Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій
(повне найменування факультету)

Кафедра програмних засобів
(повне найменування кафедри)

Пояснювальна записка

до дипломного проєкту (роботи)
бакалавр
(ступінь вищої освіти)

на тему СТВОРЕННЯ ПРОГРАМНОЇ СИСТЕМИ ПЛАНУВАННЯ ТА УПРАВЛІННЯ ЗАПИСОМ КЛІЄНТІВ САЛОНУ КРАСИ
DESIGN OF A WEB-BASED APPOINTMENT SCHEDULING AND MANAGEMENT SYSTEM FOR BEAUTY SALONS

Виконав(ла): студент(ка) 4 курсу, групи КНТ-222
Спеціальності 122 Комп'ютерні науки
(код і найменування спеціальності)

Освітня програма (спеціалізація)
Комп'ютерні науки

ІВАНОВ Д.О.
(ПРІЗВИЩЕ та ініціали)
Керівник ГОЛУБ Т.В.
(ПРІЗВИЩЕ та ініціали)
Рецензент БАБЕНКО Н.В.
(ПРІЗВИЩЕ та ініціали)

2026

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
(повне найменування закладу вищої освіти)

Факультет КНТ
Кафедра програмних засобів
Ступінь вищої освіти бакалавр
Спеціальність 122 Комп'ютерні науки
(код і найменування)
Освітня програма (спеціалізація) Комп'ютерні науки
(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
_____Сергій СУББОТІН
"____" _____2026 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

ІВАНОВА Данила Олександровича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Створення програмної системи планування та управління записом клієнтів салону краси. Design of a Web-Based Appointment Scheduling and Management System for Beauty Salons

керівник проєкту (роботи) к.т.н., доцент, ГОЛУБ Тетяна Василівна,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)
затверджені наказом закладу вищої освіти від " 07 " квітня 2026 року № 139
2. Строк подання студентом проєкту (роботи) 01 червня 2026 року
3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Аналіз предметної області та існуючих рішень для управління записом у б'юті-салонах. 2. Проєктування мобільної інформаційної системи управління записом. 3. Розробка мобільної інформаційної системи управління записом. 4. Тестування мобільної інформаційної системи управління записом.
5. Перелік графічного матеріалу (з з точним зазначенням обов'язкових креслень, кількість слайдів, плакатів)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ГОЛУБ Т.В., доцент		
Нормоконтроль	БЄЛОВА А.В., асистент		

7. Дата видачі завдання « 20 » квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області б'юті-індустрії та існуючих систем управління записом клієнтів	1 тиждень	Розділ 1
3	Розробка специфікації вимог до системи та порівняльний аналіз існуючих платформ (Fresha, Treatwell, Booksy)	2 тиждень	Розділ 1
4	Проєктування архітектури системи, UML-моделювання та розробка схеми бази даних	2 тиждень	Розділ 2
5	Розробка серверної частини на Spring Boot та реалізація REST API	3 тиждень	Розділ 3
6	Розробка мобільного додатку на React Native (Expo) для трьох ролей користувачів	3–4 тиждень	Розділ 3
7	Інтеграція платіжного шлюзу Stripe та системи автентифікації Firebase	4 тиждень	Розділ 3
8	Юніт-тестування, функціональне та навантажувальне тестування мобільної інформаційної системи	5 тиждень	Розділ 4
9	Оформлення пояснювальної записки та документів до неї	6 тиждень	Додатки
10	Нормоконтроль та рецензування	7 тиждень	
11	Захист роботи	8 тиждень	

Студент(ка) _____ Данило ІВАНОВ
(підпис) (Ім'я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Тетяна ГОЛУБ
(підпис) (Ім'я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 119 с., 14 табл., 35 рис., 2 дод., 20 джерел.

МОБІЛЬНИЙ ЗАСТОСУНОК, Б'ЮТІ-САЛОН, СИСТЕМА ЗАПИСУ КЛІЄНТІВ, ОНЛАЙН-БРОНЮВАННЯ, REACT NATIVE, SPRING BOOT.

Об'єкт дослідження – процес застосування алгоритмів та комп’ютерних обчислень для створення програмних систем автоматизації запису клієнтів і управління розкладом роботи б'юті-салону.

Предмет дослідження – програмні засоби та методи розробки мобільних інформаційних систем для автоматизації запису клієнтів б'юті-салону.

Мета роботи – підвищення ефективності діяльності б'юті-салону та зручності взаємодії з клієнтами.

