і власників, а також механізми контролю доступності ресурсів на серверній стороні.

Досягнення цієї мети потребує не просто створення форми бронювання, а побудови повного циклу роботи із заявкою. Користувач повинен мати можливість знайти майданчик, вибрати час, створити бронювання, отримати підтвердження, а після завершення послуги залишити відгук. Менеджер повинен мати інструмент для обробки заявок, а власник — доступ до аналітичних показників.

Об’єктом дослідження є процеси бронювання та оперативного управління спортивними майданчиками в цифровому середовищі. До них належать пошук доступних слотів, створення бронювань, підтвердження заявок, контроль завантаженості, обробка скасувань та аналіз статистичних даних.

Предметом дослідження є методи, моделі та програмні засоби проєктування і реалізації веб-системи, яка забезпечує контроль доступності ресурсів, статусний цикл бронювань, рольовий доступ та аналітичну обробку даних.

Для досягнення поставленої мети необхідно було вирішити такі задачі:

- провести аналіз предметної області та існуючих рішень, виявити основні проблеми сучасних booking-систем і визначити ключові вимоги до нової платформи;
- здійснити збір та аналіз вимог користувачів через реальні сценарії використання (гостьове бронювання, робота менеджера в CRM, аналітика для власника) та сформувати чіткі функціональні і нефункціональні вимоги;
- спроєктувати архітектуру програмного забезпечення, розробити модель бази даних та виконати моделювання основних бізнес-процесів;
- реалізувати серверну частину системи на ASP.NET Core, клієнтський SPA-застосунок на React та окремий CRM-модуль для внутрішнього управління;
- виконати комплексне тестування системи (unit-тести, інтеграційне та сценарне тестування), проаналізувати результати та оцінити ефективність розробленого рішення.

Поставлені задачі охоплюють повний життєвий цикл створення програмного продукту: від аналізу проблеми до перевірки готової реалізації. Така послідовність дозволяє не перескакувати одразу до програмування, а спочатку зрозуміти, які саме бізнес-правила потрібно реалізувати.

Практичне значення розробки полягає в тому, що система може бути використана спортивними об’єктами різного масштабу. Для невеликого майданчика вона може замінити ручне ведення записів, а для більшого комплексу — стати основою для централізованого керування ресурсами. Крім того, архітектура системи передбачає можливість подальшого розвитку: додавання онлайн-оплати, push-сповіщень, інтеграції з календарями або системи абонементів.

2.4 Висновки по розділу

У другому розділі було виконано аналіз вимог до системи бронювання спортивних майданчиків. На основі дослідження предметної області та розгляду сценаріїв використання визначено основні групи користувачів, їхні потреби, очікування та критичні точки майбутньої системи. Проведений аналіз дозволив встановити, що найбільший вплив на якість роботи системи мають питання конкурентного доступу до ресурсів, коректності статусних переходів, рольового контролю доступу та достовірності аналітичних даних.

Сформовані функціональні вимоги описують основний набір можливостей системи, необхідних для підтримки повного життєвого циклу бронювання. До них належать реєстрація та авторизація користувачів, створення бронювань, гостьовий режим роботи, перевірка доступності часових слотів, CRM-керування статусами, робота з відгуками та формування аналітичних показників. Водночас нефункціональні вимоги визначають якісні характеристики програмного забезпечення, зокрема безпеку, продуктивність, надійність, підтримуваність та можливість подальшого масштабування системи.

Під час аналізу було встановлено, що майбутня система повинна будуватися навколо принципу server-side контролю, за якого всі критично важливі перевірки виконуються на серверній стороні. Такий підхід дозволяє забезпечити цілісність даних, мінімізувати ризик подвійних бронювань та виключити залежність від клієнтської логіки. Також було обґрунтовано необхідність використання інстансної моделі ресурсу, що дає змогу коректно працювати з декількома фізичними майданчиками в межах одного спортивного об’єкта, та формалізованого життєвого циклу бронювання з чітко визначеними правилами переходу між статусами.

Отримані результати стали основою для прийняття подальших архітектурних рішень, проєктування структури бази даних і моделювання бізнес-процесів. Сформований набір вимог дозволив чітко визначити межі системи, її ключові компоненти та принципи їх взаємодії. Таким чином, матеріали другого розділу сформували практичну та методологічну основу для переходу до етапу проєктування програмного забезпечення.

РОЗДІЛ 3. ПРОЕКТУВАННЯ СИСТЕМИ

3.1 Розробка архітектури програмного забезпечення

Під час проєктування архітектури системи я виходив із того, що звичайна CRUD-структура для такого проєкту буде недостатньою. Система бронювання спортивних майданчиків повинна не лише зберігати записи в базі даних, а й контролювати доступність конкретного ресурсу в конкретний момент часу, перевіряти права користувача, дотримуватися статусного циклу бронювання та залишатися придатною для подальшого розширення. Через це архітектуру потрібно було будувати так, щоб бізнес-логіка не змішувалася з контролерами або кодом доступу до бази даних.

На початку проєктування я визначив дві основні архітектурні вимоги. Перша вимога — надійна робота в умовах одночасного бронювання одного й того самого слоту кількома користувачами. Друга вимога — підтримуваність коду, тобто можливість додавати нові функції без повного переписування вже реалізованих модулів. Зокрема, у майбутньому система може бути розширена онлайн-оплатою, SMS або push-сповіщеннями, інтеграцією з календарями, системою абонементів або більш складною аналітикою.

Для реалізації цих вимог мною була обрана багатошарова архітектура. Її суть полягає в тому, що кожен шар має власну відповідальність і не повинен виконувати завдання іншого шару. Такий підхід добре підходить для ASP.NET Core-застосунків, оскільки фреймворк має вбудовану підтримку Dependency Injection, middleware-конвеєра, конфігурації сервісів та розділення логіки між контролерами, сервісами й репозиторіями.

Основні шари системи такі:

- Шар представлення (Presentation) — два окремі клієнтські застосунки: публічний інтерфейс для звичайних користувачів та CRM-інтерфейс для менеджерів і власників майданчиків;
- API-шар — єдина точка входу на сервері, яка приймає запити, виконує

базову валідацію та передає обробку далі;

- Прикладний шар (Application/Domain) — тут зосереджена вся основна бізнес-логіка: перевірка доступності слотів, контроль статусних переходів, правила ролей, сценарії створення та скасування бронювань;
- Шар доступу до даних (Data Access) — робота з базою даних PostgreSQL через ORM;
- Інфраструктурний шар (Infrastructure) — фонові задачі, логування, технічні сервіси та інші перехресні функції.

Загальну структуру архітектури показано на рисунку 3.1.

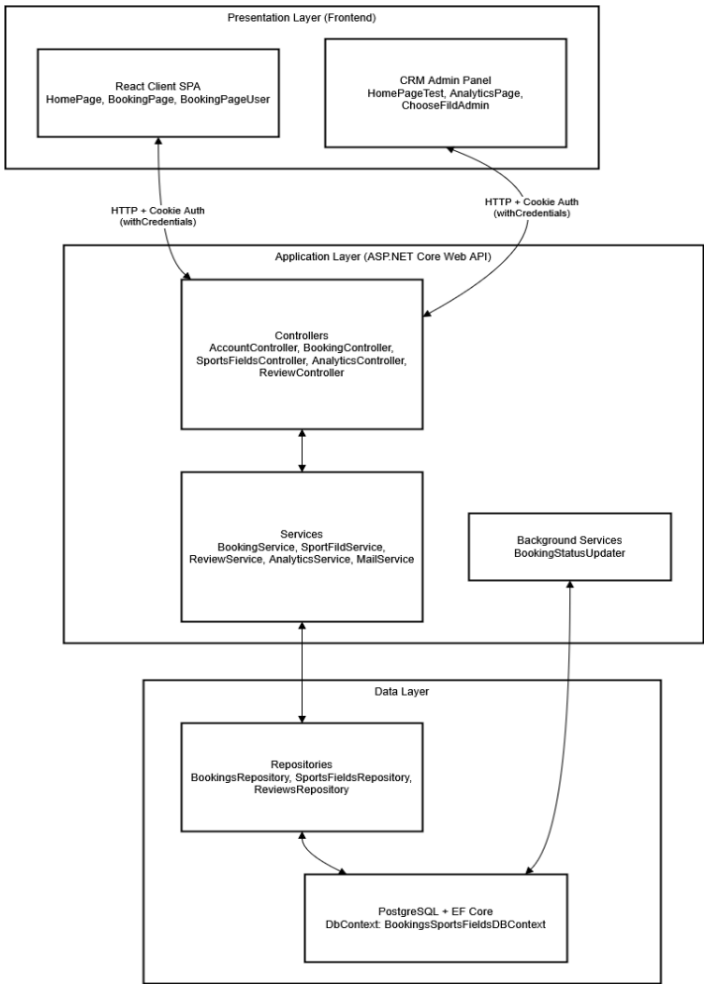

Рисунок 3.1 — Загальна архітектура системи (багатошарова модель)

У процесі проєктування я намагався не допустити ситуації, коли контролери перетворюються на місце зберігання всієї логіки. У невеликих навчальних

проєктах це іноді допускається, але для системи бронювання такий підхід швидко створив би проблеми. Наприклад, логіка перевірки доступності слоту використовується не лише при створенні бронювання, а й при попередній перевірці, відображенні доступних часових проміжків та валідації дій менеджера. Тому цю логіку доцільно винести в окремий сервіс, а не дублювати в різних контролерах.

Використання Dependency Injection у ASP.NET Core дозволило розділити залежності між компонентами. Контролер не створює сервіс напряму через оператор new, а отримує його через конструктор. Завдяки цьому конкретну реалізацію сервісу можна замінити під час тестування, наприклад, на mock-об’єкт. Це спрощує unit-тестування бізнес-логіки і зменшує зв’язаність між класами.

При проєктуванні взаємодії між шарами я також враховував принцип інверсії залежностей. Вищі рівні системи не повинні залежати від конкретних деталей збереження даних. Наприклад, сервіс бронювання має працювати з абстракцією репозиторію, а не напряму з SQL-запитами. Такий підхід дає можливість змінювати спосіб доступу до даних без суттєвого впливу на бізнес-логіку.

Потоки даних між клієнтом, API, сервісами та базою даних зображено на рисунку 3.2.

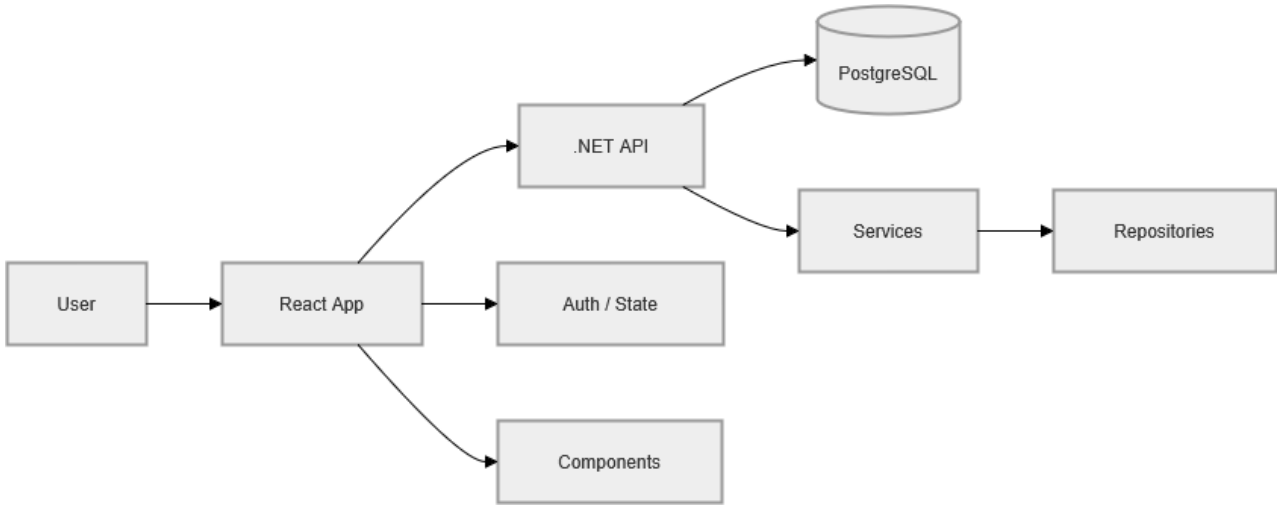

Рисунок 3.2 — Діаграма потоків даних

Під час аналізу потоків даних я визначив, які операції можуть виконуватися на стороні клієнта, а які обов’язково повинні залишатися на сервері. Наприклад, frontend може показати користувачу попередню доступність слоту, але він не повинен остаточно вирішувати, чи можна створити бронювання. Причина полягає в тому, що стан даних може змінитися між моментом завантаження сторінки і моментом підтвердження бронювання.

Ключовим архітектурним принципом стало правило server-side source of truth. Це означає, що backend є єдиним місцем, де приймаються остаточні рішення щодо доступності слоту, прав користувача, статусу бронювання та коректності бізнес-операції. Такий підхід зменшує ризик race condition, коли два користувачі одночасно намагаються зайняти один і той самий ресурс. Клієнтська частина в цьому випадку виконує роль інтерфейсу, але не джерела істини.

Для забезпечення цілісності даних у критичних сценаріях була передбачена транзакційна логіка. Операція створення бронювання повинна виконувати перевірку доступності та збереження запису як єдину логічну дію. Якщо на будь-якому етапі виявляється конфлікт, транзакція не повинна завершитися успішно. Такий підхід відповідає принципам ACID, особливо атомарності та узгодженості, що є важливими для систем бронювання.

3.2 Моделювання бізнес-процесів та системних компонентів

Для опису поведінки системи мною було використано кілька типів моделей. Це було потрібно тому, що одна діаграма не може повністю описати всі аспекти роботи системи. Use Case-діаграма дозволяє показати ролі користувачів і межі системи. Sequence-діаграми показують порядок взаємодії компонентів у часі. BPMN-модель зручна для опису бізнес-процесу обробки заявки в CRM. State Machine-діаграма використовується для формалізації життєвого циклу бронювання.

Use Case-діаграма основних ролей і функцій представлена на рисунку 3.3.

Рисунок 3.3 — Use case-діаграма ролей і функцій

На Use Case-діаграмі були виділені основні актори: гість, авторизований клієнт, менеджер і власник. Гість може переглядати майданчики та створювати бронювання без повної реєстрації. Авторизований клієнт отримує додаткові можливості: перегляд історії бронювань і залишення відгуків. Менеджер працює із заявками, підтверджує або скасовує бронювання. Власник має доступ до керування параметрами майданчика, розкладом, інстансами та аналітикою.

Таке розділення ролей потрібне не лише для зручності інтерфейсу, а й для безпеки. Наприклад, клієнт не повинен мати можливості змінювати розклад або підтверджувати власні заявки як менеджер. Тому на рівні backend необхідно перевіряти не тільки факт автентифікації, але й роль користувача та його зв’язок із конкретним майданчиком.

Найважливішою з технічної точки зору є Sequence-діаграма створення бронювання. Вона показує, як запит проходить від клієнтської частини до API, потім до сервісного шару, репозиторію та бази даних. У цьому сценарії основне значення має порядок виконання операцій: валідація вхідних даних, перевірка

доступності, перевірка розкладу, перевірка конфліктів і тільки після цього створення запису, модель наведена на рисунку 3.4.

