ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи

бакалавра
(освітньо-кваліфікаційний рівень)

на тему *«Розробка веб-системи управління ресурсами та автоматизації бронювання коворкінг-центру»*

Виконав: здобувач *3* року навчання
групи *3ПРс2*
напряму підготовки (спеціальності)
121-Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)

Кучмак І.Б.
(підпис, прізвище та ініціали)

Керівник *Величко Ю.І.*
(підпис, прізвище та ініціали)

Рецензент *к.т.н., доцент Корніловська Н.В*
(прізвище та ініціали)

Нормоконтролер *Комісаров О. С.*
(підпис, прізвище та ініціали)

Хмельницький – 2026

ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Інститут, факультет, відділення *Інформаційних технологій та дизайну*
Кафедра, циклова комісія *Програмних засобів і технологій*
Освітньо-кваліфікаційний рівень *бакалавр*
Освітньо-професійна підготовка *Програмна інженерія*
(шифр і назва)
Спеціальність *121-Інженерія програмного забезпечення*
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
програмних засобів і технологій
_____ О.Є. Огнєва
«___» _____
2026 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА

Кучмаку Івану Богдановичу
(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) *«Розробка веб-системи управління ресурсами та автоматизації бронювання коворкінг-центру»*

керівник проєкту (роботи) *доцент Величко Ю.І.*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від 16.01.2026 р. № 83-с

2. Строк подання здобувачем проєкту(роботи) 10.06.2026 р.

3. Вихідні дані до проєкту (роботи) літературні та періодичні джерела, матеріали переддипломної практики

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

1. Дослідження предметної області та аналіз існуючих систем управління коворкінгами.
2. Проєктування архітектури веб-системи та розробка структури реляційної бази даних.
3. Програмна реалізація клієнтської та серверної частин платформи FlexSpot (включаючи інтеграцію платіжної системи та авторизацію).
4. Тестування розробленого функціоналу, пошук та усунення помилок.
5. Оформлення інструкції користувача та підбиття підсумків роботи.

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

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання *05.01.2026*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів проєкту (роботи)	Примітка
1	Дослідження предметної області та аналіз аналогів.	03.02.2026 – 16.02.2026	Виконано
2	Формування вимог та проєктування архітектури ПЗ.	17.02.2026 – 02.03.2026	Виконано
3	Проєктування та розробка бази даних MySQL.	03.03.2026 – 16.03.2026	Виконано
4	Реалізація серверної частини на базі Laravel.	17.03.2026 – 06.04.2026	Виконано
5	Розробка клієнтського інтерфейсу на React.js.	07.04.2026 – 27.04.2026	Виконано
6	Інтеграція компонентів та сторонніх сервісів.	28.04.2026 – 11.05.2026	Виконано
7	Тестування системи та усунення дефектів.	12.05.2026 – 18.05.2026	Виконано
8	Оформлення звіту та підготовка до захисту.	19.05.2026 – 25.05.2026	Виконано
9	Захист кваліфікаційної роботи.		Виконано

Здобувач _____ *Кучмак І.Б.*
(підпис) (прізвище та ініціали)

Керівник проєкту (роботи) _____ *Величко Ю.І.*
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Кваліфікаційна робота бакалавра: 118 сторінок, 27 рисунків, 10 таблиць, 31 джерело, 2 додатки.

Об’єкт дослідження – процеси управління ресурсами та надання послуг у сучасних коворкінгах.

Предмет дослідження охоплює розробку програмного забезпечення, яке автоматизує бронювання місць, ведення фінансового обліку та загальне адміністрування таких центрів.

Мета роботи полягає у створенні веб-системи FlexSpot. Її головне завдання – спростити обслуговування резидентів і суттєво оптимізувати адміністративну рутину коворкінгів.

Короткий виклад. Під час роботи над дипломним проєктом розроблено повноцінну платформу FlexSpot. Вона поєднує в собі зручну клієнтську частину та розширену панель адміністратора.

За основу зберігання даних взято СУБД MySQL. Спроєктована реляційна база містить 14 взаємопов’язаних таблиць, які відповідають за ключові сутності проєкту: від профілів і фінансів (users, wallets, transactions) до інфраструктури та самих бронювань (locations, desks, bookings).

Серверна частина працює як масштабоване RESTful API, написане на PHP/Laravel. Для клієнтського інтерфейсу використано React.js (Vite) , а стилізація виконана через CSS Modules, що забезпечує надійну ізоляцію компонентів.

Щодо реалізованого функціоналу: система дозволяє шукати вільні місця та переговорні кімнати й відстежувати їхню доступність у реальному часі. Щоб уникнути прикрих ситуацій, коли двоє клієнтів одночасно бронюють одне й те саме місце, впроваджено механізм тимчасових чернеток. Вхід користувачів до системи безпечно обробляється через Firebase Authentication. Фінансові операції, такі як поповнення внутрішнього гаманця чи оплата послуг, проходять через офіційний шлюз LiqPay. Адміністратори, у свою чергу, отримали зручний

інструмент для керування локаціями, перевірки статусів ФОП-резидентів та контролю всіх транзакцій. Усі API-запити та процеси авторизації ретельно протестовано за допомогою Postman.

Практична значущість. За підсумками роботи ми отримали повністю життєздатний продукт, готовий до розгортання в реальних коворкінгах. FlexSpot бере на себе левову частку рутини адміністраторів. Завдяки системі чернеток вирішується класична проблема «овербукінгу» , а наявність гаманця та маркетплейсу тарифів робить фінансові відносини з резидентами максимально прозорими. Усе це перетворює розроблену систему на конкурентне рішення для сучасного бізнесу.

КЛЮЧОВІ СЛОВА: КОВОРКІНГ, ВЕБСИСТЕМА, АВТОМАТИЗАЦІЯ БРОНЮВАННЯ, УПРАВЛІННЯ РЕСУРСАМИ, LARAVEL, REACT.JS, РЕЛЯЦІЙНА БАЗА ДАНИХ, API.

АНОТАЦІЯ

Кучмак Іван Богданович. Розробка веб-системи управління ресурсами та автоматизації бронювання коворкінг-центру FlexSpot. – Кваліфікаційна робота бакалавра. Херсонський національний технічний університет, Хмельницький, 2026 р.

У роботі досліджується проблема оптимізації бізнес-процесів та надання послуг у сучасних коворкінг-центрах. Проведений аналіз предметної області дозволив виявити потребу у створенні єдиної платформи для зручного управління робочими просторами, фінансами та взаємодією з резидентами.

Спроєктовано клієнт-серверну архітектуру веб-системи. Розроблено структуру реляційної бази даних, яка налічує 14 взаємопов'язаних таблиць для комплексного обліку користувачів, транзакцій, локацій та бронювань. Обґрунтовано вибір технологічного стека: серверну частину реалізовано як масштабоване RESTful API на базі PHP та Laravel , а інтерактивний клієнтський інтерфейс побудовано з використанням React.js.

Реалізовано програмний продукт FlexSpot , який забезпечує розумний пошук просторів , моніторинг доступності робочих місць у реальному часі та безпечну авторизацію через Firebase. Для запобігання дублюванню замовлень розроблено двоетапний життєвий цикл бронювань із механізмом тимчасових чернеток. Фінансовий модуль інтегровано з платіжним шлюзом LiqPay. Усі API-ендпоінти успішно пройшли автоматизоване тестування за допомогою Postman.

Результати роботи можуть бути безпосередньо використані адміністраторами та власниками реальних коворкінгів для цифровізації щоденної рутини. Практична значущість проєкту полягає у створенні готового до розгортання продукту, що усуває проблему «овербукінгу», гарантує прозорий фінансовий облік та робить взаємодію команд чи ФОП із робочим простором максимально ефективною.

ABSTRACT

Ivan B. Kuchmak. Development of a web system for resource management and automation of coworking center booking FlexSpot. – Bachelor's Qualification Thesis. Kherson National Technical University, Khmelnytskyi, 2026.

The thesis explores the problem of optimizing business processes and service provision in modern coworking centers. The domain analysis revealed the need to create a unified platform for convenient management of workspaces, finances, and interactions with residents.

The client-server architecture of the web system was designed. The structure of a relational database was developed, comprising 14 interconnected tables for comprehensive accounting of users, transactions, locations, and bookings. The choice of the technological stack was justified: the server side was implemented as a scalable RESTful API based on PHP and Laravel, while the interactive client interface was built using React.js.

The FlexSpot software product was implemented, providing smart space search, real-time monitoring of workspace availability, and secure authorization via Firebase. To prevent booking duplication, a two-stage booking lifecycle with a temporary draft mechanism was developed. The financial module was integrated with the LiqPay payment gateway. All API endpoints successfully passed automated testing using Postman.

The results of the work can be directly used by administrators and owners of real coworking spaces to digitize their daily routines. The practical significance of the project lies in creating a ready-to-deploy product that eliminates the "overbooking" problem, ensures transparent financial accounting, and makes the interaction of teams or sole proprietors with the workspace as efficient as possible.

ЗМІСТ

ВСТУП

Актуальність теми. Сучасний ринок коворкінг-послуг базується на концепції гнучкого розподілу робочих зон. Головним технічним викликом для таких просторів є точна синхронізація спільних ресурсів під час активного навантаження користувачами. Йдеться про валідацію часових відрізків, обробку перетинів дат та строгий контроль лімітів посадки для різних видів залів та зон. Коли кілька клієнтів одночасно надсилають API-запити на бронювання одного місця, виникає ризик паралельних транзакцій у базі даних, а це класична технічна причина овербукінгу.

Аналіз популярних на ринку систем бронювання, таких як WeWork, Skedda, OfficeRnD та Andcards, виявив низку архітектурних та бізнес-обмежень. Здебільшого це закриті SaaS-платформи, які вимагають високих регулярних платежів та важко піддаються кастомізації під нестандартні планування приміщень невеликих коворкінгів. Така недосконаліть архітектури регулярно створює проблему як технічного характеру, таку як «овербукінг» так і завдає шкоду для самого бізнесу.

Відповідно, виникає гостра потреба у створенні власної цифрової екосистеми, яка б повністю автоматизувала перевірку місць та документообіг. Розробка вебсистеми FlexSpot є своєчасним рішенням, оскільки вона дозволяє ліквідувати людський фактор, забезпечити швидку взаємодію між коворкінгом, фізичними особами та B2B-клієнтами. А також розробка вебсистеми FlexSpot вирішує проблему конфліктів даних за рахунок оптимізованої серверної валідації. Такий підхід повністю нівелює технічні накладки та гарантує цілісність системи.

Мета і завдання дослідження. Головна мета роботи полягає у підвищенні ефективності бізнес-процесів та якості надання послуг у сучасних коворкінг-центрах шляхом проєктування та розробки комплексної вебсистеми FlexSpot. Розроблена платформа бере на себе автоматизацію управління просторами, фінансового обліку та надійного двоетапного бронювання робочих місць.

Для досягнення поставленої мети було визначено наступні завдання:

- Проаналізувати предметну галузь, дослідити специфіку ринку коворкінг-послуг та виявити функціональні недоліки існуючих систем бронювання.
- Обґрунтувати вибір технологічного стека та спроєктувати реляційну базу даних, здатну комплексно обробляти інформацію про локації, користувачів і транзакції.
- Розробити серверну логіку на базі фреймворку Laravel, впровадивши алгоритм управління чернетками, який автоматично блокує робоче місце на час оплати.
- Створити сучасний та швидкий клієнтський інтерфейс за допомогою React.js та Vite, що забезпечить плавний моніторинг доступності зон у реальному часі без перезавантаження сторінок.
- Інтегрувати сторонні сервіси для безпечної автентифікації користувачів (Firebase) та обробки платежів (LiqPay) з подальшим тестуванням готового програмного забезпечення.

Об’єкт дослідження – процеси адміністрування та розподілу ресурсів в інфраструктурі сучасних коворкінг-центрів.

Предмет дослідження – алгоритми, програмні інструменти та методи автоматизації двоетапного бронювання робочих зон.

Під час виконання роботи створено автономний програмний продукт FlexSpot. Розроблений механізм тимчасових чернеток функціонує безпосередньо на рівні бази даних MySQL і жорстко блокує обране місце. Це технічне рішення дозволило остаточно ліквідувати проблему подвійного бронювання, оскільки серверна частина самостійно відхиляє будь-які паралельні запити до моменту звільнення або успішного підтвердження поточного резерву. Адміністратори локацій більше не перевіряють графіки вручну. Налаштована інтеграція платіжного шлюзу LiqPay автоматизує безпечне проведення фінансових операцій для резидентів. Розроблена веб-система бере на себе основний масив щоденної рутини і функціонує як масштабоване інфраструктурне рішення.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1. Аналіз ринку та специфіки надання коворкінг-послуг.

Ринок нерухомості переживає масштабну реструктуризацію. Бізнес масово переходить на гібридні формати зайнятості. IT-команди відмовляються від довгострокової оренди офісів на користь коворкінг-центрів [1]. В Україні ця тенденція додатково стимулюється постійною потребою в автономній інфраструктурі, оскільки наявність промислових генераторів та терміналів Starlink перетворює робочі простори на критичні хаби для 100% безперервної діяльності.

Операційна модель коворкінгу кардинально відрізняється від класичної оренди. Простір ділиться на функціональні зони з власною монетизацією. Відвідувач Open Space купує денний доступ без прив'язки до столу. Клієнт зони Dedicated Desk оплачує закріплене місце на цілий місяць, тоді як резервування переговорних кімнат відбувається виключно за погодинною сіткою, що вимагає постійного контролю розкладу. Ця мультиформатність створює базу для подальшої алгоритмізації.

Комбінування різних тарифів вимагає точного розподілу ресурсів. Адміністрація повинна щохвилини відстежувати поточний статус кожного об'єкта. Ручний контроль такої інфраструктури гарантовано створює операційний хаос. Адміністратори програють звичайним таблицям Excel. Фізична неможливість вчасно синхронізувати записи із реальними візитами клієнтів призводить до втрати інформації, тому виникає гостра необхідність розробки надійної архітектури бази даних.

Сучасні відвідувачі гнучких просторів очікують гнучкості в оплаті послуг, що змушує коворкінги відмовлятися від фіксованої місячної оренди на користь пакетних абонементів та шерингової моделі. Користувачі купують лімітовані пакети, які містять певну кількість годин для переговорних кімнат або фіксовані дні відвідування Open Space з обмеженим терміном дії. Ручний трекінг залишків пакетного часу для десятків незалежних команд є практично неможливим. Це

призводить до регулярних суперечок між адміністрацією та резидентами. Для автоматизації такого обліку програмне ядро системи повинно підтримувати складну логіку розподілу та контролю актуальних квот. Порівняльну характеристику та специфікацію обліку основних робочих зон коворкінг-центру FlexSpot наведено в табл. 1.1.

Таблиця 1.1. Функціональні особливості та критерії обліку робочих зон коворкінгу

Тип робочої зони	**Формат посадки**	**Одиниця тарифікації та часу**	**Критерій валідації в базі даних (Логіка системи)**
Open Space	Вільна посадка на будь-яке доступне крісло у загальній робочій зоні коворкінгу.	День (фіксований разовий візит) або списання 1 одиниці з пакету візитів.	Контроль ліміту місткості: Сервер підраховує суму активних броней і чернеток на обрану дату. Новий запис дозволяється, лише якщо ця сума строго менша за максимальну місткість (max_capacity) зони.
Dedicated Desk	Закріплене конкретне місце (стіл) за резидентом.	Діапазон днів (фіксований довгостроковий період).	Унікальність об'єкта (desk_id): Перевірка відсутності активних замовлень чи чернеток на обраний стіл у межах вказаного періоду дат. Стіл не може бути зарезервований іншим user_id.
Meeting Rooms	Повне приватне бронювання окремого приміщення.	Погодинна сітка фіксованих слотів.	Перевірка перетину інтервалів (Overlap): Валідація часових меж (start_at, end_at) для обраної кімнати. Нова чернетка блокується, якщо її час частково чи повністю накладається на чинний слот.

Специфіка аудиторії коворкінгів вимагає чіткого розділення фінансових потоків на звичайні B2C-замовлення та профілі для фізичних осіб-підприємців. Фізичні особи віддають перевагу миттєвому поповненню рахунку через зручні платіжні шлюзи, тоді як корпоративні клієнти потребують збереження

юридичних реквізитів для звітності. Програмна архітектура має обробляти ці сценарії синхронно. Спліт-система списання ресурсів автоматично перевіряє наявність активних абонементів у профілі користувача. Спочатку використовуються доступні квоти, а в разі їхньої нестачі гроші автоматично знімаються з внутрішнього гаманця резидента [2]. Такий підхід повністю усуває необхідність постійного втручання менеджера у розрахунки.

Швидкість взаємодії з інтерфейсом відіграє ключову роль під час вибору робочого простору. Клієнти використовують особисті кабінети для швидкого пошуку вільних зон та перегляду статусів своїх замовлень у реальному часі. Якщо вебплатформа не забезпечує миттєве динамічне оновлення сітки часових слотів без перезавантаження сторінок, виникають затримки. Клієнти втрачають час. Саме тому інтеграція швидкого клієнтського інтерфейсу з оптимізованим бекендом, який виконує точні SQL-запити для перевірки перетинів дат, є єдиним інженерним шляхом для побудови конкурентного продукту.

1.2. Огляд та порівняльний аналіз існуючих систем бронювання.

Для об’єктивного оцінювання наявних інженерних підходів у галузі автоматизації коворкінгів проаналізовано архітектурні принципи провідних програмних платформ. Глобальні мережі гнучких офісів, такі як WeWork, використовують власні закриті екосистеми управління ресурсами. Їхня розподілена хмарна інфраструктура успішно координує роботу тисяч локацій по всьому світу в єдиному інформаційному контурі. Спеціалізовані аналітичні модулі цієї платформи у реальному часі збирають телеметрію відвідуваності, відстежують стан мережевого обладнання та автоматично коригують графіки технічного обслуговування зон. Проте цей потужний програмний комплекс розробляється виключно для внутрішніх бізнес-потреб конкретної корпорації. Він повністю закритий для інтеграції на сторонніх об'єктах. Незалежний оператор комерційного простору фізично не може отримати доступ до цього коду чи розгорнути його на власних серверах. Платформа функціонує як ізольований PropTech-продукт.

Альтернативним комерційним рішенням на ринку є спеціалізована B2B-платформа OfficeRnD [3]. Цей програмний продукт займає лідерські позиції у сегменті Enterprise-рішень, надаючи розробникам та менеджерам розгалужений набір інструментів для глобального адміністрування нерухомості. Його архітектура базується на концепції multi-tenancy, а ядро системи підтримує ведення детальних профілів компаній, облік довгострокових контрактів та динамічну візуалізацію планів поверхів з прив'язкою до датчиків присутності. Модулі білінгу автоматично взаємодіють із міжнародними банківськими системами та CRM-платформами для виставлення рахунків великим корпораціям. Масштабована структура OfficeRnD орієнтована на обслуговування великих мережевих об'єктів із десятками автономних філій [4]. Загальну концептуальну схему архітектури хмарних Enterprise SaaS-платформ корпоративного рівня приведено на рис. 1.1.

Незважаючи на високу гнучкість налаштувань, самостійні локальні коворкінг-центри часто стикаються з проблемою критичного функціонального надлишку цієї платформи. Величезна кількість сутностей, зв'язків та метаданих у структурі бази даних суттєво ускладнює процес базової конфігурації під потреби одного простору. Менеджери змушені витрачати десятки годин на ручне відключення непотрібних модулів і заплутану синхронізацію полів. Система вимагає утримання окремого технічного спеціаліста для підтримки працездатності. На додачу, універсальний підхід до обробки запитів створює відчутні затримки під час оновлення погодинних слотів для Meeting Rooms через перевантаженість серверних обробників. Висока вартість ліцензії та складність адаптації роблять інтеграцію Enterprise-систем економічно недоцільною для малих або середніх об'єктів. Локальний бізнес потребує швидкого, легкого та цільового інструменту управління.

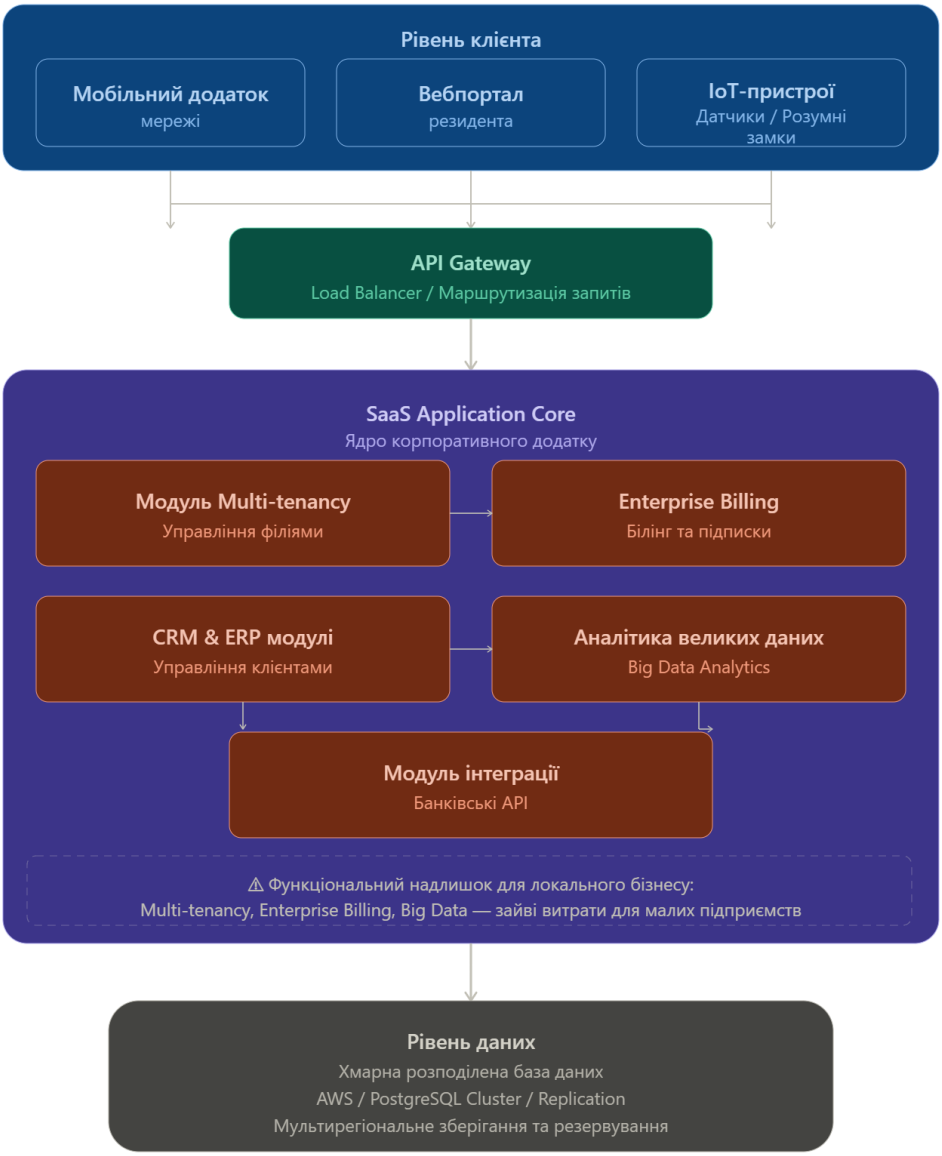


Рисунок 1.1. Блок-схема багатокористувацької архітектури Enterprise-систем управління просторами

Окрім великих Enterprise-комплексів, на ринку автоматизації представлені більш спеціалізовані рішення, орієнтовані на швидке розгортання графіків розподілу місць. Яскравим прикладом такого підходу є хмарна платформа Skedda. Ця система фокусується виключно на візуалізації простору та керуванні часовими слотами. Її інтерфейс надає інтерактивну карту приміщень, яка динамічно відображає поточну зайнятість кімнат, дозволяючи користувачам швидко обирати необхідний інтервал. Проте Skedda має суттєві архітектурні обмеження у фінансовому модулі. Система підтримує лише міжнародні платіжні

шлюзи типу Stripe, що повністю унеможливлює пряму інтеграцію з вітчизняними банківськими сервісами без використання сторонніх API-посередників. Крім того, її ядро не розраховане на спліт-систему паралельного списання балансу та пакетних квот. Це змушує менеджерів вручну комбінувати кілька різних програм для ведення повного обліку резидентів коворкінгу.

Окрему увагу в межах дослідження предметної галузі приділено українському PropTech-продукту Andcards, який є одним із найбільш затребуваних рішень на вітчизняному ринку. Платформа пропонує високу якість користувацького досвіду (UX), сучасний мобільний додаток та глибоку інтеграцію з локальними реаліями бізнесу. Програмні модулі Andcards успішно автоматизують комунікацію всередині спільноти, керують абонементами та спрощують перевірку перепусток на рецепції.

Проте, будучи комерційною закритою SaaS-платформою, Andcards використовує модель тарифікації, яка безпосередньо залежить від кількості активних користувачів або доданих робочих місць. Для незалежних коворкінгів, що розвиваються, такий підхід створює суттєве фінансове навантаження, яке зростає пропорційно до розширення резидентської бази. Більше того, закритий вихідний код повністю виключає можливість розгортання системи на власних VPS-серверах компанії. Це обмежує розробника у реалізації унікальних алгоритмів захисту від конфліктів даних – наприклад, кастомного двоетапного життєвого циклу чернеток із автоматичним похвилинним очищенням за допомогою серверного планувальника завдань.

Проведений огляд архітектурних та функціональних особливостей наявних рішень дозволяє виділити їхні спільні критичні недоліки в контексті автоматизації локальних коворкінг-центрів. Більшість комерційних SaaS-платформ використовують закритий вихідний код та монополізують розміщення інфраструктури на власних хмарних серверах. Це позбавляє локального розробника можливості точково оптимізувати SQL-запити або адаптувати бізнес-логіку під специфічні потреби конкретного приміщення. Крім того, відсутність у програмному ядрі конкурентів інтегрованих механізмів двоетапної валідації

запитів суттєво підвищує ризик виникнення конфліктів при одночасному доступі користувачів до бази даних. Клієнти отримують накладки. Залежність систем від іноземних платіжних провайдерів додатково ускладнює пряму взаємодію з локальними банківськими інструментами, змушуючи бізнес переплачувати комісії стороннім сервісам-посередникам.

Для систематизації результатів дослідження предметної галузі та обґрунтування вектору розробки проведено комплексне зіставлення аналогів із проектованою системою. Наочне порівняння ключових функціонально-технічних характеристик існуючих продуктів та розроблюваної системи наведено в табл. 1.2.

Таблиця 1.2. Порівняльний аналіз функціонально-технічних характеристик систем управління коворкінгами

Системи, що порівнюються	**Модель розгортання та код**	**Алгоритм захисту від овербукінгу**	**Пряма інтеграція з LiqPay API**	**Тип архітектури застосунку**	**Гнучкість кастомізації**	**Економічна доступність (Вартість)**
WeWork (внутрішнє ПЗ)	Закритий код (Proprietary). Внутрішня корпоративна хмара.	Внутрішня централізована валідація запитів.	Ні (глобальні корпоративні шлюзи).	Розподілена мікросервісна Enterprise-архітектура.	Низька (жорстко стандартизована під власні локації).	Не продається (лише для власної мережі).
OfficeRnD	Закритий код (SaaS). Multi-tenant хмарне розгортання.	Вбудований контроль доступності на рівні серверів платформи.	Ні (переважно Stripe, GoCardless, PayPal).	Важка B2B Enterprise-архітектура.	Висока (багато модулів), але потребує складного налаштування.	Висока (від ~$199/міс. + плата за кількість користувачів/ресурсів).

продовження табл. 1.2

Skedda	Закритий код (SaaS). Виключно хмарне розгортання.	Блокування слотів на рівні системи у реальному часі.	Ні (підтримується виключно Stripe).	Web-орієнтована платформа розкладів (Scheduling Platform).	Середня (обмежена типами «простір/час», без складного спліт-балансу).	Від безкоштовного до ~$39+/міс. за інтеграцію оплат.
Andcards	Закритий код (SaaS). Cloud-native хмарне розгортання.	Синхронна перевірка зайнятості через єдиний сервер платформи.	Так (наявні інтеграції з локальними шлюзами: Fondy, LiqPay).	Mobile-first архітектура з вебверсією.	Середня (оптимізована під стандартні бізнес-моделі коворкінгів).	Середня/Висока (щомісячна ліцензія залежить від кількості резидентів).
FlexSpot (Розроблювана система)	Відкритий код. Автономне розгортання на локальному або VPS-сервері.	Двоетапне бронювання (чернетки) із фоновим очищенням через Cron.	Так (пряма API-інтеграція без використання сторонніх конекторів).	Single Page Application (React) + REST API (Laravel).	Висока на рівні коду, але вимагає ручного втручання розробника для змін.	Безкоштовно (відсутність ліцензії), наявні витрати на утримання хостингу.

1.3. Обґрунтування необхідності розробки вебсистеми FlexSpot.

Зважаючи на результати порівняльного аналізу, використання готових комерційних продуктів для управління локальними коворкінг-центрами супроводжується низкою критичних архітектурних та економічних обмежень. Більшість існуючих хмарних платформ функціонують виключно за моделлю надання програмного забезпечення як послуги із закритою екосистемою. Цей підхід фундаментально унеможливлює ізольоване розгортання продукту на власних фізичних або віртуальних серверах компанії. Така архітектурна

закритість створює проблему жорсткої прив'язки інфраструктури бізнесу до єдиного постачальника. Відповідно, коворкінг-центр втрачає прямий контроль над власними даними та змушений сплачувати регулярні ліцензійні платежі. Економічна модель більшості конкурентних платформ базується на тарифікації за кількістю активних користувачів або доданих робочих місць, через що фінансове навантаження на бізнес непропорційно зростає разом із розширенням клієнтської бази. Крім того, залежність від універсальної хмарної інфраструктури постачальника позбавляє незалежних розробників технічної можливості оптимізувати структуру бази даних під нестандартні планування приміщень або модифікувати бізнес-логіку під специфічні потреби локації.

Окремим і надзвичайно вагомим недоліком більшості зарубіжних систем управління є повна відсутність нативних інтеграцій із популярними українськими платіжними шлюзами. Оскільки глобальні платформи орієнтовані на міжнародний ринок, вони підтримують переважно системи іноземного еквайрингу, які не функціонують повноцінно для юридичних осіб в Україні без реєстрації закордонних представництв. Неможливість прямої взаємодії з вітчизняними банками змушує адміністрацію підключати додаткові сервіси-посередники. Використання таких конекторів суттєво ускладнює прозорий фінансовий облік, вимагає постійної програмної синхронізації зовнішніх інтерфейсів та значно збільшує кінцеві транзакційні витрати через необхідність сплати подвійних комісій.

Найбільш критичним інженерним викликом у готових рішеннях залишається управління паралельними запитами під час активного резервування місць. Базові алгоритми систем конкурентів найчастіше не здатні миттєво блокувати доступ до ресурсу на етапі ініціалізації платежу. Фіксація транзакції в базі даних у таких продуктах зазвичай відбувається лише після отримання підтвердження про успішне списання коштів. Утворюється часове вікно тривалістю в кілька хвилин, під час якого система показує одне й те саме місце як вільне для кількох різних користувачів. Це створює ідеальні умови для виникнення алгоритмічних конфліктів, що неминуче призводить до регулярного

технічного овербукінгу під час пікових навантажень. Враховуючи вищезазначені інфраструктурні, фінансові та технічні проблеми, виникає об’єктивна необхідність у створенні власного програмного продукту, який би нівелював ці недоліки та надав локальному бізнесу повний контроль над внутрішніми процесами.

Проєктування та практична реалізація вебсистеми FlexSpot безпосередньо спрямовані на усунення архітектурних конфліктів і кастомну оптимізацію процесів розподілу спільної інфраструктури. Основу запропонованого інженерного рішення становить декомпозиція класичного процесу резервування місць на ізольовані етапи. Логіка бекенд-сервера, побудована на базі сучасного фреймворку Laravel, переносить первинну валідацію доступності робочого простору безпосередньо на рівень реляційної бази даних (MySQL). Замість трансляції миттєвого безповоротного запису, архітектура застосунку оперує динамічними змінами внутрішніх станів сутності бронювання. На етапі формування замовлення серверне ядро генерує тимчасовий запис із початковим статусом чернетки (Draft), виділяючи користувачеві чітко обмежене часове вікно тривалістю у п'ятнадцять хвилин для завершення взаємодії з інтерфейсом платформи.

Впровадження такого проміжного стану дозволяє ефективно блокувати будь-які паралельні конкурентні запити до прикладного програмного інтерфейсу (API) ще до моменту перенаправлення клієнта на платіжну сторінку еквайрингу [5]. Коли кілька резидентів одночасно намагаються орендувати ідентичний слот часу або конкретне робоче місце, сервер обробляє ці запити послідовно через вбудований шар валідації (Request Validation). Перший успішний POST-запит фіксує часові мітки початку та кінця візиту в таблиці бази даних, автоматично маркуючи цей часовий відрізок як зайнятий для пошукових фільтрів інших клієнтів. Наступні паралельні запити до аналогічного ресурсу миттєво відхиляються на рівні серверних перевірок, що дозволяє остаточно ліквідувати стан гонитви (Race Conditions) та повністю застрахувати систему від накладок або технічного овербукінгу.

Подальший життєвий цикл створеної чернетки повністю автоматизований та залежить від результатів взаємодії з інтегрованим платіжним шлюзом (LiqPay API). Після успішного списання коштів банківський еквайринг надсилає асинхронний зворотний виклик (Webhook) на закритий ендпоінт системи. Бекенд-контролер виконує обов'язакову криптографічну перевірку унікального цифрового підпису транзакції, після чого автоматично оновлює статус замовлення у базі даних на підтверджений (Confirmed). Якщо ж фінансова операція переривається або користувач залишає сторінку без оплати, у дію вступає автономна підсистема очищення. Фоновий планувальник завдань (Laravel Task Scheduling), підключений до системного демона сервера (Cron), щохвилини сканує таблицю бронювань. Програма автоматично виявляє прострочені за часом створення чернетки, змінює їхній статус на скасовані та миттєво повертає часові слоти у загальний доступний пул інфраструктури коворкінгу. Порівняльну схему логіки стандартного та розробленого двоетапного процесів обробки запитів приведено на рис. 1.2.

Технологічний стек системи оптимізовано для забезпечення максимальної швидкості взаємодії резидента з інтерфейсом платформи. Клієнтська частина, побудована за принципом Single Page Application на базі бібліотеки React.js та надшвидкого збирача модулів Vite, мінімізує обсяг мережевого трафіку за рахунок компонентної архітектури. Маршрутизація на стороні клієнта усуває затримки при переході між каталогом зон та особистим кабінетом. Динамічне опитування сервера та запити під час монтування компонентів дозволяють миттєво оновлювати сітку доступних місць у реальному часі без повного перезавантаження сторінок. Інтеграція набору оптимізованих векторних іконок Lucide React додатково знижує навантаження на браузер користувача під час рендерингу графічних елементів.

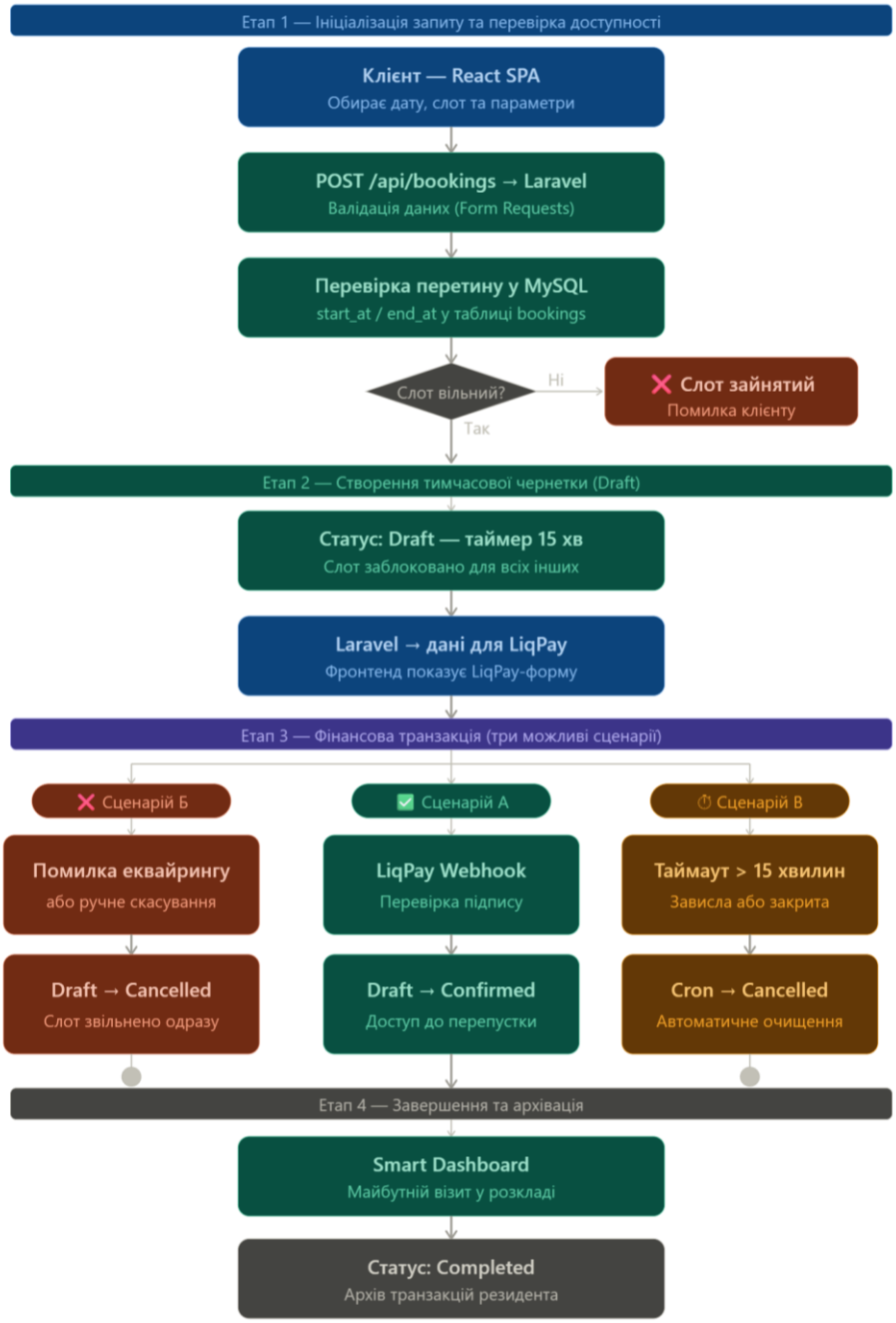


Рисунок 1.2. Концептуальна схема порівняння стандартного бронювання та двоетапного алгоритму FlexSpot з механізмом чернеток

Серверне RESTful API на базі бекенд-фреймворку Laravel гнучко взаємодіє з клієнтом виключно через обмін даними у форматі JSON. Ядро системи спирається на реляційну базу даних MySQL, яка складається з 14 таблиць із чітко прописаними зовнішніми ключами для забезпечення цілісності інформації. Зв'язки сутностей на рівні моделей спроєктовані під три різні типи зон:

безлімітна посадка, фіксовані місця та жорстка погодинна тарифікація. Застосування технології Eager Loading в Eloquent ORM запобігає появі критичної проблеми N+1 запитів під час обробки масивів інфраструктури, що суттєво зменшує час відповіді сервера. Крім того, всі фінансові операції та зміна статусів чернеток обгорнуті в транзакції бази даних, що блокує запис у разі часткового збою коду.

Економічна доцільність впровадження розроблюваного програмного забезпечення полягає в його повній автономності. Відкритий вихідний код дозволяє коворкінг-центру ізольовано розгорнути систему на власному VPS-сервері під управлінням Linux та Nginx або на локальному сервері типу Open Server без виплати регулярних ліцензійних зборів стороннім хмарним провайдерам. Використання вбудованих механізмів міграцій та сідерів дозволяє автоматично розгорнути та наповнити початкову структуру бази даних за лічені хвилини.