Матеріали, методи та технічні засоби: методи об'єктно-орієнтованого програмування, UML-моделювання, методи проєктування реляційних баз даних, шаблон Optimistic Locking; мови програмування Java та JavaScript/TypeScript, фреймворк Spring Boot 3, фреймворк React Native з платформою Expo, СКБД PostgreSQL 16, кеш-сервер Redis, платіжний шлюз Stripe, сервіс автентифікації Firebase, інструмент Apache JMeter; персональний комп'ютер під управлінням ОС Microsoft Windows 11.

Результати. Створено кросплатформений мобільний застосунок для платформ Android.

Висновки. Розроблено мобільну інформаційну систему планування та управління записом клієнтів б'юті-салону, що автоматизує процес запису.

Галузь використання – підприємства сфери б'юті-послуг: салони краси, перукарні, манікюрні та косметологічні студії, СПА-центри, барбершопи.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 119 pages, 14 tables, 35 figures, 2 appendices, 20 sources.

MOBILE APPLICATION, BEAUTY SALON, CLIENT APPOINTMENT SYSTEM, ONLINE BOOKING, REACT NATIVE, SPRING BOOT.

The object of study is the processes of planning and managing client appointments at beauty service enterprises.

The subject of study is software tools and methods for developing mobile information systems for automating client appointment booking in beauty salons.

The purpose of the work is to improve the operational efficiency of a beauty salon and the convenience of client interaction by developing a mobile information system for appointment planning and management with support for three role profiles: client, master, and administrator.

Materials, methods, and tools: methods of object-oriented programming, UML modelling, relational database design methods, the Optimistic Locking pattern; programming languages Java and JavaScript/TypeScript, the Spring Boot 3 framework, the React Native framework with the Expo platform, the PostgreSQL 16 DBMS, the Redis cache server, the Stripe payment gateway, the Firebase authentication service, the Apache JMeter load testing tool; a personal computer running the Microsoft Windows 11 operating system.

Results. A cross-platform mobile application has been developed for the Android and iOS platforms.

Conclusions. A mobile information system for planning and managing client appointments in a beauty salon has been developed. It automates the booking process and ensures a decrease in the client no-show rate from 20.5 % to 9 %.

The field of application includes beauty service enterprises: beauty salons, hairdressing salons, nail and cosmetology studios, spa centres, and barbershops.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API – Application Programming Interface;

CRUD – Create, Read, Update, Delete;

FCM – Firebase Cloud Messaging;

HTTPS – HyperText Transfer Protocol Secure;

JPA – Java Persistence API;

JSON – JavaScript Object Notation;

JWT – JSON Web Token;

KPI – Key Performance Indicator;

ORM – Object-Relational Mapping;

PCI – Payment Card Industry Data Security Standard;

REST – Representational State Transfer;

SDK – Software Development Kit;

SLA – Service Level Agreement;

SQL – Structured Query Language;

UML – Unified Modeling Language;

БД – база даних;

ПЗ – програмне забезпечення.

ВСТУП

Б'юті-індустрія є однією з найбільш стійких та динамічних галузей сфери послуг, що демонструє стабільне зростання навіть в умовах економічних криз [1]. Салони краси, барбершопи, spa-центри та студії манікюру є невід'ємною частиною міської інфраструктури та щодня обслуговують мільйони клієнтів. Незважаючи на значний обсяг ринку, значна частина закладів б'юті-сфери, особливо малого та середнього розміру, досі використовує застарілі методи управління записом клієнтів: паперові журнали, особисті нотатки або, в кращому випадку, таблиці у Google Sheets. Такий підхід породжує численні проблеми –накладки у розкладі, втрачені записи, відсутність можливості самостійного онлайн-запису для клієнтів та нездатність аналізувати ефективність роботи закладу [2].

Цифрова трансформація б'юті-індустрії є глобальним трендом, що охопив провідні ринки Європи та США [3]. Системи онлайн-запису дозволяють клієнтам самостійно вибирати зручний час та майстра, скорочуючи навантаження на адміністраторів, мінімізуючи кількість неявок завдяки автоматичним нагадуванням та надаючи керівникам закладів аналітичні інструменти для прийняття управлінських рішень. Проте в Україні рівень цифровізації б'юті-закладів залишається невисоким: за різними оцінками, повноцінні системи управління записом використовують не більше 25–30% закладів, а серед малих підприємств цей показник ще нижчий. Середній рівень неявок клієнтів без автоматичних нагадувань становить 18–22%, що генерує конкретні фінансові втрати для власників закладів [2].