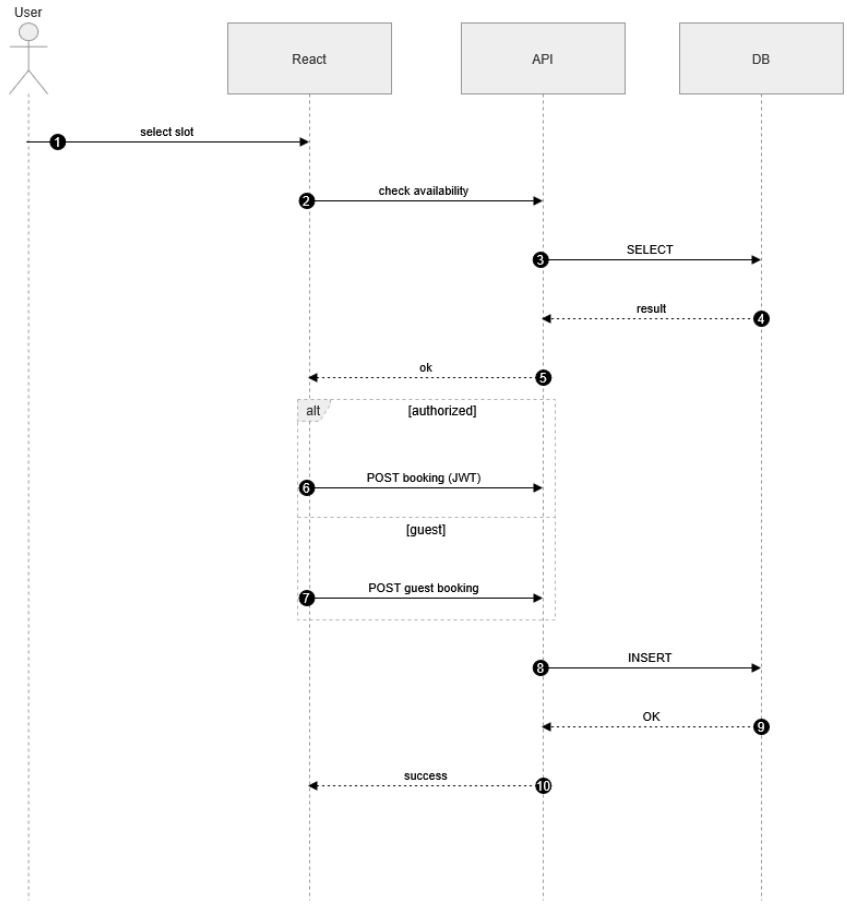

Рисунок 3.4 — Sequence-діаграма «Створення бронювання»

У межах цього сценарію перевірка доступності не може бути формальною. Система повинна враховувати перетин часових інтервалів. Наприклад, якщо вже існує бронювання з 18:00 до 19:30, то нове бронювання з 19:00 до 20:00 є конфліктним, хоча початкові часи не збігаються. Саме тому умова перевірки повинна аналізувати перетин діапазонів часу, а не лише рівність StartTime.

BPMN-модель процесу обробки заявки в CRM показана на рисунку 3.5.

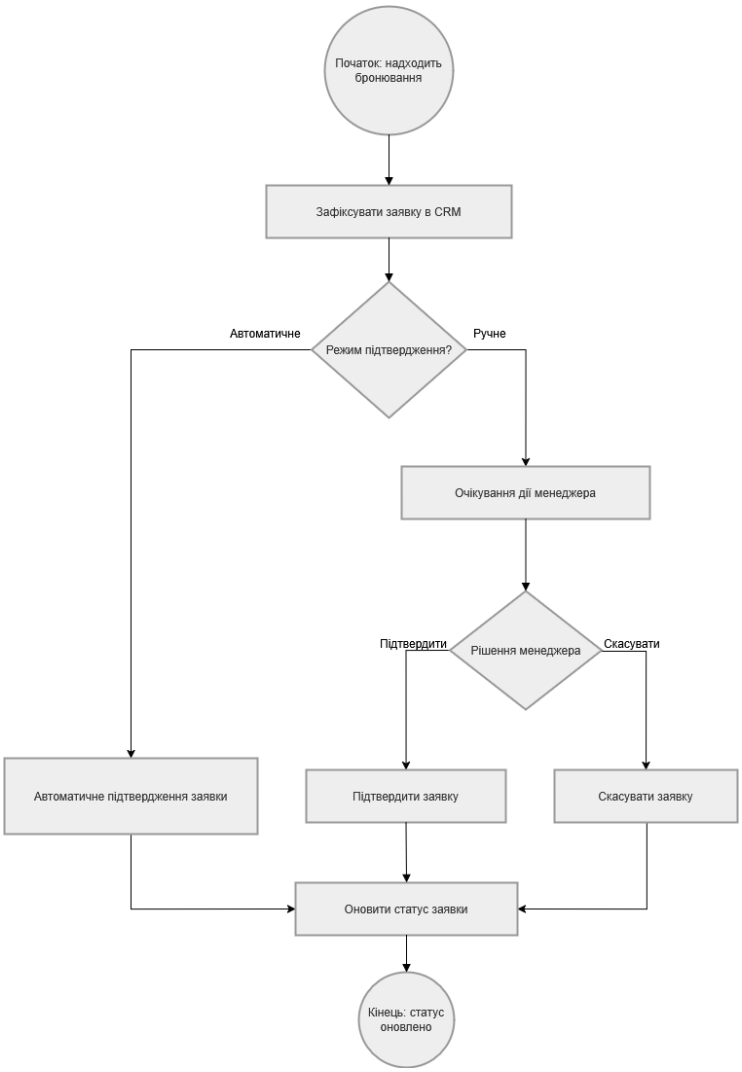

Рисунок 3.5 — BPMN-процес «Обробка заявки в CRM»

BPMN-модель була використана для опису дій менеджера після створення заявки. У реальному процесі менеджер не просто бачить новий запис, а приймає рішення: підтвердити, скасувати або залишити заявку в очікуванні. У системі ці дії повинні бути обмежені правилами статусного циклу. Наприклад, не можна підтвердити бронювання, яке вже було скасоване, і не можна завершити бронювання до настання його кінцевого часу.

Для опису статусів бронювання я використав State Machine-діаграму. Вона краще підходить для цієї задачі, ніж звичайний список статусів, оскільки дозволяє явно показати допустимі переходи. У системі бронювання статус є не просто текстовим полем, а елементом бізнес-логіки, який визначає, які дії дозволені в конкретний момент.

Життєвий цикл бронювання у вигляді State Machine представлено на рисунку 3.6.

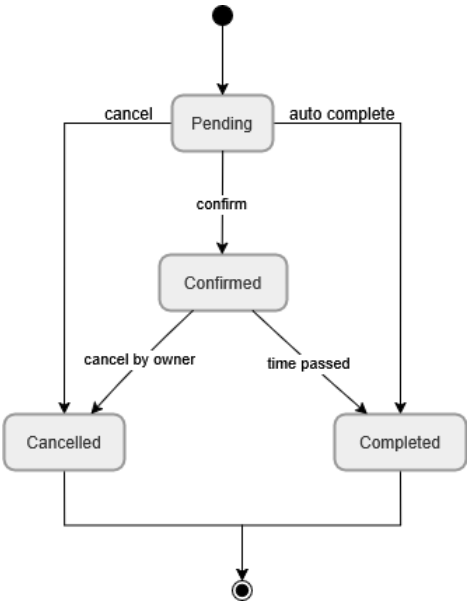

Рисунок 3.6 — State machine «Життєвий цикл бронювання»

Формалізацію статусних переходів я зібрав у таблицю 3.1. Вона показує, з якого стану в який можна перейти, хто ініціює дію та за яких умов вона дозволена.

Таблиця 3.1

Формалізація статусних переходів

Поточний стан	**Цільовий стан**	**Ініціатор**	**Умова**
Pending	Confirmed	Менеджер/власник	Валідний статус, доступ, owner match
Pending	Cancelled	Користувач/менеджер	В межах політики скасувань
Confirmed	Cancelled	Менеджер	Дозволено політикою, до старту/в межах правил
Pending/Confirmed	Completed	Системний процес	EndTime + поріг часу < now (background worker)

Ця таблиця надалі фактично стала основою для реалізації сервісу статусів. У коді такі правила не варто розкидати по різних контролерах, тому їх доцільно винести в окремий сервіс. Це спрощує тестування і дозволяє додавати нові статуси або змінювати політику переходів без переписування всієї логіки бронювання.

Формалізація статусів також зменшує ризик помилок при ручній роботі менеджера. Якщо в CRM є кнопки “Підтвердити” або “Скасувати”, backend все одно повинен перевірити, чи ця дія допустима. Інтерфейс може приховати недоступні кнопки, але остаточна перевірка має залишатися на сервері. Це узгоджується із загальним принципом server-side контролю.

3.3 Проєктування бази даних та вибір технологічного стеку

Модель бази даних я будував з урахуванням того, що предметна область має ресурсно-часову природу. Основна складність полягає в тому, що бронюється не просто майданчик як опис у каталозі, а конкретний фізичний ресурс у конкретному часовому проміжку. Тому на початку проєктування я відмовився від спрощеної моделі, де вся інформація зберігається в одній таблиці “Майданчик”. Такий підхід швидко призвів би до дублювання даних і конфліктів у розкладі.

Основними сутностями стали SportsField та SportsFieldInstance. SportsField зберігає загальну інформацію про спортивний об’єкт: назву, опис, власника, адресу та інші характеристики. SportsFieldInstance описує конкретну фізичну одиницю, яку можна бронювати: окремий корт, поле або зал. Завдяки цьому один спортивний об’єкт може мати кілька інстансів, і кожен із них може мати власну доступність.

Така модель краще відповідає реальній структурі спортивних комплексів. Наприклад, якщо комплекс має чотири тенісні корти, то вони не повинні розглядатися як один ресурс. Один корт може бути зайнятий, другий — вільний, третій — неактивний через ремонт, а четвертий — доступний тільки в певні години. Інстансна модель дозволяє відобразити ці випадки в базі даних без штучних обмежень.

При проєктуванні бази даних я враховував три основні вимоги: цілісність даних, продуктивність запитів і можливість аналітики. Цілісність забезпечується через зовнішні ключі, зв’язки між таблицями та обмеження на рівні моделі. Продуктивність залежить від структури таблиць та індексів. Аналітика потребує

збереження історичних даних і коректного статусного циклу бронювань.

Для зберігання даних було обрано PostgreSQL. Я розглядав також MySQL і Microsoft SQL Server, але PostgreSQL добре підходить для цього проєкту через підтримку транзакцій, надійність, відкритість, якісну роботу з часовими типами даних та можливість масштабування. Для системи бронювання особливо важлива підтримка ACID-транзакцій, оскільки створення бронювання має бути атомарною операцією.

Під час роботи з часовими даними було прийнято рішення зберігати часові мітки в UTC. Це дозволяє уникнути проблем, пов'язаних із часовими поясами та переходом на літній або зимовий час. Навіть якщо на frontend користувач бачить локальний час, у базі даних доцільно зберігати уніфіковане значення. Це спрощує порівняння інтервалів, фільтрацію та аналітичні запити.

Крім того, під час проєктування моделі даних враховувалися вимоги до підтримки транзакційності та конкурентного доступу. Оскільки декілька користувачів можуть одночасно виконувати операції бронювання, структура таблиць і зв'язків повинна забезпечувати коректну перевірку конфліктів та збереження узгодженого стану даних. Це особливо важливо для операцій створення бронювання, які безпосередньо впливають на доступність спортивних ресурсів.

Окремо я враховував можливість подальшого розвитку системи. На першому етапі реалізується базове бронювання, CRM і аналітика, але в майбутньому можуть з'явитися онлайн-оплата, абонементи, бронювання тренерів або інтеграція з календарями. Тому структура бази даних не повинна бути надто жорстко прив'язаною лише до поточної версії функціоналу.

Під час нормалізації даних частина інформації була винесена в окремі таблиці. Наприклад, розклад не зберігається як текстове поле в таблиці майданчика, а описується окремою сутністю SportsFieldSchedules. Такий підхід дозволяє змінювати графік роботи без зміни основних даних про майданчик. Аналогічно типи спорту та ціни винесені в окрему структуру, оскільки один об'єкт може підтримувати кілька видів активностей.

Таблиця Bookings є однією з центральних у системі. Вона зберігає інформацію про користувача, майданчик, інстанс, початок і кінець бронювання, статус та загальну вартість. Саме ця таблиця найчастіше використовується під час перевірки доступності та формування аналітики. Тому для неї потрібно передбачити індекси за полями SportsFieldInstanceId, StartTimeUtc, EndTimeUtc і Status.

Збереження історичних даних було закладене в модель з самого початку. У реальній системі не можна просто видаляти завершені бронювання, оскільки вони потрібні для статистики, фінансового аналізу та вирішення спірних ситуацій. Тому для деяких сутностей було передбачено soft delete. Додатковою перевагою такого підходу є можливість формування довгострокової аналітики без втрати історичних записів. Навіть після деактивації спортивного майданчика або окремого інстансу система зберігає зв'язки між бронюваннями, відгуками та фінансовими показниками. Це дозволяє коректно розраховувати завантаженість ресурсів, аналізувати динаміку попиту та виконувати порівняння показників за різні часові періоди. Це означає, що запис не видаляється фізично з бази даних, а позначається як неактивний або видалений через спеціальне поле, наприклад IsDeleted.

Soft delete особливо корисний для спортивних майданчиків, інстансів і відгуків. Якщо власник видаляє майданчик із публічного доступу, історія бронювань не повинна зникати. Інакше аналітика за попередні періоди стане некоректною. Такий підхід також зручний при аудиті дій та відновленні даних у разі помилки.

Відгуки були прив'язані до завершених бронювань. Це рішення було прийнято для того, щоб користувач не міг залишити оцінку до фактичного отримання послуги. З точки зору моделі даних це дозволяє зв'язати відгук не тільки з майданчиком, але й із конкретною подією бронювання. Такий підхід підвищує достовірність рейтингу.

Основні сутності бази даних наведені в таблиці 3.2.

Таблиця 3.2

Основні сутності БД

Сутність	Призначення	Ключові поля
AspNetUsers	Облікові записи і ролі	Id, Email, Role, FullName
SportsFields	Дані майданчика	Id, Name, OwnerId, IsDeleted
Locations	Локація об’єкта	Id, SportsFieldId, City, Latitude, Longitude
SportsFieldSportTypes	Типи активностей	Id, SportsFieldId, Type, PricePerHour
SportsFieldSchedules	Часові правила	Id, SportsFieldSportTypeId, DayOfWeek, AvailableFrom, AvailableTo
SportsFieldInstances	Фізичні ресурси	Id, SportsFieldId, SportTypeId, DisplayName, IsActive
Bookings	Операції бронювання	Id, UserId, SportsFieldId, SportsFieldInstanceId, StartTimeUtc, EndTimeUtc, Status, TotalPrice
Reviews	Оцінка сервісу	Id, SportsFieldId, BookingId, Rating, Comment, CreatedAt

З таблиці видно, що модель даних побудована навколо процесу бронювання. Користувач створює бронювання для конкретного майданчика та інстансу, а розклад і тип активності визначають, коли і на яких умовах цей ресурс доступний. Така структура дає змогу уникнути зберігання критичних правил у вигляді неструктурованого тексту.

Сутність SportsFieldInstance має ключове значення для коректної роботи системи. Без неї було б складно відрізнити ситуацію, коли один із кортів зайнятий, а інші залишаються доступними. У простішій моделі довелося б або дублювати майданчики, або вводити складну логіку в коді. Інстансна модель переносить цю логіку в структуру даних і робить її більш прозорою.

Під час побудови структури бази даних я намагався зберегти баланс між нормалізацією та продуктивністю. Надмірна нормалізація могла б призвести до великої кількості JOIN-запитів, а надмірна денормалізація — до дублювання даних. Тому остаточна структура була сформована після аналізу основних сценаріїв: пошуку слотів, створення бронювання, перегляду CRM-таблиці та формування аналітики.

Повна структура зв’язків між таблицями показана на ER-діаграмі (рисунок В.1), а діаграма основних сутностей, які є найважливішими в системі, показана на рисунку 3.7.

Рисунок 3.7 — ER-діаграма бази даних (основні суності)

ER-діаграма дозволила перевірити зв’язки між сутностями до початку реалізації коду. На цьому етапі було простіше побачити, де може виникнути дублювання або надмірна залежність між таблицями. Наприклад, зв’язок між Bookings і SportsFieldInstances показує, що бронювання повинно стосуватися конкретного фізичного ресурсу, а не лише загальної картки майданчика.