Завдяки такій інфраструктурі бізнес отримує інструмент із точковим контролем бази даних та власною логікою внутрішнього гаманця. Спліт-система автоматично перевіряє активні пакетні квоти резидентів перед списанням коштів, усуваючи необхідність ручного трекінгу залишків менеджерами. Створення системи FlexSpot забезпечує локальний простір масштабованим продуктом, який поєднує надійність серверної транзакційної валідації із високою продуктивністю користувацького інтерфейсу.

1.4. Постановка задачі та формування вимог до програмного забезпечення.

Головним завданням розробки є створення автономної вебсистеми FlexSpot, здатної оптимізувати процеси розподілу ресурсів та фінансового обліку в коворкінг-центрі. Програмний продукт повинен ліквідувати ручне адміністрування простору за рахунок впровадження надійної клієнт-серверної архітектури.

Для досягнення цієї мети необхідно реалізувати чіткий розподіл прав доступу. Архітектура бази даних має обробляти запити від трьох основних категорій користувачів: звичайного резидента (фізичної особи), корпоративного клієнта (B2B/ФОП) та адміністратора локації. Кожна з цих ролей отримує власний набір інструментів.

Базуючись на результатах аналізу предметної галузі, до розроблюваної вебсистеми FlexSpot висувається наступний комплекс функціональних вимог:

- Модуль автентифікації та розширеного управління профілем: Система повинна забезпечувати безпечну реєстрацію, авторизацію та відновлення доступу за допомогою JWT-токенів через інфраструктуру Firebase. Архітектура модуля має підтримувати логічний розподіл акаунтів на профілі фізичних осіб (B2C) та корпоративних клієнтів (B2B/ФОП) із можливістю збереження юридичних реквізитів. Інтерфейс особистого кабінету (Resident Dashboard) зобов'язаний автоматично підтягувати найближчі підтверджені візити, надавати доступ до цифрових перепусток та відображати деталізований стрім фінансових транзакцій.
- Каталог локацій та смарт-пошук у реальному часі: Клієнтська частина платформи має генерувати інтерактивний список доступних робочих зон із чітким поділом на типи тарифікації (вільна посадка для Open Space, фіксовані місця для Dedicated Desk, погодинна сітка для Meeting Rooms). Інтерфейс повинен підтримувати багатопараметричну фільтрацію просторів за наявністю специфічного обладнання (промислові генератори, термінали Starlink) та динамічно оновлювати індикатори кількості вільних місць без повного перезавантаження сторінки.
- Ядро бронювання (Booking Engine) та захист від конфліктів: Це критичний обчислювальний компонент платформи. Алгоритм має реалізовувати суворий двоетапний життєвий цикл обробки замовлень. Під час ініціалізації транзакції сервер створює тимчасову чернетку (Draft) та через транзакції бази

даних жорстко блокує відповідний часовий слот на 15 хвилин, відхиляючи будь-які паралельні запити. У разі відсутності підтвердження про оплату, інтегрований фоновий планувальник (Cron) повинен автоматично очищувати прострочені чернетки. Цей механізм гарантує ліквідацію проблеми технічного овербукінгу (Race Conditions).

- Фінансова підсистема та спліт-баланс: Програмний продукт повинен автономно розраховувати підсумкову вартість послуг та обробляти транзакції через API-шлюз LiqPay із обов'язковою криптографічною верифікацією callback-підписів від банку. Алгоритм списання має працювати за смарт-моделлю: система першочергово валідує наявність активних пакетних квот (передплачених годин чи днів) у профілі резидента, а за їхньої нестачі – ініціює автоматичне зняття коштів із внутрішнього гаманця (Wallet). Додатково модуль має генерувати структуровані податкові інвойси для корпоративних замовлень.

- Панель адміністратора (Admin Dashboard): Система повинна надавати закритий інтерфейс із рольовим доступом (Role-Based Access Control) для управління інфраструктурою коворкінгу. Адміністратор отримує інструменти для повного циклу CRUD-операцій над робочими зонами (створення об'єктів, зміна місткості, тимчасове блокування на ремонт), конфігурації пакетних тарифів та моніторингу активних бронювань. Окремим модулем необхідно реалізувати функціонал для швидкої валідації згенерованих PDF-перепусток резидентів безпосередньо на рецепції простору.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ПЗ

2.1. Вибір та обґрунтування технологічного стека розробки.

Для реалізації клієнтської частини вебсистеми FlexSpot було обрано архітектурну модель односторінкового застосунку (Single Page Application, SPA). Ключовим інструментом побудови користувацького інтерфейсу виступає бібліотека React.js. Вибір цієї технології зумовлений високими вимогами до інтерактивності та швидкості відгуку платформи. Оскільки інтерфейс системи оренди коворкінгу передбачає безперервну роботу з динамічними станами – локальними таймерами зворотного відліку чернеток замовлень, візуальною зміною статусів робочих місць на карті простору та каталогами зон – використання механізму Virtual DOM у React.js дозволяє точково оновлювати лише змінені компоненти інтерфейсу без повного перезавантаження вебсторінок браузером.

Компонентно-орієнтований підхід забезпечує глибоку ізоляцію логіки: окремі модулі, такі як картки пакетних тарифів, платіжні форми чи віджети інтерактивного розкладу, розробляються як незалежні одиниці коду. Це підвищує рівень повторного використання програмного коду (reusability) та суттєво спрощує подальше масштабування клієнтської архітектури. Маршрутизація користувацьких запитів (Routing) повністю перенесена на сторону клієнта за допомогою бібліотеки React Router. Це забезпечує миттєві переходи між публічними сторінками простору та захищеним особистим кабінетом резидента, формуючи плавний користувацький досвід (UX) без затримок на мережеві HTTP-запити до сервера за новими HTML-документами.

З метою забезпечення максимальної продуктивності середовища розробки та оптимізації кінцевої збірки проєкту впроваджено сучасний інструмент Vite [6]. На відміну від класичних збирачів модулів, Vite використовує нативні ES-модулі (ECMAScript) безпосередньо в браузері під час локального програмування. Такий інженерний підхід скорочує час холодного старту сервера розробника до кількох мілісекунд. Інтегрований механізм гарячої заміни модулів (Hot Module

Replacement, HMR) гарантує миттєве відображення внесених змін у вихідному коді на екрані без втрати поточного реактивного стану (state) React-компонентів. На етапі підготовки продукту до розгортання Vite виконує глибоку мініфікацію коду за допомогою Rollup та розділяє його на логічні фрагменти (code splitting), що мінімізує обсяг фінального пакета даних.

Візуальна складова клієнтського інтерфейсу доповнюється оптимізованим набором векторних графічних елементів з бібліотеки Lucide React. Використання цієї колекції дозволяє системі застосовувати алгоритм видалення неактивного коду (Tree-shaking). Завдяки цьому в кінцевий клієнтський бандл потрапляють виключно ті іконки, які фактично задіяні в інтерфейсі, що дозволяє повністю відмовитися від завантаження важких монолітних шрифтових файлів. Це архітектурне рішення критично знижує мережеве навантаження під час первинної ініціалізації клієнтського додатка FlexSpot та пришвидшує час його готовності до взаємодії з користувачем.

Організація серверного ядра та бізнес-логіки системи побудована на базі мови програмування PHP та об'єктно-орієнтованого фреймворку Laravel. Вибір цього інструментарію обґрунтовується високою масштабованістю бекенд-рішень та наявністю потужної екосистеми вбудованих компонентів. Архітектура серверної частини спроєктована за принципами RESTful API. Це забезпечує повне відокремлення клієнтського інтерфейсу від логіки обробки даних, причому обмін інформацією відбувається виключно у стандартизованому текстовому форматі JSON.

Застосування фреймворку Laravel дозволяє ефективно керувати життєвим циклом HTTP-запитів через проміжне програмне забезпечення (Middleware), що є критично важливим для перевірки прав доступу під час розмежування ролей звичайного резидента та адміністратора платформи. Крім того, наявність вбудованого планувальника завдань (Task Scheduler) забезпечує автономну роботу фонових процесів, зокрема консольних команд для автоматичного очищення протермінованих чернеток бронювань.

Для надійного зберігання й обробки інформації впроваджено класичну реляційну систему управління базами даних MySQL. Вибір реляційної моделі замість NoSQL-рішень диктується суворими вимогами до консистентності фінансових операцій та наявністю жорстких зв'язків між сутностями предметної галузі, такими як користувачі, тарифи, робочі зони та абонементи. Взаємодія програмного коду сервера з базою даних, яка налічує чотирнадцять нормалізованих таблиць, здійснюється через об'єктно-реляційне відображення Eloquent ORM. Цей патерн проєктування дозволяє маніпулювати записами таблиць як екземплярами класів, що значно пришвидшує розробку бізнес-логіки. З метою оптимізації продуктивності та запобігання появі вузьких місць у вигляді проблеми N+1 запитів під час генерації великих масивів розкладу, система повсюдно використовує механізм жадібного завантаження даних (Eager Loading).

Окрему інженерну увагу в серверній архітектурі приділено безпеці конкурентних операцій. Усі критичні етапи обробки замовлень, зокрема ініціалізація списання квот чи коштів із внутрішнього гаманця або зміна статусу часового слота, інкапсульовані в ізольовані транзакції бази даних (DB::transaction). Таке рішення взагалі гарантує дотримання принципу атомарності. У разі виникнення будь-якої програмної помилки або збою під час взаємодії із зовнішніми сервісами на проміжному етапі, ядро автоматично відкочує всі частково виконані зміни бази даних до початкового стану, унеможливлюючи виникнення фінансових чи логічних аномалій. Концептуальну схему взаємодії та обміну даними між компонентами обраного технологічного стека наведено на рис. 2.1.

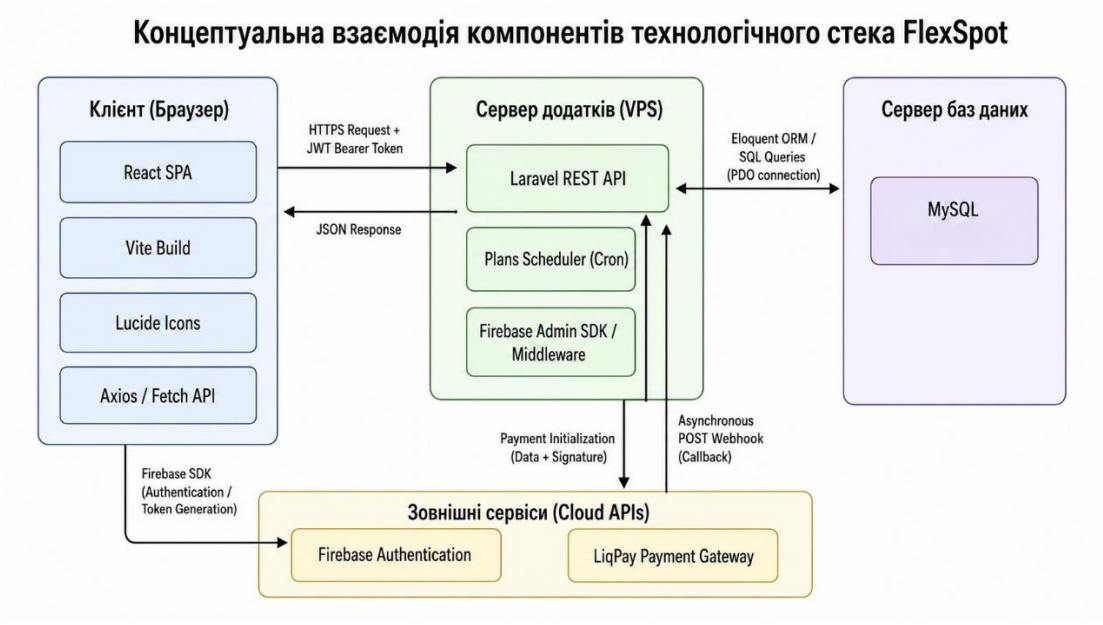


Рисунок 2.1. Блок-схема концептуальної взаємодії компонентів технологічного стека FlexSpot

Забезпечення надійної автентифікації користувачів та захисту їхніх персональних даних без збереження стану на сервері базується на інтеграції хмарної платформи Firebase Console. Такий підхід повністю децентралізує керування доступами та усуває потребу в класичних сесіях. Сервіс Firebase Authentication перебирає на себе збереження конфіденційної інформації та первинну валідацію облікових даних, завдяки чому локальна база даних стає захищеною від ризиків потенційної компрометації. Авторизація клієнтських запитів до API виконується за допомогою токенів JSON Web Token (JWT). Під час успішного входу користувача хмарний провайдер генерує унікальний токен, який клієнтський React-застосунок автоматично прикріплює до заголовків Authorization кожного наступного HTTP-запиту. Серверна частина на базі Laravel через спеціалізоване проміжне програмне забезпечення FirebaseAuthMiddleware виконує безстанову валідацію підпису отриманого токена, що гарантує високу стійкість системи до несанкційного проникнення.

Фінансова стабільність платформи та автоматизація платежів за оренду робочих зон реалізовані через пряму інтеграцію з платіжним шлюзом LiqPay API. Бізнес отримує прозорі розрахунки в національній валюті без подвійних

конвертацій. Взаємодія з платіжним сервером побудована за асинхронним принципом за допомогою механізму зворотних викликів (Webhooks). Під час ініціалізації транзакції серверне ядро FlexSpot передає шлюзу кодований у формат Base64 рядок метаданих, що містить суму, валюту та унікальний ідентифікатор операції order_id. Цілісність фінансової операції та автентичність джерела забезпечуються формуванням цифрового підпису. Спеціальний контролер бекенду після отримання результату операції у вигляді асинхронного вебхука від сервера платіжної системи самостійно обчислює очікуване значення SHA-1 хешу на основі отриманого пакета даних та секретного приватного ключа коворкінгу. Сервер порівнює згенерований хеш із надісланим підписом (сигнатурою) і лише за умови їхнього стовідсоткового збігу, що свідчить про цілісність даних та відсутність стороннього втручання, декодує вміст пакета й змінює стан замовлення у базі даних MySQL на підтверджений.

Валідація працездатності, симуляція мережевих запитів та комплексне автоматизоване тестування розроблених REST-ендпоінтів виконувалися за допомогою спеціалізованої платформи Postman. Це дозволило досконально перевірити бізнес-логику бекенду та роботу шарів авторизації ще до моменту фінальної інтеграції з клієнтським інтерфейсом. В межах проєкту було створено структуровану колекцію запитів з використанням глобальних змінних середовища (Environment Variables) для швидкого перемикання між локальним сервером розробки та віддаленим хостингом.

Завдяки вбудованому скриптовому фреймворку, Postman в автоматичному режимі перевіряв коди відповідей сервера, структуру згенерованих JSON-пакетів та швидкість виконання SQL-запитів. Окрема інженерна увага в цьому середовищі приділялася симуляції асинхронних платіжних вебхуків шлюзу LiqPay, які обробляє метод handleCallback. Такий підхід дозволив детально протестувати стійкість фінансового модуля до помилок або спроб несанкціонованої підміни параметрів оплати з боку зовнішніх вузлів.

Для ведення розробки, контролю версій вихідного коду та збереження архітектурної цілісності вебсистеми FlexSpot застосовувалася розподілена система Git спільно з хмарним сервісом GitHub. Кожен крок чітко зафіксовано в історії комітів. Стратегія розгалуження (Git Flow) дозволила ізольовано розробляти окремі компоненти, зокрема алгоритм чернеток або логіку розрахунку змішаної оплати коворкінгу, в автономних гілках. Це повністю виключило виникнення конфліктів у загальній кодовій базі застосунку. Контроль версій у поєднанні з репозиторієм GitHub виступає надійним засобом для резервного копіювання розробки та гарантує швидке розгортання (deployment) актуальної збірки продукту на VPS-сервері під управлінням Linux. Узагальнену специфікацію компонентів обраного технологічного стека та їхніх функціональних ролей у системі наведено в табл. 2.1.

Таблиця 2.1. Специфікація та функціональне призначення технологічного стека вебсистеми FlexSpot

Рівень архітектури	**Обрана технологія**	**Функціональна роль у системі FlexSpot**	**Обґрунтування вибору**
Клієнтська частина (Frontend)	React.js	Побудова користувацького інтерфейсу за принципом односторінкового застосунку.	Точкове оновлення віртуального DOM-дерева, компонентний підхід.
Збирання проєкту	Vite	Оптимізація та компіляція клієнтських модулів.	Надшвидкий холодний старт сервера, нативна підтримка модулів ECMAScript.
Графічні елементи	Lucide React	Рендеринг векторних іконок інтерфейсу.	Підтримка алгоритму повного усунення мертвого коду з фінального пакета.

продовження табл. 2.1

Серверне ядро (Backend)	PHP / Laravel	Реалізація бізнес-логіки та маршрутизації HTTP-запитів.	Надійна екосистема, наявність планувальника завдань, гнучка обробка API.
Рівень даних (Database)	MySQL	Зберігання та забезпечення консистентності інформації предметної галузі.	Реляційна модель, підтримка ACID-властивостей для фінансових транзакцій.
Автентифікація	Firebase Console	Децентралізоване керування сесіями та валідація JWT-токенів.	Безпека конфіденційних даних, зниження обчислювального навантаження на VPS.
Фінансовий шлюз	LiqPay API	Обробка безготівкових платежів та транзакцій у реальному часі.	Пряма API-інтеграція, асинхронні Webhook-виклики, відсутність подвійних комісій.

2.2. Проєктування архітектури клієнт-серверної системи.

Розробка програмного комплексу FlexSpot спирається на розподілену клієнт-серверну архітектурну модель, яка забезпечує повну ізоляцію шару представлення даних від обчислювальної логіки бекенду. Клієнтська частина функціонує як незалежний односторінковий застосунок (SPA), тоді як серверна розгорнута як автономне джерело прикладного програмного інтерфейсу (REST API). Комунікація між цими двома масивами інфраструктури реалізована за допомогою захищеного протоколу HTTPS, а обмін інформаційними пакетами відбувається виключно у стандартизованому текстовому форматі JSON. Такий підхід усуває жорстку зв'язність модулів. Це дозволяє модернізувати інтерфейс користувача без внесення деструктивних змін у ядро бази даних чи алгоритми обробки бізнес-логики [7].

Поділ системи на ізольовані рівні успішно розв'язує проблему раціонального розподілу обчислювальних потужностей. Браузер клієнта бере на себе завдання з рендерингу графічних компонентів, маршрутизації сторінок та

первинної валідації полів введення на фронтенді. Натомість віддалений сервер під управлінням ОС Linux фокусується на операціях, які вимагають підвищеної надійності та захисту. До них належать розрахунок фінансових квот, виконання транзакційних SQL-запитів до бази даних MySQL, взаємодія з платіжними серверами та регулярний запуск консольних команд за допомогою вбудованого планувальника завдань (Cron). У результаті вебсистема стає гнучкою та горизонтально масштабованою, а мережеве навантаження зводиться до мінімуму, оскільки через канали зв'язку передаються лише чисті структуровані дані замість важких HTML-сторінок.

Для формалізації вимог, візуалізації внутрішньої поведінки застосунку та точного опису взаємодії користувачів із функціональними модулями FlexSpot застосовано уніфіковану мову моделювання UML. Проєктування логічної структури починається з побудови діаграми варіантів використання, яка окреслює межі програмного продукту та визначає ролі дійових осіб. Платформа оперує трьома основними акторами, кожен з яких наділений чіткими правами доступу та власними сценаріями взаємодії з системою [8].

Першим і найбільш масовим актором виступає резидент коворкінгу, який представляє сегмент фізичних осіб. Його ключовими прецедентами є проходження процедури безстанової автентифікації за допомогою сервісів Firebase, перегляд каталогу доступних локацій та пошук вільних місць із використанням інфраструктурних фільтрів. У межах особистого кабінету резидент ініціює створення чернетки бронювання, поповнює баланс внутрішнього цифрового гаманця та взаємодіє з платіжним шлюзом LiqPay. Після успішного підтвердження операції цей актор отримує можливість генерації індивідуальної квитанції-перепустки у форматі PDF, а також доступ до перегляду повної хронології власних фінансових транзакцій.

Другим актором є корпоративний клієнт, який представляє інтереси бізнес-сегменту та юридичних осіб. Ця роль повністю успадковує базовий функціонал звичайного резидента, проте розширює архітектурні можливості системи спеціалізованими бізнес-сценаріями. Корпоративний користувач взаємодіє з

модулем управління пакетними квотами компанії, розподіляючи передплачені години оренди конференц-залів між своїми співробітниками. Крім того, його профілем затребувані прецеденти формування офіційних податкових інвойсів та завантаження структурованої фінансової звітності для бухгалтерії, що вимагає від ядра Laravel генерації складніших агрегованих вибірок із бази даних.

Третім актором є адміністратор локації, для якого спроєктовано повний операційний контроль над коворкінг-центром. Його варіанти використання зосереджені навколо керування інфраструктурним наповненням простору. Адміністратор володіє правами на виконання повного циклу операцій (CRUD) щодо таблиць робочих зон, окремих місць та пакетних тарифів. Ця роль передбачає використання модулів ручного блокування просторів на технічне обслуговування, перегляд теплової карти завантаженості локацій та формування зведених фінансових звітів за довільний часовий інтервал. Варто зауважити, що повний адміністративний функціонал та аналітичні модулі детально закладено на рівні реляційної структури бази даних та відповідних маршрутів REST API, однак розробка візуального клієнтського інтерфейсу адмінпанелі винесена за межі цієї кваліфікаційної роботи і розглядається як перспективний етап масштабування платформи.

Взаємозв'язки між прецедентами всередині архітектури FlexSpot спроєктовані з використанням суворих зв'язків включення (include) та розширення (extend), що дозволяє оптимізувати логіку на етапі розробки бекенду. Наприклад, варіанти використання, пов’язані зі створенням чернетки бронювання або переглядом балансу гаманця, в обов'язковому порядку включають у себе прецедент валідації підпису JWT-токена в проміжному програмному забезпеченні сервера (FirebaseAuthMiddleware). Водночас процес фіналізації замовлення розширюється двома альтернативними сценаріями: безпосереднім списанням накопичених годин із пакетної квоти резидента або проведенням прямої платіжної операції через шлюз LiqPay. Повну декомпозицію логічної структури платформи, що відображає її архітектурні межі та рольову

модель, наведено у вигляді діаграм варіантів використання клієнтської підсистеми на рис. 2.2 та підсистеми адміністрування на рис. 2.3.

Рисунок 2.2. Діаграма варіантів використання «Адміністративна підсистема (Адміністратор локації)»

Рисунок 2.3. Діаграма варіантів використання «Клієнтська підсистема (Резидент та Корпоративний клієнт)»

Для дослідження динаміки міжкомпонентної взаємодії та верифікації інформаційних потоків у процесі резервування робочих зон розроблено діаграму послідовності. Ця модель відображає точний хронологічний порядок обміну повідомленнями та диспетчеризації HTTP-запитів між клієнтським застосунком, серверним REST API, хмарою Firebase та платіжним шлюзом LiqPay. Процес

ініціюється відправкою POST-запиту з React-фронтенду, після чого Middleware-шар Laravel виконує перевірку та валідацію підпису JWT-токена. За умови успішної перевірки, серверне ядро застосовує песимістичне транзакційне блокування обраного часового слота в базі даних MySQL, реєструє чернетку замовлення та запускає лічильник часу експірації чернетки. Фіналізація життєвого циклу транзакції спирається на обробку асинхронного Webhook-повідомлення від еквайрингу з обов'язковою верифікацією цифрового підпису шлюзу на основі порівняння SHA-1 хешів та Base64-декодування отриманих даних. Повну схему взаємодії компонентів у часі наведено на рис. 2.4.

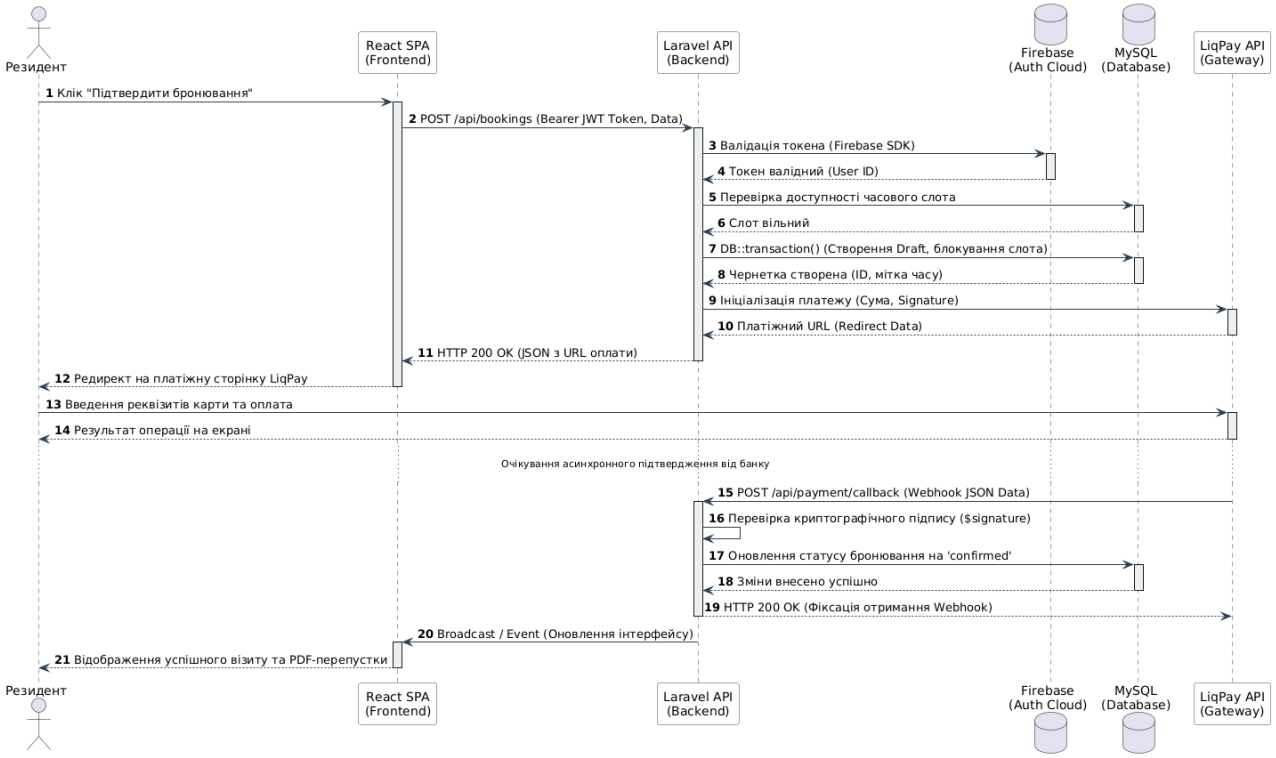


Рисунок 2.4. Діаграма послідовності (Sequence Diagram) процесу бронювання простору

Логічну послідовність алгоритмічних розгалужень та внутрішні процеси ухвалення рішень під час обробки конкурентних запитів деталізовано за допомогою діаграми діяльності (Activity Diagram). Ця модель візуалізує роботу ядра розподілу ресурсів коворкінгу. Алгоритм бекенду виконує покрокову

перевірку обраного інтервалу часу на предмет перетину з наявними записами у базі даних. У разі виявлення конфлікту сервер миттєво перериває операцію та повертає клієнту помилку зі статус-кодом 422 Unprocessable Entity.

За відсутності накладок система виконує атомарну транзакцію: фіксує чернетку із десятихвилинним лімітом блокування простору та генерує редирект на платіжну сторінку. Паралельно із цим фоновий процес планувальника завдань (Cron) безперервно аналізує часові мітки активних чернеток, автоматично анулюючи ті записи, які не отримали підтвердження у встановлений термін. Логіку функціонування ядра обробки замовлень декомпоновано на два взаємопов'язані етапи з метою збереження читабельності графічних матеріалів. Процес первинної валідації та транзакційного резервування слота наведено на рис. 2.5. Наступний етап, що включає фіналізацію фінансової операції та фоновий моніторинг життєвого циклу чернеток, ілюструє рис. 2.6. Зв'язок між двома схемами реалізовано за допомогою спеціального елемента-з'єднувача.

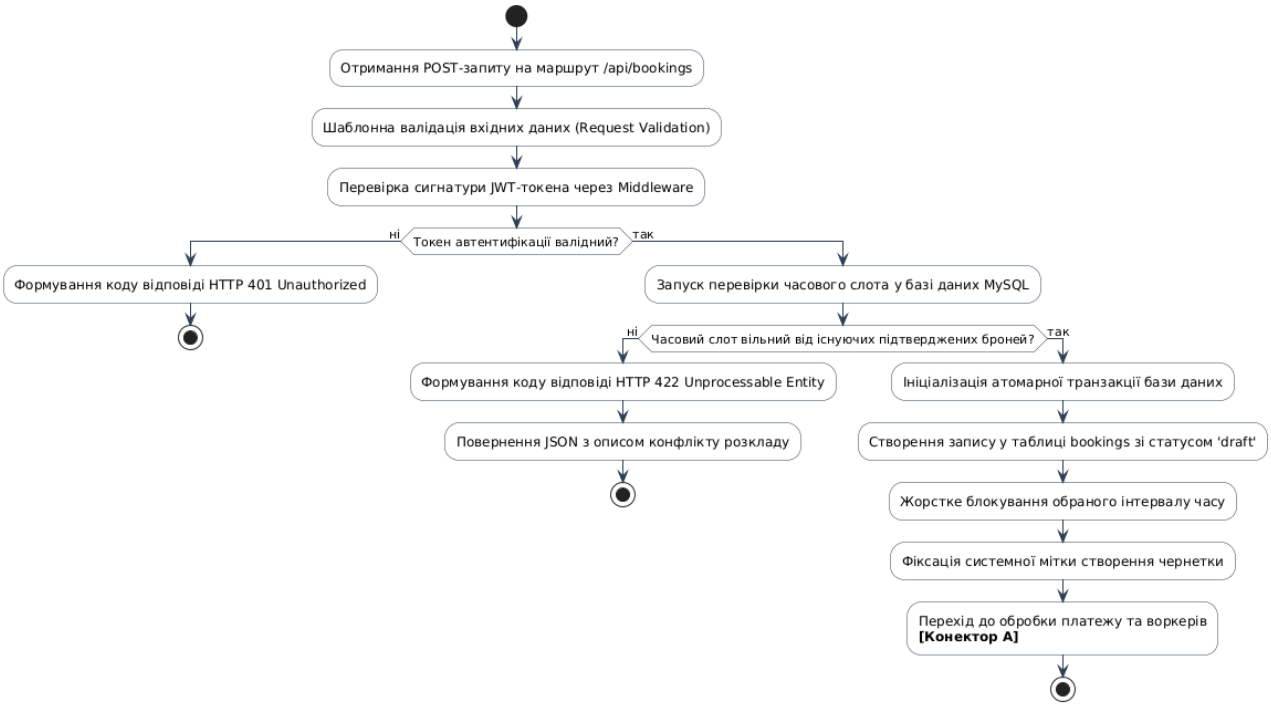


Рисунок 2.5. Діаграма діяльності алгоритму обробки замовлень: первинна валідація

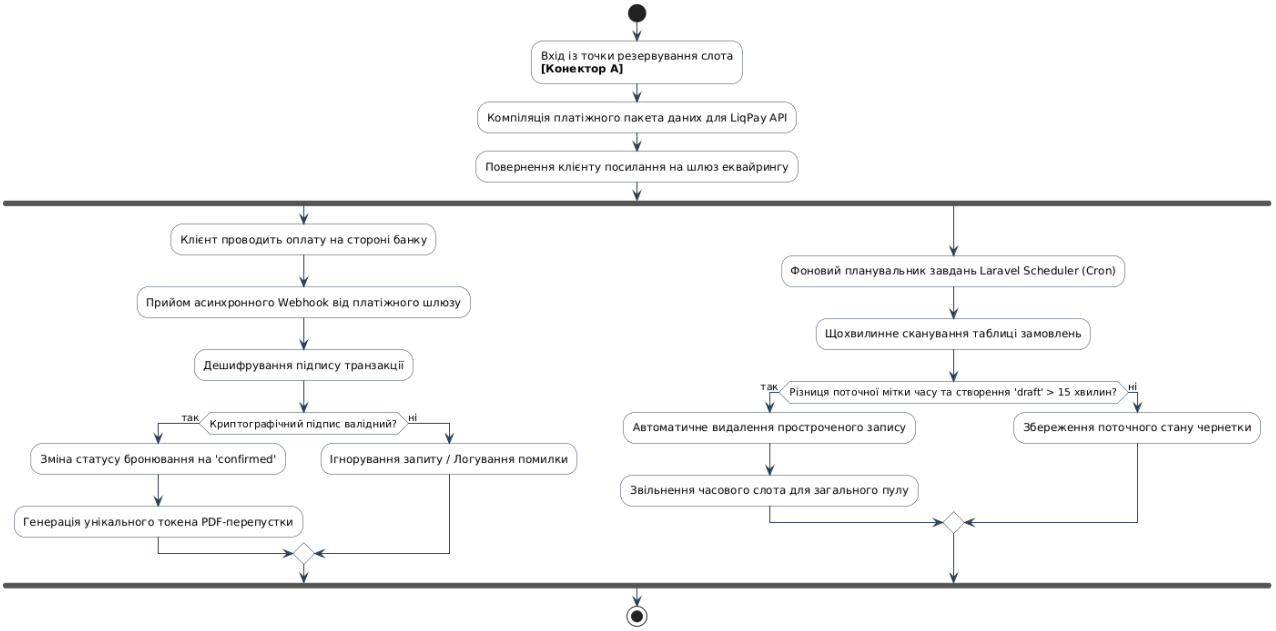


Рисунок 2.6. Діаграма діяльності алгоритму обробки замовлень: асинхронна фіналізація та очищення

Для математично точного опису та фіксації повної часової динаміки станів, у яких може перебувати ключіва сутність системи – замовлення робочої локації, розроблено діаграму станів (State Machine Diagram) [9]. Такий підхід дозволяє повністю ліквідувати логічні конфлікти на рівні бізнес-логіки бекенду ще на етапі архітектурного проєктування. Сутність бронювання володіє складним життєвим циклом, де кожен перехід між станами є строго детермінованим конкретними системними подіями, транзакціями бази даних або діями віддалених серверних воркерів. Моделювання поведінки об'єкта забезпечує абсолютну консистентність реляційних таблиць у MySQL та повністю виключає появу суперечливих чи зациклених записів під час висококонкурентного паралельного доступу резидентів до спільних ресурсів коворкінг-центра.

Початковою точкою життєвого циклу об'єкта є ініціалізація замовлення користувачем через клієнтський інтерфейс React.js. Після відправки вхідного POST-запиту на серверний ендпоінт та успішного проходження шару автентифікації, об'єкт автоматично переходить у стан чернетки (draft). На цьому

етапі ядро бекенду Laravel накладає на запис жорстке часове обмеження тривалістю в десять хвилин. Алгоритм виконує атомарну транзакцію в базі даних та ізолює обраний часовий слот від загального пулу доступних місць, запускаючи внутрішній лічильник часу життя об'єкта (Time to Live). У цей момент простір маркується як тимчасово зайнятий для інших пошукових запитів, захищаючи систему від ризику паралельних накладок на етапі очікування оплати.

Подальша зміна стану сутності цілком залежить від результатів взаємодії з платіжним шлюзом. У разі успішного списання коштів сервери LiqPay надсилають асинхронне повідомлення зворотного виклику (POST Webhook) на контролер системи FlexSpot. Серверне ядро виконує обов'язкову верифікацію цифрового підпису отриманого пакета на основі порівняння обчислених SHA-1 хеш-значень та Base64-декодування вхідних даних. Якщо перевірка успішна, статус замовлення негайно змінюється на підтверджений (confirmed). Це кінцевий успішний стан у життєвому циклі бронювання. Перехід у цей статус автоматично ініціює роботу суміжних програмних модулів: система списує пакетні квоти резидента, оновлює транзакційні логи внутрішнього гаманця (Wallet) та генерує індивідуальну цифрову перепустку у форматі PDF із унікальним токеном доступу.

Альтернативний сценарій передбачає автоматичне переведення об'єкта в стан скасованого замовлення (cancelled). Головним тригером для цієї події виступає робота автономної підсистеми очищення на базі планувальника завдань Laravel Task Scheduling. Якщо протягом встановленого десятихвилинного ліміту банківський еквайринг не фіксує оплату через відмову користувача, брак коштів на рахунку чи технічну помилку на стороні банку, фоновий процес планувальника в автоматичному режимі змінює статус чернетки на скасований. Ця операція супроводжується миттєвим звільненням заблокованого часового слота у загальний пул інфраструктури коворкінгу. З міркувань безпеки, фінансового обліку та цілісності даних зворотний рух об'єкта з фінальних термінальних станів (confirmed або cancelled) у проміжні чернетки є абсолютно неможливим, що надійно захищає екосистему від повторних помилкових

мутацій. Графічне представлення життєвого циклу сутності бронювання наведено на рис. 2.7.

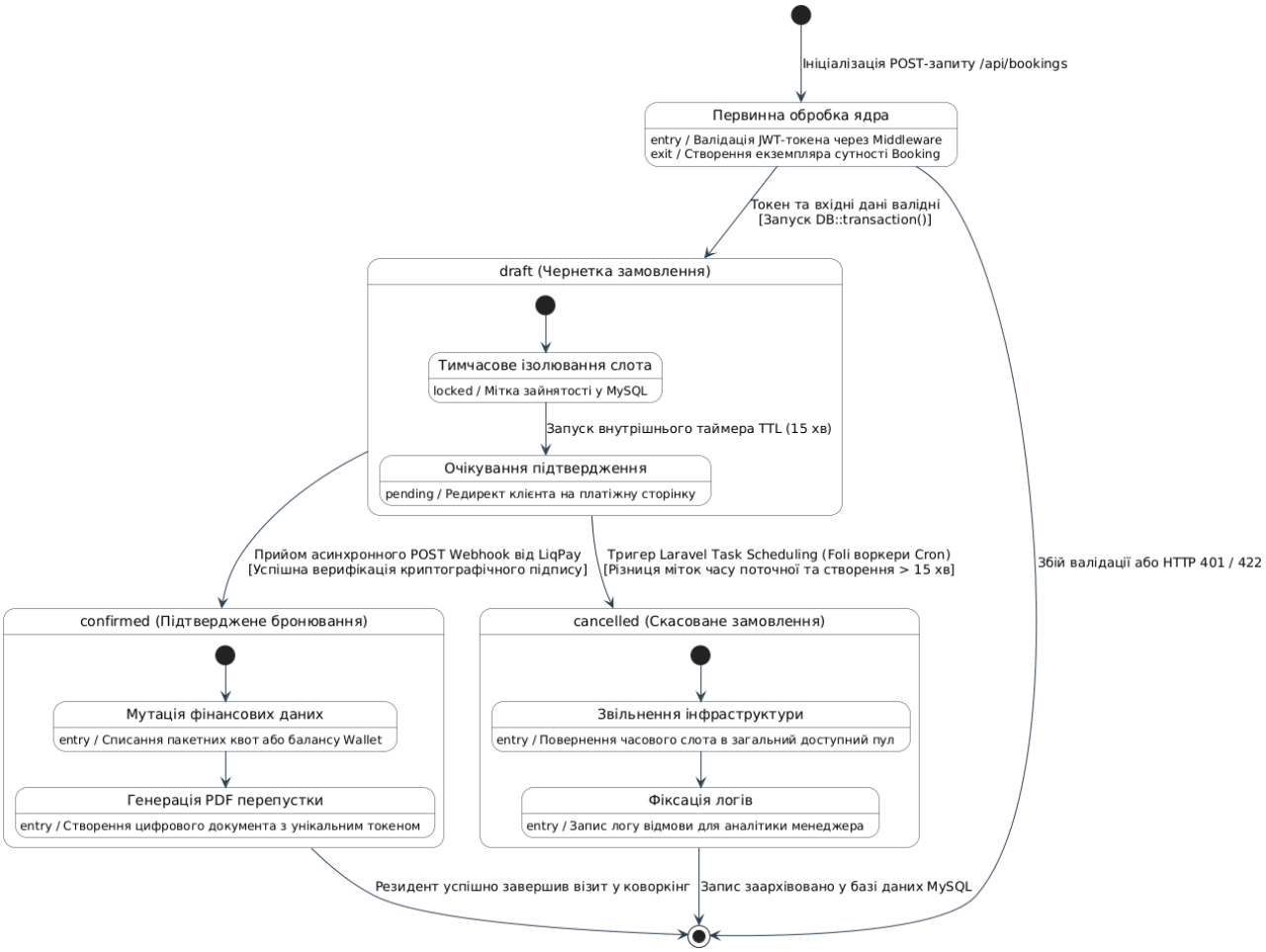


Рисунок 2.7. Діаграма станів (State Machine Diagram) сутності бронювання місць

Для комплексного аналізу статичної організації програмного коду, визначення меж модулів та візуалізації зв'язків між логічними елементами вебсистеми FlexSpot розроблено діаграму компонентів. Оскільки архітектура застосунку базується на концепції повного розділення фронтенду та бекенду (Decoupled Architecture), це моделювання дозволяє зафіксувати інтерфейси взаємодії між незалежними інфраструктурними одиницями. Логічна структура клієнтської частини, розгорнутої як односторінковий застосунок (SPA) на базі бібліотеки React.js, декомпонована на набір ізольованих користувацьких

інтерфейсів та сервісних шарів. Головний модуль представлення включає компоненти інтерактивної карти локацій, віджети вибору часових слотів розкладу та особислий кабінет резидента, тоді як взаємодія з адміністративними ендпоінтами закладена у загальну структуру клієнтських маршрутів взаємодії з API. Комунікація з бекендом ізольована всередині сервісного клієнта на основі бібліотеки Axios, який асинхронно транслює мережеві HTTPS-запити до серверного ядра, додаючи до заголовків авторизаційні JWT-токени, отримані безпосередньо через інтегрований сервісний шар Firebase SDK.