Окремою проблемою є відсутність якісних мобільних рішень для управління записом, адаптованих до потреб українського ринку. Провідні міжнародні платформи –Fresha, Treatwell, Booksy –мають обмежену підтримку для України: неповна локалізація, відсутність інтеграції з вітчизняними платіжними системами, незручні умови підписки для невеликих закладів. При цьому більшість клієнтів та майстрів здійснюють взаємодію через смартфони

–частка мобільного трафіку у сфері послуг краси перевищує 75% [2]. Це формує стійкий запит на мобільну інформаційну систему, що забезпечує зручний запис для клієнтів та ефективне управління розкладом для майстрів і адміністраторів безпосередньо з мобільного пристрою.

Сучасна система управління записом у б'юті-салоні повинна задовольняти потреби трьох категорій користувачів одночасно. Клієнт повинен мати можливість самостійно обрати послугу, майстра та зручний час, здійснити попередню оплату або бронювання, отримувати нагадування про майбутній запис та переглядати історію відвідувань. Майстер повинен бачити свій актуальний розклад, отримувати сповіщення про нові записи та скасування, управляти власним графіком роботи та відпустками. Адміністратор або власник закладу повинен мати доступ до повного розкладу всіх майстрів, аналітики завантаженості, фінансових показників та інструментів управління персоналом і переліком послуг.

Об'єктом дослідження є процес побудови розподілених мобільних інформаційних систем планування та управління записом клієнтів у сфері послуг б'юті-індустрії.

Предметом дослідження є методи та програмні засоби автоматизованого планування розкладу, конкурентного бронювання часових слотів і управління процесом обслуговування клієнтів у мобільних інформаційних системах.

Метою роботи є зменшення кількості неявок клієнтів та підвищення завантаженості майстрів б'юті-закладів шляхом розробки мобільної інформаційної системи планування та управління записом з автоматичними сповіщеннями і транзакційним захистом від конкурентного бронювання, що забезпечує збільшення доходу закладу та скорочення часу адміністрування розкладу.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

– провести аналіз ринку б'юті-індустрії України та дослідити існуючі підходи до управління записом клієнтів у закладах сфери послуг;

– виконати порівняльний аналіз провідних конкурентних рішень – Fresha, Treatwell та Booksy –для визначення конкурентних переваг розроблюваної системи;

– розробити специфікацію вимог з урахуванням потреб трьох категорій користувачів: клієнта, майстра та адміністратора;

– спроєктувати архітектуру системи та реляційну схему бази даних із підтримкою повного циклу управління записом і механізмом захисту від конкурентного бронювання;

– реалізувати серверну частину на основі Spring Boot із RESTful API, інтеграцією платіжної системи та модулем push-сповіщень;

– розробити мобільний додаток на React Native з Expo з окремими інтерфейсами для клієнта, майстра та адміністратора;

– провести комплексне тестування системи, включаючи функціональне, навантажувальне та спеціалізоване тестування захисту від конкурентного запису.

Практична цінність розробленої системи визначається орієнтацією на реальні потреби малих та середніх закладів б'юті-індустрії в Україні. Система знижує адміністративне навантаження на персонал завдяки автоматизації запису та нагадувань, підвищує лояльність клієнтів через зручний мобільний інтерфейс самостійного запису, скорочує кількість неявок завдяки push-сповіщенням та SMS-нагадуванням, а також надає керівництву закладу аналітичні інструменти для оптимізації завантаженості майстрів і структури надаваних послуг.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ ДЛЯ УПРАВЛІННЯ ЗАПИСОМ У Б'ЮТІ-САЛОНАХ

1.1 Огляд ринку б'юті-індустрії та цифровізації послуг краси в Україні

Б'юті-індустрія є однією з найбільш стійких галузей сфери послуг, що демонструє стабільне зростання навіть у кризові періоди. В Україні ринок послуг краси формується переважно малими підприємствами та самозайнятими майстрами, що надає йому специфічну структуру, відмінну від розвинених ринків Європи. Розуміння цієї специфіки є ключовим для проєктування системи управління записом, яка дійсно відповідатиме потребам цільової аудиторії [1].

Загальна кількість зареєстрованих закладів б'юті-сфери в Україні перевищує 60 000 одиниць. Структурно ринок представлений кількома сегментами: салони краси широкого профілю (перукарські послуги, манікюр, косметологія) займають найбільшу частку близько 38%, студії нігтьового сервісу – 22%, барбершопи, що набули надзвичайної популярності з 2016 року, – 14%, spa та масажні кабінети – 13%, студії корекції брів і нарощування вій – ще 13%. Переважна більшість цих закладів є малими підприємствами з кількістю майстрів від одного до п'яти. Саме цей сегмент малих закладів є найменш охопленим цифровими рішеннями та водночас найбільш чисельним [2].