Після проєктування бази даних я перейшов до вибору технологічного стеку. Для backend-частини було обрано ASP.NET Core 8. Цей фреймворк добре підходить для побудови Web API, має вбудований контейнер Dependency Injection, підтримує middleware-конвеєр, асинхронну обробку запитів і добре інтегрується з Entity Framework Core.

Асинхронна модель у .NET має практичне значення для веб-застосунків. Коли API виконує запит до бази даних, потік не повинен блокуватися на час очікування відповіді. Використання async/await дозволяє звільнити потік thread pool для обробки інших запитів, поки операція введення-виведення не завершиться. Для системи бронювання це важливо, оскільки одночасно кілька користувачів можуть перевіряти доступність слотів або створювати заявки.

Для клієнтської частини було обрано React. Він дозволяє створювати SPA-застосунки, де перехід між сторінками відбувається без повного перезавантаження браузера. Для CRM-модуля було створено окремий React-застосунок, щоб не змішувати публічний інтерфейс клієнта з адміністративними функціями.

Для доступу до PostgreSQL використовувався Entity Framework Core. Його перевага полягає в тому, що він дозволяє працювати з базою даних через C#-моделі та LINQ-запити. Міграції EF Core спростили еволюцію структури БД під час розробки. При цьому в складних місцях потрібно враховувати, який SQL-запит буде згенерований, оскільки неефективний LINQ може призвести до повільного запиту на рівні PostgreSQL.

Для автентифікації було обрано JWT. При проєктуванні контуру безпеки фактично був вибір між cookie-based сесіями та stateless-токенами. JWT зручний для API-архітектури, оскільки дозволяє передавати ідентифікатор користувача, роль та інші claims у токені. Backend може перевіряти ці дані при кожному запиті без збереження серверної сесії.

Також у стек увійшли AutoMapper, FluentValidation, Serilog і Swagger. AutoMapper використовується для перетворення між доменними моделями та DTO, FluentValidation — для опису правил валідації вхідних даних, Serilog — для структурованого логування, Swagger — для документування API та перевірки endpoint-ів під час розробки.

Обраний стек ASP.NET Core 8, React, PostgreSQL, Entity Framework Core та JWT дозволяє реалізувати багатошарову архітектуру, підтримувати транзакційність під час бронювання та розвивати систему без суттєвої перебудови її ядра.

3.4 Висновки по розділу

У третьому розділі було виконано проєктування архітектури системи, бізнес-процесів, статусної моделі бронювання, структури бази даних і

технологічного стеку. На цьому етапі було сформовано технічну основу для подальшої реалізації програмного продукту.

Багатошарова архітектура дозволила розділити відповідальність між клієнтською частиною, API, сервісним шаром, доступом до даних та інфраструктурними компонентами. Таке розділення дає можливість тестувати бізнес-логіку окремо від контролерів і бази даних, а також зменшує ризик неконтрольованих змін у коді.

Моделювання бізнес-процесів допомогло формалізувати основні сценарії: створення бронювання, роботу менеджера в CRM та життєвий цикл заявки. State Machine-модель стала основою для визначення допустимих статусних переходів, що надалі спрощує реалізацію та тестування логіки бронювання.

Проєктування бази даних було виконано з урахуванням інстансної моделі спортивних ресурсів. Окреме виділення майданчиків, фізичних інстансів, розкладів, бронювань і відгуків дозволило точніше відобразити предметну область і зменшити ризик конфліктів доступності.

Обраний технологічний стек забезпечує достатній баланс між швидкістю розробки, надійністю і можливістю подальшого розвитку. ASP.NET Core 8 підходить для побудови Web API, PostgreSQL забезпечує надійне збереження даних і транзакційність, React дозволяє створити зручний інтерфейс, а JWT забезпечує stateless-автентифікацію для клієнтських застосунків.

Таким чином, матеріали третього розділу створили основу для практичної реалізації системи. У наступному розділі описано безпосередню розробку backend-частини, клієнтського застосунку, CRM-модуля та алгоритмів, які забезпечують коректну роботу бронювання.

РОЗДІЛ 4. РЕАЛІЗАЦІЯ ПРОЕКТУ

4.1 Опис розробки програмного коду (backend)

Реалізацію серверної частини я виконував на ASP.NET Core 8 у форматі Web API. Такий підхід був обраний тому, що система має два окремі клієнтські застосунки: публічний інтерфейс для користувачів і CRM-модуль для менеджерів та власників. Обидва frontend-застосунки працюють із сервером через HTTP API, тому backend повинен був стати єдиною точкою доступу до бізнес-логіки, бази даних і механізмів авторизації.

Під час реалізації я дотримувався шарової структури, яка була спроєктована в третьому розділі. Контролери не містять складної бізнес-логіки. Їхнє завдання полягає в прийманні HTTP-запиту, перевірці моделі, отриманні ідентифікатора користувача з контексту авторизації та передачі керування у відповідний сервіс. Такий підхід дозволяє уникнути перевантаження контролерів і робить код більш придатним до тестування.

Основна бізнес-логіка була винесена в сервісний шар. Саме там реалізовано перевірку доступності слотів, створення бронювань, контроль статусних переходів, роботу з відгуками та аналітичні обчислення. Репозиторії відповідають за доступ до бази даних через Entity Framework Core. Завдяки цьому контролери, сервіси і доступ до даних мають чітко розділену відповідальність.

Під час реалізації серверної частини я намагався дотримуватися принципів SOLID. Найбільш помітно це проявилося в розділенні відповідальності між класами. Наприклад, сервіс бронювання не повинен відповідати за генерацію JWT-токенів, а сервіс аналітики — за зміну статусів заявок. Така організація коду зменшує зв’язаність між компонентами та спрощує внесення змін.

Dependency Injection використовувався як основний механізм керування залежностями. Сервіси, репозиторії, логери та допоміжні компоненти реєструються в контейнері залежностей ASP.NET Core і передаються через конструктори. Це дозволяє не створювати залежності вручну через new, а також

полегшує написання unit-тестів, оскільки реальні реалізації можна замінити mock-об’єктами.

Для автентифікації було реалізовано JWT-токени. При вході користувач отримує access token, який далі передається в заголовку Authorization. На сервері токен перевіряється middleware-компонентом, після чого з нього можна отримати claims: ідентифікатор користувача, роль та інші службові дані. Це дало змогу не довіряти userId, який потенційно може прийти з frontend, а брати ідентичність користувача з перевіреного токена [24].

При проєктуванні контуру безпеки фактично стояв вибір між cookie-based сесіями та stateless JWT-підходом. Для Web API з окремими frontend-застосунками JWT є зручним рішенням, оскільки серверу не потрібно зберігати стан сесії для кожного користувача. Кожен запит містить токен, а backend може самостійно перевірити його валідність і застосувати рольові обмеження.

Найбільш критичним модулем backend-частини став модуль бронювання. Його реалізація не обмежувалася простим додаванням запису в таблицю Bookings. Перед створенням бронювання сервер перевіряє, чи існує майданчик, чи активний потрібний інстанс, чи відповідає час розкладу, чи немає перетину з уже існуючими активними бронюваннями. Лише після цього запис може бути збережений у базі даних.

Для захисту від race condition була реалізована двофазна логіка. Спочатку користувач може отримати попередню інформацію про доступність слоту, але остаточна перевірка виконується безпосередньо перед створенням бронювання. Перевірка доступності та створення запису розглядаються як одна критична операція, яка повинна виконуватися в межах транзакційної логіки. Якщо під час перевірки виявляється конфлікт, запис не створюється.

Транзакційний підхід у цьому модулі важливий через природу самої задачі. Якщо два користувачі одночасно намагаються забронювати один і той самий інстанс у той самий часовий проміжок, система не повинна дозволити обом операціям завершитися успішно. Саме тому вся логіка створення бронювання була реалізована на сервері, а не винесена на frontend.

Окремо був реалізований сервіс керування статусами бронювання. Він перевіряє, чи дозволений перехід між поточним і новим станом заявки, хто виконує дію та чи відповідає дія часовим обмеженням. Наприклад, менеджер може підтвердити заявку у статусі Pending, але не повинен підтверджувати вже скасоване або завершене бронювання. Такі правила були винесені в окремий сервіс, щоб не дублювати їх у різних endpoint-ах.

Для автоматичного завершення прострочених бронювань я реалізував Hosted Service. Це фоновий сервіс ASP.NET Core, який періодично перевіряє бронювання, час завершення яких уже минув, і переводить їх у статус Completed. Такий механізм зменшує ручне навантаження на менеджера і підтримує актуальний стан даних без потреби постійного втручання адміністратора.

Операції взаємодії з базою даних реалізовані з використанням async/await. У ASP.NET Core це має практичне значення, оскільки запити до бази даних є I/O-bound операціями. Коли сервер очікує відповідь від PostgreSQL, потік thread pool не повинен просто блокуватися. Асинхронна модель дозволяє звільнити потік для обробки інших запитів і повернутися до виконання після завершення операції введення-виведення. Це покращує масштабованість API при одночасній роботі кількох користувачів.

Для обміну даними між frontend і backend використовувалися DTO-моделі. Вони не збігаються повністю з доменними сутностями бази даних. Це зроблено свідомо, тому що клієнт не повинен отримувати або надсилати всі внутрішні поля моделі. Наприклад, службові поля, статуси, ідентифікатори власника або ознаки soft delete не повинні змінюватися напряму з клієнтської частини. DTO дозволяють контролювати, які саме дані входять і виходять через API.

Для валідації вхідних даних використовувався підхід із перевіркою DTO до виконання бізнес-операції. Це дозволяє відхиляти некоректні запити ще до звернення до бази даних. Наприклад, система повинна перевіряти коректність email, обов’язковість контактних даних при гостьовому бронюванні, допустиму тривалість бронювання та валідність часових меж.

Для контролю роботи системи було реалізовано логування. У журнали записуються події входу користувача, помилки авторизації, невдалі бізнес-операції, зміни статусів бронювання та інші важливі дії. Для такого проєкту логування потрібне не тільки для пошуку помилок під час розробки, а й для майбутньої підтримки. Якщо користувач повідомляє про проблему, журнали дозволяють швидше зрозуміти, який endpoint викликався і на якому етапі сталася помилка.

Основні ендпоінти API наведено в таблиці 4.1. Таблиця містить маршрути, які забезпечують автентифікацію, створення бронювань, роботу менеджера, відгуки та аналітичні показники.

Таблиця 4.1

Основні API endpoints

Метод	**Endpoint**	**Призначення**
POST	/api/Account/login	Вхід користувача
POST	/api/Account/register	Реєстрація
GET	/api/Account/me	Поточний профіль
POST	/api/Booking/bookings	Створення бронювання
POST	/api/Booking/bookings/guest	Гостьове бронювання
POST	/api/Booking/check-availability	Перевірка доступності слоту
POST	/api/Booking/manager/bookings/{bookingId}/confirm	Підтвердження менеджером
POST	/api/Booking/manager/bookings/{bookingId}/cancel	Скасування менеджером
POST	/api/reviews	Додавання відгуку
GET	/api/reviews/field/{sportsFieldId}	Отримання відгуків поля
GET	/api/Analytics/occupancy/{sportsFieldId}	Завантаженість
GET	/api/Analytics/cancellations/{sportsFieldId}	Частка скасувань
GET	/api/Analytics/rating/{sportsFieldId}	Рейтинг
GET	/api/Analytics/peak-hours/{sportsFieldId}	Пікові години

При проєктуванні маршрутів я орієнтувався на REST-підхід. Для операцій читання використовуються GET-запити, а для створення або зміни стану — POST-запити. Окремі маршрути для менеджера винесені в підгрупу manager, оскільки ці дії мають інші правила авторизації, ніж звичайні клієнтські операції.

Послідовність server-side обробки бронювання показана на рисунку 4.1.

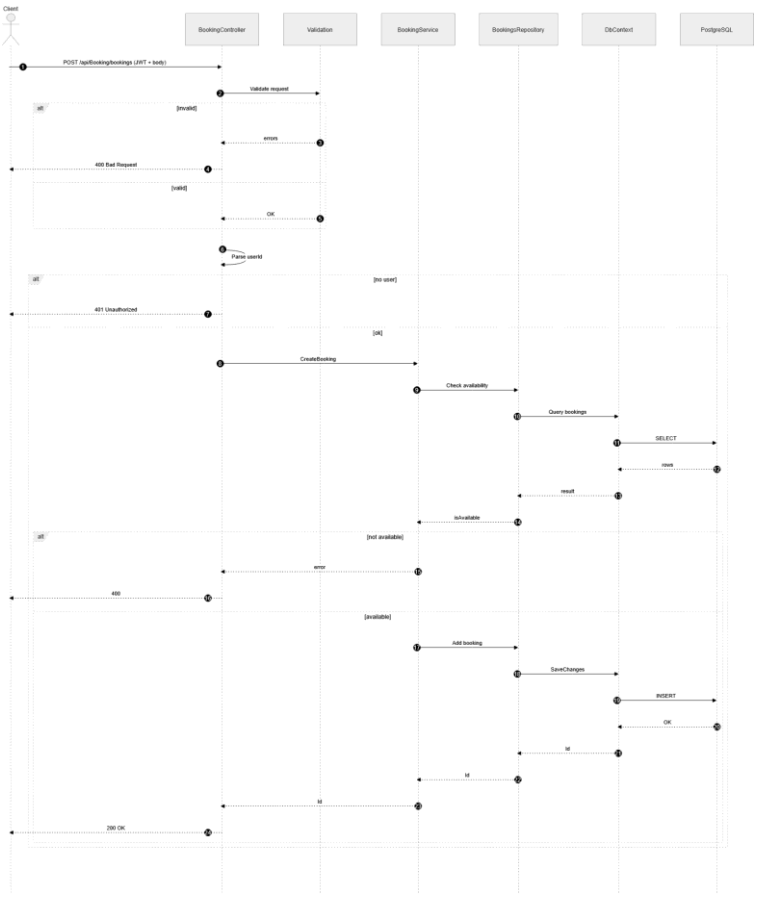

Рисунок 4.1 — Послідовність server-side обробки бронювання

Схема роботи JWT-автентифікації наведена на рисунку 4.2.

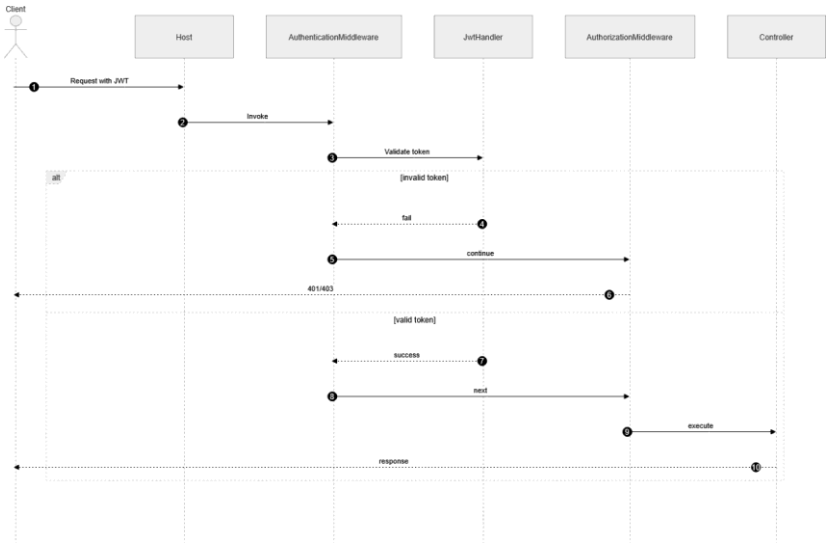

Рисунок 4.2 — Схема авторизаційного контуру JWT

4.2 Розробка клієнтської частини (frontend)

Клієнтську частину я реалізував як SPA-застосунок на React. Такий формат був обраний тому, що система передбачає активну взаємодію користувача з інтерфейсом: пошук майданчиків, перегляд карти, вибір часу, відкриття модальних вікон і підтвердження бронювання. Повне перезавантаження сторінки при кожній дії погіршувало б користувацький досвід, тому SPA-підхід є більш зручним.