Серверна частина, побудована на базі фреймворку Laravel, представлена на діаграмі у вигляді автономного монолітного ядра, яке комунікує із зовнішнім світом через чітко визначені точки входу. Шар маршрутизації REST API розподіляє вхідні JSON-пакети даних між відповідними контролерами бізнес-логіки. Безпека закритих маршрутів гарантується проміжним програмним забезпеченням FirebaseAuthMiddleware, яке виконує безстанову валідацію вхідних токенів автентифікації. Ядро системи містить окремі функціональні сервіси, що відповідають за генерацію PDF-документів квитанцій, фінансові розрахунки змішаної оплати, взаємодію з LiqPay API та менеджмент корпоративних квот. Рівень абстракції Eloquent ORM інкапсулює роботу з реляційними моделями та забезпечує прямий зв'язок із шаром збереження даних.

Зважаючи на розподілену архітектуру розроблюваної системи FlexSpot та значну кількість функціональних модулів, компонентну структуру платформи було деталізовано та розділено на дві взаємопов’язані підсистеми. На рисунку 2.8 представлено архітектуру клієнтської частини (React SPA) та її зовнішні інтеграції. Внутрішню логіку обробки запитів, структуру сервісів серверної частини (Laravel REST API) та їхню взаємодію з базою даних MySQL наведено на рисунку 2.9. Сполучною ланкою між підсистемами виступає єдиний інтерфейс REST API маршрутів. Узагальнену логічну архітектуру організації вихідного коду, шарів абстракції та міжкомпонентних залежностей наведено на рис. 2.8–2.9.

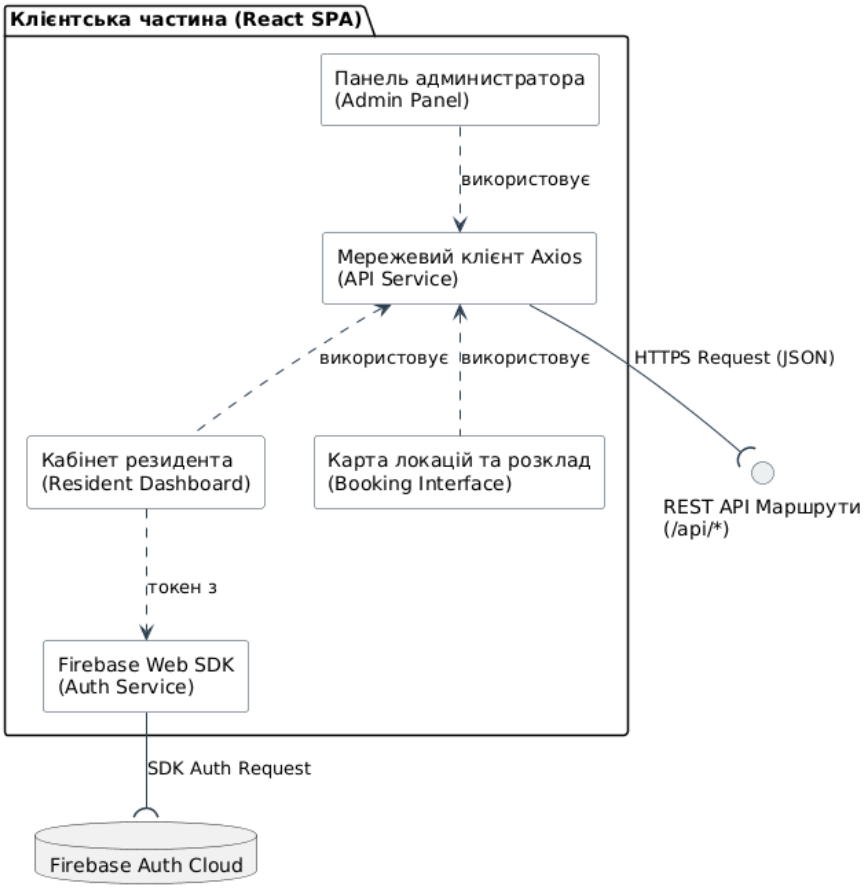


Рисунок 2.8. Діаграма компонентів (Component Diagram) архітектури клієнтської частини та взаємодії з API

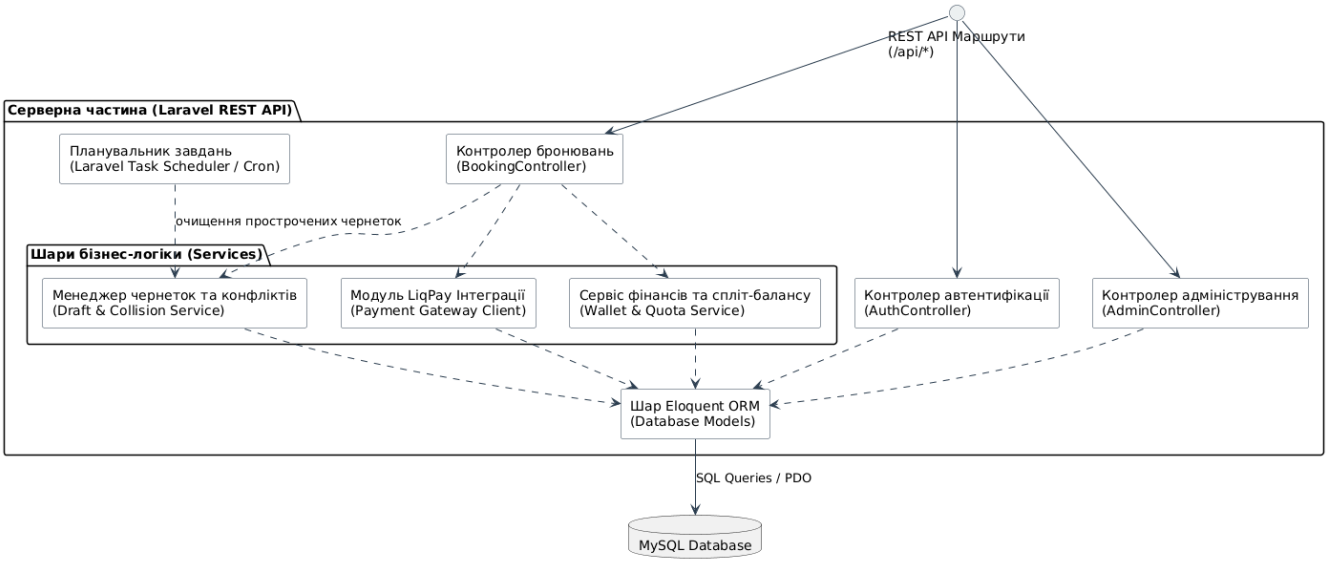


Рисунок 2.9. Діаграма компонентів (Component Diagram) внутрішньої архітектури серверної частини (Laravel)

Фізичне розгортання вебсистеми, топологія обчислювальних вузлів та мережева архітектура взаємодії програмно-апаратних комплексів формалізовані за допомогою діаграми розгортання (Deployment Diagram). Ця модель описує середовище функціонування готового програмного продукту та визначає технічні вимоги до обчислювальних ресурсів. Клієнтський рівень представлений абстрактним вузлом у вигляді персонального комп'ютера або мобільного пристрою резидента, на якому запущено сучасний веббраузер із підтримкою стандартів ECMAScript. Браузер виконує інтерпретацію скомпільованого клієнтського пакета React SPA, що завантажується за замовчуванням під час первинного звернення користувача до системи.

Центральним елементом топологічної структури виступає виділений віддалений віртуальний сервер (VPS) під управлінням операційної системи сімейства Linux. На системному рівні цей вузол містить кілька ізольованих середовищ та системних демонів, що функціонують у синергії. Первинний прийом мережевого трафіку здійснює високопродуктивний вебсервер Nginx, який виступає в ролі зворотного проксі (Reverse Proxy). Він забезпечує термінацію захищеного TLS-з'єднання, віддачу статичних файлів фронтенду та перенаправлення динамічних API-запитів на шлюз PHP-FPM.

Сервер застосунків Laravel опрацьовує бізнес-логіку, взаємодіючи через локальний сокет із сервером реляційної бази даних MySQL. Додатково на рівні операційної системи налаштовано планувальник завдань (Cron), який щохвилини ініціює виконання консольних команд Laravel для автоматичного моніторингу та очищення протермінованих чернеток бронювання. Взаємодія з хмарною інфраструктурою Firebase та платіжним сервером LiqPay здійснюється через глобальну мережу Інтернет за допомогою захищених HTTPS-запитів до зовнішніх API. Повну фізичну топологію розгортання та архітектуру мережевих вузлів екосистеми FlexSpot наведено на рис. 2.10.

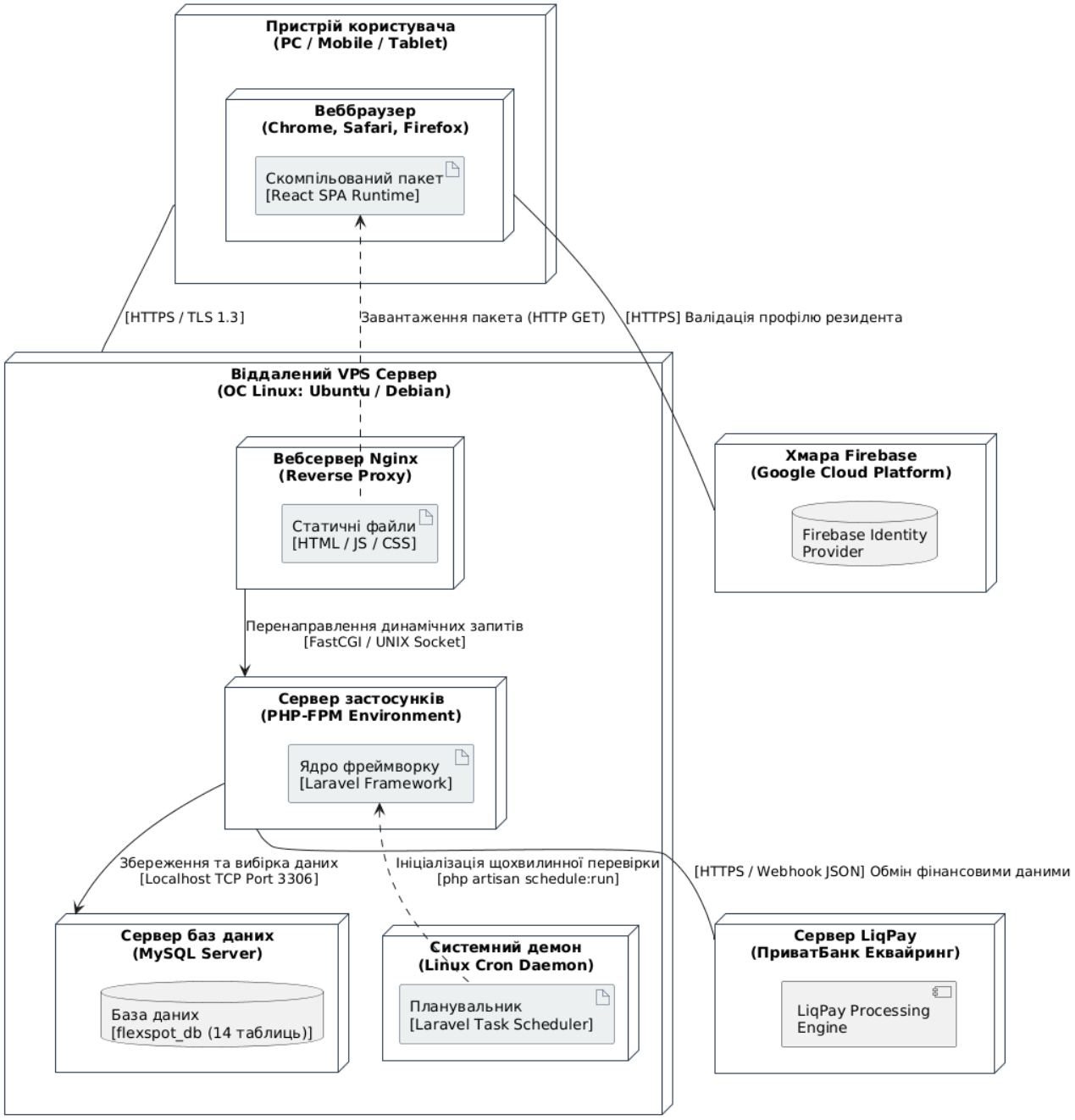


Рисунок 2.10. Діаграма розгортання (Deployment Diagram) апаратної топології системи

2.3. Проєктування структури реляційної бази даних.

Проєктування шару збереження даних вебсистеми FlexSpot повністю базується на застосуванні реляційного підходу, оскільки специфіка предметної галузі висуває підвищені вимоги до консистентності інформації. Базові бізнес-процеси платформи, до яких належать фінансовий облік внутрішніх гаманців

резидентів, контроль корпоративних лімітів та двоетапне резервування часових слотів, потребують суворого дотримання властивостей транзакційності (ACID). Використання нереляційних сховищ у цій архітектурі є недоцільним через високі ризики виникнення суперечливих станів під час обробки конкурентних паралельних запитів до розкладу спільного простору коворкінгу.

Реляційна модель MySQL (із застосуванням рушія InnoDB) гарантує абсолютну атомарність кожної фінансової або інфраструктурної операції. Якщо на проміжному етапі обробки асинхронних платіжних вебхуків від шлюзу LiqPay у методі handleCallback фіксується мережевий збій або помилка валідації підпису, серверне ядро Laravel за допомогою механізму DB::transaction виконує безпечний відкат (rollback) змін. Логічні аномалії у сховищі повністю виключаються.

Важливим етапом проєктування екосистеми є глибока нормалізація її реляційної структури. Цей процес спрямований на послідовне усунення надлишковості даних та оптимізацію виконання складних SQL-вибірок. На етапі зведення таблиць до першої нормальної форми (1NF) було забезпечено повну атомарність усіх без винятку атрибутів сутностей. Унікальні ідентифікатори користувачів, текстові описи локацій коворкінгу, мітки часу початку та завершення візитів резидентів розділені на незалежні неподільні поля таблиць. Створення складених чи повторюваних масивів у межах одного рядка бази даних є неприпустимим за архітектурними вимогами проєкту.

Для відповідності критеріям другої нормальної форми (2NF) було ліквідовано всі часткові залежності неключових полів від складених первинних ключів. Усі сутності предметної галузі, що мають власні ідентифікатори, винесені в окремі таблиці сховища. Технічні характеристики, місткість та зручності робочих зон повністю відокремлені від таблиці конкретних бронювань. Кожен інфраструктурний об'єкт існує в базі даних в єдиному екземплярі, а зв'язки між ними реалізуються через систему зовнішніх ключів (foreign keys).

Остаточним кроком структурної оптимізації стало зведення архітектури даних до третьої нормальної форми (3NF). Головною метою цього етапу є ліквідація транзитивних залежностей, за яких неключові атрибути залежать від

інших неключових атрибутів. Прикладом є розмежування загальних тарифних планів та безпосередньо сутностей робочих місць. Вартість години оренди або параметри пакетних квот компаній не зберігаються у таблиці конкретної кімнати переговорів, а винесені в окрему довідкову структуру тарифів. Точкова зміна ціни в одному пакеті більше не призводить до каскадного викривлення історичних даних про вже підтверджені замовлення клієнтів.

Завдяки доведенню структури бази даних, що складається з чотирнадцяти реляційних таблиць, до третьої нормальної форми, платформа FlexSpot повністю захищена від класичних типів архітектурних аномалій [10]:

- Аномалія вставки: усунена завдяки тому, що адміністратор коворкінгу має технічну можливість додавати нові типи пакетних квот, локацій чи робочих місць у базу даних ще до того, як перший резидент здійснить їхнє реальне бронювання або оплату;
- Аномалія оновлення: нівелюється точковим редагуванням конкретних полів. Зміна юридичних реквізитів або назви корпоративного B2B-клієнта відбувається строго в одному рядку таблиці компаній, а зміни миттєво відображаються для всіх пов'язаних профілів співробітників через зовнішні ключі;
- Аномалія видалення: ліквідована за рахунок суворого налаштування зв'язків моделей. Автоматичне очищення консольною командою планувальника завдань протермінованої десятихвилинної чернетки бронювання видаляє лише тимчасовий запис із розкладу. Обліковий запис самого резидента, стан його гаманця та фізичні параметри робочого простору залишаються в абсолютній цілісності.

Таблиця 2.2. Аналіз інженерних рішень усунення аномалій даних за допомогою 3НФ

Тип аномалії	**Прояв у ненормалізованій структурі (приклад)**	**Інженерне рішення у FlexSpot (3НФ)**
Аномалія вставки (Insertion)	Неможливо додати нову робочу зону (zones) або її ціну, поки в коворкінгу не з'явиться перший резидент, який її забронює.	Повний поділ сутностей. Створено незалежний довідник zones, що дозволяє адмініструвати простори незалежно від наявності бронювань.
Аномалія оновлення (Update)	При зміні адреси або назви локації коворкінгу довелося б перезаписувати ці дані у кожному рядку таблиці історії бронювань.	Винесення даних у таблицю locations. В інших таблицях зберігається лише числовий ідентифікатор location_id (зовнішній ключ).
Аномалія видалення (Deletion)	При видаленні аккаунта користувача з бази даних автоматично стерлася б уся фінансова звітність та історія транзакцій компанії.	Впровадження зв'язків із обмеженням ON DELETE SET NULL для фінансового блоку таблиці transactions.

Логічна декомпозиція бази даних вебсистеми FlexSpot передбачає розподіл 14 нормалізованих таблиць за чотирма взаємопов'язаними функціональними доменами. Взаємодія між ними забезпечується системою зовнішніх ключів та індексів, що дозволяє оптимізувати SQL-запити до СУБД MySQL та ізолювати обчислювальні процеси на сервері.

Домен користувачів та автентифікації містить таблиці users та roles із зв'язком «один-до-багатьох». Оскільки верифікація облікових даних делегована хмарі Firebase, локальна таблиця users зберігає лише базові профілі резидентів та

їхні ідентифікатори Firebase UID. Це дозволяє проміжному шару FirebaseAuthMiddleware на бекенді безстаново ідентифікувати користувача та розмежовувати права доступу між резидентами, офіс-менеджерами та адміністраторами платформи.

Домен інфраструктурного каталогу описує фізичні ресурси коворкінгу і містить таблиці locations, zones, desks, amenities та проміжну таблицю зв'язку location_amenity. Відношення «один-до-багатьох» між locations та zones відображає наявність кількох просторів (Open Space, кабінети, переговорні кімнати) в межах однієї будівлі коворкінг-центру. Робочі зони наповнюються конкретними столами та місцями із таблиці desks, а для прив'язки оснащення та зручностей реалізовано зв'язок «багато-до-багатьох» через проміжну реляційну структуру.

Домен ядра чернеток та бронювань забезпечує роботу сервісу розкладу і базується на таблиці bookings. У ній фіксуються ідентифікатори користувачів, обраних робочих місць чи зон, загальна вартість, а також поточний статус оренди (draft, confirmed, cancelled). Це дозволяє серверному коду контролювати сітку доступності простору під час обробки нових запитів і надійно блокувати спроби конкурентних накладок (овербукінгу).

Фінансовий домен та менеджмент B2B-квот призначений для автоматизації розрахунків і охоплює таблиці companies, wallets, transactions та user_packages. Корпоративні клієнти реєструються в таблиці companies, яка через зв'язок «один-до-багатьох» асоціюється з працівниками в users. Баланс фізичних осіб контролюється через таблицю wallets, що знаходиться у відношенні «один-до-одного» з користувачем. Високорівневу декомпозицію реляційної структури FlexSpot та розподіл таблиць за архітектурними доменами представлено на рис. 2.11.

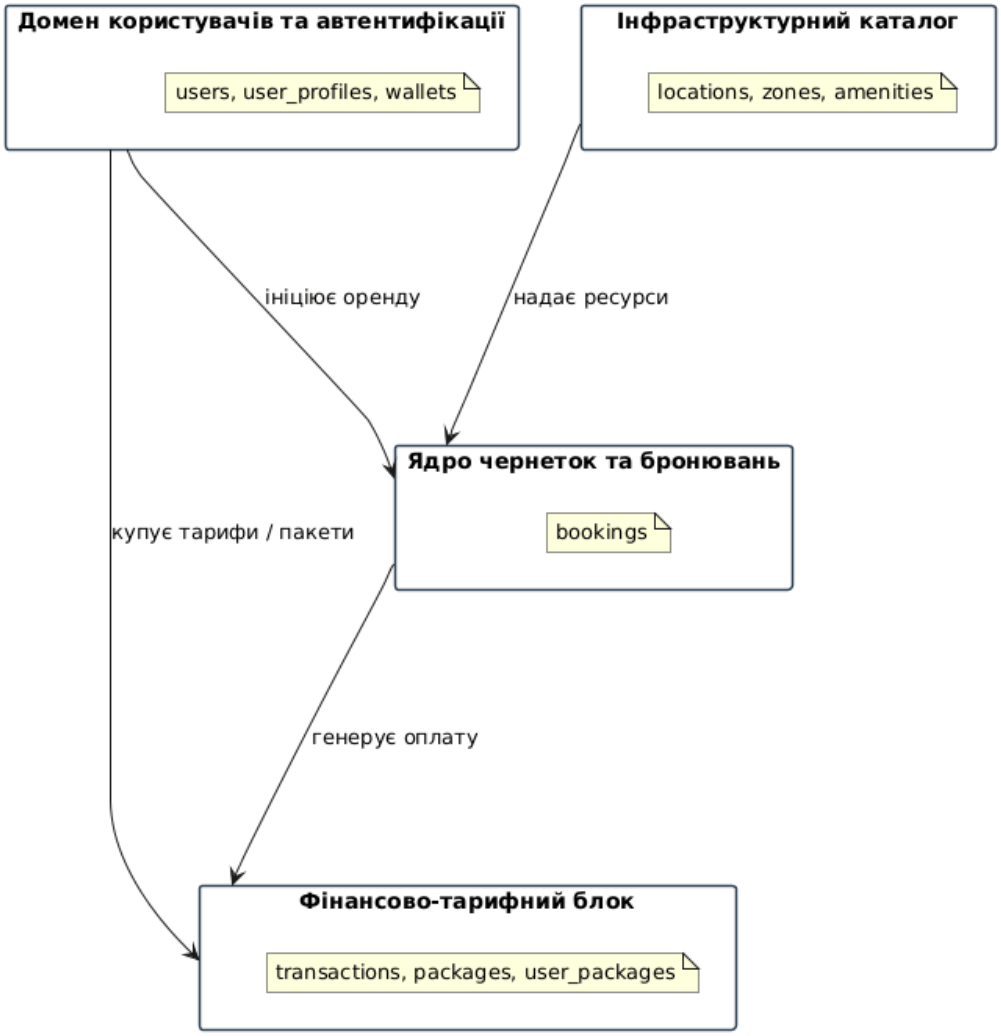


Рисунок 2.11. Схема високорівневої декомпозиції структури даних за функціональними блоками.

Будь-яка зміна балансу, включаючи безготівкове поповнення рахунку через платіжний еквайринг або автоматичне списання за оренду мітинг-руму, обов'язково реєструється в транзакційній таблиці логів transactions. Для управління лімітами юридичних осіб розроблено таблицю user_packages, яка зберігає обсяги передплаченого часу та пов'язана з довідником тарифних планів packages. Шар абстракції Eloquent ORM використовує ці реляційні зв'язки для автоматичного контролю залишку годин компанії під час ініціалізації бронювання її співробітниками. Структурний розподіл розроблених реляційних таблиць бази даних вебсистеми FlexSpot за логічними доменами та їхніми цільовими операційними задачами представлено в табл. 2.3.

Таблиця 2.3. Розподіл реляційних таблиць FlexSpot за функціональними доменами

Назва домену (блоку)	**Системні таблиці, що входять до блоку**	**Основна функціональна задача**
Автентифікація та профілі	users, user_profiles, wallets	Збереження Firebase UID, балансу гаманця та B2B/B2C реквізитів резидентів.
Інфраструктурний каталог	locations, zones, amenities	Опис фізичних характеристик офісів, кімнат переговорів та їх оснащення.
Чернетки та бронювання	bookings	Фіксація часових слотів, керування життєвим циклом чернеток та статусів оренди.
Фінансово-тарифний модуль	transactions, packages, user_packages	Інтеграція з LiqPay, облік куплених B2B-пакетів та списання лімітів годин.

Фізична реалізація розробленої схеми в середовищі системи управління базами даних MySQL орієнтована на мінімізацію дискового простору та прискорення індексації рядків. Основним рушієм зберігання для всіх чотирнадцяти таблиць обрано InnoDB [11]. Дане рішення зумовлене повною підтримкою зовнішніх ключів, механізмів транзакційного блокування на рівні рядків та високою стійкістю до системних збоїв. Сувора типізація атрибутів безпосередньо впливає на швидкість обчислень серверного ядра. Для всіх первинних ключів встановлено тип даних BIGINT UNSIGNED із атрибутом AUTO_INCREMENT. Цей крок повністю унеможливлює переповнення лічильників у таблицях логів та часових слотів розкладу, де обсяг записів зростає найшвидше.

Особливу інженерну увагу приділено фінансовим атрибутам у таблицях гаманців та транзакцій. Використання типів із плаваючою комою (FLOAT, DOUBLE) було повністю відкинуто через високі ризики накопичення похибок

округлення під час математичних операцій. Натомість для зберігання сум та вартості тарифів впроваджено фіксований точний тип даних DECIMAL(10, 2).

Для роботи з часовими інтервалами, фіксації періодів оренди та забезпечення десятихвилинного ліміту утримання чернеток замовлень застосовано тип даних DATETIME. Важливо підкреслити, що на відміну від типу TIMESTAMP, тип DATETIME є бінарно стабільним і зберігає абсолютні значення календаря у незмінному вигляді, тобто він технічно не підтримує автоматичну конвертацію часу на основі динамічних зміщень часового поясу самого сервера СКБД. Тому для збереження цілісності даних у системі FlexSpot приведення всіх часових предикатів до єдиного часового стандарту (UTC) виконується суворо на рівні прикладного програмного коду (за допомогою вбудованого інструментарію бібліотеки Carbon у Laravel) безпосередньо перед надсиланням SQL-запиту на збереження. Це гарантує абсолютну точність порівняння часових діапазонів і захищає систему від логічних збоїв розкладу незалежно від локальних налаштувань часу на фізичному сервері. Рядові текстові поля обмежені типом VARCHAR від 64 до 255 символів, що оптимізує розмір індексних сторінок.

Цілісність реляційних даних та автоматичний захист від руйнування логічної структури забезпечуються на рівні обмежень зовнішніх ключів (FOREIGN KEY CONSTRAINTS). Коли фоновий процес планувальника завдань анулює або очищує записи протермінованих чернеток, СКБД MySQL автоматично видаляє пов'язані проміжні часові інтервали завдяки активованому режиму ON DELETE CASCADE. Навпаки, для сутностей предметної області, які мають безпосередню фінансову цінність або історичну значущість для стрічки аудиту, застосовано суворий захисний режим ON DELETE RESTRICT. Це повністю унеможливлює випадкове каскадне видалення профілів B2B-компаній чи облікових записів резидентів за наявності активних фінансових транзакцій, проведених платежів або діючих абонементів у таблиці user_packages.

Для суттєвого прискорення пошукових операцій під час генерації розкладу та оптимізації виконання складних SQL-запитів із фільтрацією, поля зовнішніх ключів, операційних статусів та календарних дат проіндексовані у СКБД за

допомогою збалансованих індексів типу B-Tree. Крім того, на критичні поля, такі як унікальні ідентифікатори Firebase UID користувачів, накладено обмеження унікальності UNIQUE. Повну фізичну структуру реляційної бази даних вебсистеми FlexSpot, включаючи назви таблиць, їхні атрибути, первинні та зовнішні ключі, типи даних та логічні зв'язки між сутностями предметної галузі, продемонстровано у вигляді деталізованої ER-діаграми на рис. 2.12.

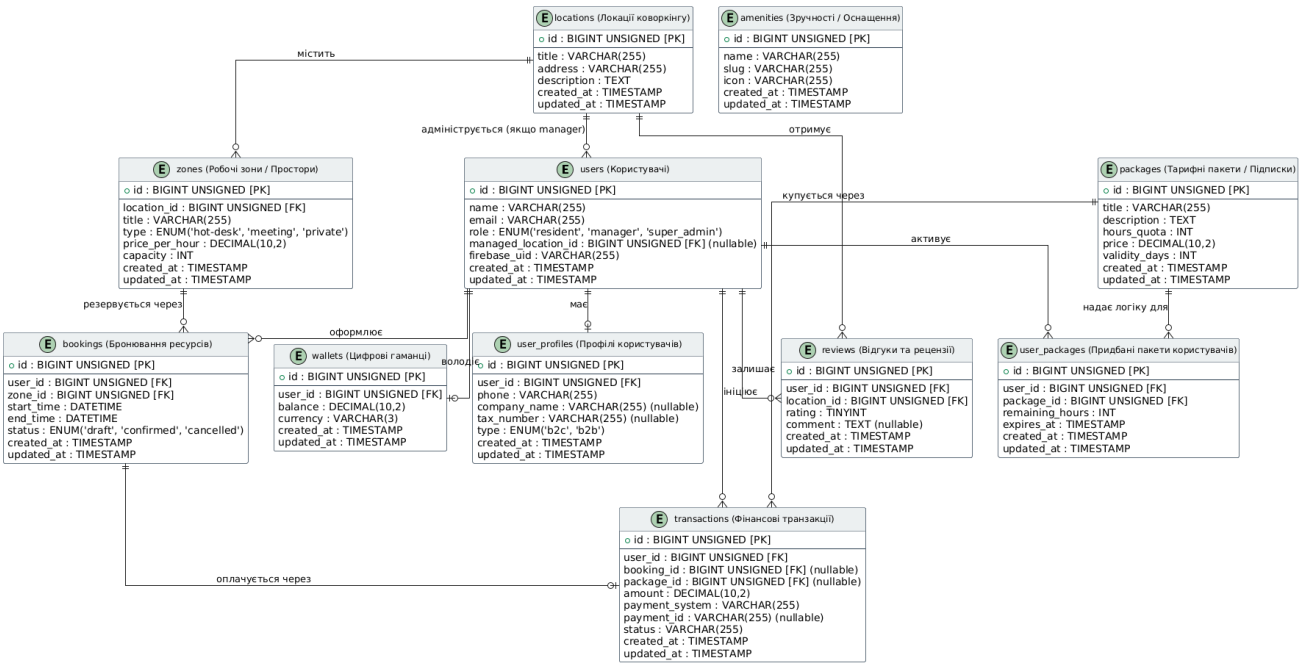


Рисунок 2.12. ER-діаграма фізичної структури реляційної бази даних вебсистеми FlexSpot

2.4. Проєктування інтерфейсу користувача (UI/UX).

Проєктування інтерфейсу користувача вебсистеми FlexSpot підпорядковане меті мінімізації когнітивного навантаження та створення безперервного користувацького досвіду під час взаємодії з функціональними модулями автоматизації коворкінгу. Ергономіка взаємодії людини з комп'ютером розроблена на основі глибокого аналізу поведінкових патернів цільових аудиторій системи. Клієнтська частина платформи оперує двома кардинально протилежними сценаріями використання, що вимагає впровадження гнучких макетів відображення даних.

Для основної маси користувачів – резидентів коворкінгу – застосовано стратегію проєктування Mobile-First. Оскільки бронювання часових слотів або перевірка доступності мітинг-румів найчастіше виконуються безпосередньо з мобільних пристроїв або смартфонів під час перебування у просторі, мобільний макет інтерфейсу є пріоритетним. Всі елементи навігації, інтерактивні схеми та платіжні кнопки оптимізовані під сенсорне керування, мають велику площу натискання та підтримують інтуїтивні жести гортання. Навпаки, для панелі адміністрування та кабінетів корпоративних клієнтів пріоритетом виступає широкоформатний настільний макет Desktop-First. Менеджери локацій потребують одночасного відображення великих аналітичних таблиць, теплових карт завантаженості та графіків фінансової звітності. Це вимагає максимального використання робочої площі моніторів високої роздільної здатності [12].

Релізація клієнтської логіки інтерфейсу спирається на компонентно-орієнтований підхід бібліотеки React.js. Кожен візуальний елемент – від простого текстового поля введення до складної сітки вибору розкладу – розроблений як незалежний модуль. Це забезпечує візуальну консистентність усього застосунку. Колірна палітра інтерфейсу та контрастність елементів підібрані відповідно до міжнародних стандартів доступності вебконтенту Web Content Accessibility Guidelines. Візуальне навантаження збалансоване за рахунок використання векторних графічних елементів з колекції Lucide React. Векторні іконки адаптуються під будь-яку щільність пікселів екрана без втрати чіткості. Одночасно з цим архітектура збірки проєкту через Vite відсікає невикористані графічні ресурси. Клієнт отримує надзвичайно швидкий інтерфейс.

Інформаційна архітектура платформи має чітку трирівневу ієрархію, яка візуалізує карту сайту та розділяє інформаційні потоки. Перший рівень – публічна зона, що містить маркетингові сторінки, загальний каталог локацій та інформацію про тарифи. Ця зона доступна неавторизованим відвідувачам. Другий рівень – захищений особистісний кабінет резидента, доступ до якого відкривається виключно після успішної автентифікації. Тут зосереджена персональна інформація: баланс внутрішнього гаманця, історія бронювань,

інструменти керування B2B квотами компанії та генератор PDF перепусток. Третій рівень – ізольована панель адміністрування. Вона має окрему навігаційну структуру і призначена для CRUD операцій, керування інфраструктурою та перегляду теплових карт завантаженості.

Моделювання користувацьких шляхів дозволяє детально проаналізувати логіку руху користувача крізь інтерфейсні екрани. Ключовий сценарій взаємодії резидента із системою починається з екрана автентифікації. Процес обробки вхідних даних делеговано хмарі Firebase. Після миттєвої перевірки профілю користувач перенаправляється в особистий кабінет до інтерактивної карти робочих зон. Вибір зони активує динамічний віджет розкладу, де користувач відзначає потрібні часові слоти. Після кліку на кнопку підтвердження інтерфейс блокує обрані ресурси на п'ятнадцять хвилин, відображаючи таймер зворотного відліку. Користувач переходить до платіжної форми, де обирає метод списання – кошти з гаманця або пряму оплату. Вибір прямої оплати ініціює відкриття вбудованого віджета LiqPay API. Після успішного завершення транзакції та отримання асинхронного підтвердження сервером, інтерфейс автоматично оновлює стан екрана, переводячи користувача на сторінку успішного візиту з можливістю завантаження згенерованої PDF перепустки.

Деталізація архітектури користувацького інтерфейсу FlexSpot охоплює специфікацію ключових екранних форм, які матеріалізують розроблену бізнес-логіку на стороні клієнта. Кожна сторінка спроєктована як сукупність реактивних компонентів під управлінням єдиного шару стану застосунку. Мутації даних на сервері Laravel миттєво викликають локальне перемальовування відповідних інтерфейсних блоків без повного оновлення сторінок браузера. Основний упор зроблено на три критичні функціональні вузли: модуль інтерактивного бронювання з графіком розкладу, персональний кабінет резидента з фінансовим профілем та аналітичну панель управління для адміністратора локації.

Центральним елементом інтерфейсу користувача є модуль інтерактивної карти коворкінгу, інтегрований із сіткою вибору часових слотів. Цей блок поєднує в собі векторну схему зон простору та динамічну часову шкалу у форматі

Timeline layout. Користувач візуально обирає необхідну зону, що ініціює асинхронний запит через Axios-клієнт для отримання актуальних замовлень з таблиці booking_slots на поточну дату. Графічний макет головного екрана вибору місць та часових інтервалів наведено на рис. 2.13.

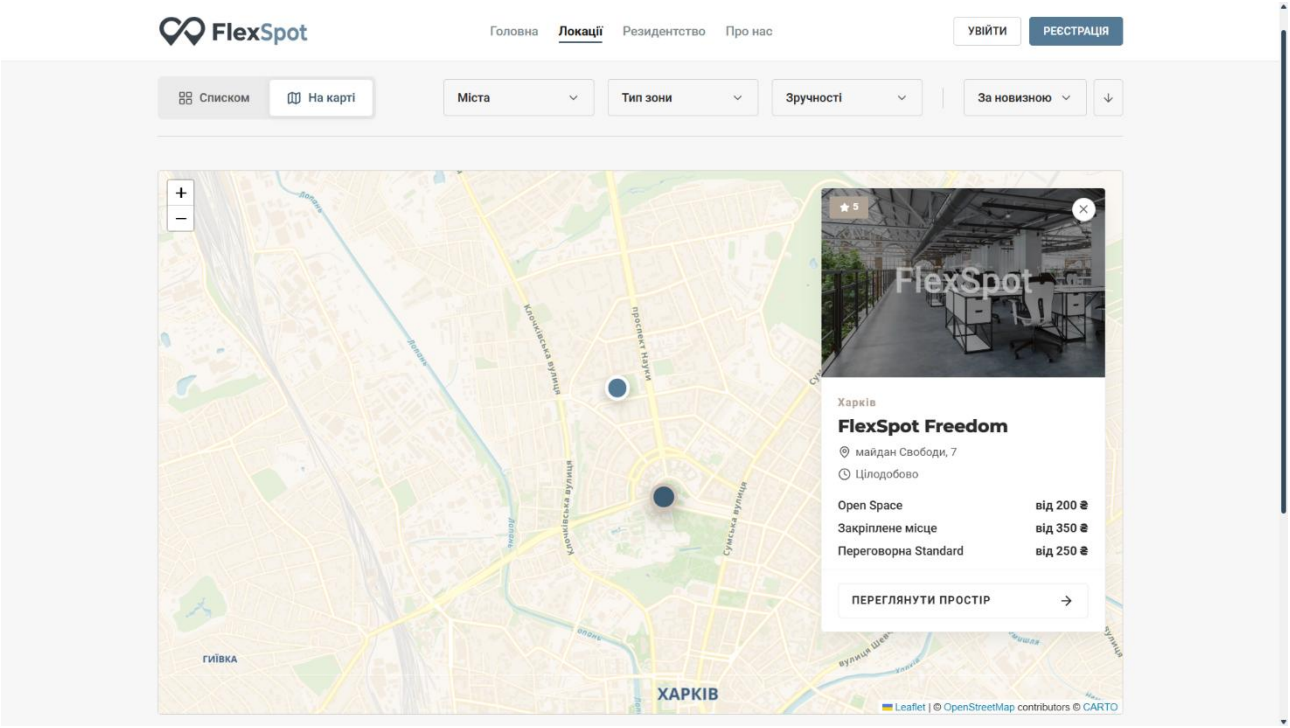


Рисунок 2.13. Інтерфейсна форма інтерактивної карти

Екранна форма особистого кабінету резидента зосереджена навколо фінансових інструментів та управління персональною інфраструктурою. Інтерфейс у реальному часі відображає точний баланс цифрового гаманця, зчитуючи дані з реляційної моделі wallets. Для корпоративних користувачів передбачено окремий кастомний віджет, який демонструє залишок доступних годин із пакетної квоти компанії. Сторінка містить інтерактивну стрічку фінансової історії та списки активних і минулих візитів. Будь-яка зміна стану, наприклад, безготівкове поповнення рахунку або списання коштів за оренду мітинг-руму, викликає миттєве оновлення числових показників на екрані. Безпека платежів забезпечується інтегрованим платіжним вікном LiqPay API, яке відкривається у вигляді безпечного модального фрейму всередині SPA-

застосунку. Після успішного завершення операції на екрані кабінету стає доступною кнопка для завантаження згенерованої PDF-перепустки. Візуальну структуру кабінету резидента приведено на рис. 2.14.