Для маршрутизації використовувався React Router. Він дозволяє організувати переходи між сторінками без перезавантаження браузера. Для збереження частини стану застосунку використовувалися Context API та локальні стани компонентів через useState. Такий підхід достатній для цього проєкту, оскільки стан не є надто складним і не потребує окремого глобального state manager.

Під час розробки інтерфейсу я орієнтувався на те, щоб користувач міг створити бронювання за мінімальну кількість кроків. Процес був реалізований у два етапи. На першому етапі користувач обирає майданчик, дату, час, тривалість та інші параметри. Після цього frontend звертається до backend для перевірки доступності. На другому етапі користувач бачить узагальнену інформацію і підтверджує бронювання.

Двокрокова логіка була обрана не випадково. Якщо одразу створювати бронювання без попереднього підтвердження, користувач може помилитися в часі або майданчику. Якщо ж зробити процес занадто довгим, користувач може не завершити його. Тому модальні вікна були використані як компроміс: вони не перевантажують сторінку і дозволяють поетапно провести користувача через процес бронювання.

Для відображення майданчиків на карті було інтегровано Leaflet. Карта потрібна для того, щоб користувач міг оцінити розташування майданчика, а не лише прочитати адресу. Це особливо зручно в міських умовах, коли вибір спортивного об’єкта часто залежить від відстані, транспорту або району.

Взаємодія з backend була винесена в окремий API-сервіс. Це дозволило централізовано обробляти HTTP-запити, додавати JWT-токен до заголовків і перехоплювати типові помилки. Якби запити були розкидані по компонентах, підтримувати код було б складніше. Окремий API-шар на frontend робить структуру застосунку більш зрозумілою.

Помилки серверної перевірки також обробляються на клієнтській стороні. Якщо backend повертає повідомлення про зайнятий слот, відсутність прав або некоректні дані, користувач отримує зрозуміле повідомлення в інтерфейсі. Це важливо, бо користувач не повинен бачити технічні тексти помилок або HTTP-коди без пояснення.

Адаптивність інтерфейсу була важливою вимогою, оскільки бронювання спортивних майданчиків часто виконується зі смартфона. Користувач може шукати майданчик уже дорогою з роботи або під час листування з друзями. Тому сторінки, картки майданчиків, модальні вікна та кнопки були адаптовані під мобільні екрани.

Основні екрани клієнтської частини показані на рисунках 4.3, 4.4, 4.5, 4.6.

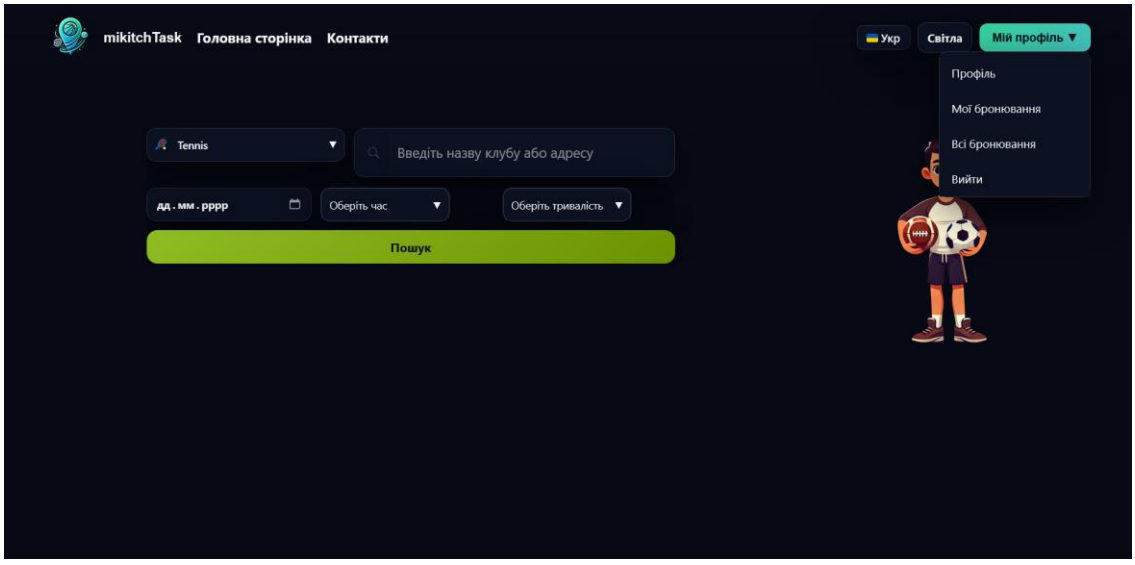

Рисунок 4.3 — Головна сторінка клієнтського застосунку

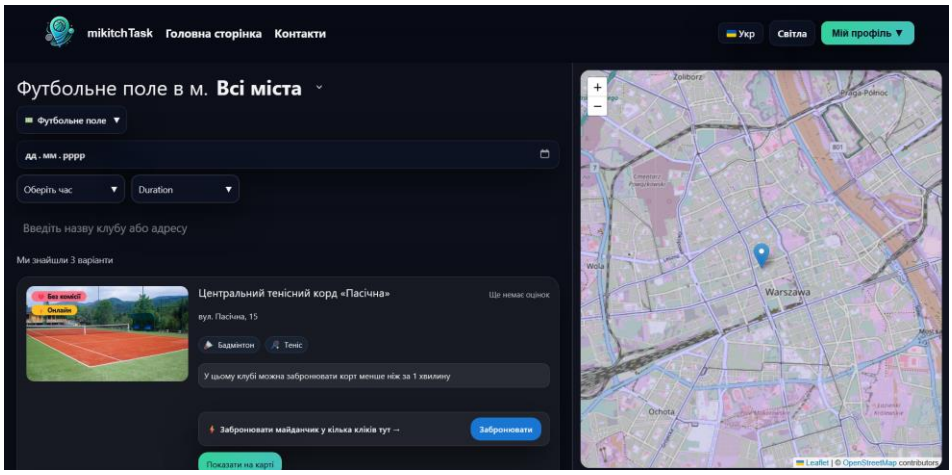

Рисунок 4.4 — Сторінка результатів пошуку з картою

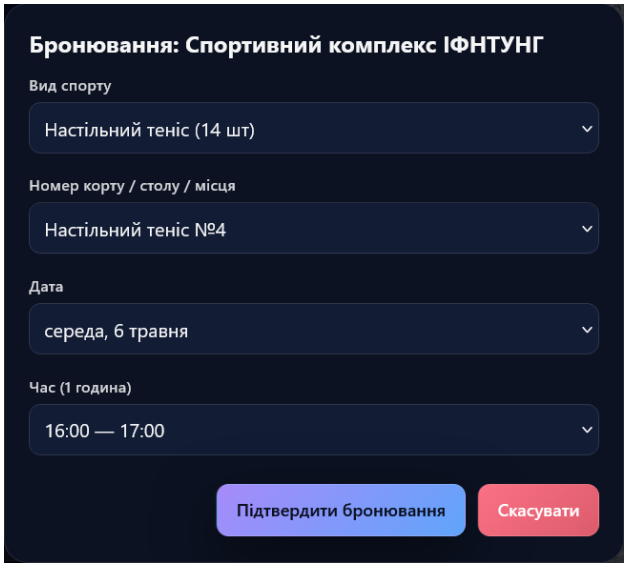

Рисунок 4.5 — Модальне вікно першого кроку бронювання

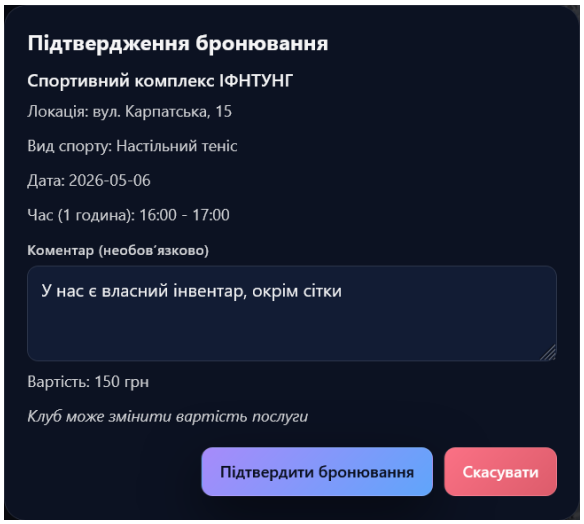

Рисунок 4.6 — Модальне вікно другого кроку бронювання

Реалізована клієнтська частина не приймає критичних рішень щодо бронювання самостійно. Вона лише збирає дані, показує користувачу доступну інформацію і передає запит на backend. Такий підхід відповідає загальній архітектурі системи, де сервер залишається єдиним джерелом істини.

4.3 Реалізація CRM-модуля

CRM-модуль був реалізований як окремий React-застосунок. Я прийняв таке рішення, оскільки публічний інтерфейс і внутрішній інтерфейс менеджера мають різні задачі. Клієнту потрібен швидкий пошук і бронювання, а менеджеру — таблиця заявок, фільтрація, зміна статусів, редагування параметрів майданчика та перегляд аналітики. Якщо об’єднати все в одному інтерфейсі, система стала б перевантаженою.

Головний екран CRM містить таблицю бронювань. Для менеджера важливо швидко бачити час, контактні дані, тип активності, статус і суму. З цієї таблиці можна виконати основні дії: підтвердити або скасувати заявку. При цьому frontend лише ініціює дію, а backend перевіряє, чи дозволений такий статусний перехід.

Менеджер також може редагувати параметри майданчика, розклад, типи спорту та інстанси. Усі такі зміни проходять через серверну перевірку. Це потрібно для того, щоб зміна розкладу або деактивація інстансу не створила конфлікт із уже існуючими бронюваннями. Наприклад, не можна просто видалити ресурс, якщо в історії є записи, пов’язані з ним. Для таких випадків краще використовувати деактивацію або soft delete.

Аналітичний модуль у CRM був реалізований для власників і менеджерів, які хочуть бачити не тільки поточні заявки, але й загальну картину роботи об’єкта. У системі відображаються завантаженість, частка скасувань, рейтинг, пікові години та інші показники. Ці дані допомагають оцінити, які години є найбільш популярними, а які ресурси використовуються недостатньо.

CRM-модуль був організований як єдина точка доступу до управлінських функцій. Усі ключові дії виконуються в одному інтерфейсі, а backend забезпечує

перевірку прав і цілісність даних.

Екрани CRM-модуля наведено на рисунках 4.7, 4.8, 4.9, 4.10.

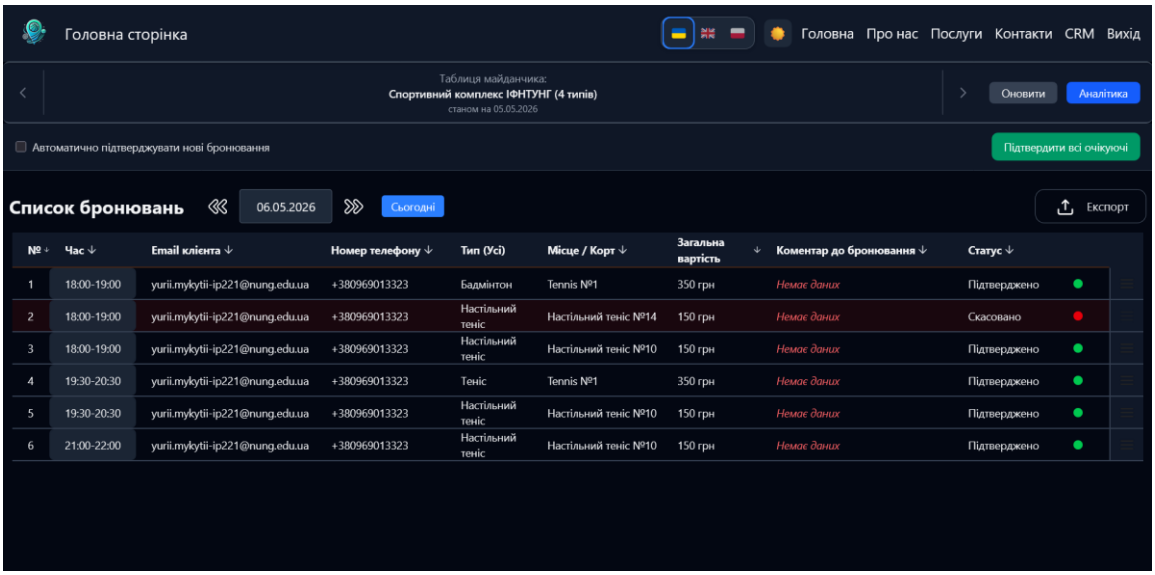

Рисунок 4.7 — Головний екран CRM (таблиця бронювань)