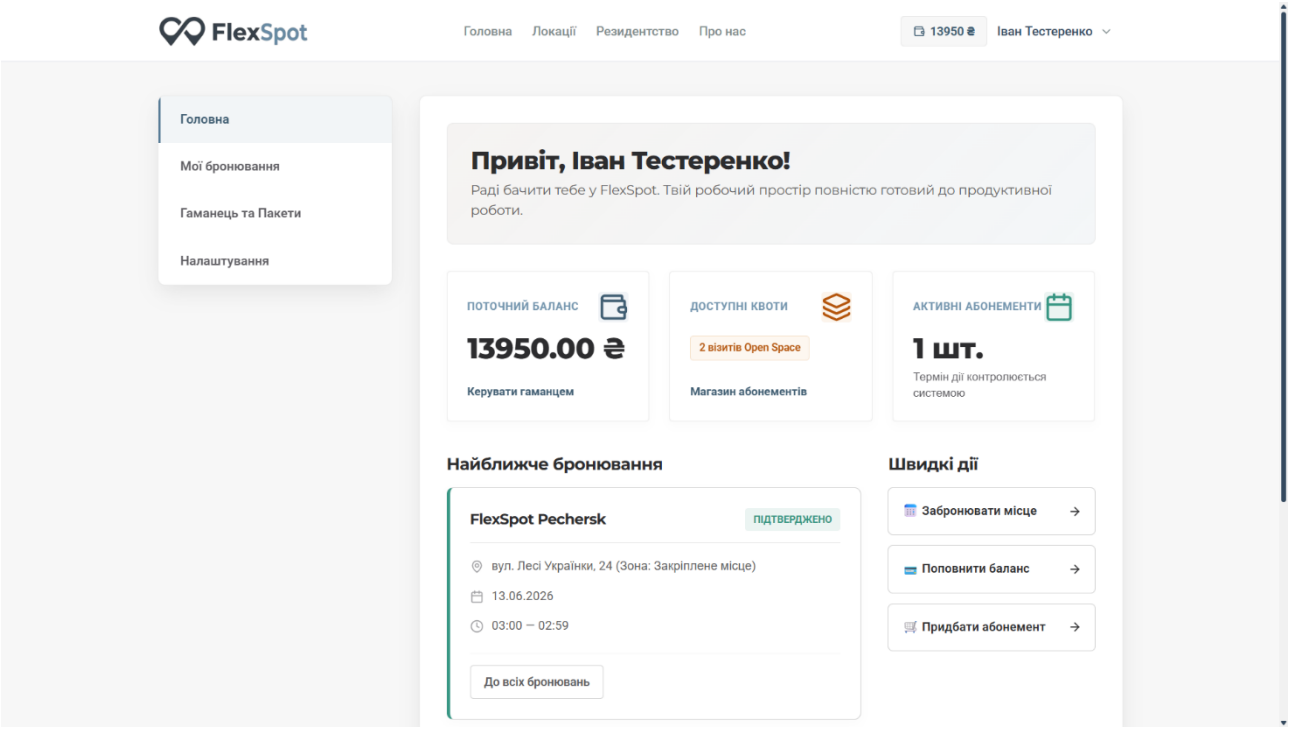


Рисунок 2.14. Інтерфейсна форма особистого кабінету резидента коворкінгу

Аналітична панель адміністратора коворкінгу являє собою широкоформатний робочий простір із підвищеною щільністю інформації. Інтерфейс бек-офісу надає менеджменту повний набір інструментів для виконання CRUD-операцій над сутностями робочих місць, тарифів та локацій. Сторінка насичена інтерактивними графіками та тепловими картами завантаженості простору. Вони візуалізують статистичні вибірки з бази даних MySQL за допомогою діаграм, згрупованих за часовими інтервалами. Адміністратор має можливість відстежувати пікові години навантаження на Open Space, контролювати сумарний фінансовий потік замовлень та виконувати ручне блокування окремих кімнат на технічне обслуговування. Для швидкого обслуговування клієнтів на рецепції екранна форма інтегрована з функціоналом

сканування перепусток резидентів для миттєвої валідації унікальних токенів доступу.

РОЗДІЛ 3. РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1. Реалізація клієнтської частини вебсистеми.

Практична розробка користувацького SPA-інтерфейсу екосистеми коворкінгу FlexSpot розпочалася з етапу конфігурування базового каркаса додатка. Як основне інфраструктурне середовище збірки та автоматизації модулів було обрано інструмент Vite, що функціонує в синергії з бібліотекою React.js. На відміну від класичних рішень пакетування, Vite повністю покладається на роботу нативних ES-модулів (EcmaScript Modules), які інтерпретуються безпосередньо сучасними веббраузерами [13]. Це технічне рішення дозволило кардинально скоротити тривалість запуску локального сервера розробки, оскільки процес попереднього збирання (Pre-bundling) залежностей виконується надшвидким компілятором esbuild.

Під час редагування компонентів механізм гарячої заміни модулів (Hot Module Replacement) оновлює виключно змінені файли без повного скидання стану інтерфейсу додатка, що істотно прискорює тестування логіки на фронтенді. Системне налаштування середовища виконується у файлі vite.config.js. За обробку синтаксису JSX відповідає інтегрований платіж @vitejs/plugin-react. Для оптимізації архітектури імпортів та ліквідації глибоких відносних шляхів на кшталт ../../components у конфігурації прописано аліас "@", який жорстко зв'язаний із кореневою папкою вихідного коду за допомогою утиліт Node.js path.resolve та fileURLToPath. Системні параметри збірки клієнтського застосунку та конфігурацію шляхів у файлі vite.config.js приведено в лістингу 3.1.

Лістинг 3.1. Конфігурація збирача модулів та налаштування проксі-сервера

```
import { defineConfig } from "vite";
import react from "@vitejs/plugin-react";
import { fileURLToPath } from "url";
import path from "path";

const __dirname = path.dirname(fileURLToPath(import.meta.url));

export default defineConfig({
    plugins: [react()],
    resolve: {
        alias: {
            "@": path.resolve(__dirname, "./src")
```

```
        }
    }
});
```

Логічна організація вихідного коду клієнтського застосунку всередині робочої директорії src повністю підпорядкована архітектурному принципу суворого розподілу відповідальності (Separation of Concerns). Робочий простір структуровано у вигляді набору ізольованих каталогів, кожен з яких інкапсулює окремий технологічний шар системи [14]. Повну структурну декомпозицію каталогів вихідного коду фронтенду та їхній безпосередній інженерний зв’язок із шарами системи відображено в табл. 3.1.

Таблиця 3.1. Структурна організація та призначення каталогів вихідного коду фронтенду

Назва каталогу (src/*)	**Технологічний шар системи**	**Функціональне призначення в екосистемі FlexSpot**
/api	Сервісний шар мережевих запитів	Конфігурація клієнта Axios (axios.js), опис методів взаємодії з тарифами (packageService.js) та еквайрингом (paymentService.js).
/assets/images	Рівень статичних ресурсів	Збереження оптимізованих зображень у форматі WEBP для інтерфейсу локацій, галеркї та типів робочих місць.
/components	UI-шар (Шаблони представлення)	Багаторазові ізольовані компоненти (віджети бронювання зон booking/*, системні банери, блоки Header та Footer).
/context	Глобальний стейт-менеджмент	Реалізація провайдерів контексту (AuthContext.jsx, AuthProvider.jsx) для наскрізного поширення стану сесії.
/hooks	Шар абстракції логіки	Кастомні React-хуки (useAuth.js) для швидкого доступу компонентів до бізнес-даних та методів авторизації.
/layouts	Рівень структурних макетів	Каркасні обгортки інтерфейсу. Головний макет MainLayout.jsx та ізольований профільний макет ProfileLayout.

продовження табл. 3.1

/pages	Рівень екранних подань (Views)	Високорівневі компоненти-сторінки, які агрегують логічні блоки у завершені екрани (Home, Locations, BookingPage, Profile).

Життєвий цикл ініціалізації односторінкового застосунку (SPA) починається зі зчитування браузером кореневого документа index.html, який запускає головний скрипт точки входу main.jsx. Скрипт активує монтирування додаткових провайдерів та розгортає віртуальну копію інтерфейсу в оперативній пам'яті. Безпосереднє управління навігаційним графом та переходами резидента між екранами коворкінгу делеговано файлу App.jsx. Усередині цього компонента розгорнуто декларативне дерево маршрутизації на базі бібліотеки react-router-dom. Застосування клієнтського роутингу дозволяє системі миттєво перемальовувати вміст сторінки відповідно до зміни URL-адреси. Це повністю позбавляє браузер необхідності виконувати перезавантаження сторінки та повторно запитувати важкі HTML-файли у сервера, забезпечуючи високу швидкість відгуку системи [15].

Для публічних сторінок, таких як головна екранна форма (/), каталог коворкінг-центрів (/locations), сторінка деталей конкретного об'єкта (/locations/:id), перегляд доступних пакетів абонементів (/packages) та сторінка опису сервісу (/about), застосовано загальний каркасний макет MainLayout. На етапі перемикання маршрутів компонент ScrollReset примусово скидає координати прокрутки вікна до нульової мітки, ліквідуючи аномалії збереження положення екрана. Глобальне сповіщення користувачів про хід транзакцій або помилки валідації реалізовано через підключення реактивного контейнера ToastContainer.

Безпека закритих зон та розмежування прав доступу на фронтенді контролюється за допомогою спеціалізованих компонентів-обгорток, інтегрованих безпосередньо в ієрархію дерева <Routes>. Для захисту приватних даних резидентів від несанкціонованого доступу всі внутрішні екрани особистого кабінету – загальна панель стану (ProfileDashboard), історія

замовлень (ProfileBookings), цифровий гаманець (ProfileWallet), налаштування акаунта (ProfileSettings), а також інтерактивний модуль бронювання (BookingPage) – обгорнуті компонентом ProtectedRoute та підключені до ізольованого макета ProfileLayout [16]. Цей модуль зчитує поточний стан автентифікації користувача з провайдера AuthProvider.

Якщо користувач намагається отримати доступ до кабінету з пустим або невалідним токеном, система виконує миттєву деструктуризацію запиту і перенаправляє трафік на сторінку входу. Навпаки, маршрути авторизації (/login), реєстрації нового профілю (/register) та відновлення втраченого доступу (/forgot-password) захищені компонентом GuestRoute. Цей елемент блокує автентифікованим резидентам можливість повторного переходу на екрани введення облікових даних, автоматично повертаючи їх усередину захищеного периметра. Декларативну структуру побудови навігаційного графа та логіку ізоляції захищених маршрутів у файлі App.jsx приведено в лістингу 3.2.

Лістинг 3.2. Ініціалізація декларативного дерева маршрутизації

```
<Routes>
    <Route element={<MainLayout />}>
        <Route path="/" element={<Home />} />
        <Route path="/locations" element={<Locations />} />
        <Route
            path="/locations/:id"
            element={<LocationDetails />}
        />
        <Route
            path="/booking/:zoneId"
            element={<BookingPage />}
        />
        <Route path="/packages" element={<Packages />} />
        <Route path="/about" element={<About />} />
        <Route
            path="/profile"
            element={
                <ProtectedRoute>
                    <ProfileLayout />
                </ProtectedRoute>
            }
        >
            <Route index element={<ProfileDashboard />} />
            <Route
                path="bookings"
                element={<ProfileBookings />}
            />
            <Route path="wallet" element={<ProfileWallet />} />
            <Route
                path="settings"
                element={<ProfileSettings />}
```

```
            />
        </Route>

        <Route
            path="/admin"
            element={
                <ProtectedRoute allowedRoles={["admin"]}>
                    <div
                        style={{
                            padding: "100px",
                            textAlign: "center"
                        }}
                    >
                        <h1>Панель адміністратора</h1>
                        <p>Доступ дозволено тільки для Admin</p>
                    </div>
                </ProtectedRoute>
            }
        />
    </Route>

    <Route
        path="/register"
        element={
            <GuestRoute>
                <Register />
            </GuestRoute>
        }
    />
    <Route
        path="/login"
        element={
            <GuestRoute>
                <Login />
            </GuestRoute>
        }
    />
    <Route
        path="/forgot-password"
        element={<ForgotPassword />}
    />
</Routes>
```

Програмна реалізація ядра бронювання вебсистеми FlexSpot базується на патерні розділення компонентів на контейнери («розумні» компоненти, Smart Components) та презентаційні елементи («тупі» компоненти, Dumb Components), що взаємодіють у межах єдиної декларативної парадигми React.js. Центральним вузлом архітектури виступає сторінка-оркестратор BookingPage, яка акумулює в собі повний бізнес-стан поточної сесії оренди місць, здійснює координацію життєвого циклу тимчасових чернеток та виконує високорівневі математичні розрахунки фінансових квот.

Комунікація між батьківською формою та дочірніми модулями просторів (OpenSpaceWidget, DedicatedDeskWidget, MeetingRoomWidget) підпорядкована архітектурному принципу підняття стану (Lifting State Up) та концепції односпрямованого потоку даних (Unidirectional Data Flow). Батьківський компонент виступає єдиним джерелом істини (Single Source of Truth), спускаючи вниз конфігураційні пропси та асинхронні колбеки для зворотного зв'язку, тоді як віджети інкапсулюють ізольовану логіку відображення специфічних для конкретної зони інтерфейсних елементів. Архітектурну топологію взаємодії реактивних компонентів ядра бронювання та суміжних шарів системи відображено за допомогою UML-діаграми на рис. 3.1.

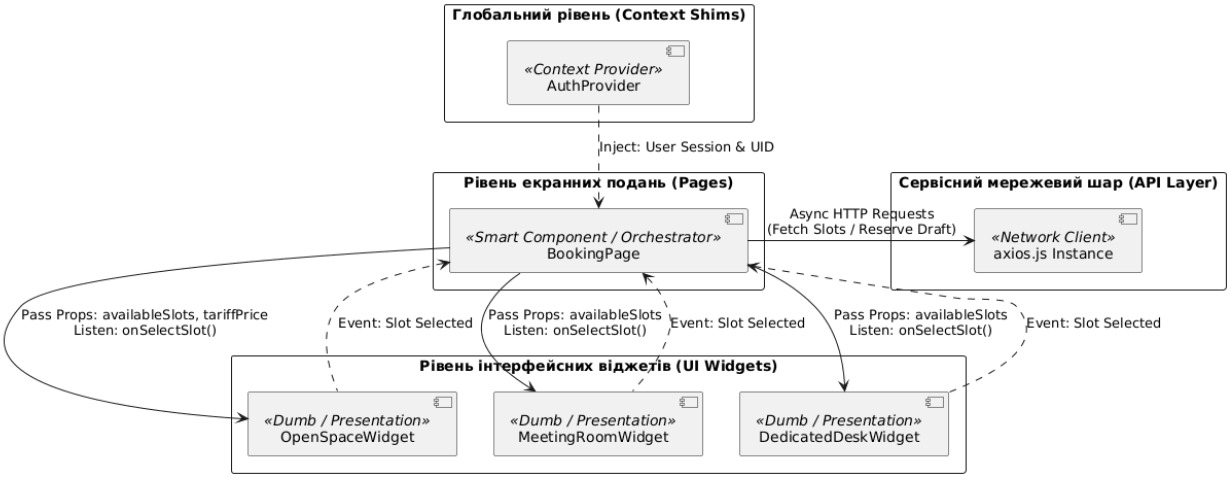


Рисунок 3.1. Діаграма компонентної структури реактивного ядра бронювання

Контейнер BookingPage динамічно ініціалізується в пам'яті клієнта після зчитування параметра zoneId із навігаційного URL-маршруту за допомогою хука useParams. Компонент негайно запускає асинхронний ланцюжок через хук useEffect, відправляючи GET-запит до ендпоінкта /zones/${zoneId} для отримання повних метаданих поточної інфраструктурної зони: її текстової назви, типу простору, конфігурації батьківської локації, адреси та базової тарифної сітки. Одночасно з цим через кастомний хук useAuth здійснюється мапування глобального масиву активних чернеток користувача (drafts), що зберігаються в пам'яті спільного контексту.

Алгоритм виділяє цільову чернетку (currentZoneDraft), яка відповідає ідентифікатору поточної зони, та формує об'єкт activeSource, що є реактивним міксом між зафіксованою на сервері чернеткою та динамічними змінами інтерфейсу на фронтенді (liveDetails). Повну програмну реалізацію сторінки-оркестратора, що координує процеси розрахунку вартості та життєвого циклу замовлення, наведено в лістингу 3.3.

Лістинг 3.3. Фрагмент коду методу handleConfirmPayment з алгоритмом інтерактивної обробки фінансових помилок

```
const handleConfirmPayment = async () => {
        setIsConfirming(true);
        try {
            await api.post(`/bookings/${currentZoneDraft.booking_id}/confirm`, {
                use_quota: useQuota
            });
            toast.success("Бронювання успішно підтверджено!");
            await fetchActiveDrafts();
            navigate("/profile/bookings");
        } catch (err) {
            const errorData = err.response?.data;
            if (errorData?.code === "INSUFFICIENT_FUNDS") {
                toast.error(
                    ({ closeToast }) => (
                        <div className={s.toastContainer}>
                            <span>{errorData.error}</span>
                            <button
                                type="button"
                                className="btn-primary btn-sm"
                                onClick={() => {
                                    closeToast();
                                    navigate("/profile/wallet");
                                }}
                            >
                                Поповнити баланс
                            </button>
                        </div>
                    ),
                    { autoClose: 8000 }
                );
            } else {
                toast.error(errorData?.message || "Помилка при оплаті.");
            }
        } finally {
            setIsConfirming(false);
        }
    };
```

Важливою інженерною частиною сторінки бронювання є внутрішній алгоритм динамічного розрахунку корпоративних B2B-квот резидента у режимі

реального часу. Система підтримує кілька логічних гілок обчислення вартості залежно від типу об'єкта, що орендується:

- Для зон вільного планування (Open Space) необхідний ліміт відповідає прямій кількості обраних місць, а списання днів проводиться з доступного пакета послуг користувача.
- Для закріплених робочих місць та кімнат переговорів обсяг потрібної квоти розраховується на основі співвідношення загальної вартості сесії до базової ціни ресурсу, що дозволяє уникнути розбіжностей при довгостроковому або погодинному резервуванні.

На наступному етапі алгоритм аналізує активні абонементи користувача для визначення сумарного доступного залишку годин або днів. Кінцева сума до сплати вираховується за принципом пропорційного покриття: система визначає частку, яку покриває корпоративний пакет, а залишок вартості автоматично округлюється та виводиться на платіжний віджет інтерфейсу.

Презентаційний шар платформи реалізовано у вигляді модульних інтерфейсних компонентів. Ключовий елемент сторінки містить інтерактивний календар із локалізованим вибором дат. При взаємодії з графічною сіткою розкладу система асинхронно надсилає запити до серверної частини для отримання даних про завантаженість коворкінгу на обраний період. Отримані дані містять актуальну інформацію про кількість вільних місць на кожен день та прапорці загальної доступності обраної зони.

Будь-яка зміна конфігурації замовлення (кількості місць або дати візиту) миттєво синхронізується з батьківським компонентом для актуалізації стану системи. Для збереження візуальної чистоти користувацького інтерфейсу стандартні кнопки надсилання форми приховані, а передача даних реалізована через програмний виклик подій між компонентами. Алгоритм обробки форми та валідації прав доступу резидента наведено в лістингу 3.4.

Лістинг 3.4. Фрагмент коду методу handleSubmit з алгоритмом валідації прав доступу резидента

```
const handleSubmit = async (e) => {
        if (e) e.preventDefault();

        if (!user) {
            toast.info(
                <div>
                    <strong>Потрібна авторизація</strong> <br />
                    Будь ласка, увійдіть або зареєструйтесь для створення
                    бронювання.
                </div>,
                { autoClose: 5000 }
            );
            return navigate("/login", {
                state: { from: `/booking/${zone.id}` }
            });
        }

        if (user && !user.email_verified_at) {
            return toast.warning(
                <div>
                    <strong>Увага!</strong> <br />
                    Для створення бронювання необхідно підтвердити вашу
                    електронну адресу.
                </div>,
                { autoClose: 6000 }
            );
        }

        if (!dateString || quantity < 1 || quantity > maxSpots) return;

        setIsSubmitting(true);
        try {
            const response = await api.post("/bookings/open-space/draft", {
                zone_id: zone.id,
                date: dateString,
                quantity: quantity
            });
            toast.success(response.data.message);
            onDraftCreated();
        } catch (err) {
            toast.error(
                err.response?.data?.message ||
                    "Помилка при створенні бронювання."
            );
        } finally {
            setIsSubmitting(false);
        }
    };
```

Управління життєвим циклом часової чернетки реалізовано на сторінці BookingPage через щосекундний моніторинг часових міток. Після підтвердження сервером створення чернетки та повернення атрибута expires_at, клієнтський застосунок ініціює інтервальний таймер setInterval із тактом оновлення в 1000

мс. Цей алгоритм обчислює різницю між поточним системним часом пристрою та міткою закінчення резерву. Отриманий інтервал розкладається на хвилини та секунди з форматуванням рядка через метод padStart(2, "0") для коректного відображення таймера у вигляді MM:SS.

Коли лічильник досягає нульової позначки, таймер очищується за допомогою clearInterval, стан інтерфейсу перемикається в режим блокування (isExpired = true), а застосунок автоматично викликає фоновий метод fetchActiveDrafts() для синхронізації стану з сервером. Це дозволяє своєчасно звільнити заблоковані ресурси коворкінгу для інших користувачів у реальному часі.

Взаємодія на фінальному етапі бронювання координується асинхронним методом handleConfirmPayment. Функція переводить інтерфейс у стан завантаження для запобігання повторним клікам користувача та відправляє POST-пакет на ендпоінт /bookings/${booking_id}/confirm. Архітектурною особливістю обробки виняткових ситуацій (Exception Handling) на фронтенді є перехоплення та аналіз специфічних помилок бізнес-логіки сервера.

У випадку відхилення транзакції через брак коштів на внутрішньому балансі користувача, мережевий шар на базі Axios розпізнає константний код помилки INSUFFICIENT_FUNDS. Замість виведення статичного тексту інтерфейс FlexSpot динамічно інжектує всередину модального вікна react-toastify кастомний JSX-контейнер. Цей блок містить кнопку швидкого доступу, яка закриває повідомлення та перенаправляє користувача до розділу цифрового гаманця (/profile/wallet) для поповнення рахунку через платіжний шлюз LiqPay. Такий підхід забезпечує безперервність користувацького досвіду (User Flow) та покращує загальну ергономіку системи.

Обробка мережевих потоків та взаємодія клієнтської частини з REST API бекенду побудована на базі ізольованого сервісного шару. Для винесення логіки низькорівневих HTTP-запитів за межі візуальних компонентів розроблено виділений мережевий клієнт. Таке архітектурне рішення дозволило централізовано керувати базовою URL-адресою сервера, типами контенту та

заголовками обміну даними у форматі JSON, забезпечуючи високу модульність клієнтського застосунку.

Безпека передачі даних та наскрізна автентифікація запитів забезпечується конвеєром перехоплювачів (Interceptors). Перед відправленням кожного HTTP-виклику клієнтський перехоплювач автоматично звертається до сервісу автентифікації Firebase для перевірки стану поточної сесії. Якщо користувач успішно авторизований у системі, алгоритм асинхронно зчитує криптографічний токен доступу та інжектує його в заголовок безпеки як Bearer-носій. Повний життєвий цикл асинхронної взаємодії між компонентами інтерфейсу, конвеєром мережевих перехоплювачів та зовнішніми сервісами під час виконання фінансових операцій наочно продемонстровано за допомогою UML-діаграми послідовності на рис. 3.2.

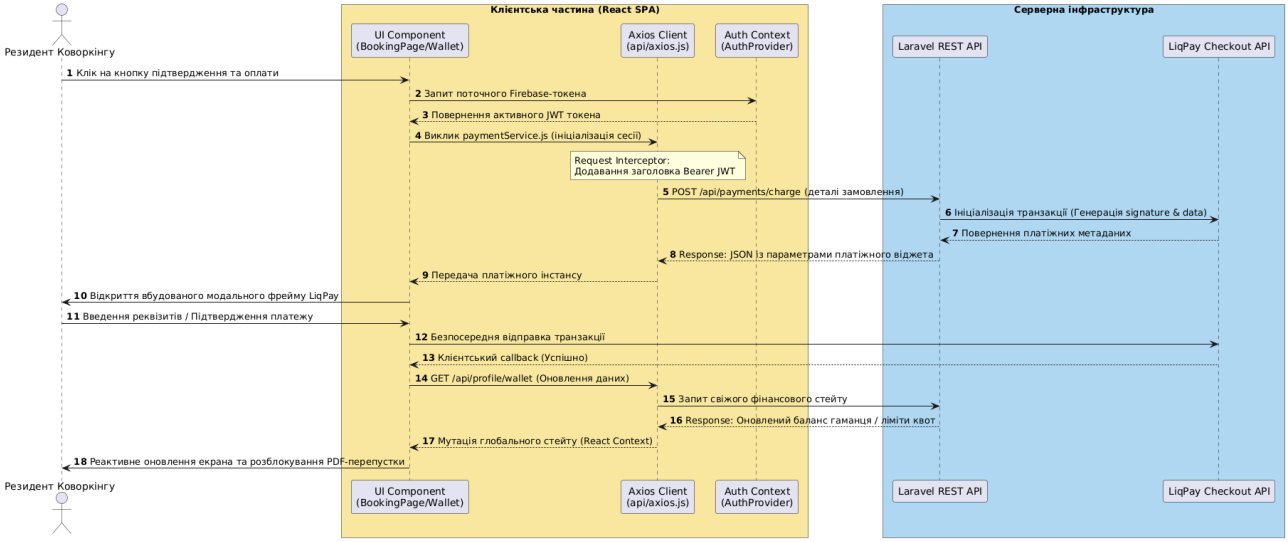


Рисунок 3.2. Діаграма послідовності процесу координації асинхронних платіжних потоків

Особливу увагу при проєктуванні мережевого шару приділено забезпеченню відмовостійкості системи у випадку завершення терміну дії сесії резидента. Якщо віддалений сервер відхиляє запит і повертає помилку авторизації зі статус-кодом 401, у дію вступає перехоплювач відповідей. Алгоритм автоматично блокує потік, встановлює прапорець повторної спроби

для запобігання зацикленню мережевих викликів та ініціює примусове оновлення токена через Firebase SDK.

Після успішного отримання свіжого ключа доступу система оновлює заголовки і прозоро для користувача повторно відправляє початковий запит, що суттєво підвищує надійність SPA-платформи. Програмну реалізацію конфігурації клієнта з двостороннім конвеєром JWT-перехоплювачів приведено в лістингу 3.5.

Лістинг 3.5. Програмний код конфігурації мережевого клієнта та JWT-перехоплювачів токенів

```
import axios from "axios";
import { auth } from "@/firebase";

const api = axios.create({
    baseURL: "http://127.0.0.1:8000/api",
    headers: {
        "Content-Type": "application/json",
        Accept: "application/json"
    }
});

api.interceptors.request.use(
    async (config) => {
        const user = auth.currentUser;

        if (user) {
            const token = await user.getIdToken();
            config.headers.Authorization = `Bearer ${token}`;
        }

        return config;
    },
    (error) => {
        return Promise.reject(error);
    }
);

api.interceptors.response.use(
    (response) => {
        return response;
    },
    async (error) => {
        const originalRequest = error.config;

        if (
            error.response &&
            error.response.status === 401 &&
            !originalRequest._retry
        ) {
            originalRequest._retry = true;

            const user = auth.currentUser;
            if (user) {
                try {
```

```
                    const newToken = await user.getIdToken(true);

                    originalRequest.headers.Authorization = `Bearer ${newToken}`;

                    return api(originalRequest);
                } catch (refreshError) {
                    console.error(
                        "Не вдалося примусово оновити токен Firebase:",
                        refreshError
                    );
                }
            }
        }

        return Promise.reject(error);
    }
);

export default api;
```

На базі налаштованого мережевого інстансу спроєктовано шар сервісів, відповідальних за інтеграцію фінансового модуля та взаємодію з LiqPay. Замість написання розрізнених мережевих викликів усередині UI-елементів гаманця, вся логіка інкапсульована у спеціалізованому платіжному сервісі. Сервіс містить три основні асинхронні методи, що забезпечують повноцінний фінансовий життєвий цикл клієнта:

- Метод ініціалізації транзакції, який передає суму поповнення на сервер для генерації захищеного платіжного віджета та цифрового підпису.
- Метод скасування фінансової сесії у разі відмови резидента від проведення платежу, що анулює операцію за ідентифікатором замовлення.
- Метод запиту повної історії транзакцій користувача для формування деталізованої стрічки фінансового аудиту в особистому кабінеті.

Такий підхід гарантує абсолютну чистоту кодової бази і дозволяє реактивним компонентам фронтенду оперувати готовими бізнес-даними. Програмний код реалізації сервісного шару управління фінансовими транзакціями приведено в лістингу 3.6.

Лістинг 3.6. Програмний код сервісу управління фінансовими транзакціями

```
import api from "./axios";

export const initiateTopUp = async (amount) => {
```

```
    const response = await api.post("/payments/initiate", { amount });
    return response.data;
};

export const cancelTransaction = async (orderId) => {
    const response = await api.post("/payments/cancel", { order_id: orderId });
    return response.data;
};

export const getTransactionsHistory = async () => {
    const response = await api.get("/payments/history");
    return response.data;
};
```

3.2. Розробка серверної частини та бізнес-логіки

Серверне ядро вебсистеми FlexSpot функціонує як безстановий (Stateless) REST API застосунок, розроблений на базі фреймворку Laravel. Вибір безстанової архітектури зумовлений необхідністю забезпечення високої масштабованості системи та її повної незалежності від клієнтського React-інтерфейсу. Сервер не зберігає стан сесії користувача у файловій системі чи оперативній пам’яті (на відміну від класичних сесійних Cookie-механізмів), а кодує ідентифікаційні дані всередині криптографічних JWT-токенів, які фронтенд прикріплює до кожного запиту. Такий підхід дозволяє оптимізувати розподіл апаратних ресурсів сервера, ізолювати бізнес-логіку обробки ресурсів коворкінгу та забезпечити уніфікований інтерфейс взаємодії.

Вся мережева маршрутизація та декларація доступних точок входу зосереджена всередині системного файла routes/api.php. Шар маршрутизації виступає першим рівнем диспетчеризації вхідних HTTP-пакетів, розподіляючи потоки даних між відповідними контролерами предметної області. Для уникнення хаотичної структури та спрощення клієнтської інтеграції всі API-ендпоінти системи строго уніфіковані відповідно до архітектурних конвенцій REST. Повну інженерну специфікацію ключових REST API ендпоінтів екосистеми FlexSpot, їхніх HTTP-методологій та зв'язків із шаром контролерів відображено в табл. 3.2.

Таблиця 3.2. Специфікація ключових REST API ендпоінтів екосистеми FlexSpot

HTTP-метод	URL-адреса ендпоінту	Метод класу Контролера	Рівень доступу та Middleware
GET	/api/locations	LocationController@index	Публічний (Без обмежень)
GET	/api/locations/{id}	LocationController@show	Публічний (Без обмежень)
GET	/api/zones/{id}	LocationController@showZone	Публічний (Без обмежень)
GET	/api/zones/{id}/open-space-availability	BookingController@getOpenSpaceAvailability	Захищений (auth.firebase)
POST	/api/bookings/open-space/draft	BookingController@createOpenSpaceDraft	Захищений (auth.firebase, firebase.verified)
POST	/api/bookings/{id}/confirm	BookingController@confirmBooking	Захищений (auth.firebase, firebase.verified)
POST	/api/payments/initiate	PaymentController@initiateTopUp	Захищений (auth.firebase)
GET	/api/payments/history	PaymentController@getHistory	Захищений (auth.firebase)

Для забезпечення безпеки ядра автоматизації бронювання у файлі маршрутів реалізовано механізм ієрархічного групування роутів за допомогою методів Route::middleware(). Ендпоінти, що відповідають за публічний перегляд локацій та опис робочих зон, винесені за межі авторизаційного периметра, що мінімізує затримки при первинному завантаженні сайту неавторизованими відвідувачами. Навпаки, критичні бізнес-операції – створення чернеток, фінансові транзакції гаманця та підтвердження оренди місць – жорстко інкапсульовані всередині захищених блоків під управлінням кастомних посередників auth.firebase та firebase.verified. Фрагмент програмної конфігурації архітектури маршрутів у файлі routes/api.php приведено в лістингу 3.7.

Лістинг 3.7. Фрагмент конфігурації файлу маршрутів routes/api.php із групуванням закритих ендпоінтів

```
Route::middleware('auth.firebase')->group(function () {
    Route::get('/user', function (Request $request) {
```

```
                    return    $request->user()->load(['profile',    'wallet',
'userPackages.package']);
    });

    Route::post('/user/profile', [ProfileController::class, 'update']);

    Route::get('/user/dashboard', function (Request $request) {
         $nextBooking = \App\Models\Booking::where('user_id', $request->user()-
>id)
            ->where('status', 'confirmed')
            ->where('end_at', '>', now())
            ->orderBy('start_at', 'asc')
            ->with('zone.location')
            ->first();

        return response()->json([
            'success' => true,
            'next_booking' => $nextBooking
        ]);
    });

    Route::get('/user/bookings', [BookingController::class, 'getUserBookings']);
             Route::get('/user/active-drafts',      [BookingController::class,
'getActiveDrafts']);

    Route::middleware('verified.firebase')->group(function () {
          Route::post('/bookings/open-space/draft',  [BookingController::class,
'createOpenSpaceDraft']);
           Route::post('/bookings/dedicated/draft',  [BookingController::class,
'createDedicatedDeskDraft']);
            Route::post('/bookings/meeting/draft',   [BookingController::class,
'createMeetingRoomDraft']);
             Route::post('/bookings/{id}/confirm',   [BookingController::class,
'confirmBooking']);
         Route::post('/bookings/{id}/cancel-draft',  [BookingController::class,
'cancelDraft']);
              Route::get('/bookings/{id}/receipt',   [ReceiptController::class,
'downloadBookingReceipt']);

                Route::post('/payments/initiate',    [PaymentController::class,
'initiateTopUp']);
                  Route::post('/payments/cancel',    [PaymentController::class,
'cancelTransaction']);
                  Route::get('/payments/history',    [PaymentController::class,
'getHistory']);
                                Route::get('/payments/transactions/{id}/receipt',
[ReceiptController::class, 'downloadTopUpReceipt']);

                Route::post('/packages/purchase',    [PackageController::class,
'purchase']);
    });
});
```

Важливим інженерним досягненням при розробці бекенду стало впровадження проміжного шару трансформації даних (Data Transformation Shims) за допомогою механізму Laravel API Resources [17]. Класичний підхід прямого повернення моделей Eloquent з контролера є архітектурно вразливим,

оскільки створює жорстке зчеплення (Tight Coupling) між фізичною структурою таблиць MySQL та клієнтським застосунком, а також призводить до надлишкового мережевого трафіку. Для розв'язання цієї проблеми було спроєктовано класи LocationResource та LocationDetailResource.

Серверний шар ресурсів виконує роль декларативного фільтра: метод toArray($request) перехоплює сирий об'єкт моделі, вилучає службові поля бази даних (такі як created_at, updated_at, внутрішні ідентифікатори системних логів) та формує чисту структуру JSON-пакета із примусовим приведенням типів даних. Дане рішення повністю ізолює внутрішнє представлення бази даних, захищає систему від потенційного витоку внутрішніх атрибутів сутностей та оптимізує швидкість парсингу об'єктів на фронтенді. Реалізацію мапування та трансформації сутностей у класі LocationResource приведено в лістингу 3.8.

Лістинг 3.8. Реалізація мапування сутностей у класі LocationResource

```
public function toArray(Request $request): array
    {
        $amenities = $this->amenities->map(function ($amenity) {
            return [
                'name' => $amenity->name,
                'slug' => $amenity->slug,
                'icon' => $amenity->icon,
            ];
        });

        if ($this->is_24_7) {
            $amenities->prepend([
                'name' => 'Доступ 24/7',
                'slug' => '24-7',
                'icon' => 'Clock',
            ]);
        }

        return [
            'id' => $this->id,
            'title' => $this->title,
            'city' => $this->city->name,
            'city_slug' => $this->city->slug,
            'address' => $this->address,
            'lat' => (float) $this->lat,
            'lng' => (float) $this->lng,
            'rating' => (float) $this->rating,
            'open_time' => $this->open_time,
            'close_time' => $this->close_time,
            'is_24_7' => (bool) $this->is_24_7,
           'image_url' => $this->images->where('is_main', true)->first()->url ??
null,
            'zones' => $this->zones->map(function ($zone) {
                return [
                    'id' => $zone->id,
```

```
                    'name' => $zone->name,
                    'type' => $zone->type,
                    'min_price' => $zone->min_price,
                ];
            }),
            'amenities' => $amenities,
        ];
    }
```

Взаємодія серверного ядра вебсистеми FlexSpot із реляційною базою даних MySQL реалізована за допомогою вбудованої об'єктно-реляційній моделі Eloquent ORM, яка базується на архітектурному патерні Active Record. У межах цієї парадигми кожна таблиця бази даних асоціюється з відповідним класом моделі (User, Location, Zone, Booking, Wallet), а кожен рядок таблиці представляє окремий інстанс цього класу. Такий підхід дозволив інкапсулювати низькорівневі SQL-операції всередині зрозумілих методів об'єктно-орієнтованого програмування PHP. Це значно підвищило швидкість розробки та забезпечило надійний захист від ін'єкцій завдяки автоматичному екрануванню параметрів на рівні драйвера PDO.

Для відтворення спроєктованої інфраструктури коворкінгу всередині програмного коду було налаштовано декларативні реляційні зв'язки між моделями. Базова сутність Location (конкретний коворкінг-центр) зв'язана відношенням «один-до-багатьох» (hasMany) із сутністю Zone (функціональні простори типу Open Space чи кімнати переговорів). У свою чергу, модель Zone належить (belongsTo) до локації та володіє множинними зв'язками з моделями тимчасових замовлень Booking та столів Desk. Окремо реалізовано логіку зв'язку з географічними сутностями міст City та додаткових зручностей Amenity [18]. Програмне визначення реляційних зв'язків та конфігурацію інкапсульованих моделей на прикладі сутності Zone приведено в лістингу 3.9.

Лістинг 3.9. Декларація реляційних зв'язків у моделі сутності Zone

```
class Zone extends Model
{
    protected $guarded = [];

    public function location()
    {
        return $this->belongsTo(Location::class);
    }
```

```
    public function desks()
    {
        return $this->hasMany(Desk::class);
    }

    public function bookings()
    {
        return $this->hasMany(Booking::class);
    }
}
```

Під час проєктування високонавантажених REST API систем на базі ORM критичною проблемою є виникнення архітектурної вразливості N+1 запитів до бази даних. Ця аномалія з'являється тоді, коли додаток ініціює один первинний запит для отримання масиву батьківських сутностей (наприклад, усіх локацій коворкінгу), а потім у циклі виконує додатковий SQL-запит для кожної окремої локації, щоб витягнути її зв'язані зони або зображення. У результаті для виведення списку з 20 об'єктів сервер робить 21 транзакцію до MySQL, що миттєво блокує пул з'єднань СУБД та призводить до експоненціального зростання затримок відповіді сервера.

Для ліквідації проблеми N+1 та оптимізації споживання оперативної пам'яті в шарі контролерів FlexSpot було примусово впроваджено механізм жадібного завантаження (Eager Loading). Замість лінивого зчитування даних, серверні методи контролерів викликають побудову SQL-запитів із використанням оператора with(). Це змушує Eloquent ORM виконати оптимізовану операцію об'єднання через єдиний нативний оператор IN на рівні бази даних, завантажуючи всі вкладені масиви міст, зображень та робочих зон лише за два такти звернення до диска. Додатково для швидкого виведення лічильників відгуків та доступних місць без завантаження важких текстових полів застосовано метод withCount(). Фрагмент оптимізованої вибірки колекцій у класі LocationController приведено в лістингу 3.10.

Лістинг 3.10. Оптимізація вибірки даних у контролері за допомогою механізму Eager Loading

```
class LocationController extends Controller
{
    public function index(Request $request)
```

```
{
    $query = Location::with(['city', 'zones', 'images', 'amenities']);

    if ($request->has('cities') && $request->cities !== '') {
        $cities = explode(',', $request->cities);
        $query->whereHas('city', function ($q) use ($cities) {
            $q->whereIn('slug', $cities);
        });
    }

    if ($request->has('zones') && $request->zones !== '') {
        $zones = explode(',', $request->zones);
        $query->whereHas('zones', function ($q) use ($zones) {
            $q->whereIn('type', $zones);
        });
    }

    if ($request->has('amenities') && $request->amenities !== '') {
        $amenities = explode(',', $request->amenities);

        if (in_array('24-7', $amenities)) {
            $query->where('is_24_7', true);
            $amenities = array_diff($amenities, ['24-7']);
        }

        foreach ($amenities as $amenity) {
            $query->whereHas('amenities', function ($q) use ($amenity) {
                $q->where('slug', $amenity);
            });
        }
    }

    $sort = $request->input('sort', 'newest');
    $order = $request->input('order', 'desc') === 'asc' ? 'asc' : 'desc';

    if ($sort === 'popularity') {
        $query->orderBy('rating', $order);
    } elseif ($sort === 'price') {
        $query->orderBy(
            Zone::select('min_price')
                ->whereColumn('location_id', 'locations.id')
                ->orderBy('min_price', 'asc')
                ->limit(1),
            $order
        );
    } else {
        $query->orderBy('created_at', $order);
    }

    return LocationResource::collection($query->get());
}

public function show(int $id)
{
    $location = Location::with([
        'city',
        'zones',
        'images',
        'amenities',
        'reviews.user.profile'
    ])->findOrFail($id);

    return new \App\Http\Resources\LocationDetailResource($location);
```

```
    }
}
```

Застосування такого підходу дозволило суттєво знизити час відповіді сервера (Response Time) під час масових вибірок інфраструктурних об'єктів. Завдяки перенесенню обчислювального навантаження з виконання циклічних SQL-запитів на рівень оптимізованих внутрішніх операцій самої СУБД MySQL, суттєво зменшилося споживання оперативної пам'яті сервера застосунків. Це забезпечує лінійну швидкість рендерингу та завантаження повного каталогу коворкінг-центрів на стороні клієнта, мінімізуючи мережеві затримки навіть за умов високої щільності одночасних запитів від резидентів платформи.

Організація безпеки та розмежування прав доступу на рівні серверної частини вебсистеми FlexSpot підпорядкована концепції наскрізного конвеєра обробки запитів (Request Pipeline). Головним архітектурним викликом при проєктуванні цієї підсистеми стала необхідність синхронізації розподіленої інфраструктури користувачів. Системі довелося об'єднати локальну реляційну базу даних MySQL, де зберігаються фінансові баланси гаманців та деталі замовлень, із зовнішнім хмарним провайдером ідентифікації Google Firebase Cloud Auth. Для розв'язання цієї задачі, замість використання стандартних закритих сесій, було реалізовано ланцюжок кастомних компонентів проміжного шару (FirebaseAuthMiddleware), які послідовно аналізують, валідують та фільтрують кожен вхідний HTTP-пакет.

Першим етапом захисту безстанового API виступає посередник FirebaseAuthMiddleware. Його алгоритм перехоплює вхідний пакет і вилучає Bearer-токен із заголовків авторизації. У разі відсутності ключа конвеєр негайно переривається з поверненням клієнту помилки авторизації. Якщо токен надано, додаток взаємодіє із хмарним Firebase SDK, виконуючи дешифрування та перевірку цифрового підпису.

Головна інженерна особливість цього Middleware полягає в реалізації механізму динамічної синхронізації профілів (Distributed Identity Management): якщо хмарна валідація пройшла успішно, але користувача з таким

ідентифікатором ще немає в локальній базі MySQL, система автоматично створює новий обліковий запис резидента та ініціалізує його профіль, копіюючи базові атрибути. Після цього об'єкт користувача автентифікується всередині поточного такту роботи фреймворку. Повний логічний шлях проходження клієнтського запиту крізь фільтри проміжного шару до моменту допуску до контролерів відображено на рис. 3.3.

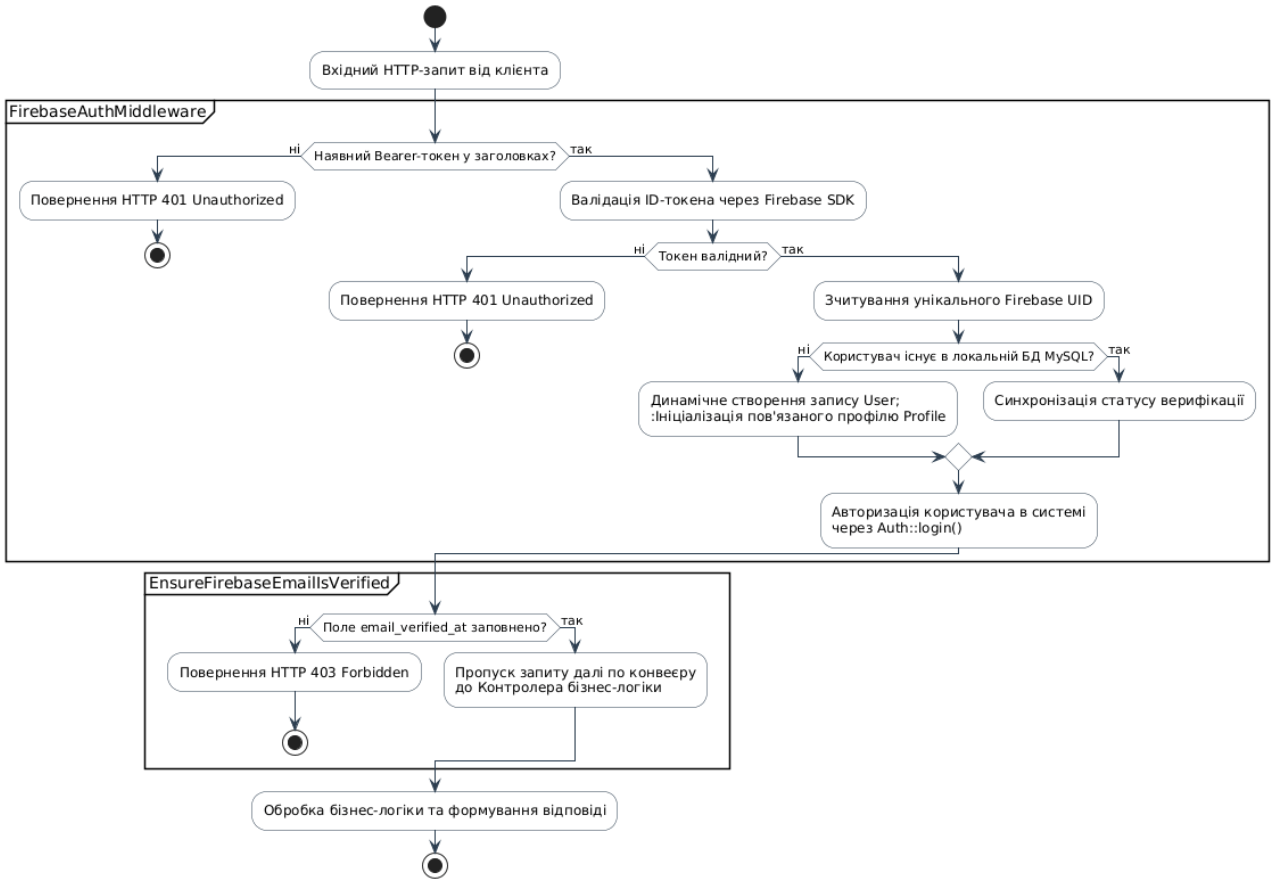


Рисунок 3.3. Діаграма діяльності конвеєра обробки HTTP-запитів Middleware-шаром

Розподіл логіки на ізольовані класи посередників дозволяє гнучко налаштовувати рівні безпеки для різних типів операцій. Програмний алгоритм фонової синхронізації облікових записів у класі FirebaseAuthMiddleware приведено в лістингу 3.11.

Лістинг 3.11. Фрагмент логіки автентифікації та перевірки токенів у FirebaseAuthMiddleware

```
public function handle(Request $request, Closure $next)
    {
        $token = $request->bearerToken();

        if (!$token) {
            return response()->json(['message' => 'Unauthorized. Token not provided.'], 401);
        }

        try {
            $auth = Firebase::auth();
            $verifiedIdToken = $auth->verifyIdToken($token);
            $firebaseUid = $verifiedIdToken->claims()->get('sub');

           $isEmailVerified = $verifiedIdToken->claims()->get('email_verified');

            $user = User::where('firebase_uid', $firebaseUid)->first();

            if (!$user) {
                $firebaseUser = $auth->getUser($firebaseUid);

                $user = User::create([
                    'firebase_uid' => $firebaseUid,
                    'email' => $firebaseUser->email,
                    'role' => 'user',
                    'email_verified_at' => $isEmailVerified ? now() : null,
                ]);

                $user->profile()->create([
                            'name' => $firebaseUser->displayName ?? explode('@', $firebaseUser->email)[0],
                ]);
            } else {
                if ($isEmailVerified && is_null($user->email_verified_at)) {
                    $user->update(['email_verified_at' => now()]);
                }
            }

            Auth::login($user);

            return $next($request);
        } catch (Throwable $e) {
            return response()->json([
                'message' => 'Unauthorized. Invalid token.',
                'error' => $e->getMessage()
            ], 401);
        }
    }
```

Другим етапом захисного периметра виступає посередник EnsureFirebaseEmailIsVerified. Поділ процесу безпеки на два окремі Middleware реалізує архітектурний принцип єдиної відповідальності (Single Responsibility). Це дозволяє системі розрізняти користувачів, які просто увійшли в акаунт (і

мають право, наприклад, переглядати історію платежів), від тих, хто намагається взаємодіяти безпосередньо з ядром автоматизації та створювати чернетки бронювання місць.

Посередник верифікації жорстко перевіряє наявність тимчасового штампа підтвердження електронної пошти в об'єкті авторизованого резидента. Якщо поштову адресу не підтверджено у хмарі Firebase, алгоритм блокує виконання запиту, повертаючи HTTP-код 403 Forbidden. Дане технічне рішення надійно захищає ядро бази даних від спам-реєстрацій та автоматизованих спроб штучного блокування вільних стільців у коворкінг-центрі. Програмний алгоритм захисту критичних точок входу в класі EnsureFirebaseEmailIsVerified приведено в лістингу 3.12.

Лістинг 3.12. Програмний алгоритм валідації облікового запису в EnsureFirebaseEmailIsVerified

```
public function handle(Request $request, Closure $next)
    {
        $user = $request->user();

        if (!$user || is_null($user->email_verified_at)) {
            return response()->json([
                 'message' => 'Ваш email не підтверджено. Будь ласка, перевірте пошту.'
            ], 403);
        }

        return $next($request);
    }
```

Впровадження такого двоступеневого конвеєра проміжного шару дозволило надійно ізолювати критичні бізнес-процеси автоматизації від несанкціонованого або зловмисного втручання. Завдяки делегуванню операцій криптографічної валідації токенів та верифікації поштових скриньок на рівень ізольованих посередників, шар контролерів оперує виключно довіреними та повністю перевіреними об'єктами користувачів. Це гарантує цілісність реляційних даних у MySQL, повністю запобігає засміченню бази даних фіктивними замовленнями та істотно підвищує загальний рівень відмовостійкості й безпеки серверного ядра системи.

3.3. Реалізація алгоритму бронювання та системи управління чернетками

Проєктування базового функціоналу автоматизації коворкінгу FlexSpot вимагало створення надійної логіки розподілу обмежених інфраструктурних ресурсів центру. Головна алгоритмічна задача цієї підсистеми полягає у запобіганні овербукінгу – критичній ситуації, за якої одне й те саме робоче місце (desk) чи переговорна кімната можуть бути одночасно заброньовані кількома резидентами. Для розв'язання цієї проблеми на рівні архітектури бекенду було впроваджено концепцію двофазного фіксування замовлень за допомогою інструменту тимчасових чернеток (drafts). Замість миттєвого створення незворотного фінансового запису в базі даних, система спочатку ізолює обраний простір, надаючи користувачеві часове вікно для спокійного завершення транзакції.

Процес ініціалізації оренди починається на стороні BookingController з обов'язкової атомарної валідації параметрів вхідного пакета. Сервер перевіряє наявність цільової зони, коректність обраної дати, а також ліміти запитуваних місць. Ключовим елементом алгоритму захисту від паралельних запитів є розрахунок поточного реального навантаження простору. Проте суто серверна валідація за допомогою умовних операторів не здатна захистити платіжний контур від конкурентних конфліктів на мілісекундному рівні, коли кілька запитів на одне й те саме місце приходять одночасно.

Для повного усунення стану гонитви (race conditions) алгоритм обчислення поточного навантаження виконується всередині ізольованої транзакції бази даних із застосуванням песимістичного блокування рядків за допомогою методу lockForUpdate(). Під час виконання SQL-запиту СКБД блокує наявні релевантні записи в таблиці й підсумовує не лише підтверджені замовлення, а й усі активні чернетки інших користувачів, термін дії яких ще не вичерпано. Якщо сумарне навантаження разом із новим запитом перевищує паспортну місткість зони коворкінгу, виконання транзакції примусово переривається, а клієнту

повертається відповідний JSON-пакет із описом помилки зі статус-кодом 422 Unprocessable Entity. Такий дворівневий захист (бізнес-валідація в поєднанні з транзакційним блокуванням на рівні бази даних) гарантує стовідсоткову стійкість і консистентність даних під час пікових навантажень. Фрагмент програмного коду методу ініціалізації та транзакційної перевірки лімітів простору наведено в лістингу 3.13.

Лістинг 3.13. Метод створення тимчасової чернетки замовлення з валідацією місткості простору

```
public function createOpenSpaceDraft(Request $request)
{
    $request->validate([
        'zone_id' => 'required|exists:zones,id',
        'date' => 'required|date|after_or_equal:today',
        'quantity' => 'required|integer|min:1',
    ]);

    return DB::transaction(function () use ($request) {
        $zone = Zone::lockForUpdate()->findOrFail($request->zone_id);

        if ($zone->type !== 'open_space') {
            return response()->json(['success' => false, 'message' => 'Цей ендпоінт призначений тільки для Open Space зон.'], 400);
        }

        $bookingDate = Carbon::parse($request->date)->startOfDay();
        $startAt = $bookingDate->copy()->startOfDay();
        $endAt = $bookingDate->copy()->endOfDay();
        $quantity = $request->quantity;

        $activeBookingsCount = Booking::where('zone_id', $zone->id)
            ->where(function ($q) {
                $q->where('status', 'confirmed')
                    ->orWhere(function ($sub) {
                        $sub->where('status', 'draft')
                            ->where('expires_at', '>', Carbon::now());
                    });
            })
            ->where('start_at', '<=', $endAt)
            ->where('end_at', '>=', $startAt)
            ->sum('quantity');

        if (($activeBookingsCount + $quantity) > $zone->capacity) {
            return response()->json(['success' => false, 'message' => 'На жаль, на вибрану дату немає достатньо вільних місць.'], 422);
        }

        $totalPrice = $zone->min_price * $quantity;

        $booking = Booking::create([
            'user_id' => $request->user()->id,
            'zone_id' => $zone->id,
            'desk_id' => null,
            'quantity' => $quantity,
            'start_at' => $startAt,
```

```
            'end_at' => $endAt,
            'total_price' => $totalPrice,
            'status' => 'draft',
            'expires_at' => Carbon::now()->addMinutes(10),
        ]);

        return response()->json([
            'success' => true,
            'message' => 'Місце тимчасово заблоковано за вами на 10 хвилин для
завершення бронювання.',
            'data' => [
                'booking_id' => $booking->id,
                'total_price' => $booking->total_price,
                'expires_at' => $booking->expires_at->toIso8601String()
            ]
        ], 201);
    });
}
```

У разі успішного проходження фільтрів валідації та накладання песимістичного блокування, алгоритм розраховує фінансову вартість сесії та реєструє новий рядок у таблиці bookings зі специфічним статусом draft. Для жорсткого обмеження часу блокування робочих місць об'єкту примусово присвоюється часова мітка expires_at, яка розраховується додаванням десяти хвилин до поточної системної мітки за допомогою інструментів бібліотеки Carbon. Дане інженерне рішення забезпечує тимчасову фіксацію інфраструктурного ресурсу за конкретним резидентом платформи в межах поточного безстанового HTTP-запиту.

Проте динамічне виділення місць породжує суміжний виклик: якщо користувач закриє вкладку браузера або відмовиться від оплати, заблоковані ресурси залишаться недоступними для інших відвідувачів. Для розв'язання цієї проблеми та автоматизації санітарії бази даних MySQL у серверній частині FlexSpot розгорнуто фонову службу планування завдань на базі вбудованого інструменту Laravel Task Scheduling (Scheduler). Цей модуль щохвилини координується системним демоном Cron на рівні операційної системи Linux, в автоматичному режимі перевіряє часові штампи, анулює застарілі чернетки та повертає слоти у загальний пул доступної інфраструктури коворкінгу. Програмну реалізацію консольної команди очищення протермінованих чернеток наведено в лістингу 3.14.

Центральним інструментом автоматичного вивільнення заблокованих просторів є розроблена консольна команда CancelExpiredBookings, яка зареєстрована в системі під унікальним сигналом bookings:cancel-expired. Замість ресурсомісткого перебору записів у циклі засобами мови PHP, метод handle() виконує один оптимізований атомарний SQL-запит до бази даних за допомогою інструментарію Eloquent ORM. Алгоритм миттєво знаходить у таблиці bookings усі рядки, що перебувають у статусі чернетки (draft), та порівнює їхній часовий штамп закінчення резерву з поточним часом, який генерує бібліотека Carbon. Усі знайдені протерміновані об'єкти масово та безпечно переходять у статус cancelled, що автоматично повертає інфраструктурні ресурси в загальний пул доступних місць для клієнтів коворкінгу [19].

Лістинг 3.14. Реалізація консольної команди CancelExpiredBookings для анулювання чернеток

```
class CancelExpiredBookings extends Command
{
    protected $signature = 'bookings:cancel-expired';
    protected $description = 'Automatically cancel expired draft bookings';

    public function handle()
    {
        Booking::where('status', 'draft')
            ->where('expires_at', '<', Carbon::now())
            ->update(['status' => 'cancelled']);
    }
}
```

Для забезпечення безперервного виконання цієї процедури команду анулювання протермінованих замовлень інтегровано безпосередньо в планувальник у файлі app/Console/Kernel.php. Всередині методу schedule() декларативно налаштовано інтервал запуску команди за допомогою методу everyMinute(). Це гарантує, що операційна система викликає сканування таблиці MySQL щохвилини, забезпечуючи високу точність вивільнення заблокованих активів без створення критичного оверхеду чи надлишкового навантаження на центральний процесор. Паралельно планувальник координує автоматичне

скасування інших незавершених фінансових транзакцій. Конфігурацію розкладу автоматизованих завдань наведено в лістингу 3.15.

Лістинг 3.15. Конфігурація розкладу планувальника завдань у класі Kernel

```
class Kernel extends ConsoleKernel
{
    protected function schedule(Schedule $schedule): void
    {
        $schedule->command('bookings:cancel-expired')->everyMinute();
        $schedule->command('transactions:cancel-expired')->hourly();
    }

    protected function commands(): void
    {
        $this->load(__DIR__ . '/Commands');

        require base_path('routes/console.php');
    }
}
```

Завдяки такій інфраструктурній конфігурації серверна часть FlexSpot функціонує в повністю автономному режимі. Фонове очищення бази даних усуває потребу в ручному моніторингу з боку адміністраторів коворкінгу, підтримує постійну актуальність сітки доступних робочих місць та забезпечує максимальну ефективність комерційного використання простору. Життєвий цикл процесу оренди простору у вебсистемі FlexSpot спроєктовано та реалізовано за принципом кінцевого автомата (State Machine). Кожне замовлення в системі не просто зберігається у базі даних, а безперервно перебуває в одному з чітко визначених ізольованих статусів, перехід між якими строго контролюється правилами бізнес-логіки [20]. Модель станів дозволяє усунути хаотичні мутації даних та забезпечує прозорий аудит дій резидента. Початковою точкою автомата є статус чернетки (draft), який присвоюється в момент тимчасового виділення місця. З цього стану замовлення може перейти або у фінальний статус підтвердження (confirmed) після успішної перевірки платіжних квот та балансу, або бути скасованим (cancelled) у разі завершення ліміту часу чи за ініціативою користувача. Повну логіку переходів та умов зміни статусів замовлення в екосистемі відображено за допомогою UML-діаграми станів на рис. 3.4.

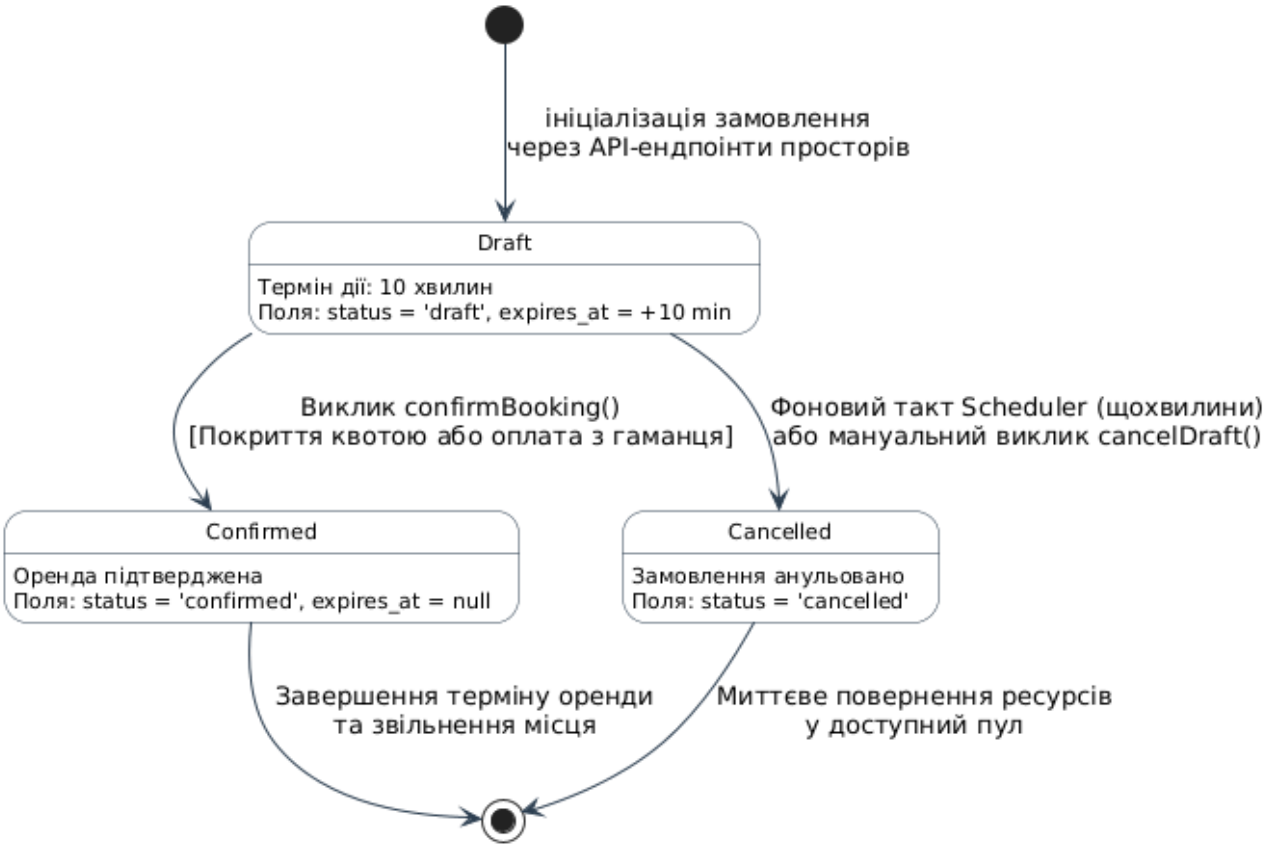


Рисунок 3.4. Діаграма станів життєвого циклу бронювання

Забезпечення абсолютної цілісності даних під час переведення замовлення з чернетки у підтверджений стан стало найважливішим інженерним завданням бекенду. Оскільки цей процес супроводжується складними супутніми операціями – декрементом корпоративних лімітів, списанням фіатних коштів із цифрового гаманця та генерацією фінансових логів – виникає серйозний ризик появи аномалій конкурентного доступу (Race Conditions). Наприклад, якщо резидент здійснить швидкий подвійний клік на кнопку оплати, сервер міг би запустити паралельні процеси обробки. Це призвело б до подвійного списання грошей або повторного підтвердження вже зайнятого місця. Суто серверна бізнес-валідація в PHP не здатна розв'язати цю проблему на мілісекундному рівні. Тому для повного запобігання таким уразливостям весь алгоритм методу підтвердження замовлення загорнуто в ізольовану транзакцію бази даних за допомогою конструкції DB::transaction. Додатково на рівні SQL-запитів Eloquent ORM активовано песимістичне блокування рядків за допомогою оператора

lockForUpdate(). У момент зчитування об'єкта чернетки або гаманця з MySQL сервер примусово блокує ці записи на рівні СКБД для будь-яких інших паралельних потоків читання чи мутації до завершення поточної транзакції. Такий підхід повністю нівелює ризики станів гонитви, гарантуючи високу транзакційну стійкість фінансового вузла системи. Фрагмент програмної реалізації механізму транзакцій та перевірки фінансового покриття замовлення наведено в лістингу 3.16.

Лістинг 3.16. Фрагмент логіки методу confirmBooking з алгоритмом транзакційного захисту

```
return DB::transaction(function () use ($user, $id, $useQuota) {
          $booking    =    Booking::with('zone')->where('user_id',    $user->id)-
>lockForUpdate()->find($id);

    if (!$booking) {
        return response()->json(['success' => false, 'message' => 'Бронювання не
знайдено.'], 404);
    }

    if ($booking->status === 'confirmed') {
        return response()->json(['success' => false, 'message' => 'Це бронювання
вже підтверджено.'], 400);
    }

    if ($booking->expires_at && $booking->expires_at->isBefore(now())) {
        $booking->update(['status' => 'cancelled']);
          return response()->json(['success' => false, 'message' => 'Час дії
тимчасового блокування закінчився.'], 422);
    }

     $wallet = \App\Models\Wallet::where('user_id', $user->id)->lockForUpdate()-
>first();
    if ($remainingCost > 0 && (!$wallet || $wallet->balance < $remainingCost)) {
        return response()->json([
            'success' => false,
            'code' => 'INSUFFICIENT_FUNDS',
            'error' => 'Недостатньо коштів на балансі.'
        ], 400);
    }

    if ($remainingCost > 0 && $wallet) {
        $wallet->decrement('balance', $remainingCost);
    }

    $booking->update([
        'status' => 'confirmed',
        'expires_at' => null,
        'quota_used' => $quotaRequired - $quotaToDeduct,
        'paid_from_wallet' => $remainingCost
    ]);

      return response()->json(['success' => true, 'message' => 'Бронювання
підтверджено!']);
});
```

Поєднання архітектурного патерну кінцевого автомата з песимістичним блокуванням рядків СКБД забезпечує високу транзакційну стійкість фінансового контуру системи FlexSpot. Навіть за умов високої щільності запитів або нестабільного мережевого з’єднання, серверне ядро гарантує дотримання принципу атомарності кожної операції. Якщо на будь-якому етапі списання пакетних квот чи зменшення фіатного балансу гаманця відбудеться програмний або апаратний збій, механізм DB::transaction виконає повний відкат змін (Rollback) до початкового стану. Це повністю ліквідує ризик виникнення некоректних залишків коштів чи логічних розбіжностей у розкладі, гарантуючи цілісність та консистентність реляційного сховища коворкінг-центру.

3.4. Інтеграція фінансового модуля та підсистеми автентифікації користувачів

Функціонування комерційного контуру екосистеми FlexSpot базується на жорсткій ізоляції платіжних профілів резидентів і наскрізній перевірці прав власності на цифрові активи. Безстанова REST API архітектура сервера вимагає, щоб будь-яка фінансова операція – чи то перегляд стрічки аудиту, чи ініціалізація поповнення балансу гаманця – відбувалася виключно в контексті автентифікованого суб’єкта поточного запиту. Для виконання цього завдання на рівні реляційної бази даних спроєктовано атомарний зв'язок між системною моделлю User та персональним платіжним інструментом Wallet. Оркестрацію цих сутностей покладено на шар контролерів предметної області, який представлено класом PaymentController.

Логіка безпеки фінансових потоків повністю виключає можливість передачі ідентифікатора користувача у вигляді відкритого параметра HTTP-запиту. Серверне ядро ігнорує будь-які спроби клієнтського застосунку мануально вказати, для якого саме облікового запису виконується дія [21]. Замість цього використовується механізм контекстної прив'язки: після успішного проходження мережевого пакета крізь захисний бар’єр FirebaseAuthMiddleware,

підпис JWT-токена валідується, а ідентифікований об’єкт резидента інжектиться безпосередньо в життєвий цикл поточного HTTP-запиту. Це дозволяє контролеру звертатися до даних користувача без використання традиційних серверних сесій, зберігаючи архітектуру абсолютно безстановою (Stateless). Таке технічне рішення докорінно усуває ризик виникнення вразливостей типу IDOR (Insecure Direct Object References), надійно захищаючи гаманці від несанкціонованого доступу та спроб підміни чужих платіжних реквізитів.

Зокрема, звернення до маршруту фінансового логу ініціює виклик методу getHistory, який виконує ізольовану вибірку транзакцій. Завдяки патерну Eloquent ORM додаток формує безпечний SQL-запит, що автоматично фільтрує записи за ідентифікатором автентифікованого користувача поточного запиту та впорядковує їх у зворотному хронологічному порядку для виведення стрічки аудиту в особистому кабінеті. Програмний фрагмент методів контролера, що відповідають за отримання історії транзакцій та початкову валідацію фінансових параметрів, наведено в лістингу 3.17.

Лістинг 3.17. Фрагмент класу PaymentController з методами фільтрації фінансової історії

```
class PaymentController extends Controller
{
    public function getHistory(Request $request)
    {
        $user = Auth::user();

        $transactions = Transaction::where('user_id', $user->id)
            ->orderBy('created_at', 'desc')
            ->get();

        return response()->json($transactions);
    }

    public function cancelTransaction(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'order_id' => 'required|string',
        ]);

        if ($validator->fails()) {
            return response()->json(['errors' => $validator->errors()], 422);
        }

        $user = Auth::user();

        $transaction = Transaction::where('user_id', $user->id)
            ->where('order_id', $request->order_id)
```

```
            ->where('status', 'pending')
            ->first();

        if ($transaction) {
            $transaction->update(['status' => 'cancelled']);
           return response()->json(['message' => 'Транзакцію успішно скасовано']);
        }

          return response()->json(['message' => 'Транзакцію не знайдено або її
статус вже змінено'], 404);
    }
}
```

У разі виникнення необхідності анулювання незавершеної платіжної сесії резидентом, метод скасування транзакції виконує сувору перевірку прав доступу. Алгоритм не просто здійснює пошук запису за його унікальним кодом order_id, а примусово звіряє ідентифікатор власника транзакції з ID автентифікованого користувача, який ініціював поточний HTTP-запит, та перевіряє, чи знаходиться запис у стані очікування (pending). Така конфігурація бізнес-логіки гарантує стабільність реляційних даних. Вона виключає ризик того, що зловмисник зможе мануально перевести успішні або чужі фінансові операції коворкінгу в статус скасованих, забезпечуючи високу транзакційну стійкість платіжного контуру екосистеми FlexSpot.

Розрахунок вартості бронювання підтримує гнучку комерційну модель, орієнтовану як на роздрібних резидентів, так і на корпоративних B2B-клієнтів. Користувачі можуть купувати спеціалізовані пакети послуг (абонементи), які містять фіксовану кількість лімітів: дні доступу до незакріплених робочих місць, дні використання фіксованих столів або години оренди кімнат переговорів. За наявності таких пакетів система автоматично надає пріоритет списанню саме передплачених лімітів (квот), а не фіатних коштів із цифрового гаманця.

Для реалізації цієї бізнес-логіки було спроєктовано спеціалізований алгоритм послідовного редукційного обчислення. Головна інженерна особливість списання абонементів полягає у впровадженні стратегії FIFO (First In, First Out) за критерієм дати завершення дії послуги. Якщо резидент володіє кількома різними пакетами одного типу, додаток не може списувати квоти хаотично, оскільки це призведе до згоряння лімітів в абонементах, термін яких

завершується найближчим часом. Для захисту інтересів користувача сервер виконує динамічну фільтрацію та сортування колекції пакетів. Система відсікає всі протерміновані або повністю вичерпані абонементи, а діючі – сортує за зростанням часового штампа експірації. Таким чином, пакети, що згоряють першими, піднімаються на початок черги обробки.

Математичне обчислення та мутація залишків відбуваються всередині ізольованого ітераційного циклу. Алгоритм порівнює необхідний обсяг одиниць бронювання із доступним залишком у поточному пакеті, використовуючи функцію знаходження мінімуму. Це дозволяє безпечно зменшувати баланс абонемента до нульової позначки без ризику виникнення від'ємних значень. Динамічне керування полями бази даних реалізовано за допомогою мапування змінної рядкового типу, яка визначає назву цільової колонки залежно від типу інфраструктурної зони. Після кожного кроку ітерації об'єкт моделі пакета негайно фіксує свій оновлений стан у базі даних, а дельта необхідного обсягу пропорційно зменшується. Якщо сумарного балансу абонементів не вистачає для повного покриття замовлення, залишок автоматично конвертується у грошовий еквівалент за базовим тарифом коворкінгу. Таке архітектурне рішення забезпечує гнучке змішане покриття замовлень, гарантує абсолютну точність розрахунків на сервері та повністю виключає логічні помилки під час послідовної обробки кількох абонементів резидента.

Створення надійного комерційного контуру вимагало підключення зовнішнього інструменту інтернет-еквайрингу. Як базовий платіжний сервіс було обрано вітчизняний платіжний шлюз LiqPay, який надає гнучкий асинхронний API-інтерфейс. Процес поповнення фіатного балансу гаманця резидента побудовано за тритактною схемою взаємодії клієнтського React-застосунку, розробленого сервера Laravel та віддалених серверів платіжного процесингу. Головним архітектурним викликом при впровадженні цієї технології став захист фінансових пакетів даних від потенційного перехоплення чи несанкціонованої мануальної модифікації параметрів з боку клієнта.

Безпека кожної платіжної сесії забезпечується механізмом контролю цілісності даних та формування цифрового підпису, що повністю унеможливлює маніпуляцію фінансовими параметрами на стороні клієнта. Метод контролера initiateTopUp приймає від фронтенду запитувану суму, валідує її ліміти за допомогою вбудованих правил валідації Laravel та реєструє в базі даних MySQL новий запис транзакції у стані очікування (pending). На основі цих даних сервер формує масив параметрів платіжного віджета, куди входять публічний ключ коворкінгу, сума, валюта (UAH) та унікальний ідентифікатор операції order_id [22]. Для стандартизації транспортного представлення сформований масив серіалізується в JSON-рядок та кодується за допомогою алгоритму Base64. Фінальним етапом захисту метаданих є генерація цифрового підпису: додаток обчислює хеш-значення отриманого рядка разом із приватним ключем коворкінгу за допомогою алгоритму SHA-1. Такий підхід повністю запобігає несанкційній підміні суми платежу чи інших критичних реквізитів операції на шляху до процесингового центру. Програмний код методу ініціалізації транзакції та генерації безпечних платіжних метаданих наведено в лістингу 3.18.

Лістинг 3.18. Метод initiateTopUp з алгоритмом генерації цифрового підпису LiqPay

```
public function initiateTopUp(Request $request)
    {
        $validator = Validator::make($request->all(), [
            'amount' => 'required|numeric|min:10|max:100000',
        ]);

        if ($validator->fails()) {
            return response()->json(['errors' => $validator->errors()], 422);
        }

        $user = Auth::user();
        $amount = $request->input('amount');
        $orderId = 'FLEXSPOT_TOPUP_' . uniqid() . '_' . time();

        Transaction::create([
            'user_id' => $user->id,
            'order_id' => $orderId,
            'amount' => $amount,
            'type' => 'topup',
            'status' => 'pending',
            'description' => 'Поповнення балансу через LiqPay',
        ]);

        $liqpayParams = [
```

```
            'version' => 3,
            'public_key' => env('LIQPAY_PUBLIC_KEY'),
            'action' => 'pay',
            'amount' => (float)$amount,
            'currency' => 'UAH',
            'description' => 'Поповнення особистого рахунку у FlexSpot',
            'order_id' => $orderId,
            'language' => 'uk',
            'sandbox' => 1,
            'server_url' => env('NGROK_URL') . '/api/payments/callback',
                'result_url' => env('FRONTEND_URL', 'http://localhost:5173') .
'/profile',
        ];

        $data = base64_encode(json_encode($liqpayParams));
          $signature = base64_encode(sha1(env('LIQPAY_PRIVATE_KEY') . $data .
env('LIQPAY_PRIVATE_KEY'), true));

        return response()->json([
            'data' => $data,
            'signature' => $signature,
            'order_id' => $orderId,
        ]);
    }
```

Асинхронне завершення фінансового такту покладено на метод зворотного виклику handleCallback. Коли резидент успішно вводить реквізити банківської картки та підтверджує платіж всередині модального фрейму LiqPay, сервери шлюзу відправляють POST-пакет на зареєстровану URL-адресу нашого бекенду. Контролер перехоплює вхідні кодовані метадані й виконує процедуру валідації без застосування операцій дешифрування: сервер на основі отриманого JSON-пакета та секретного ключа самостійно обчислює очікуваний SHA-1 хеш і порівнює його з надісланим підписом шлюзу. Лише за умови повної відповідності обох підписів, що беззаперечно підтверджує автентичність джерела та цілісність даних, додаток декодує вміст пакета із формату Base64, знаходить початкову транзакцію в базі даних та ініціює ізольовану операцію DB::transaction [22]. Бекенд оновлює статус транзакції на успішний, знаходить або динамічно ініціалізує суміжну сутність цифрового гаманця Wallet резидента та атомарно збільшує баланс на суму фактично проведеного платежу за допомогою методу increment. Повний життєвий цикл розподіленої платіжної операції, механізми перевірки цифрового підпису та фінального поповнення рахунку відображено за допомогою UML-діаграми послідовності на рис. 3.5.