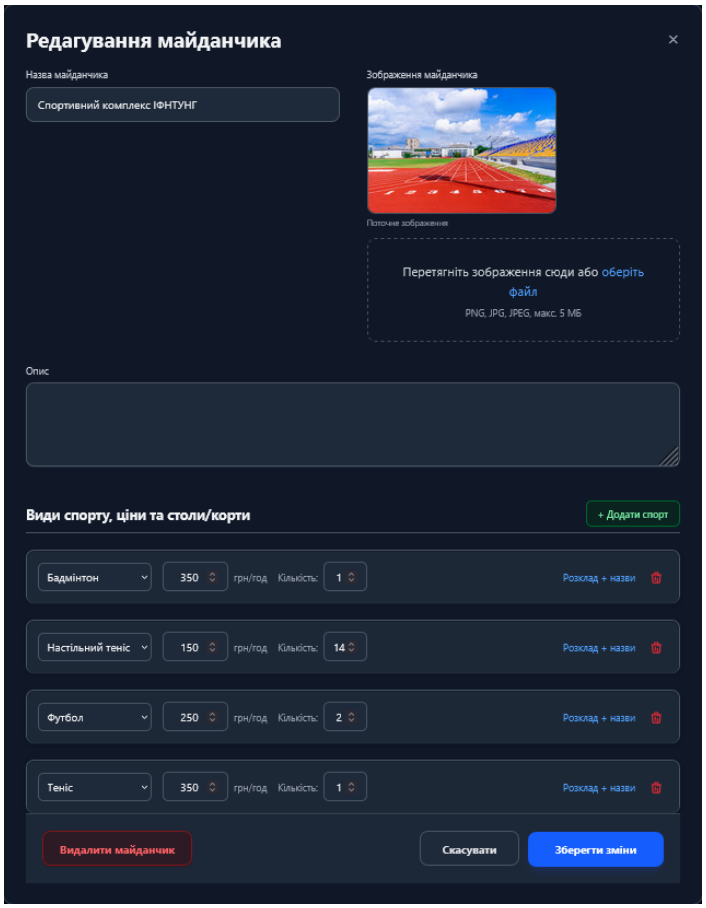

Рисунок 4.8 — Форма редагування параметрів майданчика

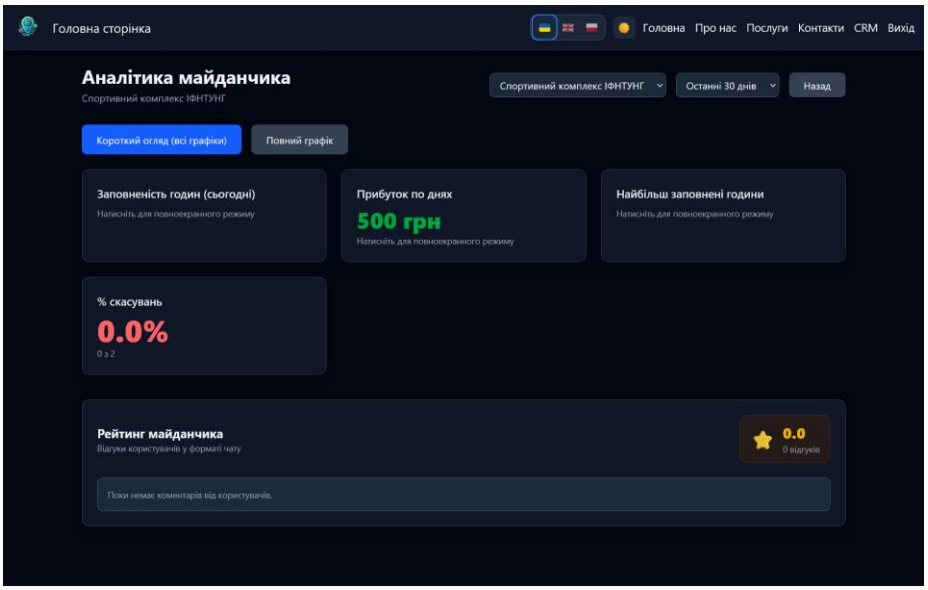

Рисунок 4.9 — Аналітичний модуль

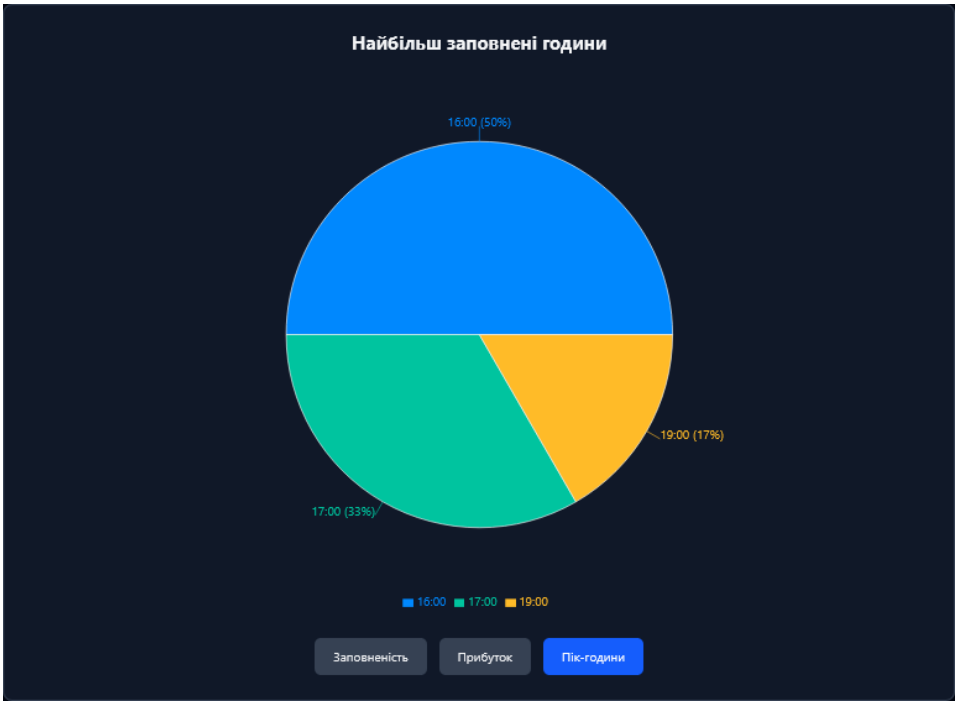

Рисунок 4.10 — Графік найбільш заповнених годин

Реалізований CRM-модуль дозволяє поєднати оперативну роботу із заявками та аналіз діяльності спортивного об’єкта. Це робить систему корисною не тільки для клієнта, який створює бронювання, але й для персоналу, який щоденно керує майданчиками.

4.4 Використані алгоритми та структури даних

Найважливішим алгоритмом у системі є перевірка доступності слоту. На перший погляд може здатися, що достатньо перевірити, чи існує бронювання з таким самим часом початку. Але в реальній системі цього недостатньо. Потрібно перевіряти перетин часових інтервалів, активність інстансу, статуси вже існуючих бронювань, відповідність розкладу та роль користувача.

Алгоритм перевірки доступності працює у два етапи. Перший етап використовується для попереднього відображення доступності на frontend. Другий етап виконується на backend безпосередньо перед створенням бронювання. Саме другий етап є критичним, оскільки він працює з актуальним станом бази даних.

Перетин часових інтервалів перевіряється за логікою, де новий інтервал конфліктує з існуючим, якщо його початок менший за кінець існуючого бронювання, а його кінець більший за початок існуючого бронювання. Це дозволяє виявити не тільки повний збіг часу, але й часткові перетини.

Під час реалізації алгоритму я враховував також статус бронювання. Наприклад, скасоване бронювання не повинно блокувати слот, а підтверджене або очікуване — повинно враховуватися при перевірці. Це дозволяє уникнути ситуації, коли система помилково вважає ресурс зайнятим або, навпаки, дозволяє конфліктний запис.

Алгоритм зміни статусів бронювання був винесений в окремий сервіс. Для кожного переходу визначено початковий стан, цільовий стан, роль користувача і додаткові умови. Такий підхід фактично реалізує спрощену State Machine у коді. Завдяки цьому логіка переходів не дублюється в контролерах і може бути протестована окремо.

Фоновий алгоритм завершення бронювань працює через Hosted Service. Він періодично перевіряє записи, у яких EndTimeUtc уже менший за поточний час, і переводить їх у статус Completed. Це дозволяє підтримувати актуальний стан бронювань без ручного втручання менеджера.

Для аналітичного модуля використовуються агрегуючі запити. Вони групують бронювання за датами, годинами, статусами або майданчиками. Наприклад, для визначення пікових годин потрібно згрупувати записи за годиною початку та підрахувати кількість бронювань. Для рейтингу використовуються агрегати по таблиці Reviews.

Щоб зменшити навантаження на базу даних, результати окремих аналітичних запитів кешуються за допомогою IMemoryCache. Аналітичні показники не потребують оновлення щосекунди, тому кешування на 10–15 хвилин є прийнятним компромісом між актуальністю і продуктивністю.

Структури даних у системі відповідають ER-моделі, описаній у третьому розділі. Для пошуку і фільтрації активно використовуються LINQ-запити, які Entity Framework Core перетворює на SQL. Під час написання таких запитів потрібно враховувати, що не кожен зручний LINQ-вираз генерує ефективний SQL, тому складні вибірки перевірялися окремо.

Окремо під час реалізації алгоритмів враховувалися вимоги до підтримуваності програмного коду. Логіка перевірки доступності слотів, зміни статусів та аналітичних розрахунків була розділена між окремими сервісами відповідно до принципу Single Responsibility Principle. Це дозволяє змінювати або розширювати окремі алгоритми без впливу на інші компоненти системи. Крім того, такий підхід значно спрощує написання unit-тестів та подальший супровід програмного забезпечення.

Усі часові дані зберігаються в UTC. Це спрощує порівняння інтервалів і зменшує ризик помилок, пов’язаних із часовими поясами. На frontend час може відображатися у локальному форматі, але backend і база даних працюють із єдиним часовим стандартом.

Під час тестування система перевірялася на наборі даних із понад 500 бронюваннями. Суттєвого зниження продуктивності при типових сценаріях не спостерігалося. Це підтвердило, що обрана структура даних і алгоритми достатні для навчального прототипу та можуть бути розширені в майбутньому.

4.5 Висновки по розділу

У четвертому розділі було описано реалізацію backend-частини, клієнтського SPA-застосунку, CRM-модуля та основних алгоритмів системи. Реалізація виконувалася відповідно до архітектурних рішень, сформованих у попередньому розділі.

Серверна частина на ASP.NET Core 8 забезпечує автентифікацію, авторизацію, перевірку доступності слотів, створення бронювань, статусні переходи, роботу з відгуками та аналітику. Основна бізнес-логіка винесена в сервісний шар, що спрощує тестування та підтримку коду.

Клієнтська частина на React реалізує зручний процес пошуку та бронювання майданчиків. Вона не приймає критичних рішень самостійно, а працює з backend через API. Це дозволяє зберегти server-side контроль і уникнути помилок, пов’язаних із застарілими даними на клієнті.

CRM-модуль забезпечує менеджерам і власникам інструменти для керування заявками, редагування параметрів майданчиків і перегляду аналітики. Завдяки цьому система охоплює не тільки клієнтський сценарій бронювання, але й внутрішню операційну роботу спортивного об’єкта.

Реалізовані алгоритми перевірки доступності, контролю статусів, фонового завершення бронювань і аналітичної агрегації забезпечують коректну роботу основних бізнес-процесів. Практична реалізація підтвердила працездатність архітектурних рішень, прийнятих на етапі проєктування. Розділення відповідальності між шарами, використання серверного контролю та формалізованих бізнес-правил дозволили побудувати систему, яка залишається зрозумілою для подальшого розвитку та підтримки. Крім того, сформована структура проєкту створює основу для інтеграції додаткових модулів без суттєвої зміни існуючого коду. Наступним етапом стало тестування системи, результати якого наведено в п’ятому розділі.

РОЗДІЛ 5. ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ

5.1 Стратегії та методи тестування

Тестування розробленої системи я організував за багаторівневим підходом. Для такого проєкту недостатньо перевірити лише те, що сторінки відкриваються, а кнопки виконують певні дії. Система бронювання працює з часовими інтервалами, ролями користувачів, статусами заявок, транзакційною логікою та аналітичними розрахунками. Тому тестування повинно було охоплювати не тільки інтерфейс, але й backend-логіку, API-контракти, авторизацію, негативні сценарії та поведінку системи при одночасних запитах.

У процесі перевірки я використав три основні рівні тестування: модульне тестування, інтеграційне тестування та ручне сценарне тестування. Додатково проводилася перевірка навантаження на найбільш важливі API-методи та тестування стійкості системи до помилкових запитів. Такий підхід дав змогу перевірити систему з різних боків: окремі сервіси, взаємодію між компонентами та повний користувацький сценарій.

Під час планування тестування я окремо виділив позитивні та негативні сценарії. Позитивні сценарії показують, що система правильно працює у стандартних умовах: користувач вводить коректні дані, має потрібні права, обирає доступний слот і успішно створює бронювання. Негативні сценарії потрібні для перевірки того, як система реагує на помилки: відсутність JWT-токена, невалідний статусний перехід, спробу доступу до чужого майданчика або бронювання зайнятого часу.

Модульне тестування охоплювало найважливіші частини backend-логіки. Основна увага була зосереджена на сервісі бронювання, перевірці статусних переходів, правилах доступності слотів, роботі з відгуками та аналітичних обчисленнях. Саме ці частини системи є найбільш критичними, оскільки помилка в них може призвести до подвійного бронювання, некоректного статусу заявки або неправильних KPI в аналітичному модулі.

Для модульного тестування я використовував xUnit. Цей фреймворк добре підходить для .NET-проєктів і дозволяє описувати тестові кейси у вигляді окремих методів. Для ізоляції залежностей застосовувалася бібліотека Moq. Вона дозволяє підміняти реальні залежності тестованого сервісу mock-об’єктами. Наприклад, під час тестування сервісу бронювання не обов’язково звертатися до реальної бази даних, якщо можна змоделювати відповідь репозиторію.

Під час написання unit-тестів я дотримувався структури AAA: Arrange, Act, Assert. На етапі Arrange готуються вхідні дані та mock-залежності. На етапі Act виконується метод, який тестується. На етапі Assert перевіряється очікуваний результат. Такий підхід робить тести зрозумілими і дозволяє швидко побачити, яка саме умова перевіряється.

Окремо перевірялися граничні випадки. Наприклад, тестувалися перетини часових інтервалів, спроби бронювання неактивного інстансу, створення заявки без ідентифікатора користувача, невалідні переходи зі статусу Cancelled або Completed, а також запити без авторизації. Такі сценарії важливі, бо в реальній експлуатації користувачі не завжди виконують дії в очікуваному порядку.

Сценарії конкурентного доступу також були винесені в окрему групу перевірок. У системах бронювання це один із найризикованіших випадків. Якщо два користувачі одночасно намагаються зайняти один і той самий слот, backend повинен дозволити лише одну операцію. Для цього перевірялася логіка повторної server-side перевірки перед створенням бронювання та поведінка системи при наявності вже існуючого активного запису.

Інтеграційне тестування було спрямоване на перевірку взаємодії між компонентами системи. Якщо unit-тест перевіряє окремий сервіс із заміненими залежностями, то інтеграційний тест дозволяє переконатися, що API, сервісний шар, репозиторії та база даних працюють узгоджено. Для цього використовувалися Postman-запити та інтеграційні тести в .NET.

Для перевірки роботи з реальною базою даних я використовував підхід із Testcontainers. Він дозволяє запускати тестове середовище з базою даних окремо від основної БД. Такий підхід корисний, оскільки дає змогу перевірити не тільки

C#-код, але й фактичну роботу Entity Framework Core, міграцій, SQL-запитів, зовнішніх ключів і обмежень на рівні бази даних.

Під час інтеграційного тестування було перевірено HTTP-контракти між frontend, CRM-модулем і backend. Особливо важливо було переконатися, що API повертає коректні статуси: 200 або 201 для успішних операцій, 400 для невалідних даних, 401 для відсутньої авторизації, 403 для заборонених дій, 404 для неіснуючих ресурсів. Це потрібно не лише для backend, але й для frontend, який на основі цих відповідей показує повідомлення користувачу.

Ручне сценарне тестування проводилося за ролями користувачів: гість, авторизований клієнт, менеджер і власник. Для кожної ролі перевірявся власний набір дій. Гість створював бронювання без реєстрації, авторизований клієнт переглядав історію та залишав відгук, менеджер підтверджував і скасовував заявки, власник переглядав аналітику та редагував параметри майданчика.

Під час ручного тестування я перевіряв повний цикл роботи системи: пошук майданчика, вибір часу, перевірку доступності, створення бронювання, підтвердження в CRM, завершення бронювання та перегляд результатів в аналітиці. Такий підхід дозволив перевірити не тільки окремі endpoint-и, а й логіку всієї системи як єдиного продукту.

Окремий блок перевірок стосувався адаптивності інтерфейсу. Система тестувалася у браузерах Chrome, Firefox та Edge, а також на екранах різного розміру. Перевірялися картки майданчиків, модальні вікна бронювання, карта, таблиці CRM та аналітичні графіки. У процесі тестування було виявлено та виправлено окремі проблеми з відображенням модальних вікон і карти на мобільних пристроях.

Для додаткової перевірки навантаження я імітував одночасну роботу 15–20 користувачів за допомогою Apache JMeter. Основний акцент був зроблений на endpoint-и перевірки доступності та створення бронювання, оскільки саме вони є найбільш критичними для продуктивності. Результати показали, що при короткочасному навантаженні час відповіді API залишався в межах приблизно 300–600 мс.

Також перевірялася стійкість системи до помилок. Я надсилав запити з невалідними даними, пробував звертатися до неіснуючих ресурсів, виконував операції без JWT-токена або з некоректним контекстом користувача. Такі перевірки потрібні для того, щоб система не завершувала роботу з необробленими винятками і не відкривала доступ до дій, які користувач не має права виконувати.

Під час тестування я вів журнал виявлених недоліків. У ньому фіксувалися опис помилки, умови відтворення, очікувана поведінка, фактичний результат і спосіб виправлення. Такий підхід допоміг не втрачати дрібні помилки та перевіряти, чи не з'являються вони повторно після внесення змін у код.

5.2 Результати тестування системи

Після завершення основної реалізації мною було проведено комплексну перевірку системи. Результати тестування використовувалися для оцінки того, наскільки реалізований програмний продукт відповідає функціональним і нефункціональним вимогам, сформульованим у другому розділі. Основний акцент був зроблений на backend-логіці, оскільки саме вона відповідає за цілісність даних, статусні переходи та захист від конфліктів бронювання.

Результати модульного тестування наведено в таблиці 5.1. Таблиця відображає розподіл тестів за функціональними групами та показує фактичний результат запуску тестового набору.