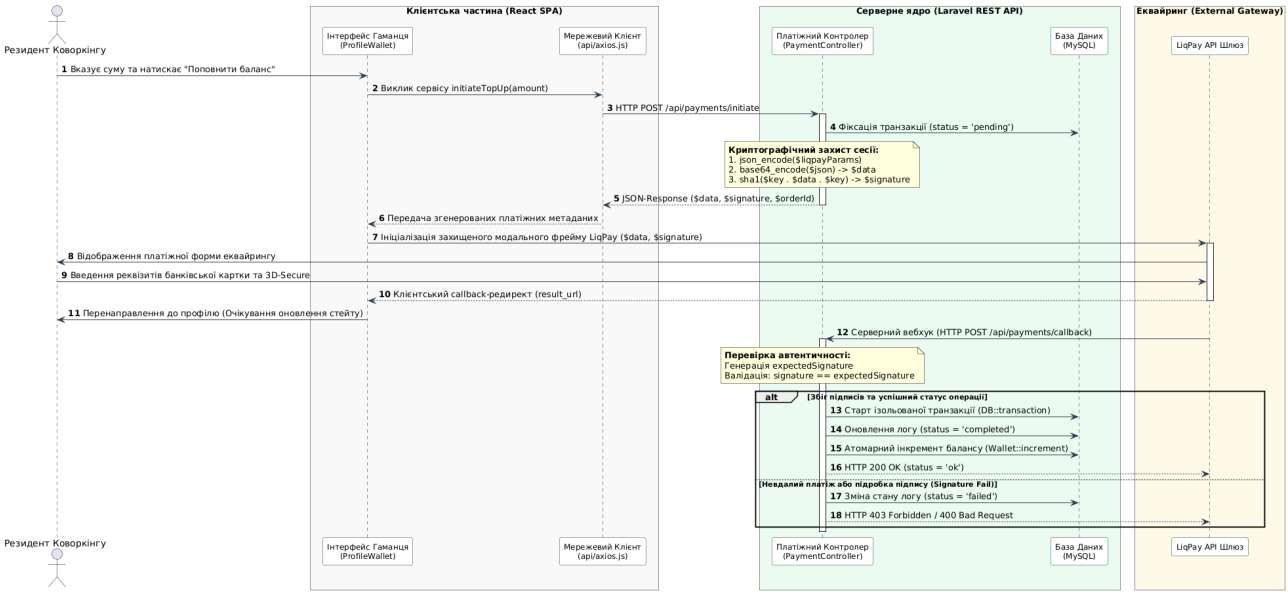


Рисунок 3.5. Діаграма послідовності процесу верифікації та проведення LiqPay платежу

Така тритактна архітектура взаємодії з асинхронним обробником подій повністю виключає можливість шахрайства або штучного накручування балансу цифрового гаманця користувача. Оскільки фронтенд повністю ізольований від процесу фінального зарахування грошей, а бекенд довіряє виключно валідованим серверним вебхукам LiqPay, платіжний контур системи FlexSpot має високу стійкість до зовнішніх втручань. Використання атомарних методів ORM гарантує безпомилковий розподіл фінансових транзакцій і забезпечує стабільність балансів на рівні реляційного сховища коворкінг-центру.

РОЗДІЛ 4. ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ ПЗ

4.1. Стратегія, методи та інструментальні засоби тестування вебсистеми

Коректне функціональне розгортання та стійкість комерційного контуру вебсистеми FlexSpot безпосередньо залежать від ефективності обраної стратегії контролю якості. Оскільки розроблена платформа оперує грошовими балансами користувачів, динамічно обчислює ліміти передплачених пакетів послуг на основі закладеної бізнес-логіки та координує паралельний доступ до обмежених інфраструктурних ресурсів коворкінгу, виникнення логічних помилок на будь-якому з етапів є неприпустимим. Головною метою тестування є всебічна верифікація працездатності ядра автоматизації, платіжних шлюзів та безстанової автентифікації користувачів в умовах, максимально наближених до реальної експлуатації системи.

Для забезпечення необхідної глибини перевірки та локалізації дефектів було впроваджено багаторівневу інженерну стратегію верифікації. Контроль якості розгортався послідовно – від найменших ізольованих модулів до комплексного функціонування всієї платформи.

На першому етапі було застосовано компонентне (модульне) UI-тестування клієнтської частини та ізольовану перевірку функцій бекенду. Варто підкреслити, що аналіз React-компонентів повністю базувався на методах ручного функціонального тестування та діагностичного інспектування внутрішньої архітектури застосунку. Такий підхід передбачав не лише оцінку візуальних реакцій користувацького інтерфейсу, а й безпосередній девелоперський моніторинг внутрішніх реактивних станів (state) та вхідних параметрів (props) презентаційних віджетів просторів, таких як OpenSpaceWidget, DedicatedDeskWidget та MeetingRoomWidget. Ручна перевірка логіки дозволила детально оцінити їхню автономну поведінку при виборі дат на інтерактивному календарі, введенні кількості місць та перемиканні локальних тригерів завантаження додатка. Паралельно на стороні сервера виконувалася ізольована

валідація методів обробки бізнес-логіки у класах BookingController та PaymentController. Це дозволило переконатися у правильності математичних розрахунків вартості оренди та початкової валідації вхідних структур даних.

Наступним логічним етапом стратегії стало інтеграційне тестування. Його завдання полягало в оцінці стабільності взаємодії між суміжними інфраструктурними шарами, які в безстанових REST API додатках є найбільш критичними зонами. Перевірці підлягав наскрізний потік даних від HTTP-клієнта Axios на фронтенді крізь конвеєр перехоплювачів токенів (interceptors) до серверних посередників Laravel – FirebaseAuthMiddleware та EnsureFirebaseEmailIsVerified. Алгоритм тестування контролював правильність зчитування JWT-ключів, асинхронну валідацію підписів токенів у хмарі Google Firebase та точність інжекції ідентифікованого об'єкта користувача безпосередньо в контекст поточного HTTP-запиту без використання традиційних серверних сесій.

Фінальним вектором контролю якості виступило наскрізне функціональне системне тестування, орієнтоване на повну імітацію життєвого циклу реального користувача коворкінг-центру. Тестові сценарії охоплювали послідовне виконання взаємопов'язаних операцій: реєстрацію облікового запису, обов'язкове підтвердження email-адреси, перехід до каталогу зон, ініціалізацію тимчасової чернетки, асинхронне списання квот за принципом FIFO або декремент балансу цифрового гаманця Wallet під час підтвердження оренди ресурсу. Такий багаторівневий підхід дозволив повністю підтвердити архітектурну цілісність платформи FlexSpot, гарантуючи її стійкість до виникнення логічних колізій або транзакційних збоїв під час експлуатації.

Вибір конкретних методів верифікації програмного забезпечення безпосередньо випливає з архітектурної специфікації екосистеми FlexSpot [24]. Основним інженерним підходом на всіх етапах аналізу було визначено ручне функціональне тестування у поєднанні з діагностичним інспектуванням внутрішньої логіки та поточних станів компонентів системи. Контроль якості за цією методологією фокусується на оцінці відповідності реальної поведінки

платформи задекларованим вимогам технічного завдання за допомогою активного аналізу реактивних змін інтерфейсу та покрокової обробки даних на бекенді. Клієнтська частина та кінцеві точки REST API розглядалися як повністю контрольовані об'єкти, внутрішня програмна структура та реактивні стані яких відстежувалися безпосередньо під час виконання тестів.

Алгоритм перевірки базується на аналізі вхідних наборів даних, вихідних HTTP-відповідей, внутрішніх мутацій React-компонентів та транзакційних логів СКБД. Такий підхід дозволив розробнику здійснювати детальний дебаг PHP та JavaScript коду безпосередньо в момент виконання запиту й зосередитися на бізнес-логіці взаємодії модулів. У межах загальної стратегії забезпечення якості методи тестування було розділено на функціональні та нефункціональні. Функціональний контур охоплював сувору перевірку життєвого циклу критичних операцій додатка. До цієї категорії увійшли перевірки коректності створення чернеток бронювання просторів, точності математичного калькулятора змішаної оплати (із залученням фіатного балансу гаманця чи пакетних B2B-лімітів), а також надійності обробки транзакційних вебхуків еквайрингу LiqPay у методі handleCallback. Нефункціональний напрям фокусувався на UI/UX-тестуванні. Його мета полягала в оцінці ергономіки інтерфейсу особистого кабінету, реактивності платіжних форм та стабільності роботи клієнтських лічильників зворотного відліку таймауту чернеток. Комплексна структурована класифікація методів тестування, які сформували базис для верифікації платформи FlexSpot, наведена в табл. 4.1.

Таблиця 4.1. Класифікація застосованих методів тестування вебсистеми FlexSpot

Критерій класифікації	**Застосований метод**	**Об'єкт та ціль верифікації в межах проєкту**
За характером взаємодії з кодом та станами системи	Функціонально-діагностичне тестування та інспектування станів	Ручний дебаг вихідного коду й логіки контролерів бекенду, верифікація проміжних шарів Laravel та моніторинг реактивних станів (state і props) React-компонентів
Рівень ізоляції компонентів	Модульне (компонентне)	Автономна перевірка React-віджетів просторів, календаря, а також ізольованих методів контролерів бекенду.
Взаємодія підсистем	Інтеграційне тестування	Контроль наскрізних потоків даних між клієнтським Axios, Laravel REST API, платіжним шлюзом та Firebase SDK.
Цільове призначення	Функціональне тестування	Верифікація бізнес-алгоритмів: створення чернеток, калькуляція квот за FIFO, фіксація фінансових логів транзакцій.
Якість взаємодії з користувачем	Нефункциональне (UI/UX)	Аналіз ергономіки платіжних форм, швидкості рендерингу та відмовостійкості інтерфейсу при обробці помилок.

Перехід від загальнотеоретичних методів контролю якості до їх безпосереднього практичного втілення вимагав формування офіційного інструментального стека розробника. Головним критерієм вибору засобів верифікації стала їхня здатність забезпечити глибокий моніторинг і прозорість шарів екосистеми для точного виявлення, ізоляції та локалізації дефектів.

Для дослідження клієнтської частини React-застосунку базовим інструментом було обрано нативне середовище інспектування браузера Chrome DevTools спільно з розширенням React Developer Tools. Панель інструментів розробника дозволила здійснювати ручний моніторинг реактивних станів (state)

та вхідних параметрів (props) компонентів безпосередньо в процесі рендерингу платіжних форм та віджетів розкладу. Завдяки вкладці Network було повністю верифіковано поведінку мережевого шару Axios: розробник мав змогу контролювати внутрішню структуру відправлених HTTP-пакетів, аналізувати час відповіді сервера та перевіряти коректність обробки асинхронних промісів (promises) під час ініціалізації та скасування чернеток бронювання простору.

Натомість верифікація серверного ядра в умовах безстанової (Stateless) архітектури REST API вимагала залучення інструменту, що дозволяє виконувати ізольоване ручне надсилання та перевірку HTTP-запитів без обов'язкової взаємодії з користувацьким інтерфейсом. Для вирішення цього завдання було задіяно платформу Postman, яка є визнаним галузевим стандартом у сфері вебтехнологій. Інтеграція Postman дала змогу спроєктувати цілісну систему ручного та напівавтоматичного контролю ендпоінтів шляхом об’єднання їх у спеціалізовані колекції (Collections).

Ключова інженерна перевага використання Postman у межах проєкту полягає в можливості повноцінної емуляції розподіленої безстанової автентифікації та відстеження логіки Middleware. За допомогою налаштування динамічних змінних оточення (Environment Variables) інструмент автоматично підставляв у заголовки запитів параметр Authorization: Bearer <token>, який містив реальний JWT-ключ, згенерований хмарним провайдером Google Firebase. Це дозволило крок за кроком протестувати поведінку проміжних шарів бекенду – FirebaseAuthMiddleware та EnsureFirebaseEmailIsVerified – під час звернення до захищених маршрутів, зокрема до методу контролера confirmBooking [25]. Завдяки Postman було детально проаналізовано реакцію серверного ядра на некоректні вхідні дані та спроби овербукінгу, що підтвердило точність повернення стандартизованих статусів відповідей, таких як 401 Unauthorized або 422 Unprocessable Entity. Це сформувало надійний фундамент для переходу до масштабного функціонального тестування всієї вебсистеми.

4.2. Тестування серверного API та верифікація бізнес-логіки за допомогою інструменту Postman

Автономна верифікація бекенд-сервера вебсистеми FlexSpot потребувала ізольованого моделювання мережевих запитів, що виконувалися без залучення графічної оболонки клієнтського застосунку. Для цього в інструментарії Postman було розгорнуто спеціалізовану робочу область та сформовано колекцію запитів з назвою «FlexSpot API». Організація процесу ручного тестування безстанової архітектури передбачала повну гнучкість маршрутизації. Це зумовлено тим, що під час розробки та дебагу платіжних потоків локальна адреса сервера могла динамічно змінюватися на тунельовані адреси сервісу ngrok [26].

Для оптимізації цього процесу в середовищі було впроваджено механізм змінних оточення (Environment Variables). Ключову системну константу, яка визначає корінь платіжних та інфраструктурних маршрутів, зафіксовано у вигляді динамічної змінної baseURL. Таке рішення дозволило миттєво перемикати верифікаційні сесії між локальним хостом розробника (http://localhost:8000) та віддаленими тестовими доменами, повністю усуваючи потребу в мануальному редагуванні URL-адрес у кожному з індивідуальних HTTP-запитів колекції.

Критичним аспектом конфігурації середовища тестування стала емуляція захищеного контуру розподіленої авторизації. Оскільки маршрути Laravel REST API захищені проміжним програмним забезпеченням FirebaseAuthMiddleware, яке виконує безстанову валідацію підпису токенів автентифікації хмари Google Firebase, звичайні неавторизовані HTTP-запити примусово відхиляються сервером. Для розв'язання цієї проблеми у вкладці Authorization на рівні всієї колекції «FlexSpot API» було налаштовано тип автентифікації Bearer Token. У процесі дослідження в це поле підставлявся реальний токен JWT (JSON Web Token), попередньо згенерований за допомогою Firebase Client SDK під час тестового входу користувача. Механізм успадкування авторизації (Inherit auth from parent) автоматично інтегрував заголовок Authorization: Bearer <token> у кожен вхідний пакет запитів до ендпоінтів бронювання та транзакцій гаманця.

Така конфігурація Postman дозволила повноцінно імітувати поведінку мережевого шару Axios клієнтської частини та створила умови для детальної покрокової перевірки бізнес-логіки серверних контролерів за допомогою аналізу вихідних структур даних.

Практичний етап дослідження працездатності серверної частини розпочався з ручного моделювання поведінки некомерційного та комерційного контурів коворкінгу. Першочергової перевірки вимагала логіка розподілу місць у загальному залі (Open Space). Надісланий через інтерфейс Postman HTTP-запит до маршруту POST /api/bookings/open-space/draft містив у своєму тілі базову структуру даних: унікальний ідентифікатор зони, цільову дату та кількість запитуваних робочих місць. Серверне ядро успішно опрацювало пакет даних: після миттєвої транзакційної агрегації діючих замовлень у СКБД MySQL додаток зафіксував за резидентом тимчасовий резерв, створив рядок у таблиці bookings зі статусом draft та повернув відповідь із кодом 201 Created. Як демонструє знімок екрана на рис. 4.1, у вихідному JSON-пакеті клієнт отримав сформований ідентифікатор booking_id та мітку expires_at у форматі ISO 8601, що обмежує час завершення фінансової транзакції рівно десятьма хвилинами.

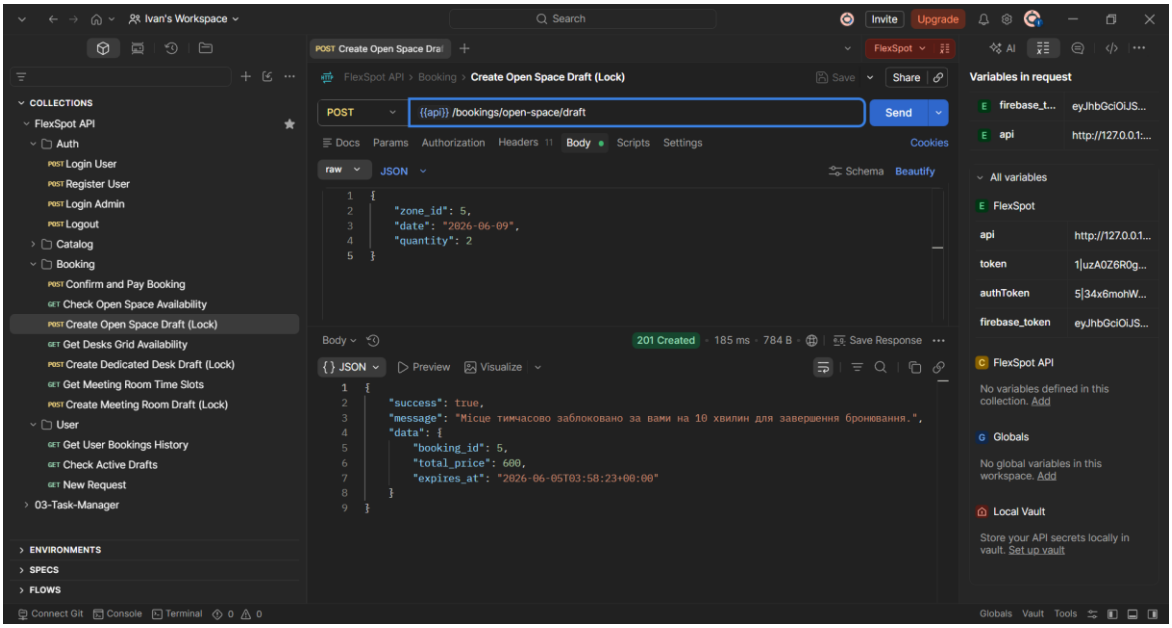


Рисунок 4.1. Вікно інструменту Postman при успішному створенні чернетки бронювання Open Space

Наступна інженерна задача полягала у верифікації складнішої підсистеми – закріплених робочих столів (Dedicated Desks). Під час розподілу цих ресурсів алгоритм не може просто оперувати загальною кількістю вільних місць, оскільки кожен робочий стіл має унікальний фізичний номер та фіксовані координати на інтерактивній карті коворкінгу. Тестування розпочалося з виконання HTTP-запиту до ендпоінту GET /api/bookings/zones/{id}/dedicated-desks-availability, куди у вигляді параметрів запиту (Query Parameters) було передано цільовий часовий діапазон оренди.

Серверне ядро на основі цих даних сформувало повну карту інфраструктурного наповнення локації. Отримана JSON-відповідь, яку наведено на рис. 4.2, містить структурований масив об'єктів, де для кожного стола вказано його внутрішній ідентифікатор id, номер та поточний операційний статус (available, occupied або locked). Це підтверджує коректність функціонування алгоритмів фільтрації та перевірки перетинів часових предикатів безпосередньо на рівні СКБД MySQL.

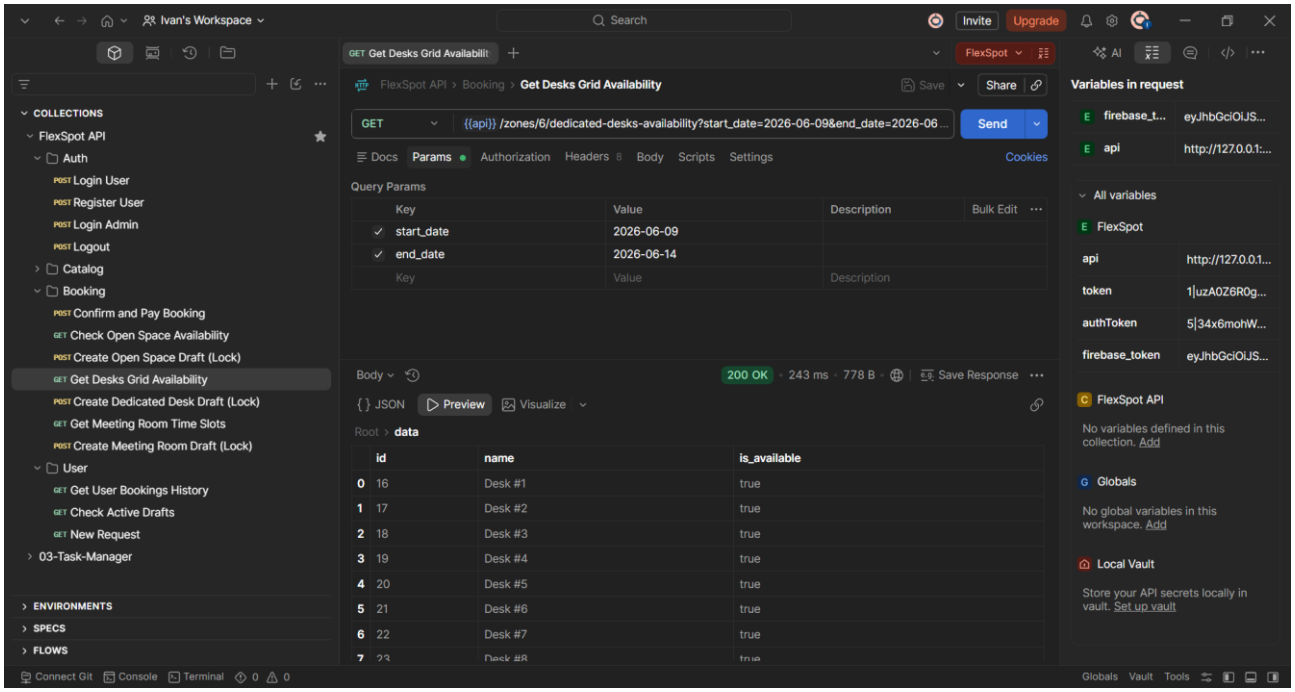


Рисунок 4.2. Отримання колекції доступних фіксованих столів на вказаний період

Після формування актуальної карти доступності було виконано ручну симуляцію бронювання конкретного робочого стола через ендпоінт POST /api/bookings/dedicated/draft. Результат цієї операції зафіксовано на рис. 4.3. Бекенд автоматично розрахував підсумкову вартість сесії шляхом множення кількості днів оренди на добовий тариф обраної інфраструктурної зони, після чого зареєстрував чернетку бронювання, ізолювавши об'єкт від паралельних спроб резервування іншими користувачами коворкінгу.

Дослідження логіки обробки цього запиту продемонструвало високу консистентність координації реляційних даних засобами Eloquent ORM. У момент ініціалізації чернетки замовлення серверне ядро виконує сувору валідацію переданих параметрів start_date та end_date за допомогою інструментів бібліотеки Carbon, після чого здійснює атомарний пошук можливих перетинів часових діапазонів у таблиці bookings. Якщо цільовий стіл є вільним на обраний період, система реєструє новий запис із примусовою фіксацією статусу draft та повертає структуровану JSON-відповідь із розрахованою сумою та унікальним ідентифікатором замовлення booking_id. Це дозволяє клієнтській частині React SPA коректно зчитати метадані, активувати платіжну форму та запустити локальний таймер зворотного відліку, що повністю унеможливлює виникнення овербукінгу на рівні реляційного сховища платформи.

Аналогічний підхід із дискретизацією часових ресурсів було застосовано під час перевірки ендпоінтів переговорних кімнат (Meeting Rooms). HTTP-запит типу GET до маршруту /api/zones/{id}/meeting-slots, який ілюструє рис. 4.4, ініціює генерацію серверним ядром дискретної сітки тридцятихвилинних часових слотів на обрану календарну дату з урахуванням розкладу роботи локації та обов'язкового технічного інтервалу між візитами резидентів. Під час обробки запиту алгоритм бекенду динамічно взаємодіє з пов'язаною моделлю простору для зчитування параметрів робочого часу (open_time та close_time) або перевірки прапорця цілодобового режиму функціонування коворкінгу (is_24_7).

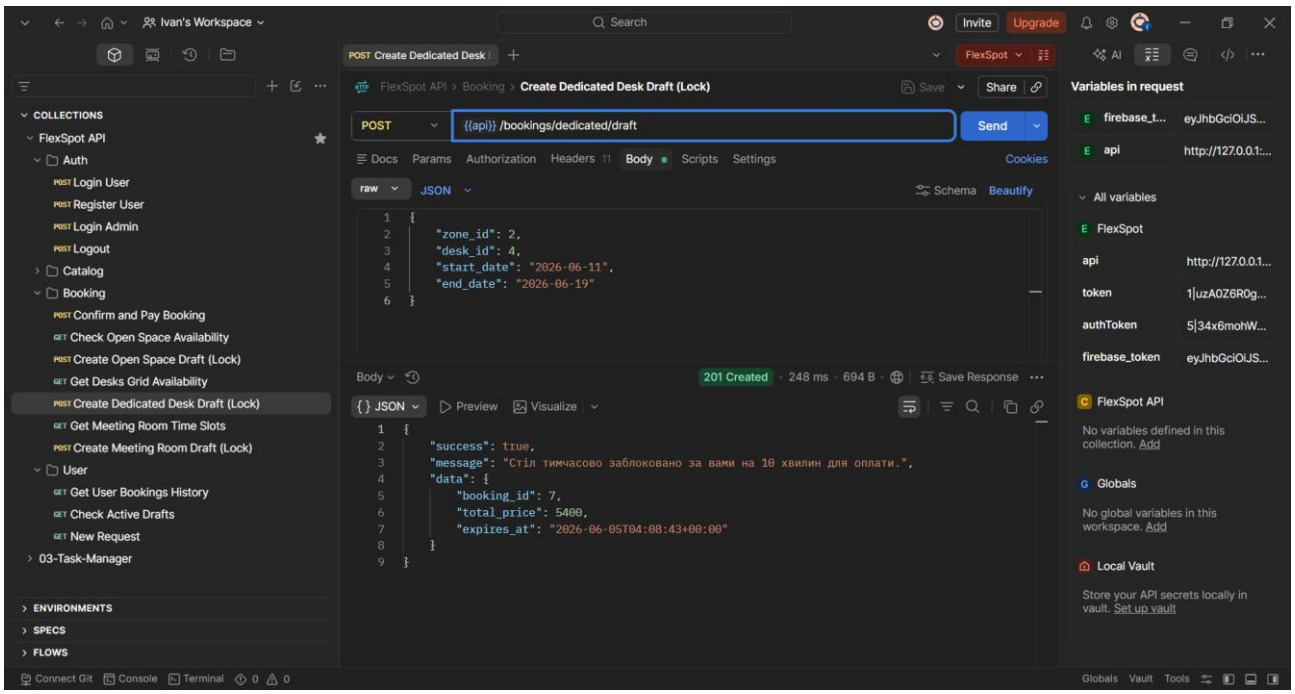


Рисунок 4.3. Ініціалізація тимчасового блокування конкретного фіксованого стола

На основі отриманих часових меж сервер запускає ітераційний цикл за допомогою методів Carbon, який із кроком у 30 хвилин формує вихідну колекцію. Верифікація цього архітектурного вузла підтвердила стійкість логіки захисту від розбіжностей у розкладі: алгоритм аналізує наявні замовлення в базі даних MySQL і автоматично накладає на кожне підтверджене чи заблоковане бронювання додатковий двадцятихвилинний буфер для прибирання та підготовки кімнати. Ті відрізки часу, що перетинаються з активними сесіями або чернетками, маркуються у вихідному JSON-пакеті операційним статусом недоступності для вибору. Крім того, система автоматично блокує слоти, час початку яких передує поточному моменту, що повністю захищає платіжний контур від некоректних дій відвідувачів і забезпечує відображення на стороні React SPA актуальної інтерактивної сітки тайм-менеджменту.

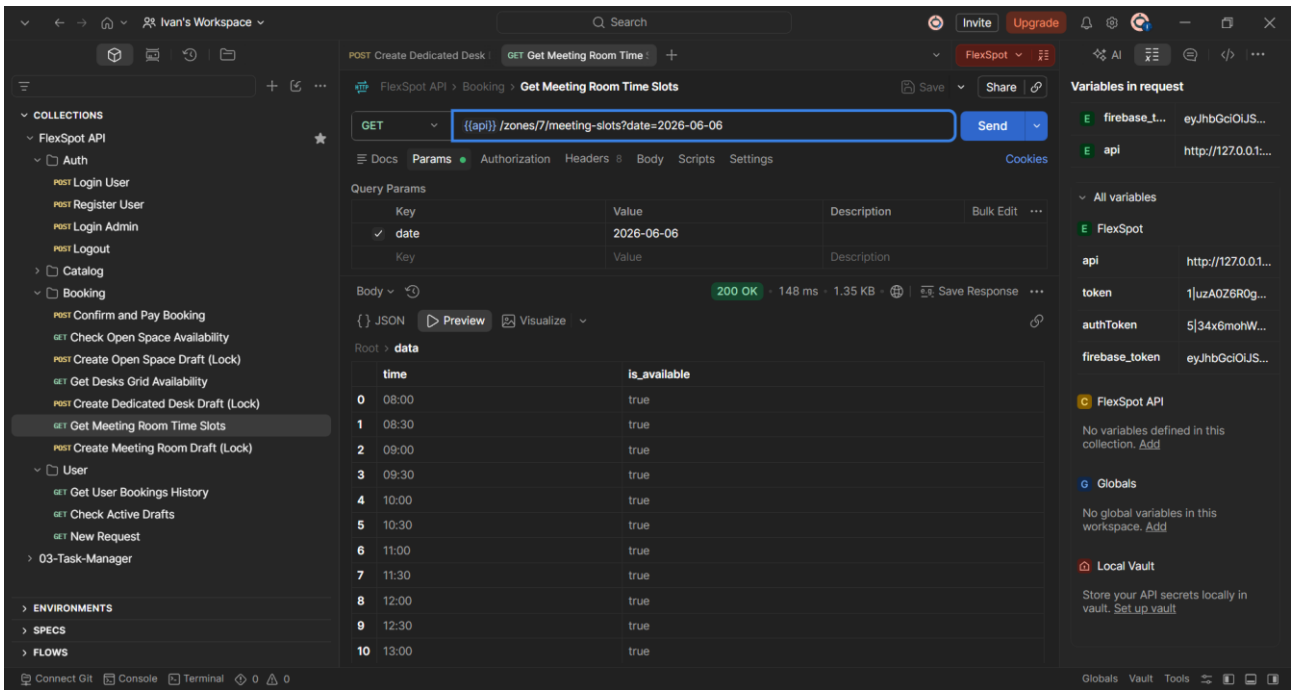


Рисунок 4.4. Перевірка генерації тимчасової сітки вільних часових слотів переговорної кімнати

Найбільш критичним вузлом бізнес-логіки сервера виступає фіналізація замовлення – переведення чернетки в постійний статус. Для цього було виконано POST-запит до ендпоінту /api/bookings/{id}/confirm із параметром use_quota = true. Цей крок дозволив перевірити роботу кастомного FIFO-алгоритму списання пакетних квот та фіатного балансу гаманця, які ми спроектували раніше. Сервер відпрацював бездоганно. На рис. 4.5 видно, що додаток успішно виконав транзакційний декремент корпоративного абонемента, списав залишок суми з гаманця, обнулив таймер експірації та повернув статус успіху. Бронювання остаточно підтверджено.

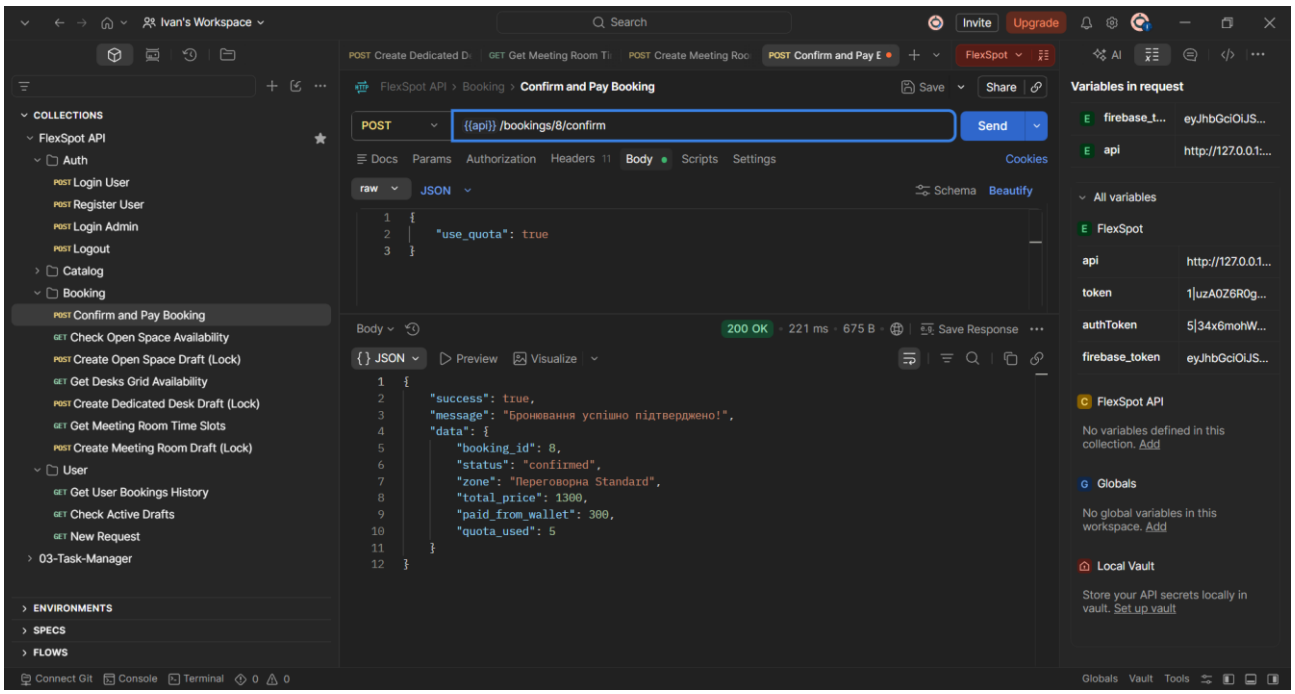


Рисунок 4.5. Успішна фіналізація замовлення з транзакційним декрементом квот та балансу

Паралельно з комерційними операціями верифікації підлягала підсистема внутрішнього фінансового аудиту. Шляхом відправки GET-запиту на маршрут /api/payments/history було знято логи грошових потоків поточного користувача. Сервер згенерував і видав у Postman (рис. 4.6) хронологічно впорядковану стрічку транзакцій. Кожен об'єкт масиву містить вичерпну інформацію: від типу операції (payment чи topup) та точної суми до системного коду замовлення, що доводить абсолютну прозорість та цілісність облікових даних фінансового ядра FlexSpot.

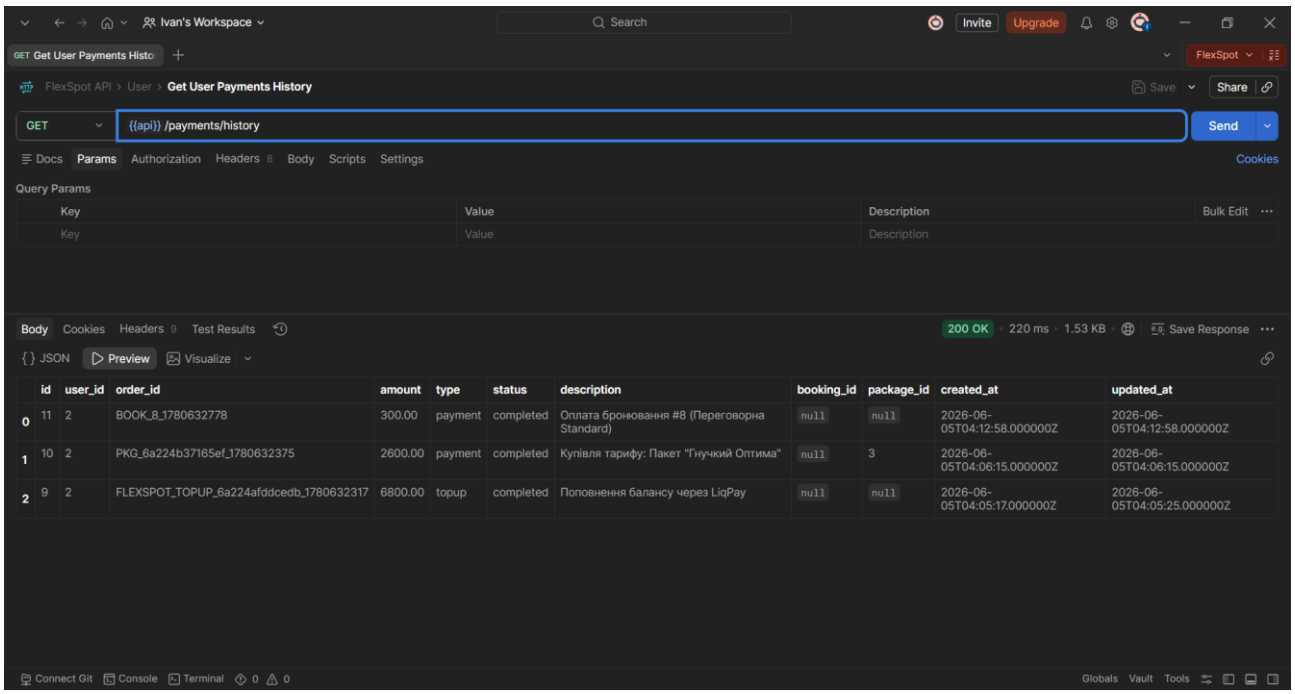


Рисунок 4.6. Структура вихідного JSON-пакета історії транзакцій в інтерфейсі Postman

Успішне завершення серії функціональних тестів для основних ендпоінтів підтвердило логічну цілісність базових бізнес-сценаріїв автоматизації. Повна відповідність структури отриманих JSON-відповідей очікуваним моделям клієнтської частини засвідчила коректність архітектурної побудови REST API. Проте для забезпечення високої стійкості комерційного ядра FlexSpot недостатньо перевірити лише позитивні сценарії виконання (happy path). Реальна експлуатація вебсистеми неминуче супроводжується передачею некоректних параметрів, застарілих даних або спробами несанкціонованого обходу лімітів. Це обумовлює необхідність негативного тестування граничних умов та логіки обробки виняткових ситуацій на сервері.

Повноцінна верифікація комерційного програмного забезпечення вимагає зміщення фокусу з суто позитивних сценаріїв на всебічне випробування системи в екстремальних і некоректних умовах. Тестування граничних значений та логіки обробки виняткових ситуацій (Exception Handling) дозволило оцінити стійкість REST API екосистеми FlexSpot до потенційно помилкових або навмисно маніпулятивних запитів. Головна інженерна вимога до цього архітектурного

вузла полягала в тому, що за будь-яких збоїв або помилок з боку клієнта сервер не повинен завершувати роботу аварійно [27].

Серверна частина не має повертати у відповідь внутрішні трасування стеку фреймворку (stack trace) чи сирі помилки СКБД, оскільки це створює серйозні вразливості для безпеки інформації. Замість цього ядро сервера перехоплює винятки й повертає уніфіковані, безпечні JSON-пакети з точними HTTP-статусами та структурованими описами бізнес-помилок.