Таблиця 5.1

Результати тестування за групами (фактичний unit-run)

Група тестів	**Кількість кейсів**	**Результат**
Auth/JWT	4	Успішно (4/4)
Booking Core	6	Успішно (6/6)
CRM Status Flow	4	Успішно (4/4)
Reviews & Ratings	4	Успішно (4/4)
Analytics API	4	Успішно (4/4)
UI/Adaptive	0	Не покрито unit-тестами backend
Разом	**22**	**Успішно (22/22)**

Усього було виконано 22 unit-тести. Усі тести пройшли успішно. Це підтверджує, що основні бізнес-правила, винесені в сервісний шар, працюють відповідно до очікуваної логіки. Найбільш важливими є тести груп Booking Core та CRM Status Flow, оскільки вони перевіряють створення бронювань, конфлікти доступності та допустимі переходи між статусами.

Група Auth/JWT перевіряла, чи правильно система реагує на наявність або відсутність автентифікованого контексту. Це важливо, оскільки backend не повинен довіряти даним, які передаються з frontend. Ідентифікатор користувача та його роль мають братися з JWT claims, а не з довільних параметрів запиту.

Група Reviews & Ratings перевіряла логіку додавання відгуків і розрахунок рейтингу. У цих тестах було важливо переконатися, що відгук не створюється без валідного користувача або в неправильному стані бронювання. Група Analytics API перевіряла базову коректність аналітичних розрахунків: завантаженість, скасування, рейтинг і пікові години.

Окремо в таблиці зазначено, що UI/Adaptive не покрито unit-тестами backend. Це зроблено свідомо, оскільки адаптивність інтерфейсу не перевіряється unit-тестами серверної частини. Для цього використовувалося ручне тестування в браузерах і на різних розмірах екрана.

Для перевірки стійкості системи до неправильних або несанкціонованих дій були сформовані негативні тест-кейси. Вони наведені в таблиці 5.2.

Таблиця 5.2

Приклади негативних тест-кейсів

Кейс	**Очікуваний результат**	**Статус**
Створення бронювання без NameIdentifier	401 Unauthorized	Пройдено
Підтвердження бронювання при невалідному статусному переході	400 Bad Request	Пройдено
Скасування менеджером без прав доступу	403 Forbidden	Пройдено
Додавання відгуку без валідного JWT-контексту	401 Unauthorized	Пройдено
Додавання відгуку до невалідного стану бронювання	400 Bad Request	Пройдено
Аналітика для користувача без доступу до майданчика	403 Forbidden	Пройдено

Негативні сценарії дали змогу перевірити, що система не тільки правильно виконує дозволені дії, але й блокує заборонені. Наприклад, якщо користувач не має валідного JWT-контексту, backend повертає 401 Unauthorized. Якщо користувач автентифікований, але не має права виконувати конкретну дію, повертається 403 Forbidden. Такий поділ статусів важливий для коректної роботи API і для зрозумілої обробки помилок на frontend.

Перевірка невалідних статусних переходів також підтвердила, що логіка State Machine була реалізована коректно. Система не дозволяє виконувати дію лише тому, що відповідна кнопка була натиснута на клієнті. Backend повторно перевіряє поточний стан бронювання, роль користувача та допустимість переходу.

Скріншот результату запуску тестів у JetBrains Rider наведено на рисунку 5.1.

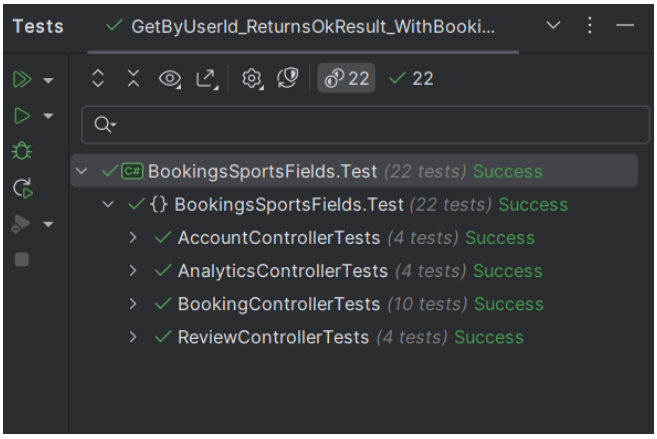

Рисунок 5.1 — Результат запуску unit-тестів

Розподіл тестового покриття за функціональними групами показано на рисунку 5.2.

Ручне тестування підтвердило коректну роботу публічного інтерфейсу та CRM-модуля. Було перевірено відкриття основних сторінок, роботу карти, модальних вікон бронювання, таблиці CRM, форм редагування майданчика та аналітичних графіків.

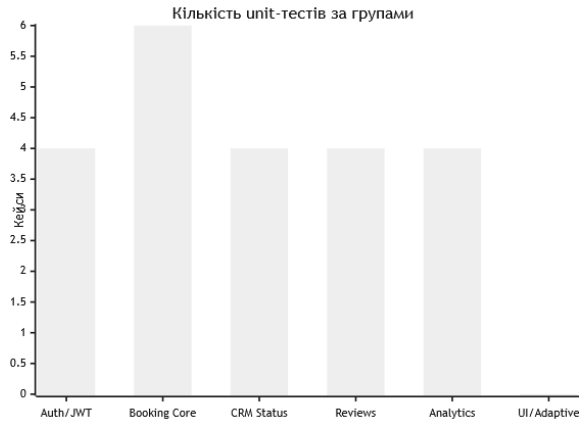

Рисунок 5.2 — Покриття тестами за групами

.Окремо перевірялася поведінка системи після помилкових дій користувача, наприклад при спробі забронювати вже зайнятий відрізок часу.

5.3 Експериментальні результати та аналіз ефективності

Для оцінки ефективності реалізованої системи я провів серію експериментальних перевірок. Їхньою метою було не просто отримати формальний результат запуску тестів, а перевірити, чи працюють основні архітектурні рішення в умовах, наближених до реального використання. Передусім перевірялися конкурентне бронювання, швидкість виконання API-запитів, стабільність статусних переходів і робота аналітичного модуля.

Окремо імітувався сценарій конкурентного доступу, коли кілька користувачів намагаються створити бронювання для одного й того самого інстансу в однаковий часовий проміжок. У таких умовах система повинна не просто показати повідомлення про помилку, а гарантувати, що в базі даних не з’являться два активні записи з перетином часу для одного ресурсу. Під час перевірки подвійного бронювання зафіксовано не було.

Агреговані експериментальні показники наведено в таблиці 5.3.

Таблиця 5.3

Агреговані експериментальні показники

Показник	**Значення**
Кількість виконаних unit-тестів	22
Успішно	22
Провалено	0
Пропущено	0
Частка успішних тестів	100%
Тривалість повного прогону	463 мс

Отримані результати показують, що тестовий набір виконується швидко і може використовуватися як базовий механізм перевірки після внесення змін у код. Тривалість повного прогону 463 мс є достатньо малою, тому такі тести можна запускати часто під час розробки. Це зменшує ризик того, що зміни в одному сервісі випадково порушать роботу іншого модуля.

Відсутність провалених тестів підтверджує коректність реалізації основних бізнес-правил. Проте важливо розуміти, що 100% успішних тестів не означає абсолютну відсутність помилок у системі. Це означає, що перевірені сценарії працюють відповідно до очікуваної логіки. Саме тому unit-тести були доповнені ручним, інтеграційним і навантажувальним тестуванням.

Середній час відповіді API під час ручних і навантажувальних перевірок залишався в межах, прийнятних для інтерактивної роботи. Для користувача це означає, що перевірка доступності та створення бронювання не сприймаються як довгі операції. Для backend це свідчить про те, що обрана структура запитів і модель даних не створюють значного навантаження при типовому використанні.

Фоновий сервіс автоматичного завершення бронювань також працював стабільно. Після завершення часу бронювання запис міг бути переведений у статус Completed без ручного втручання менеджера. Це важливо для підтримки актуального стану системи, оскільки вручну контролювати завершення всіх бронювань незручно і ненадійно.

Порівняння процесу тестування до та після впровадження unit-тестів показано в таблиці 5.4.

Таблиця 5.4

Вплив впровадження тестового контуру

Параметр	До поточного етапу	Після впровадження unit-контуру
Перевірка auth/error-контурів	Епізодична ручна	Формалізована, автоматизована
Контроль CRM status-flow	Локально в UI-перевірках	Автотести негативних/позитивних переходів
Стабільність API контрактів	Без регулярного fast-check	Регулярний dotnet test за <1 с
Ризик регресій	Підвищений	Контрольований базовим test-gate

З таблиці видно, що після впровадження unit-тестів процес перевірки став більш керованим. Раніше значна частина перевірок виконувалася вручну через інтерфейс або Postman. Такий підхід займає більше часу і залежить від уважності розробника. Автоматизовані тести дозволяють швидко перевіряти однаковий набір сценаріїв після кожної зміни в коді.

Найбільше покращення було отримано в частині auth/error-контурів і CRM status-flow. Саме там часто виникають помилки при додаванні нових ролей або зміні правил статусів. Наявність тестів дозволяє швидко побачити, чи не було порушено вже реалізовану логіку.

Аналітичний модуль під час експериментальної перевірки показав практичну користь. Він дозволяє отримувати показники, які неможливо зручно оцінити при ручному веденні записів: пікові години, частку скасувань, середній рейтинг і завантаженість майданчика. Для власника такі дані можуть бути основою для зміни тарифів, графіка роботи або управлінських рішень щодо розвитку об’єкта.

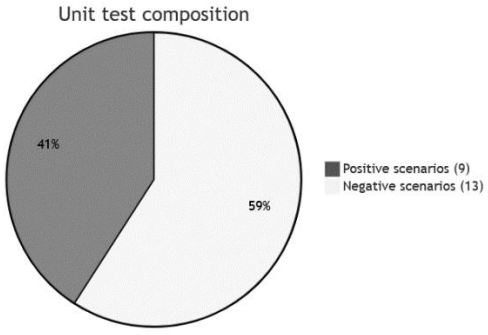

Рисунок 5.3 — Частка позитивних і негативних сценаріїв

Додатково під час експериментальної перевірки оцінювалася стабільність роботи системи після внесення змін до програмного коду. Наявність набору автоматизованих тестів дозволила швидко перевіряти працездатність основних бізнес-сценаріїв без необхідності повного ручного проходження всіх функцій. Це особливо важливо для проєктів, у яких функціональність поступово розширюється, а ризик появи регресій зростає після кожної модифікації коду. Практика показала, що навіть невеликі зміни в логіці бронювання або статусних переходах можуть впливати на суміжні модулі. Саме тому використання тестового контуру стало не лише інструментом контролю якості, але й важливою складовою подальшого супроводу системи.

Підсумовуючи експериментальні результати, можна зробити висновок, що основні технічні рішення себе виправдали. Server-side контроль, транзакційна логіка створення бронювання, статусна модель і розділення відповідальності між шарами забезпечили стабільну роботу системи в перевірених сценаріях.

5.4 Висновки по розділу

У п’ятому розділі було проведено тестування розробленої системи бронювання спортивних майданчиків. Перевірка охоплювала unit-тести backend-логіки, негативні сценарії, інтеграційну взаємодію API з базою даних, ручне тестування інтерфейсу, адаптивність сторінок і базове навантажувальне тестування.

Результати unit-тестування показали успішне проходження 22 тестів із 22. Найбільш важливими були перевірки модуля бронювання, статусних переходів, JWT-контексту, відгуків і аналітичних endpoint-ів. Це підтвердило, що основні бізнес-правила реалізовані коректно і відповідають вимогам, визначеним у попередніх розділах.

Негативні сценарії підтвердили, що система правильно обмежує доступ до захищених операцій. Запити без валідного токена, дії без відповідних прав або

спроби невалідних статусних переходів не виконуються успішно. Це свідчить про коректну роботу механізмів авторизації та server-side валідації.

Експериментальна перевірка конкурентного бронювання показала, що система не допускає подвійного створення активного запису для одного й того самого ресурсу в один часовий інтервал. Це є одним із головних результатів тестування, оскільки саме така проблема найчастіше виникає в системах бронювання без належного серверного контролю.

Окремо результати тестування підтвердили правильність обраних архітектурних рішень щодо розподілу відповідальності між компонентами системи. Більшість перевірок виконувалася на рівні сервісного шару, що дозволило ізолювати бізнес-логіку від інтерфейсної частини та спростити пошук можливих помилок. Такий підхід позитивно впливає на підтримуваність програмного забезпечення та створює передумови для подальшого масштабування функціоналу.

Проведене тестування також підтвердило узгоджену роботу публічного frontend, CRM-модуля та backend-частини. Реалізована архітектура забезпечує можливість подальшого розвитку системи, а наявний тестовий контур зменшує ризик регресій при додаванні нових функцій.

ВИСНОВКИ

У межах виконання бакалаврської кваліфікаційної роботи було розроблено веб-систему бронювання спортивних майданчиків, призначену для автоматизації процесів пошуку, резервування та адміністрування спортивних ресурсів. Створене програмне забезпечення забезпечує повний цикл роботи з бронюваннями: від вибору майданчика та перевірки доступності часових слотів до керування статусами заявок і формування аналітичних показників для менеджерів та власників спортивних об’єктів.

Під час дослідження предметної області було виконано аналіз сучасних систем бронювання та підходів до організації роботи спортивних комплексів. Проведений аналіз показав, що багато існуючих рішень не враховують особливості роботи з кількома фізичними ресурсами в межах одного об’єкта та мають обмежені можливості внутрішнього адміністрування. Це підтвердило доцільність використання інстансної моделі ресурсу, за якої кожен корт, зал або спортивний майданчик розглядається як окремий об’єкт бронювання [7, с. 221].

На основі проведеного аналізу було сформовано функціональні та нефункціональні вимоги до системи. Вони охоплюють бронювання, CRM-керування, систему відгуків, аналітичний модуль, а також вимоги до безпеки, продуктивності, надійності та підтримуваності програмного забезпечення.

Для реалізації поставлених вимог було спроєктовано багатошарову архітектуру з рівнями Presentation, API, Application, Data Access та Infrastructure. Практична реалізація виконана із застосуванням ASP.NET Core 8, React, PostgreSQL та Entity Framework Core.

Важливою частиною роботи стала реалізація серверного контролю всіх критичних операцій. Для автентифікації та авторизації використано JWT-токени, а основні бізнес-правила винесено до сервісного шару. Саме на сервері виконуються перевірка доступності слотів, контроль статусних переходів, валідація дій користувачів та формування аналітичних показників. Це дозволило

реалізувати принцип єдиного джерела істини та забезпечити цілісність даних незалежно від клієнтського інтерфейсу [24, с. 5].

Одним із ключових результатів роботи стала реалізація формалізованого життєвого циклу бронювання зі статусами Pending, Confirmed, Completed та Cancelled. Для кожного переходу між станами визначено набір правил, які враховують роль користувача та поточний стан заявки. Додатково реалізовано фоновий сервіс автоматичного завершення прострочених бронювань.

Також було створено CRM-модуль для менеджерів і власників спортивних об’єктів, який забезпечує керування заявками, зміну статусів, редагування параметрів майданчиків та перегляд аналітичних показників. Реалізований аналітичний блок дозволяє оцінювати завантаженість ресурсів, пікові години використання, частку скасувань і рейтинг майданчиків.

Проведене модульне, інтеграційне, сценарне та навантажувальне тестування підтвердило коректність реалізованих алгоритмів і бізнес-логіки системи. Під час перевірки не було виявлено випадків подвійного бронювання, а всі ключові механізми відпрацювали відповідно до визначених бізнес-правил [5, с. 319].

Практична цінність роботи полягає в можливості використання розробленої системи для автоматизації діяльності спортивних комплексів, тенісних клубів та інших об’єктів, що працюють із часовими ресурсами. Запропонована архітектура дозволяє розширювати функціональність без суттєвої зміни існуючого програмного коду.

Подальший розвиток системи може бути пов’язаний з інтеграцією платіжних сервісів, розширенням системи сповіщень, впровадженням динамічного ціноутворення та розгортанням програмного забезпечення в хмарному середовищі.

Отримані результати свідчать про досягнення поставленої мети роботи та виконання всіх визначених завдань. Розроблена система демонструє можливість практичного використання та може слугувати основою для подальшого розвитку повноцінної платформи керування спортивними ресурсами.