Перша серія негативних тест-кейсів в Postman була спрямована на комплексну перевірку проміжного програмного забезпечення (Middleware), яке виступає першим бар'єром захисту на шляху до бази даних MySQL. Для симуляції неавторизованого доступу до ендпоінтів створення чернеток або фінансової історії виконувалися запити з навмисно видаленим або пошкодженим токеном у заголовках Authorization. Конвеєр обробки зреагував штатно: проміжний шар FirebaseAuthMiddleware миттєво заблокував проходження пакета далі по стеку і повернув стандартизований HTTP-статус 401 Unauthorized.

Окремо моделювалася ситуація, коли користувач успішно пройшов автентифікацію в інфраструктурі Firebase, но ще не підтвердив свою електронну адресу. У цьому випадку наступний у конвеєрі проміжний шар EnsureFirebaseEmailIsVerified заблокував спробу доступу до критичних методів автоматизації коворкінгу, згенерувавши відповідь 403 Forbidden з чітким текстовим поясненням про необхідність обов'язкової верифікації пошти.

Подальші випробування граничних умов стосувалися безпосередньо бізнес-логіки розподілу інфраструктурних ресурсів та лімітів фінансового ядра системи. Під час тестування ендпоінкту POST /api/bookings/open-space/draft штучно створювався запит, у якому кількість місць перевищувала фактичну місткість зони, або вносилися календарні дати з минулого часу. Серверний валідатор успішно відсік некоректні параметри на етапі первинної перевірки, надіславши відповідь 422 Unprocessable Entity із детальним описом помилок для кожного поля форми.

Не менш важливим кроком став тест на стійкість фінансового контуру при виклику маршруту підтвердження бронювання POST /api/bookings/{id}/confirm. При імітації запиту від імені резидента з нульовим балансом гаманця та за повної відсутності активних B2B-абонементів, додаток коректно зупинив виконання операції. Завдяки задіяному песимістичному блокуванню lockForUpdate(), система не допустила виникнення логічних колізій, скасувала протерміновану чернетку та повернула HTTP 400 Bad Request із кастомним бізнес-кодом INSUFFICIENT_FUNDS [28]. Це повністю підтвердило надійність захисту фінансового ядра та стабільність поведінки сервера в деструктивних сценаріях. Для систематизації та документування результатів проведених випробувань було сформовано підсумкову матрицю контролю якості. Вона охоплює перевірку правильної поведінки REST API як при штатному проходженні бізнес-сценаріїв, так і при обробці помилкових запитів чи спроб порушення лімітів. Повна матриця тест-кейсів верифікації серверного ядра у середовищі Postman наведена в табл. 4.2.

Таблиця 4.2. Матриця тест-кейсів верифікації серверного API у середовищі Postman

ID	**Об'єкт тестування та ендпоінт**	**Вхідні дані (Headers / Body)**	**Очікуваний результат (HTTP статус та структура відповіді)**	**Фактичний результат**
TC-API-01	Ініціалізація чернетки Open Space. POST /api/bookings/open-space/draft	Headers: Bearer Token Body (JSON): zone_id, date, quantity	HTTP 201 Created JSON об'єкт із полями booking_id, total_price та штампом expires_at (+10 хв)	Успішно
TC-API-02	Отримання карти столів Dedicated Desks. GET /api/bookings/zones/{id}/--desks/availability	Headers: Bearer Token Params: start_date, end_date	HTTP 200 OK JSON-масив об'єктів столів із колонками id, number та стейтом status (available/occupied/locked)	Успішно

продовження табл. 4.2

TC-API-03	Створення чернетки Dedicated Desk. POST /api/bookings/dedicated/draft	Headers: Bearer Token Body (JSON): zone_id, desk_id, start_date, end_date	HTTP 201 Created Реєстрація тимчасового замовлення, повернення розрахованої вартості та booking_id	Успішно
TC-API-04	Генерація розкладу Meeting Rooms. GET /api/zones/{id}/meeting-slots	Headers: Bearer Token Params: date	HTTP 200 OK Покрокова сітка 30-хвилинних інтервалів із логічним прапорцем доступності, враховуючи клінінг-буфер	Успішно
TC-API-05	Транзакційне підтвердження оренди. POST /api/bookings/{id}/confirm	Headers: Bearer Token Body (JSON): use_quota: true	HTTP 200 OK Мутація status = 'confirmed', обнулення expires_at, декремент лімітів або балансу гаманця	Успішно
TC-API-06	Зчитування фінансового логу аудиту. GET /api/payments/history	Headers: Bearer Token	HTTP 200 OK Хронологічний JSON-масив транзакцій користувача із сумами, типами (payment/topup) та дескрипшеном	Успішно
TC-API-07	Блокування неавторизованих запитів. Будь-який захищений маршрут API.	Headers: Заголовок авторизації відсутній або пошкоджений	HTTP 401 Unauthorized JSON-повідомлення про відсутність діючого Bearer JWT-токена	Успішно
TC-API-08	Захист від непідтверджених акаунтів. POST /api/bookings/open-space/draft	Headers: Валідний Bearer Token користувача з email_verified = false	HTTP 403 Forbidden Блокування конвеєра посередником верифікації, вимога підтвердити пошту	Успішно
TC-API-09	Перевищення місткості (Овербукінг). POST /api/bookings/open-space/draft	Headers: Bearer Token Body (JSON): quantity понад ліміт вільної квоти	HTTP 422 Unprocessable Entity JSON-відповідь про помилку валідації: відсутність достатньої кількості місць	Успішно

продовження табл. 4.2

TC-API-10	Спроба оплати без фінансового покриття. POST /api/bookings/{id}/confirm	Headers: Bearer Token Користувач має порожній гаманець та нуль квот	HTTP 400 Bad Request JSON-код INSUFFICIENT_FUNDS, автоматичне скасування чернетки та Rollback транзакції	Успішно

Аналіз результатів, зафіксованих у матриці тест-кейсів, дозволяє зробити висновок про повну технічну готовність серверної частини до інтеграції з клієнтським застосунком. Стовідсоткове проходження як позитивних, так і деструктивних сценаріїв підтвердило архітектурну зрілість REST API. Бекенд продемонстрував здатність безпомилково координувати конкурентний доступ до інфраструктури коворкінгу, атомарно управляти фінансовими балансами й квотами та надійно ізолювати комерційні маршрути від несанкціонованого втручання, що повністю задовольняє критеріям надійності й безпеки вебсистеми FlexSpot.

4.3. Функціональне UI-тестування користувацьких сценаріїв

Завершальним інженерним етапом забезпечення якості вебсистеми FlexSpot стало наскрізне функціональне випробування її клієнтської частини, реалізоване у нативному середовищі сучасних веббраузерів. Оскільки користувацький інтерфейс побудовано на базі бібліотеки React.js та інструменту збірки Vite, контроль працездатності UI фокусувався на перевірці реактивності компонентів при моделюванні реальних призначених для користувача сценаріїв. Головне завдання ручного тестування полягало у верифікації відповідності візуального відображення внутрішнім станам програми, точності обробки подій введення та стабільності інтеграції з REST API. Дослідження охоплювало повноцінний флоу резидента, починаючи від спроби взаємодії з екранними формами до фінального рендерингу підтверджених замовлень.

Особливу увагу в QA-циклі було приділено контролю станів компонентів під час виконання асинхронних операцій. Безстанова архітектура взаємодії з

бекендом вимагає від фронтенду суворого захисту від повторних мережевих запитів, які користувач може спровокувати подвійним натисканням на кнопки відправки форм. Для вирішення цієї проблеми у кодову базу React-віджетів було закладено спеціалізовані булеві прапорці, такі як isLoading або isSubmitting [29]. Під час тестування форми підтвердження оренди ресурсу було перевірено логіку активації цих станів. У момент ініціалізації клієнтського запиту через Axios прапорець переводиться у положення true, що примусово додає атрибут disabled до цільової кнопки та змінює її візуальний стиль. Це повністю унеможливлює виникнення гонок умов (race conditions) на етапі очікування відповіді від Laravel-сервера. Інтерфейс залишається стабільним, інформуючи резидента про тривалу операцію за допомогою інтерактивного лоадера.

Не менш важливим аспектом ручного UI-тестування став аналіз відмовостійкості екранів при отриманні помилок валідації або критичних відповідей сервера. Додаток не може залишати користувача у стані невизначеності при виникненні позаштатних ситуацій, наприклад, при спробі забронювати стіл за відсутності коштів. Для забезпечення якісного фідбеку у вебсистему інтегровано механізм спливаючих тост-сповіщень. Ручна симуляція негативних сценаріїв підтвердила точність перехоплення HTTP-помилок конвеєром перехоплювачів Axios: при отриманні коду відповіді 422 або 400 додаток моментально ініціює виклик тоста із зрозумілим текстовим дескрипшеном, що дозволяє відвідувачу оперативно скоригувати свої дії.

Фінальним вектором модульної верифікації клієнтського шару стала перевірка інтерактивних лічильників зворотного відліку. Коли сервер успішно реєструє чернетку, React-компонент зчитує з отриманого JSON-пакета штамп часу expires_at і запускає внутрішній таймер за допомогою хука useEffect та системної функції setInterval [30]. Тестування зафіксувало абсолютну синхронність роботи інтерфейсу та серверної логіки: таймер щосекунди мутує локальний стейт залишку часу, забезпечуючи плавне візуальне зменшення таймера на екрані коворкінгу. При досягненні нульової позначки компонент бездоганно виконує очищення інтервалу, блокує платіжні елементи керування та

автоматично повертає резидента на сторінку каталогу зон. Така поведінка UI повністю дублює логіку автоматичного очищення протермінованих чернеток у базі даних MySQL, гарантуючи високу ергономіку та надійність платформи FlexSpot.

Результати ручних випробувань візуальних та інтерактивних модулів клієнтської частини React SPA було зведено в єдину матрицю контролю якості. Це дозволило структурувати процес перевірки фронтенду та зафіксувати поведінку системи при виконанні базових користувацьких сценаріїв (User Stories). Матриця охоплює найбільш критичні точки взаємодії резидента з інтерфейсом, включаючи клієнтську валідацію форм, динамічний рендеринг карти робочих місць та поведінку інтерактивних елементів у моменти інтеграції з сервером [31]. Повна матриця функціональних тест-кейсів користувацького інтерфейсу наведена в табл. 4.3.

Таблиця 4.3. Матриця функціональних тест-кейсів користувацького інтерфейсу

ID	**Об'єкт тестування**	**Початкові умови та дії користувача**	**Очікуваний результат**	**Фактичний результат**
TC-UI-01	Форма автентифікації та реєстрації	Введення некоректного формату email або пароля коротше 6 символів, клік на кнопку відправки.	Блокування відправки запиту. Рендеринг локальних червоних попереджень під відповідними інпутами.	Успішно
TC-UI-02	Інтерактивна карта простору DedicatedDeskWidget	Перехід до карти зони, клік на графічний маркер вільного робочого стола.	Зміна кольору маркера стола на активний. Запис ID стола у стейт React, активація кнопки «Забронювати».	Успішно
TC-UI-03	Екран очікування та підтвердження замовлення	Успішне отримання відповіді від API про створення чернетки бронювання.	Поява модального оверлею з деталями замовлення та запуском щосекундного лічильника зворотного відліку.	Успішно

продовження табл. 4.3

TC-UI-04	Модуль поповнення гаманця через LiqPay	Введення суми в особистому кабінеті, натискання на кнопку «Поповнити баланс».	Активація лоадера isLoading, успішне зчитування Base64 метаданих та рендеринг офіційного платіжного віджета.	Успішно
TC-UI-05	Система зворотного зв'язку (Toast-сповіщення)	Отримання клієнтським Axios помилки від сервера (наприклад, HTTP 422 при овербукінгу).	Спливання кастомного Toast-вікна у верхньому правому кутку екрана з точним описом проблеми.	Успішно
TC-UI-06	Обробка події експірації таймера	Зменшення лічильника зворотного відліку чернетки до позначки 00:00.	Автоматичне закриття форми оплати, очищення стейтів, виклик попереджувального тоста та редирект до каталогу.	Успішно

Аналіз результатів, зафіксованих у процесі UI-тестування, демонструє повну відповідність розробленого інтерфейсу сучасним вимогам до вебзастосунків. Стоввідсоткове проходження задекларованих тест-кейсів підтвердило, що React-клієнт коректно реагує на дії відвідувача, надійно захищає систему від повторних помилкових кліків за допомогою реактивних прапорців завантаження та забезпечує високий рівень інформативності завдяки тост-сповіщенням.

Підбиваючи підсумок усього четвертого розділу, можна стверджувати, що розгорнутий комплекс модульно-інтеграційних та функціональних випробувань повністю довів технічну зрілість і стабільність екосистеми FlexSpot. Автономна перевірка REST API ендпоінтів у середовищі Postman разом із ручним UI-інспектуванням у Chrome DevTools дозволили локалізувати й усунути дрібні дефекти ще на етапі розробки бізнес-логіки. Спроєктоване програмне забезпечення повністю відповідає критеріям надійності, відмовостійкості та безпеки, є цілісним і абсолютно готовим до промислового впровадження й експлуатації в реальних коворкінг-центрах.

ВИСНОВКИ

Загальним результатом виконання кваліфікаційної роботи є проєктування та програмна реалізація вебсистеми FlexSpot, призначеної для комплексного управління інфраструктурними ресурсами та автоматизації процесів резервування у просторах коворкінг-центрів. На етапі системного аналізу предметної області та розробки шару збереження даних було спроєктовано нормалізовану реляційну структуру бази даних у середовищі СКБД MySQL на базі рушія InnoDB, яка охоплює 14 взаємопов'язаних таблиць. Зведення цієї схеми до критеріїв третьої нормальної форми дозволило повністю нівелювати ризики виникнення архітектурних аномалій модифікації, вставки та видалення записів, що сформувало консистентний інфраструктурний та фінансовий базис платформи.

Завдання із побудови серверної архітектури було успішно вирішено шляхом створення автономного ядра на базі фреймворку Laravel у форматі RESTful API, де обмін інформацією виконується виключно у стандартизованому форматі JSON. Для забезпечення транзакційної стійкості комерційного контуру та захисту від овербукінгу в умовах висококонкурентних паралельних запитів на бронювання, у бізнес-логіку серверних контролерів інтегровано механізм песимістичного блокування рядків на рівні СКБД за допомогою оператора lockForUpdate(). Виконання обчислень реального навантаження простору всередині ізольованих транзакцій DB::transaction дозволило повністю нейтралізувати стани гонитви під час пікових навантажень на сервіс.

Важливим архітектурним результатом проєкту стало впровадження повністю безстанової (Stateless) моделі автентифікації та захисту даних. Керування доступами децентралізовано за допомогою інтеграції хмарного сервісу Google Firebase Cloud Auth, а авторизація клієнтських HTTPS-запитів виконується через токени JWT, безстанову валідацію підписів яких здійснює проміжний шар FirebaseAuthMiddleware. Фінансовий вузол платформи успішно реалізує алгоритм змішаної оплати, де декремент корпоративних B2B-лімітів за

принципом FIFO поєднується зі списанням коштів із цифрового гаманця резидента, поповнення якого автоматизовано через асинхронні вебхуки LiqPay API із застосуванням транспортного кодування Base64 та SHA-1 валідації сигнатури.

Етап верифікації розробленого програмного забезпечення підтвердив високу стабільність та надійність функціонування всіх модулів екосистеми FlexSpot. Комплексна перевірка серверного REST API у середовищі Postman за принципом «чорної скриньки» у поєднанні з ручним функціональним тестуванням інтерфейсу та інспектуванням реактивних станів (state та props) компонентів у Chrome DevTools довело коректність обробки виняткових ситуацій (Exception Handling). Система штатно блокує спроби несанкційного доступу чи порушення лімітів, повертаючи клієнту структуровані JSON-пакети з точними HTTP-кодами.

Підсумовуючи, можна стверджувати, що створена вебсистема є повністю завершеним інженерним рішенням, яке відповідає сучасним вимогам до безпеки, швидкодії та консистентності даних. Проєкт має високу практичну цінність і повністю готовий до розгортання в комерційному секторі коворкінг-центрів із перспективою масштабування у вигляді майбутньої інтеграції з фізичними системами контролю та управління доступом.

Посилання на репозиторій з кодом програми URL – https://github.com/wanderrlust/flexspot.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Servcorp. 2025 Global Coworking Space Statistics and Growth Insights. URL: https://www.servcorp.com.au/en/blog/business-networking/how-many-coworking-spaces-are-there-in-the-world-2024-coworking-statistics/ (дата звернення: 08.05.2026).

2. Nexudus. Automated billing and payments for coworking spaces. URL: https://nexudus.com/automated-billing-payments/ (дата звернення: 08.05.2026).

3. OfficeRnD. The Coworking Space Software That Grows Your Revenue. URL: https://www.officernd.com/coworking-software/ (дата звернення: 09.05.2026).

4. OfficeRnD Flex Help Center. Welcome to OfficeRnD Flex. URL: https://help.officernd.com/en/articles/271498-welcome-to-officernd-flex (дата звернення: 09.05.2026).

5. Ghanbari M. Avoid Race Conditions with Reservation Pattern. URL: https://medium.com/@mmdGhanbari/avoid-race-conditions-with-reservation-pattern-bc4846602417 (дата звернення: 13.05.2026).

6. Vite. Why Vite. URL: https://vitejs.dev/guide/why (дата звернення: 15.05.2026).

7. Laravel. Frontend. URL: https://laravel.com/docs/13.x/frontend (дата звернення: 15.05.2026).

8. Lucidchart. UML Use Case Diagram Tutorial. URL: https://www.lucidchart.com/pages/tutorial/uml-use-case-diagram (дата звернення: 11.05.2026).

9. Visual Paradigm. What is State Machine Diagram? URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-state-machine-diagram/ (дата звернення: 11.05.2026).

10. DigitalOcean. Database Normalization: 1NF, 2NF, 3NF & BCNF Examples. URL: https://www.digitalocean.com/community/tutorials/database-normalization (дата звернення: 12.05.2026).

11. Rick James. Best Practices for Datatypes in a MySQL Schema. URL: https://mysql.rjweb.org/doc.php/schema_best_practices_mysql (дата звернення: 12.05.2026).

12. Codica. Mobile First Design: Guide and Best Practices. URL: https://www.codica.com/blog/mobile-first-design/ (дата звернення: 15.05.2026).

13. Vite. Getting Started. URL: https://vite.dev/guide/ (дата звернення: 16.05.2026).

14. Robin Wieruch. React Folder Structure Best Practices [2026]. URL: https://www.robinwieruch.de/react-folder-structure/ (дата звернення: 16.05.2026).

15. React Router. Official Documentation. URL: https://reactrouter.com/start/framework/routing (дата звернення: 17.05.2026).

16. Robin Wieruch. React Router 7: Private Routes (alias Protected Routes). URL: https://www.robinwieruch.de/react-router-private-routes/ (дата звернення: 17.05.2026).

17. Laravel. Eloquent: API Resources. URL: https://laravel.com/docs/13.x/eloquent-resources (дата звернення: 19.05.2026).

18. Laravel. Eloquent: Relationships. URL: https://laravel.com/docs/13.x/eloquent-relationships (дата звернення: 19.05.2026).

19. AddWeb Solution. Laravel Task Scheduling & Automation: Simplify Cron Jobs with the Scheduler. URL: https://dev.to/addwebsolutionpvtltd/laravel-task-scheduling-automation-simplify-cron-jobs-with-the-scheduler-1j4j (дата звернення: 20.05.2026).

20. Steve King. Implementing the Saga Pattern in a Laravel Monolith. URL: https://www.juststeveking.com/articles/implementing-the-saga-pattern-laravel (дата звернення: 20.05.2026).

21. Pentest Testing Corp. Preventing Insecure Direct Object References (IDOR) in Laravel. URL: https://pentest-testing-corp.medium.com/preventing-insecure-direct-object-references-idor-in-laravel-9b8ef97866cb (дата звернення: 21.05.2026).

22. LiqPay. API Documentation - Callback. URL: https://www.liqpay.ua/en/doc/api/callback (дата звернення: 22.05.2026).

23. Reintech. Handling Payment Notifications and Webhooks with LiqPay API. URL: https://reintech.io/blog/handling-payment-notifications-liqpay-api (дата звернення: 22.05.2026).

24. BrowserStack. What is Black Box Testing: Types, Tools & Examples. URL: https://www.browserstack.com/guide/black-box-testing (дата звернення: 24.05.2026).

25. Postman. API Authentication. URL: https://www.postman.com/api-platform/api-authentication/ (дата звернення: 24.05.2026).

26. Ngrok. Test Webhooks Locally. URL: https://ngrok.com/docs/guides/share-localhost/webhooks (дата звернення: 23.05.2026).

27. Postman. Best Practices for API Error Handling. URL: https://blog.postman.com/best-practices-for-api-error-handling/ (дата звернення: 25.05.2026).

28. RESTfulAPI.net. HTTP Status Codes. URL: https://restfulapi.net/http-status-codes/ (дата звернення: 25.05.2026).

29. Newline. Race Conditions in React: What They Are and How to Avoid Them. URL: https://www.newline.co/@RichardBray/race-conditions-in-react-what-they-are-and-how-to-avoid-them--675702e6 (дата звернення: 18.05.2026).

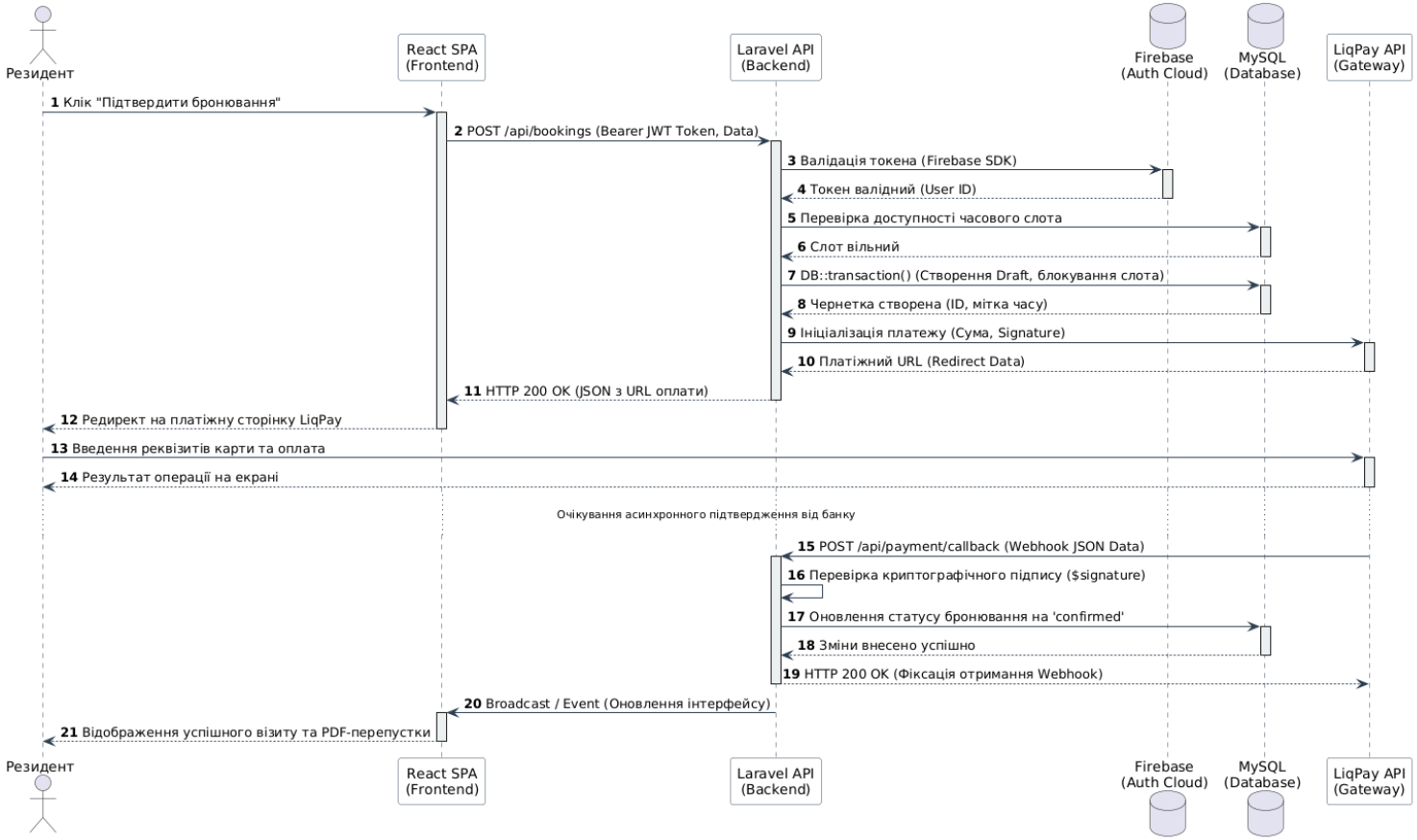
30. Overreacted (Dan Abramov). Making setInterval Declarative with React Hooks. URL: https://overreacted.io/making-setinterval-declarative-with-react-hooks/ (дата звернення: 18.05.2026).

31. GeeksforGeeks. Queue Data Structure. URL: https://www.geeksforgeeks.org/dsa/introduction-to-queue-data-structure-and-algorithm-tutorials/ (дата звернення: 21.05.2026).

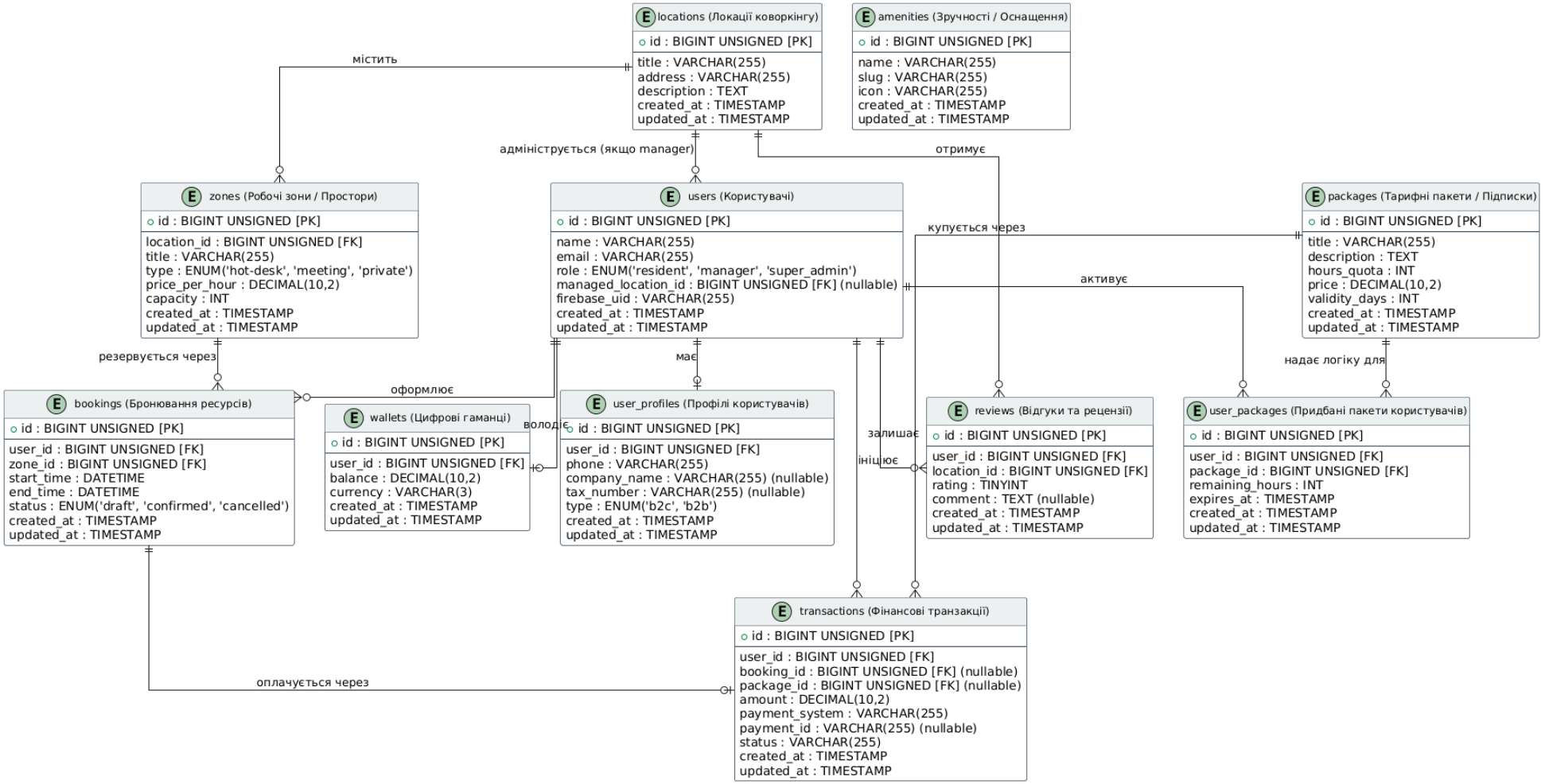
ДОДАТОК А

СХЕМИ АЛГОРИТМІВ ТА ГРАФІЧНІ ДІАГРАМИ ФУНКЦІОНУВАННЯ ВЕБСИСТЕМИ

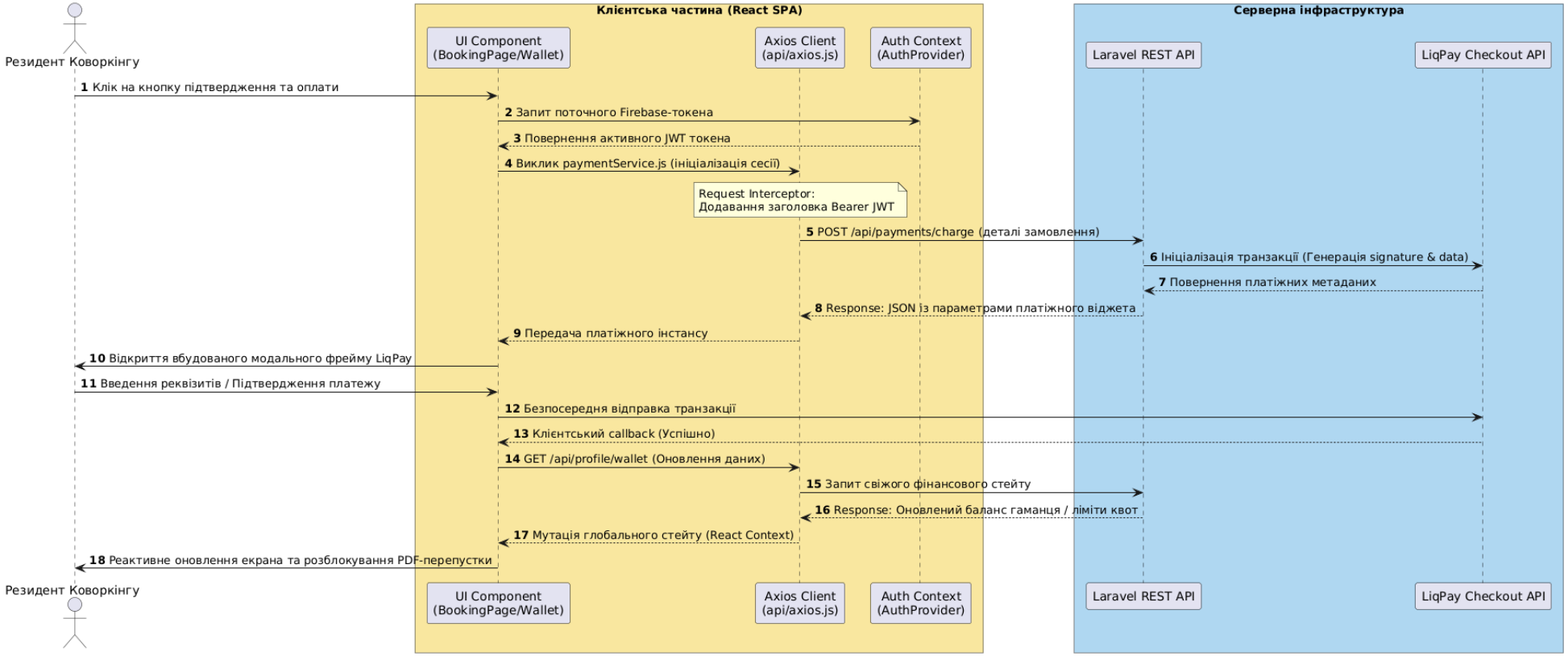
А.1. Діаграма послідовності (Sequence Diagram) процесу бронювання простору



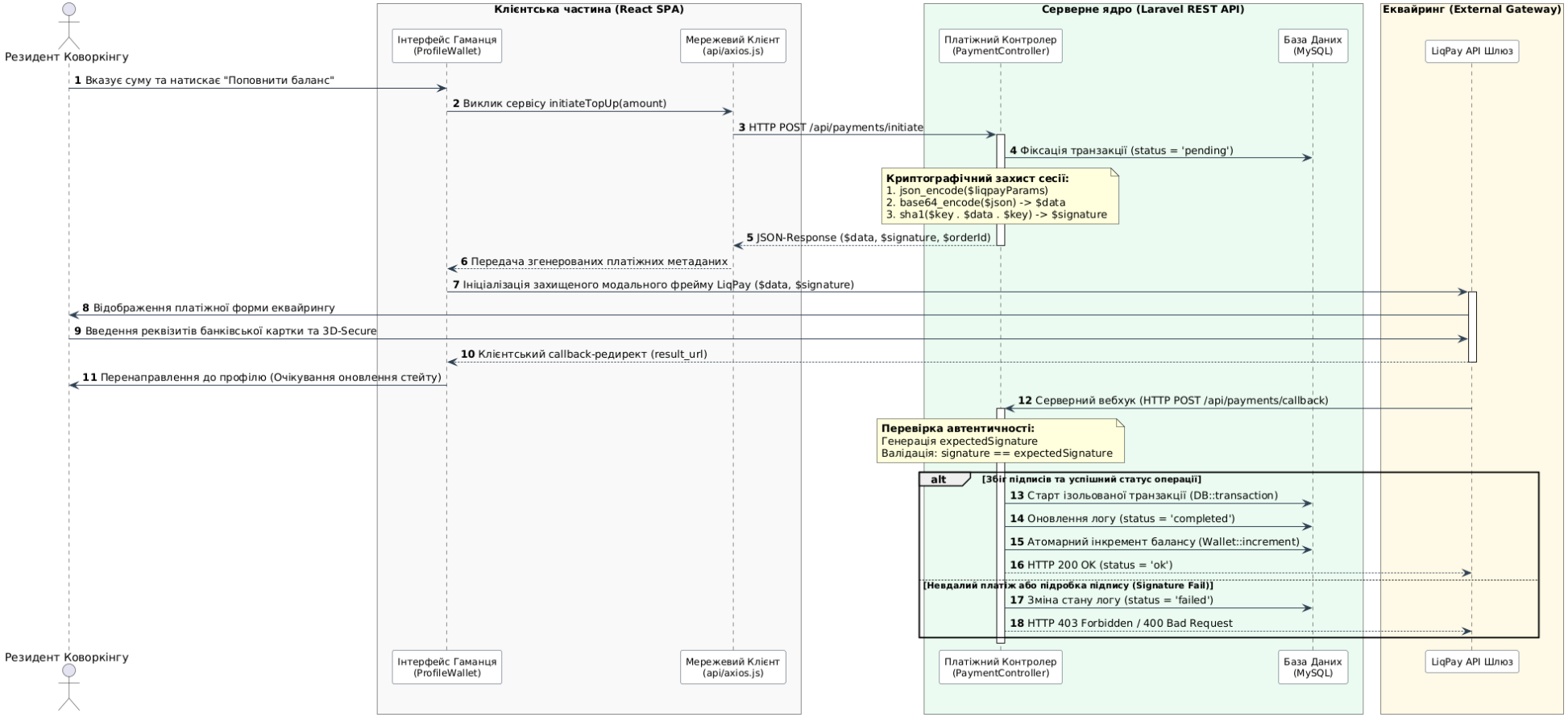
А.2. ER-діаграма фізичної структури реляційної бази даних вебсистеми FlexSpot



А.3. Діаграма послідовності процесу координації асинхронних платіжних потоків



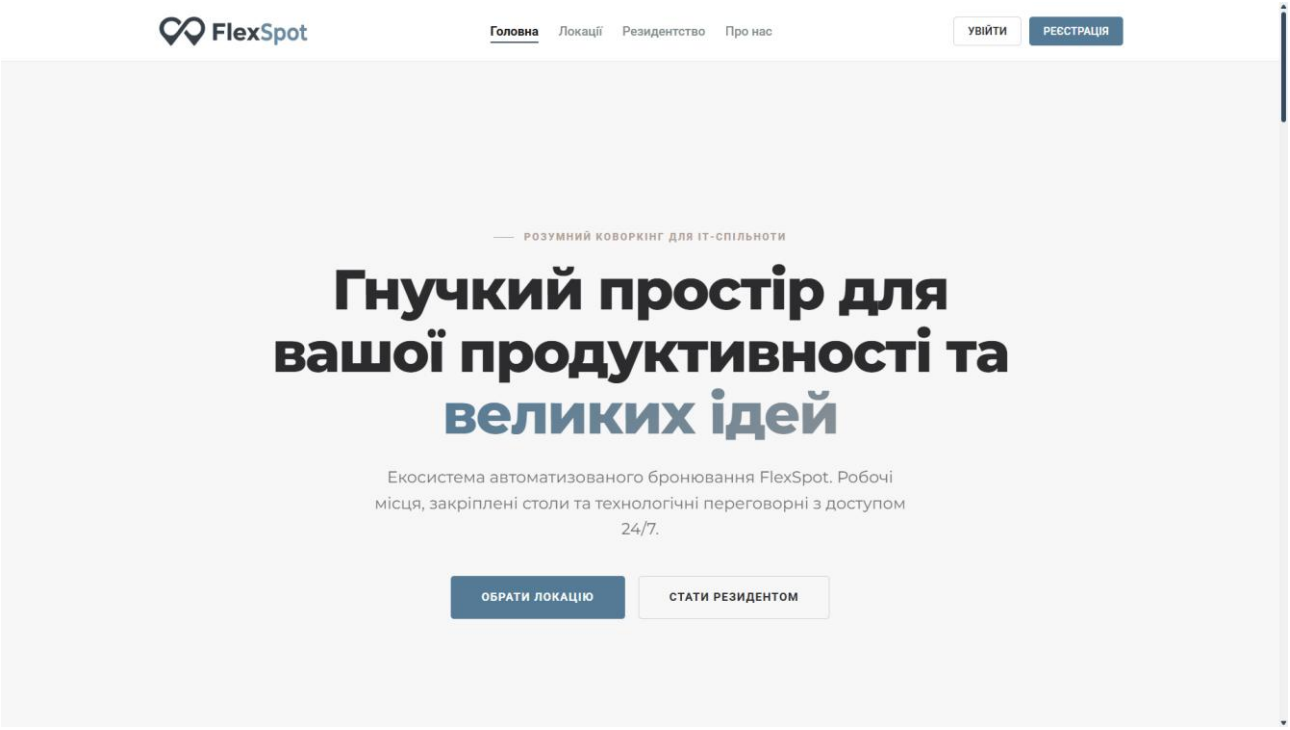
А.4. Діаграма послідовності процесу верифікації та проведення LiqPay платежу



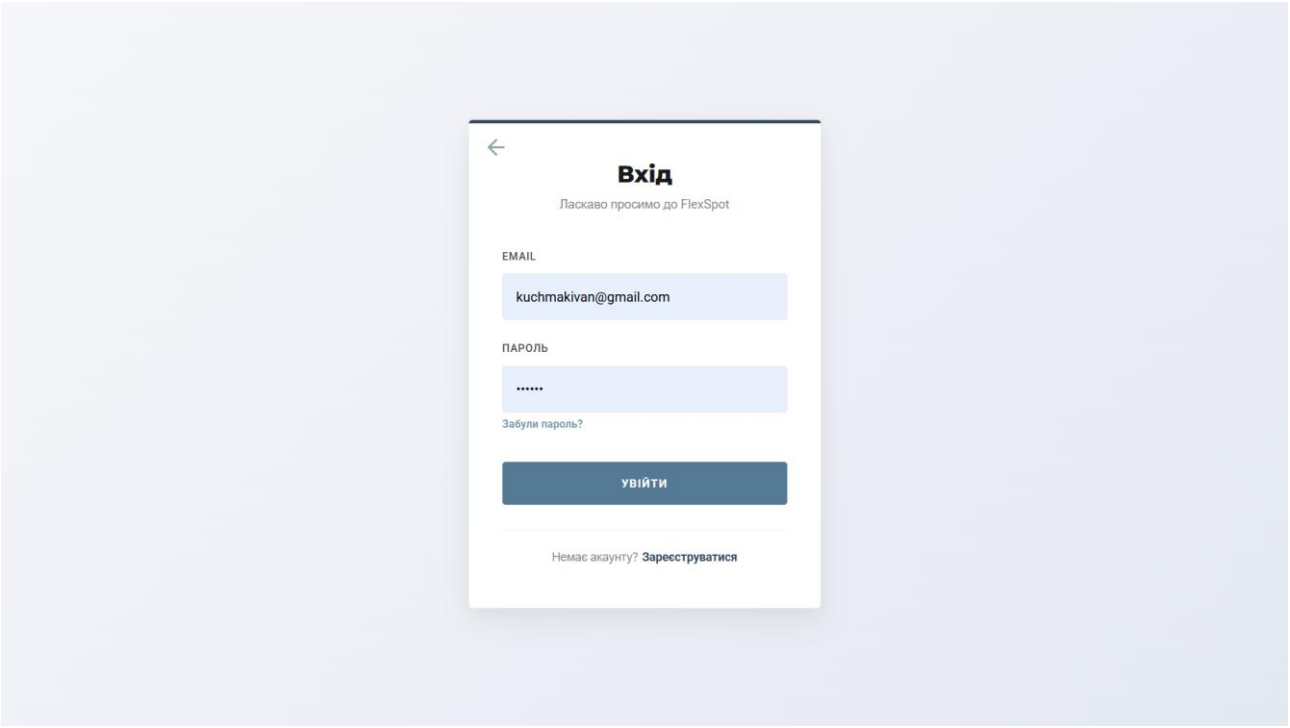
ДОДАТОК Б

ІНТЕРФЕЙС ВЕБСИСТЕМИ FLEXSPOT

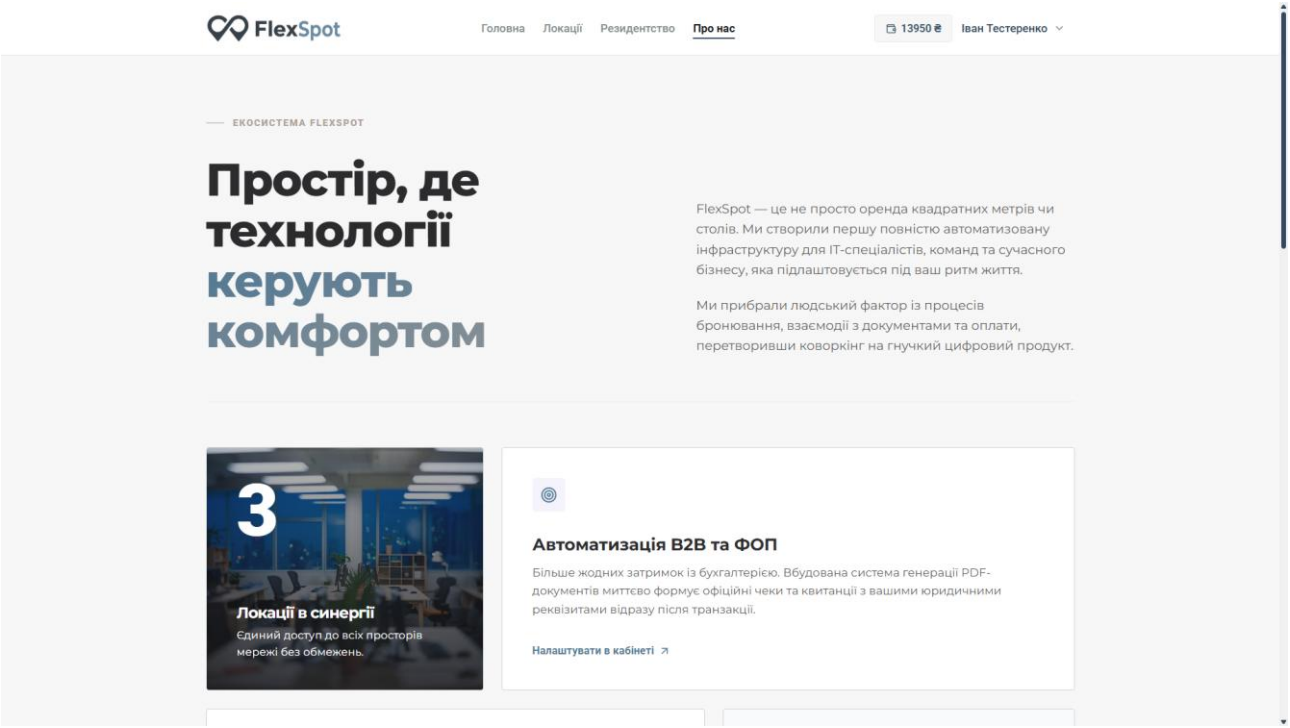
Б.1. Головна сторінка



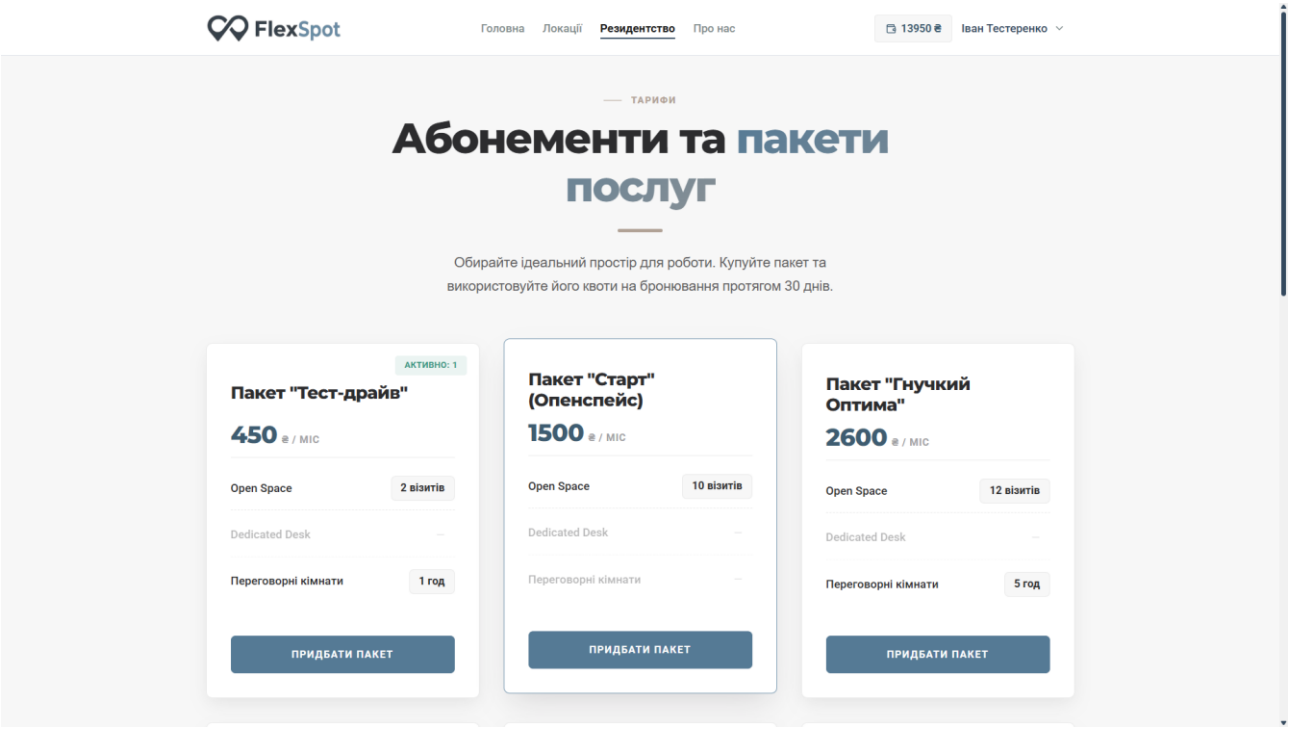
Б.2. Сторінка авторизації



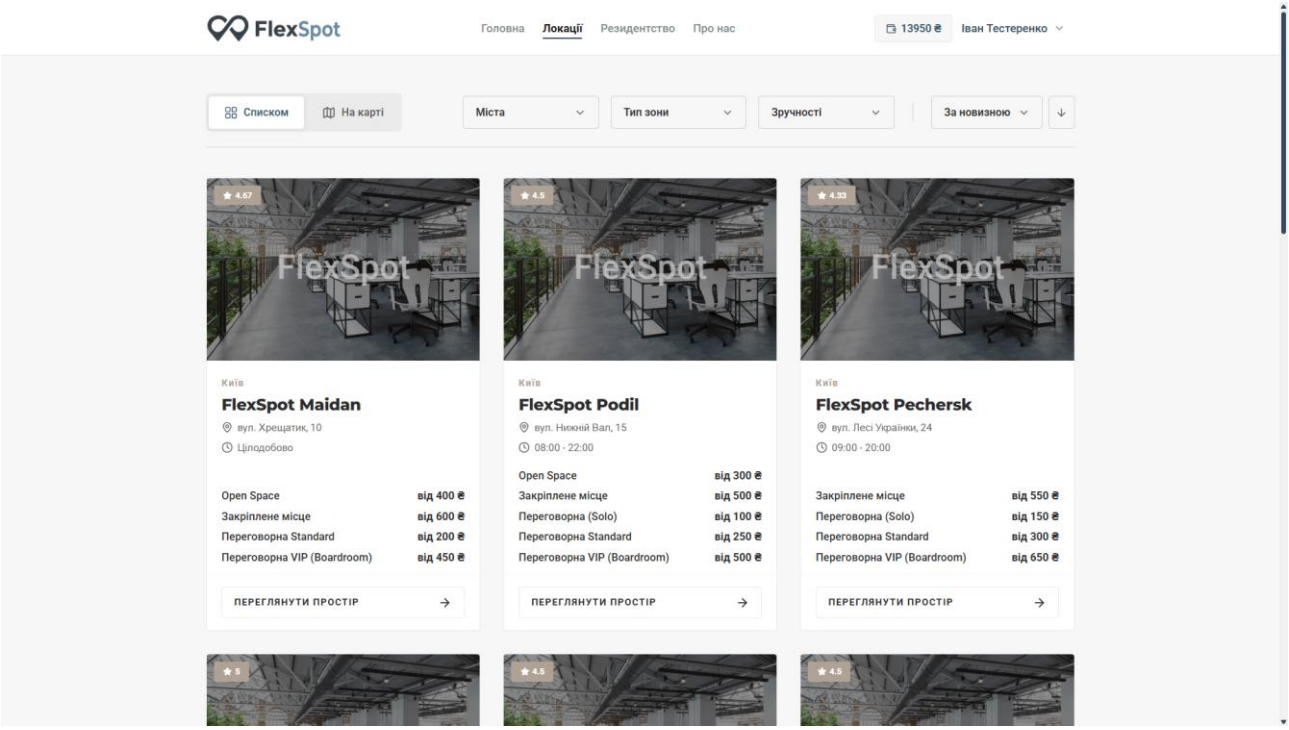
Б.3. Сторінка «Про нас»



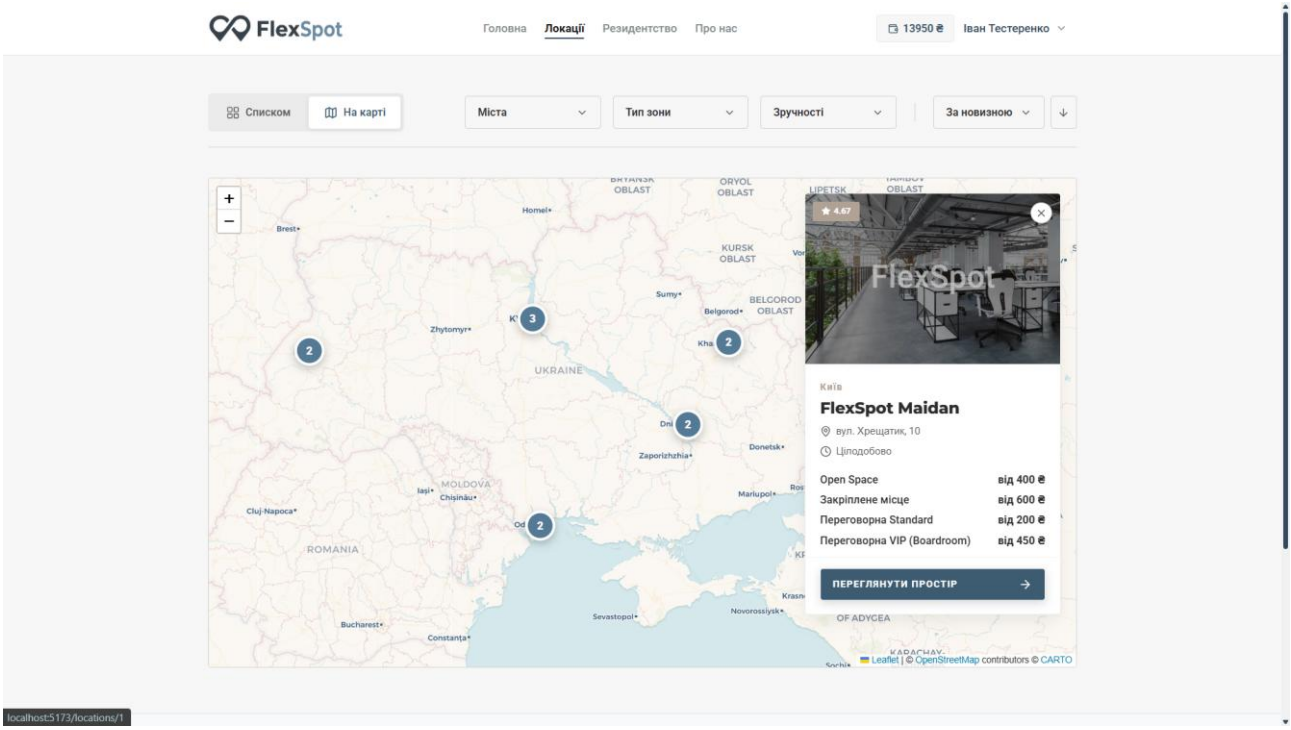
Б.4. Сторінка «Резидентство» (сторінка пакетів послуг)



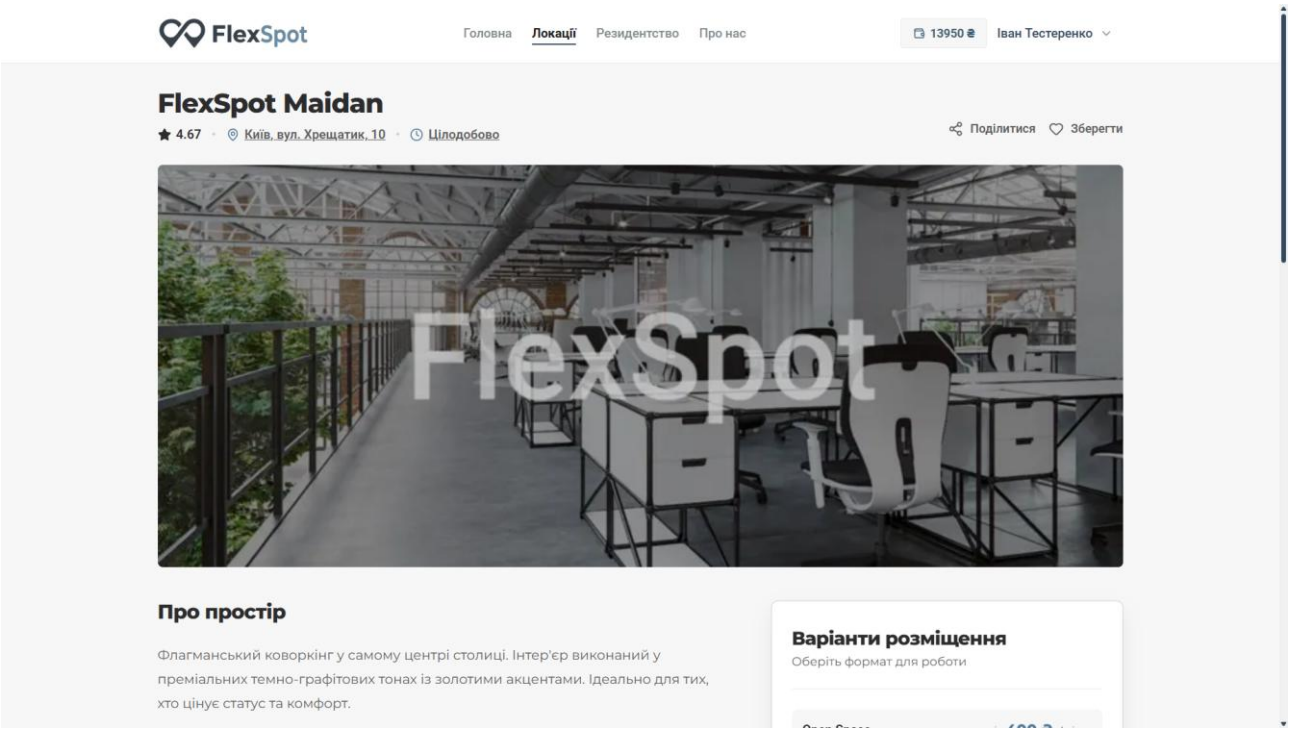
Б.5. Сторінка «Локації» (каталог коворкінг-центрів, режим перегляду – сітка карток)



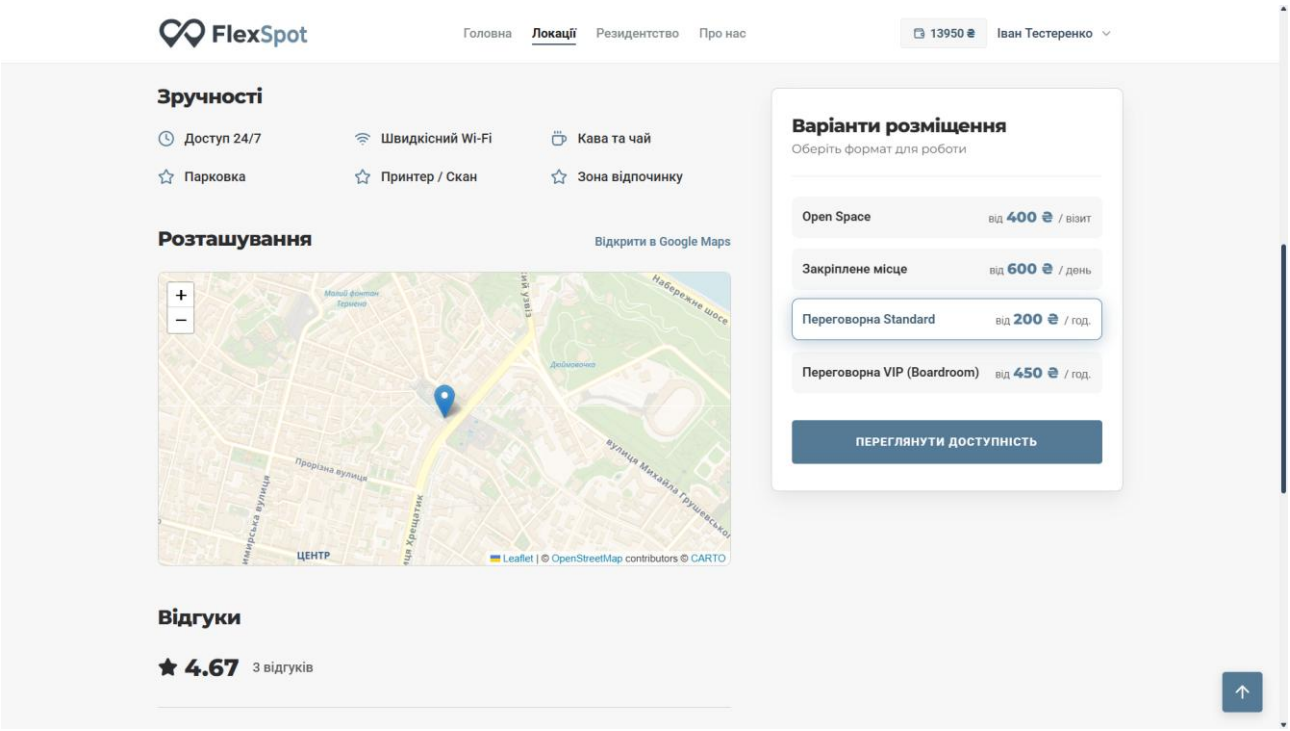
Б.6. Сторінка «Локації» (каталог коворкінг-центрів, режим перегляду – інтерактивна карта)



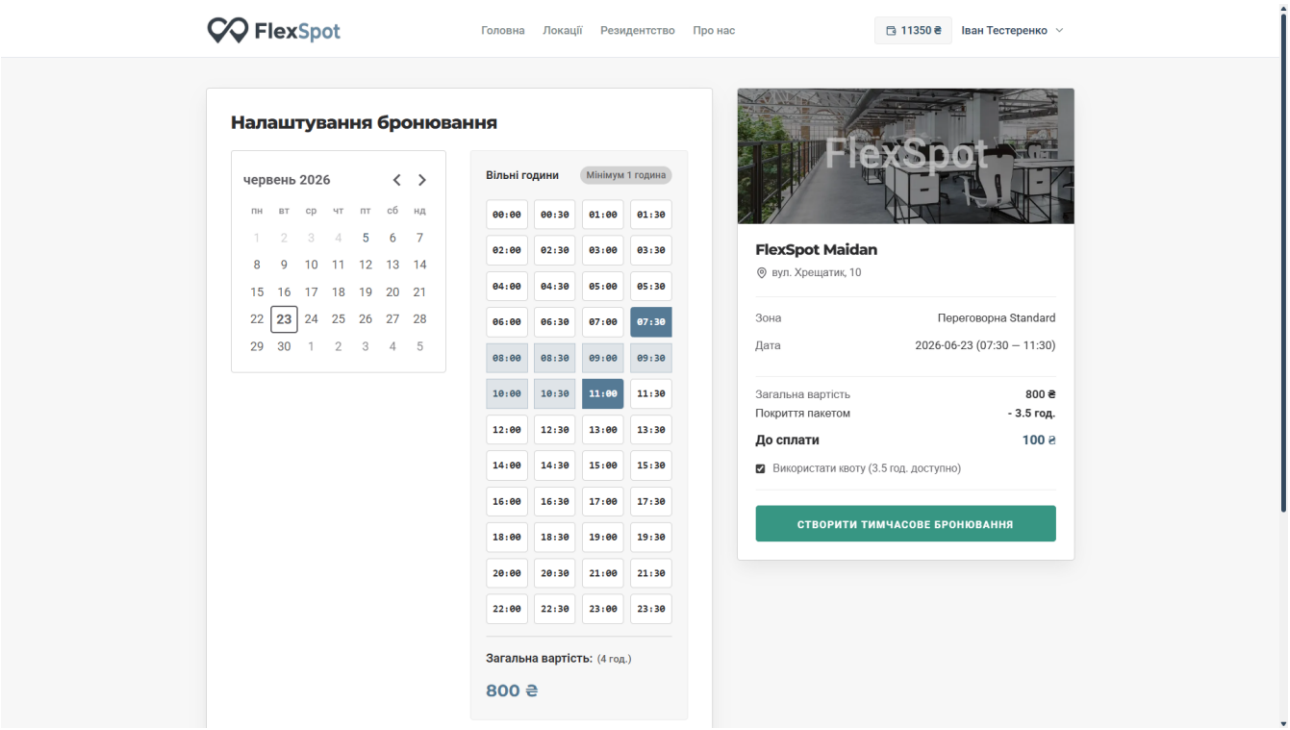
Б.7. Сторінка конкретної локації – частина 1



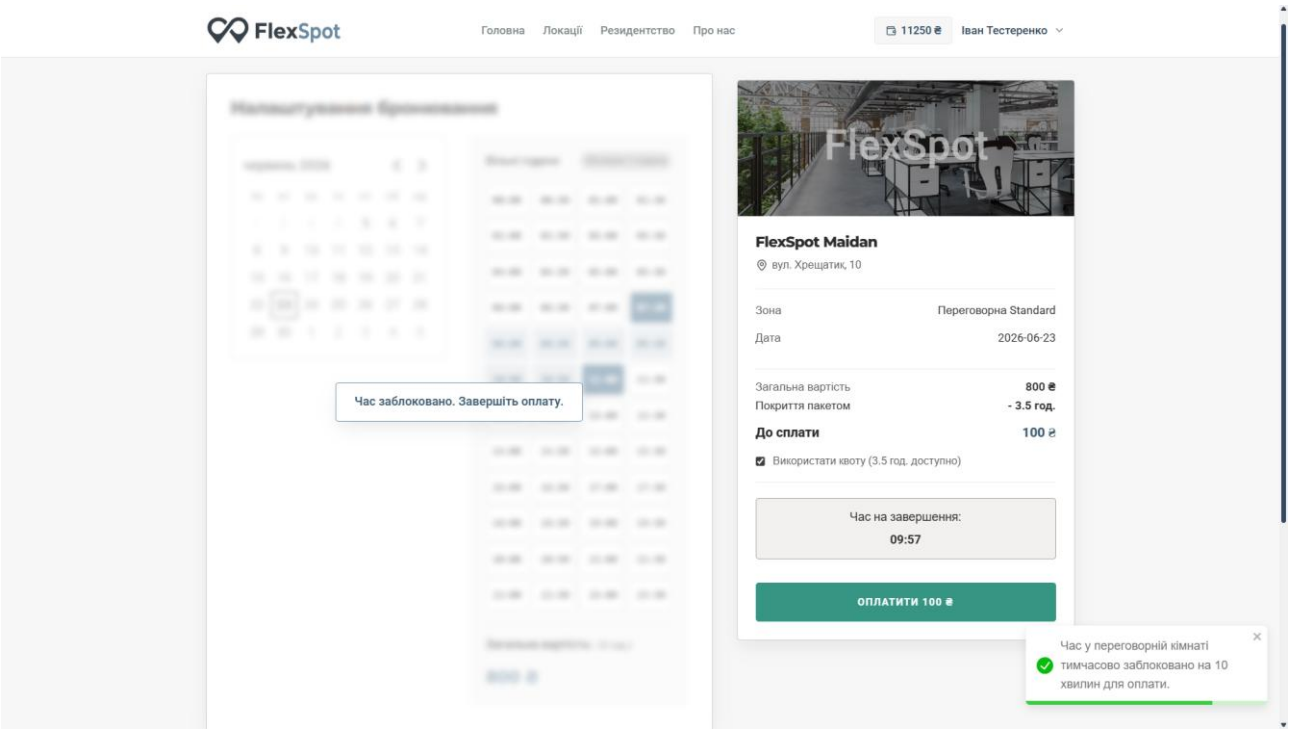
Б.8. Сторінка конкретної локації – частина 2



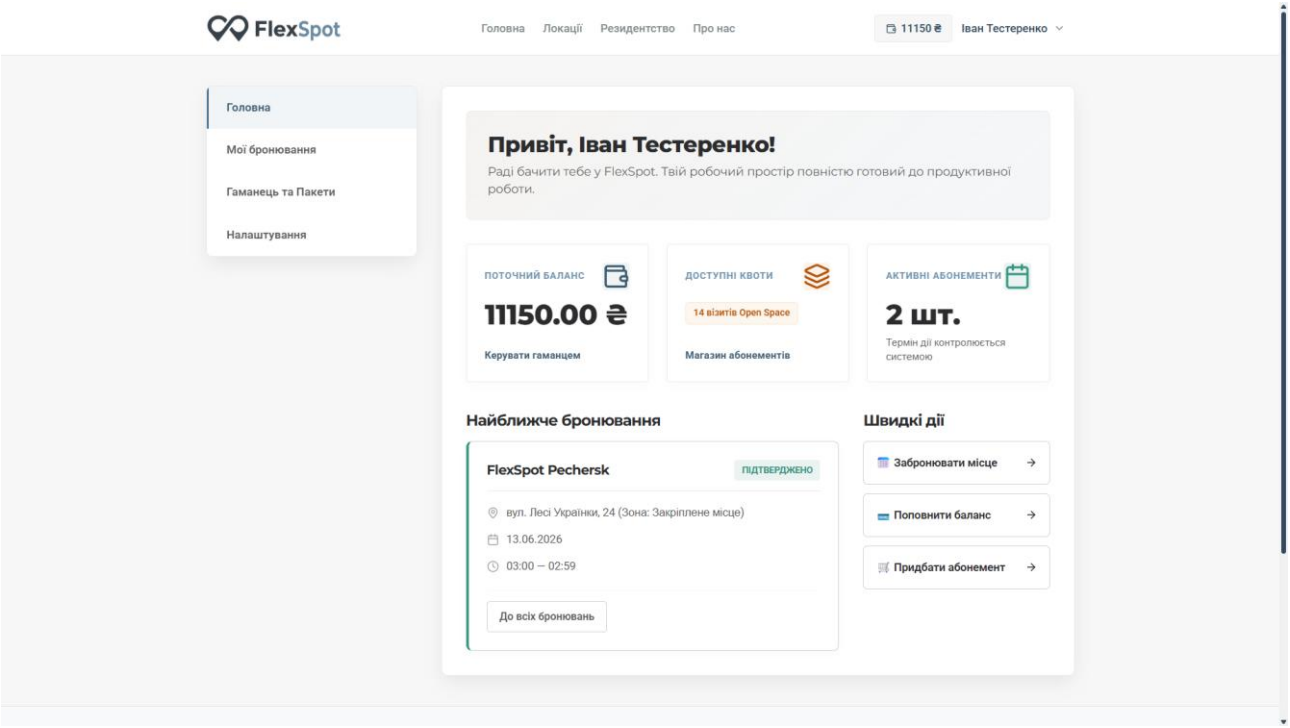
Б.9. Сторінка налаштування бронювання – стан 1



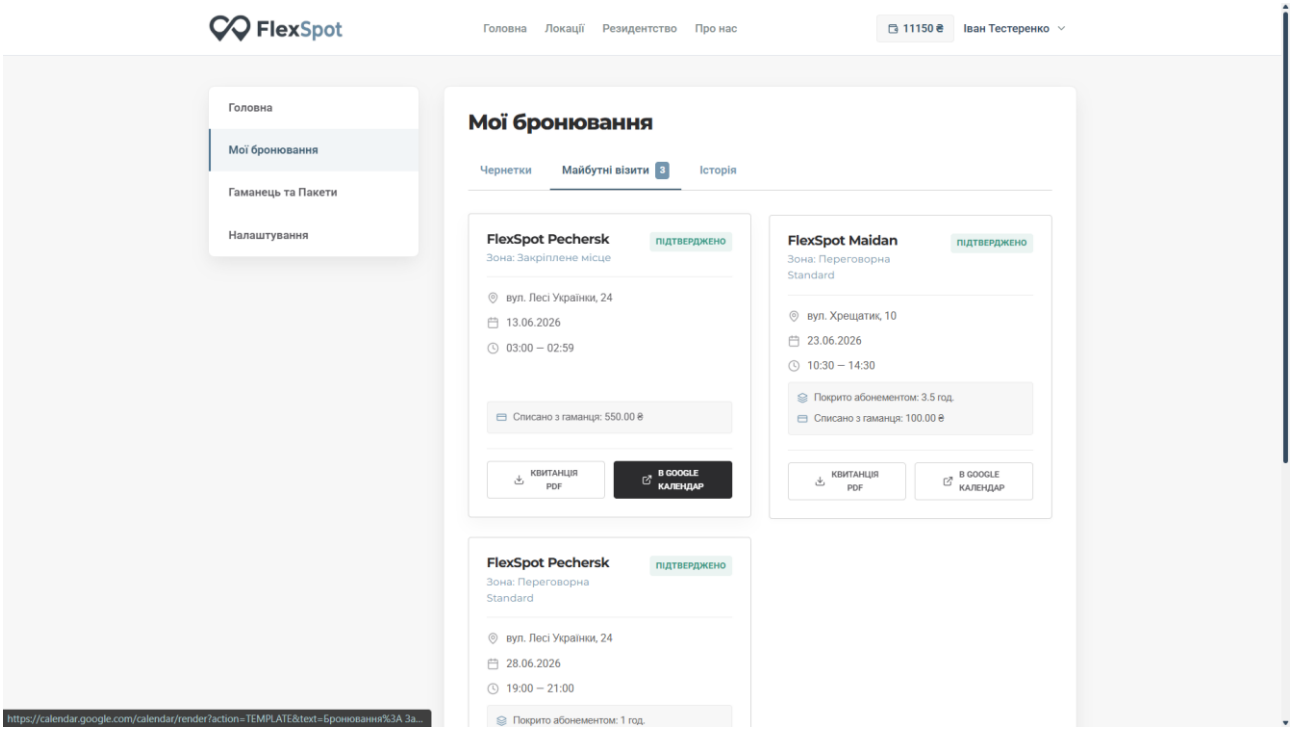
Б.10. Сторінка налаштування бронювання – стан 2



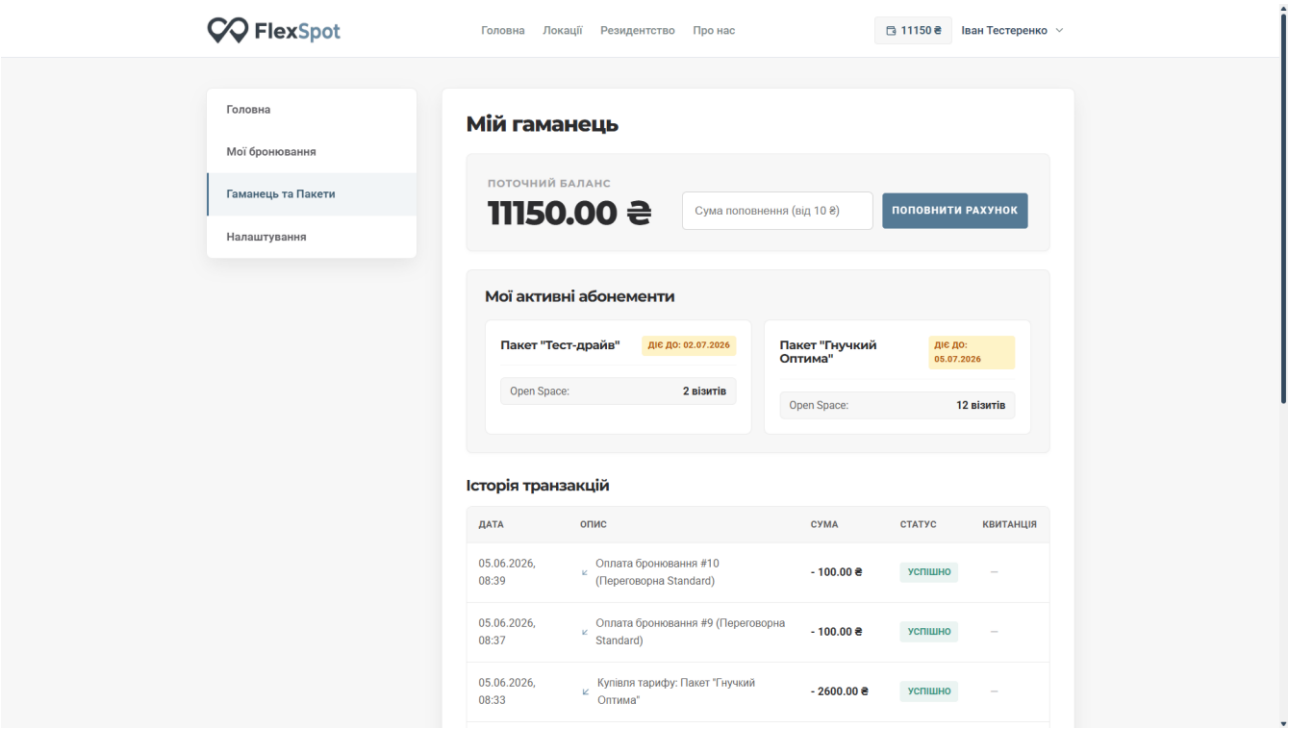
Б.11. Сторінка кабінету користувача – Головна



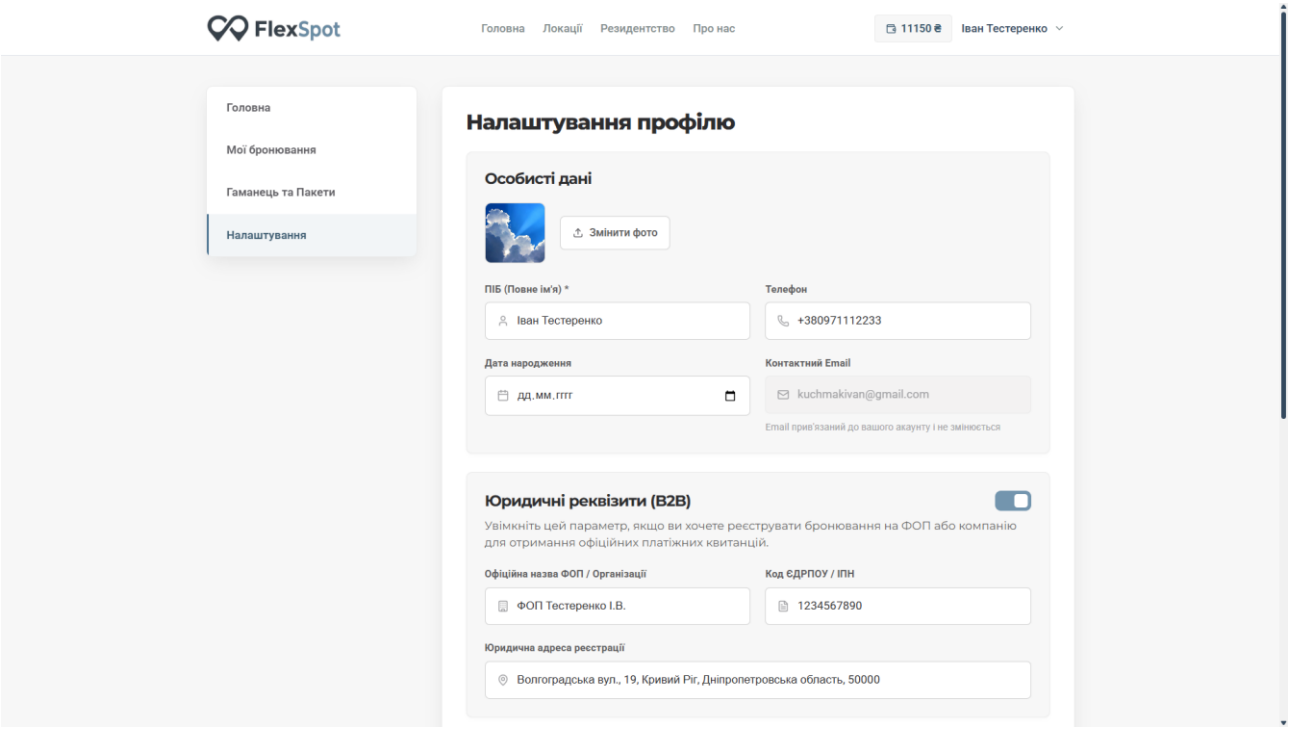
Б.12. Сторінка кабінету користувача – Мої бронювання



Б.13. Сторінка кабінету користувача – Гаманець та Пакети



Б.14. Сторінка кабінету користувача – Налаштування



Б.15. Вигляд оформленої квитанції поповнення рахунку

Б.16. Вигляд оформленої квитанції бронювання