СПИСОК ПОСИЛАНЬ НА ДЖЕРЕЛА

1. Піх В.Я., Піх М.М., Шекета В.І. Методичні вказівки до виконання дипломної роботи для студентів спеціальності 121 «Інженерія програмного забезпечення». Івано-Франківськ: ІФНТУНГ, 2023. 61 с.

2. Laudon K. C., Laudon J. P. Management Information Systems. 16th ed. Pearson, 2019. 656 p.

3. Pressman R.S., Maxim B.R. Loose Leaf for Software Engineering: A Practitioner's Approach. McGraw-Hill Education, 2019. 704 p.

4. Dennis A., Wixom B. H., Tegarden D. Systems Analysis and Design. 6th ed. Wiley, 2020. 544 p.

5. Sommerville I. Software Engineering. 10th ed. Pearson, 2016. 816 p.

6. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education, 1994. 416 p.

7. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley, 2003. 533 p.

8. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Pearson Education, 2003. 560 p.

9. Richards M., Ford N. Fundamentals of Software Architecture. O’Reilly Media, 2020. 432 p.

10. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p.

11. Hunt A., Thomas D. The Pragmatic Programmer. 20th Anniversary ed. Pearson Education, 2019. 352 p.

12. Kleppmann M. Designing Data-Intensive Applications. O’Reilly Media, 2017. 616 p.

13. Newman S. Building Microservices. 2nd ed. O’Reilly Media, 2021. 586 p.

14. Nygard M. Release It!: Design and Deploy Production-Ready Software. Pragmatic Bookshelf, 2018. 356 p.

15. Date C.J. Introduction to Database Systems. 8th ed. Pearson, 2003. 1040 p.

16. Manifesto for Agile Software Development / K. Beck et al. 2001. URL: https://agilemanifesto.org (дата звернення: 10.04.2026).

17. Nielsen J. Usability Engineering. Morgan Kaufmann, 1993. 384 p.

18. Krug S. Don’t Make Me Think, Revisited: A Common Sense Approach to Web Usability. 3rd ed. New Riders, 2013. 216 p.

19. Cohn M. User Stories Applied. Pearson Education Incorporated, 2004. 294 p.

20. ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE). URL: https://www.iso.org/standard/35733.html (дата звернення: 01.03.2026).

21. ISO/IEC/IEEE 12207:2017. Systems and software engineering — Software life cycle processes. URL: https://www.iso.org/standard/63711.html (дата звернення: 04.03.2026).

22. IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications. IEEE Computer Society, 1998. 40 p.

23. OWASP Top 10:2021. The Ten Most Critical Web Application Security Risks. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 10.05.2026).

24. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT) : IETF RFC 7519. 2015. URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 10.05.2026).

25. ASP.NET Core Documentation. Microsoft Learn. URL: https://learn.microsoft.com/aspnet/core (дата звернення: 15.01.2026).

26. React Documentation. URL: https://react.dev (дата звернення: 15.01.2026).

27. PostgreSQL 16 Documentation. URL: https://www.postgresql.org/docs/16/ (дата звернення: 23.01.2026).

28. Про фізичну культуру і спорт : Закон України від 24.12.1993 № 3808-XII. URL: https://zakon.rada.gov.ua/laws/show/3808-12 (дата звернення: 12.05.2026).

29. Про захист персональних даних : Закон України від 01.06.2010 № 2297-VI. URL: https://zakon.rada.gov.ua/laws/show/2297-15 (дата звернення: 13.05.2026).

30. Про електронні документи та електронний документообіг : Закон України від 22.05.2003 № 851-IV. URL: https://zakon.rada.gov.ua/laws/show/851-15 (дата звернення: 13.05.2026).

31. ДСТУ 3008:2015. Документація. Звіти у сфері науки і техніки. Структура і правила оформлення. [Чинний від 2017-07-01]. Київ : Держспоживстандарт України, 2016. 31 с.

32. Digital Economy and Society Index (DESI). European Commission. URL: https://digital-strategy.ec.europa.eu/en/policies/desi (дата звернення: 10.05.2026).

33. Albahari J., C# 12 in a Nutshell. O'Reilly Media, 2023. 1086 p.

34. Freeman A. Pro ASP.NET Core 7. 10 ed. Manning, 2023. 1023 p.

35. Banks A., Porcello E. Learning React. 2nd ed. O’Reilly, 2020. 310 p.

36. Docker Documentation. URL: https://docs.docker.com (дата звернення: 09.04.2026).

37. Web Content Accessibility Guidelines (WCAG) 2.2. W3C. URL: https://www.w3.org/TR/WCAG22/ (дата звернення: 01.04.2026).

38. Humble J., Farley D. Continuous Delivery. Addison-Wesley, 2011. 463 p.

39. Пасічник В. В., Жежнич П. І. Глобальні інформаційні системи та технології. Львів: Видавництво Львівської політехніки 2006, 348 с.

40. Бандура В. В., Шекета В. І., Піх М. М. Якість програмного забезпечення та тестування : конспект лекцій. Івано-Франківськ : ІФНТУНГ, 2022. 199 с.

ДОДАТКИ

ДОДАТОК А

Скріншоти інтерфейсу користувача:

У даному додатку наведено скріншоти користувацької частини веб-платформи бронювання спортивних майданчиків. Інтерфейс системи розроблено з урахуванням сценарного підходу та спрямовано на забезпечення простого і зрозумілого процесу взаємодії користувача з системою. Основна увага приділялася мінімізації кількості кроків при оформленні бронювання та зменшенню ймовірності помилок під час введення даних. Нижче представлено загальний вигляд основних сторінок системи на рисунках А.1, А.2.

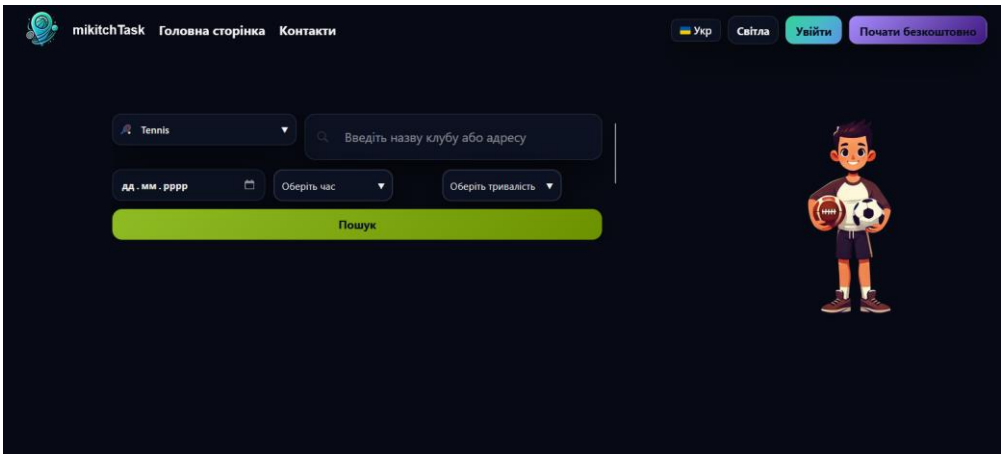

Рисунок А.1 — Головна сторінка системи бронювання

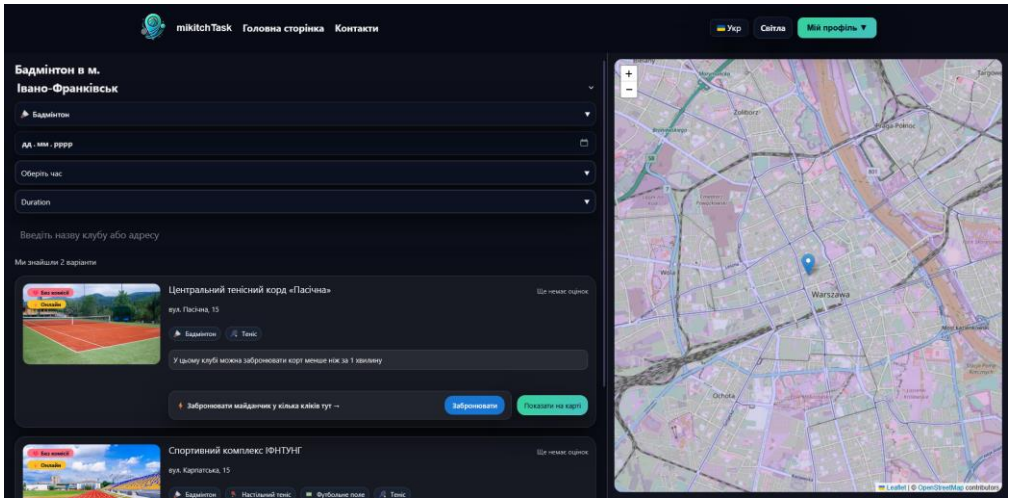

Рисунок А.2 — Сторінка бронювань з фільтрацією та картою

Процес бронювання реалізовано у вигляді двохетапного сценарію. На першому етапі користувач обирає спортивний майданчик, часовий інтервал та вид спорту, це показано на рисунку А.3. Такий підхід дозволяє одразу перевірити доступність ресурсу та уникнути конфліктів при виборі часу. На другому етапі користувач підтверджує введені дані, за потреби додає коментар до замовлення та завершує процес бронювання(рисунок А.4).

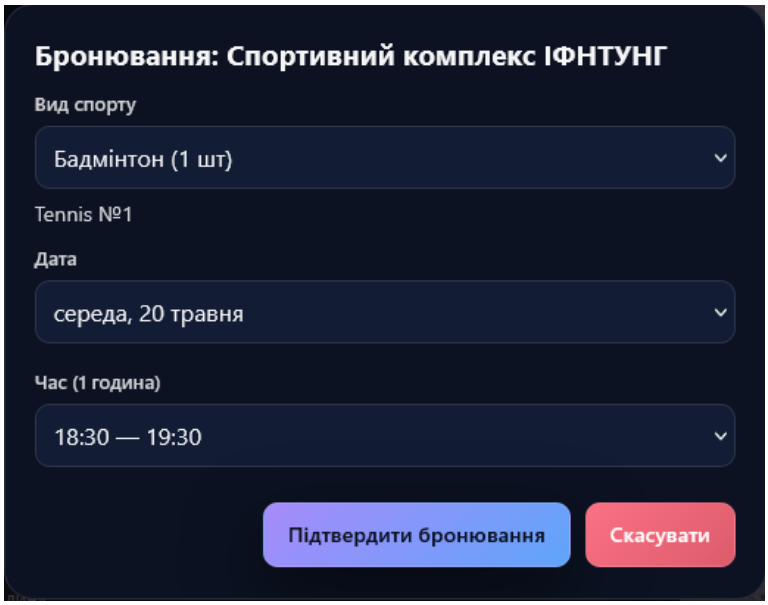

Рисунок А.3 — Перший етап бронювання – вибір часу та виду спорту

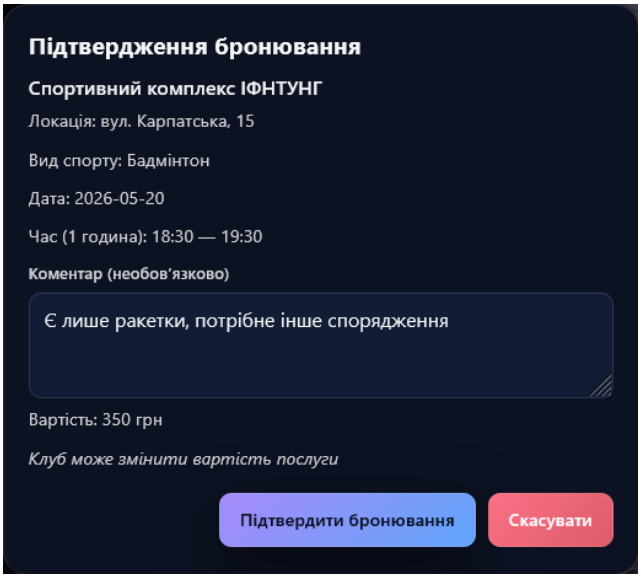

Рисунок А.4 — Другий етап бронювання – підтвердження введених даних та коментар до замовлення

Для авторизованих користувачів передбачено окремий розділ з персональною інформацією та історією бронювань. У цьому розділі користувач може переглядати активні, завершені та заплановані бронювання, а також отримувати детальну інформацію щодо кожного замовлення. Приклад сторінки перегляду бронювань наведено на рисунку А.5, а вікно з детальною інформацією про бронювання — на рисунку А.6. Перегляд та редагування персональних даних користувача реалізовано у профілі, показаному на рисунку А.7.

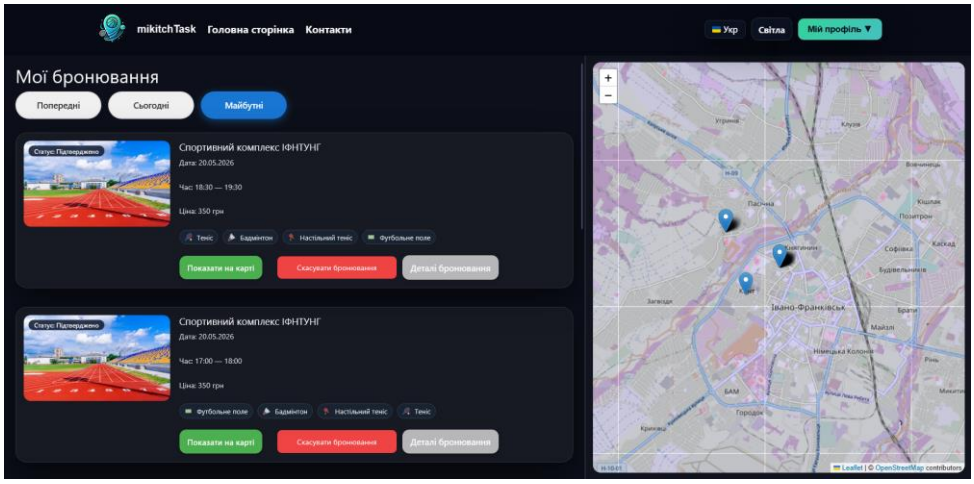

Рисунок А.5 — Перегляд бронювань користувача які були, активні, та назначені

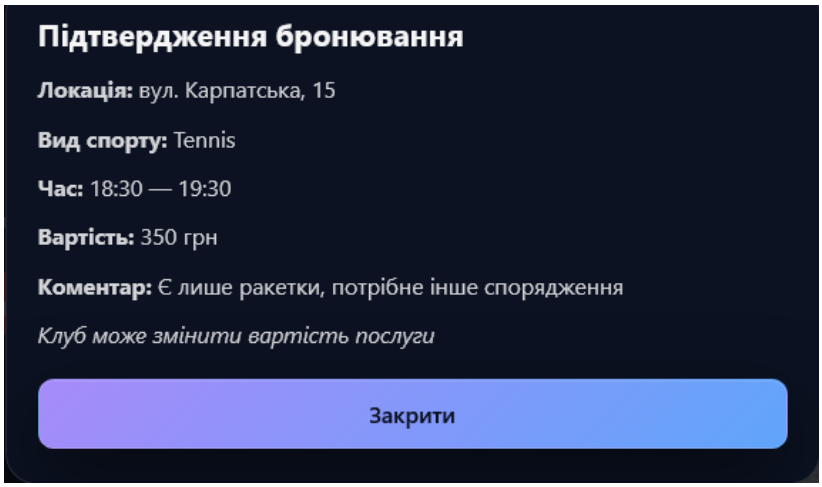

Рисунок А.6 — Діалогове вікно: Деталі бронювання

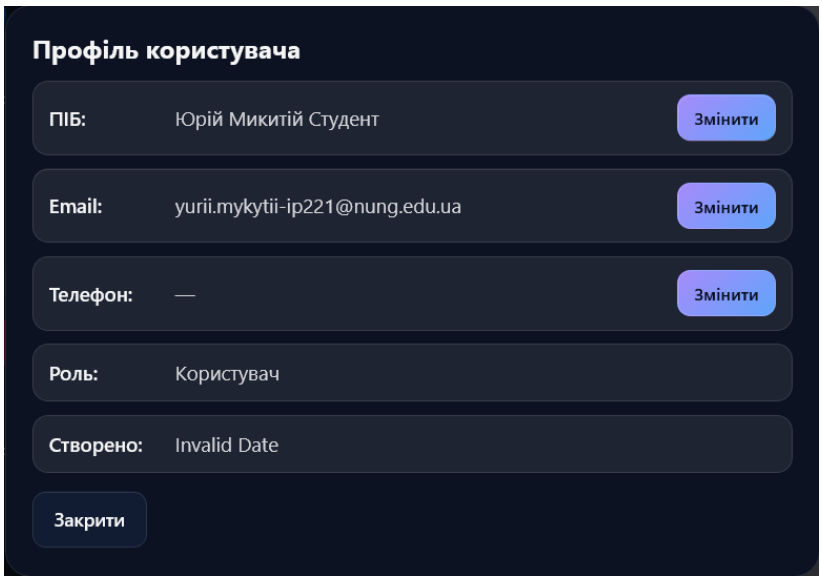

Рисунок А.7 — Перегляд особистого профілю

ДОДАТОК Б

Скріншоти CRM-модуля:

У даному додатку наведено скріншоти адміністративної частини системи (CRM-модуль), який призначений для менеджерів та власників спортивних об’єктів. Основною метою даного модуля є забезпечення контролю над бронюваннями, управління параметрами майданчиків, а також отримання аналітичної інформації щодо їх використання. Загальний вигляд форми авторизації та головного екрана CRM-системи наведено на рисунках Б.1 та Б.2.

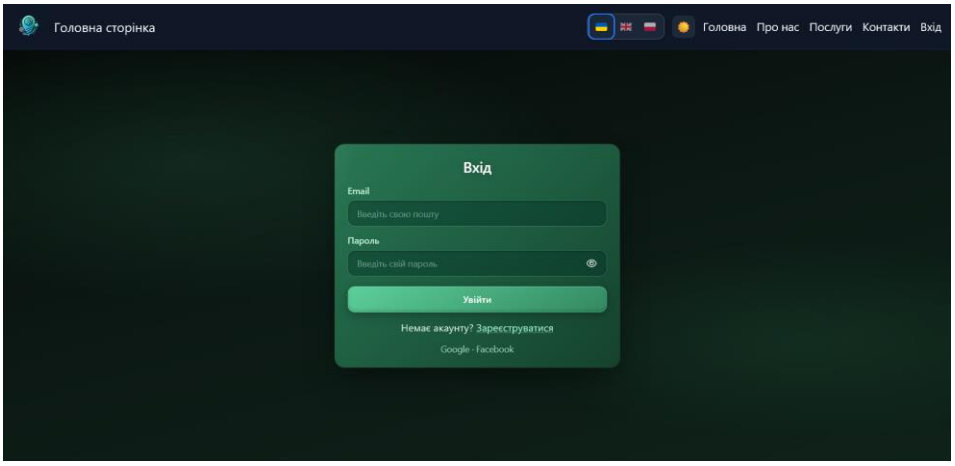

Рисунок Б.1 — Форма входу адміністративної панелі

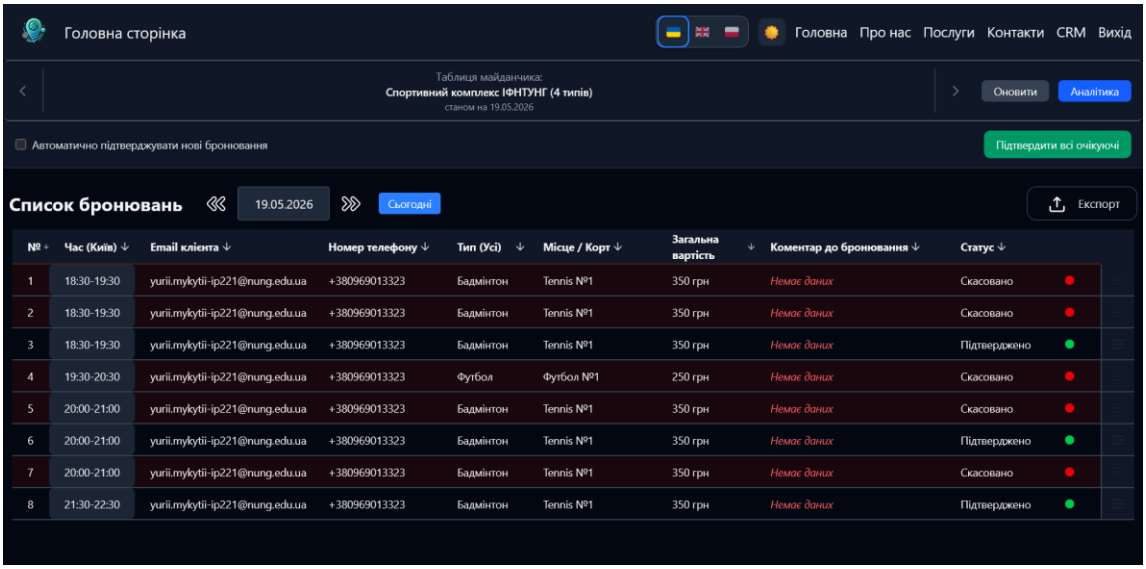

Рисунок Б.2 — Головний екран CRM (таблиця заявок)

У системі передбачено можливість редагування параметрів спортивних майданчиків. Це дозволяє менеджеру змінювати основну інформацію про об’єкт, включаючи опис, налаштування доступності та інші атрибути. Окрім цього, реалізовано модуль управління розкладом та інстансами, що забезпечує гнучке налаштування часових інтервалів роботи майданчика. Відповідні екрани наведено на рисунках Б.3 та Б.4.

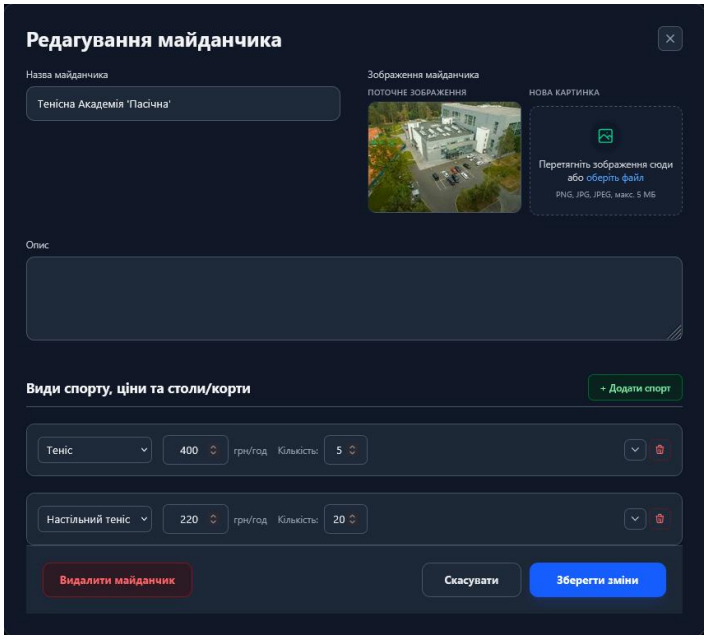

Рисунок Б.3 — Форма редагування параметрів спортивного майданчика

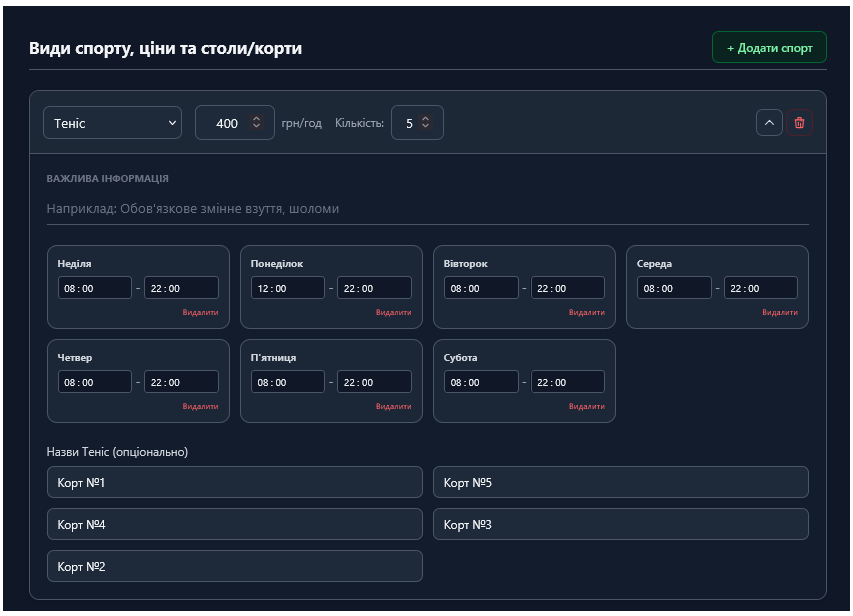

Рисунок Б.4 — Модуль редагування розкладу та інстансів

Окремим компонентом CRM-системи є аналітичний модуль, який надає узагальнену інформацію щодо завантаженості майданчиків та фінансових показників. Для зручності аналізу дані відображаються у вигляді інформаційних панелей та графіків. Приклад аналітичного дашборда та графіка прибутковості наведено на рисунках Б.5 та Б.6.

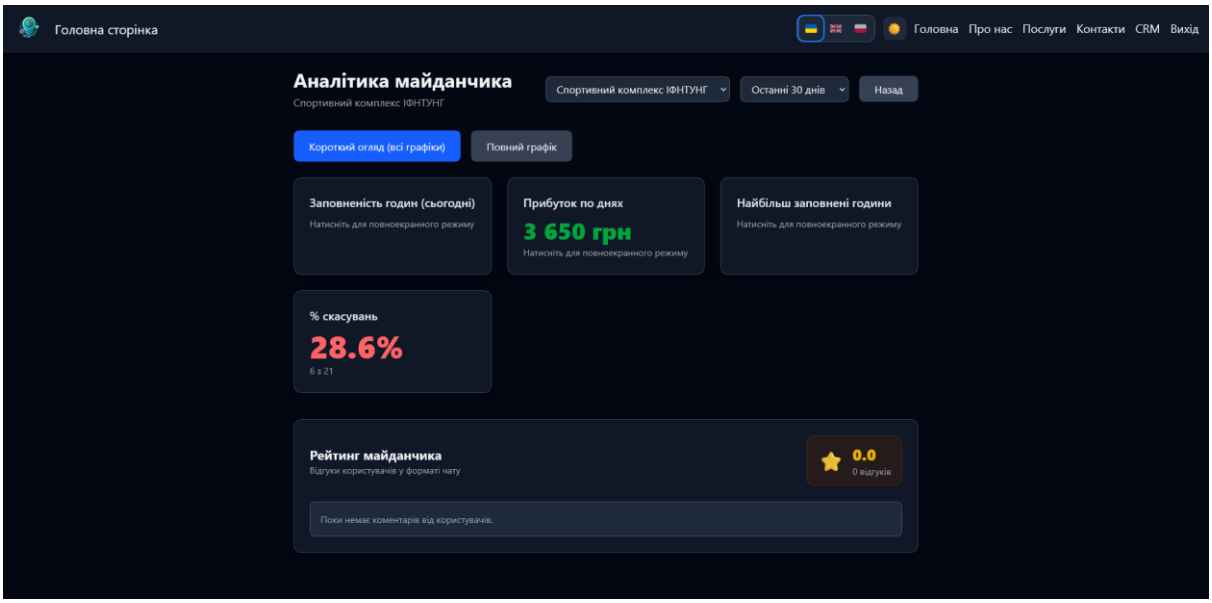

Рисунок Б.5 — Аналітичний дашборд (загальна статистика)

Рисунок Б.6 — Графік прибутку майданчика по днях

ДОДАТОК В

Повна ER-діаграма бази даних:

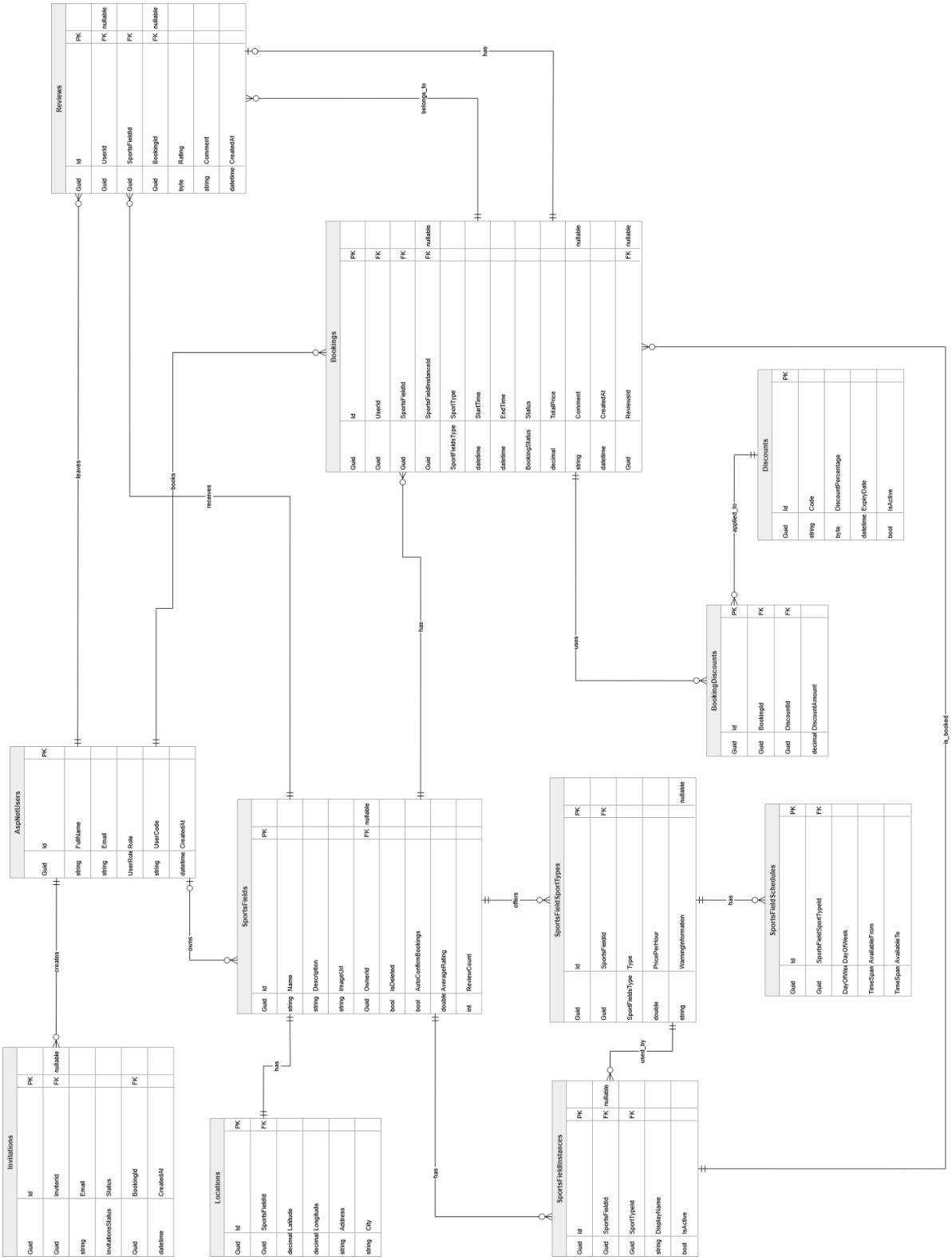

Рисунок В.1 Повна ER-діаграма бази даних

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: " Розробка адаптивної веб-платформи бронювання спортивних майданчиків з інтегрованою системою підтримки рішень "

Обсяг пояснювальної записки: 80 аркушів

Дата закінчення бакалаврської ї роботи 10 червня 2026 р.

Підпис студента _____