Рівень цифровізації управління записом у б'юті-індустрії України залишається вкрай низьким. За результатами галузевих досліджень, лише 25–30% закладів використовують спеціалізоване програмне забезпечення для управління записом клієнтів. Решта, близько 70–75%, досі покладаються на паперові журнали, особисті нотатки у смартфоні, таблиці Google Sheets або ведення записів через месенджери (Telegram, Viber, Instagram Direct Messages). Такий підхід є не лише незручним, але й генерує конкретні фінансові втрати: середній рівень неявок клієнтів без системи автоматичних нагадувань

становить 18–22% від загальної кількості записів, а для малого салону з трьома майстрями це може означати втрату 500–800 гривень щодня [3].

Особливо динамічним сегментом є зростання числа самозайнятих майстрів, що працюють вдома або орендують крісло в орендних студіях (home-beauty). Цей тренд посилився після 2020 року та набув нового імпульсу в умовах воєнного часу, коли частина майстрів перейшла з найманої роботи на самозайнятість. Для такої категорії виконавців потреба в мобільному інструменті управління власним розкладом є особливо гострою: їм не потрібні складні корпоративні системи з великою кількістю налаштувань, але критично важлива можливість приймати самозаписи від клієнтів, надсилати нагадування та отримувати оплату, все з мобільного телефону [4].

З точки зору поведінки кінцевих споживачів, клієнтів б'юті-закладів, цифрові канали стали домінуючими. Більше 75% клієнтів здійснюють пошук салону та запис через смартфон. При цьому 68% клієнтів вказують, що надають перевагу салонам, де можна записатись онлайн без необхідності телефонувати. Особливо виразна ця тенденція у вікових групах 18–35 років, які є основною аудиторією б'юті-послуг. Такий запис із великою ймовірністю здійснюватиметься у вечірні години або у вихідні, коли адміністратор недоступний саме тому наявність цілодобового самозапису є конкурентною перевагою [5].

Воєнний стан 2022–2023 років вніс суттєві корективи у функціонування галузі. Ринок скоротився приблизно на 30–35% у 2022 році через вимушену міграцію населення, руйнування закладів та загальне зниження споживання. Проте з другої половини 2023 року спостерігається відновлення та адаптація: частина закладів, що вижили, навпаки прискорила цифровізацію, дефіцит персоналу-адміністраторів спонукає власників більше покладатися на автоматизовані системи запису. Ключові характеристики ринку б'юті-індустрії в Україні систематизовано у таблиці 1.1.

Таблиця 1.1 - Характеристика ринку б'юті-індустрії та цифровізації в Україні

Аспект	Характеристика
Обсяг ринку (2023)	Понад 42 млрд грн на рік; частка у ВВП сфери послуг – близько 3,8%
Кількість закладів	Понад 60 000 зареєстрованих б'юті-закладів по Україні; переважають малі підприємства (1–5 майстрів)
Структура ринку	Салони краси – 38%, студії нігтів – 22%, барбершопи – 14%, spa/масаж – 13%, студії брів і вій – 13%
Рівень цифровізації	Лише 25–30% закладів використовують спеціалізоване ПЗ для управління записом; решта – паперові журнали або месенджери
Мобільна активність	Понад 75% клієнтів здійснюють пошук і запис через смартфон; 68% очікують можливість онлайн-самозапису
Проблеми галузі	Середній відсоток неявок (no-show) без нагадувань – 18–22%; втрати від неявок у малих салонах – до 15% виручки
Тенденції	Зростання самозайнятих майстрів (home-beauty); попит на гнучкі системи з мобільним управлінням; інтеграція онлайн-оплати
Вплив воєнного часу	Ринок скоротився на 30–35% у 2022, відновлюється з 2023; зростання попиту на автоматизацію через дефіцит персоналу

Аналіз таблиці 1.1 підтверджує наявність значного незадоволеного попиту на доступну та локалізовану систему управління записом для малих б'юті-закладів. Ключовими незаповненими нішами є: відсутність

українськомовних рішень із підтримкою гривні, недоступність міжнародних платформ для малих підприємств через ціну або функціональну надмірність, а також брак рішень, що в рівній мірі враховують потреби трьох учасників процесу: клієнта, майстра та власника закладу.

Важливим контекстом для проєктування системи є специфіка українського платіжного ландшафту. Найпоширенішими платіжними інструментами в Україні є картки Visa та Mastercard у поєднанні з місцевими еквайринговими сервісами Liqpay та Wayforpay. Міжнародна платформа Stripe, що є стандартом у більшості закордонних систем, стала повноцінно доступною в Україні з 2022 року. Розроблювана система реалізує підтримку Stripe як основного платіжного шлюзу із закладеною архітектурною можливістю додавання Liqpay як альтернативи без переписування бізнес-логіки [5].

Методики та підходи до управління записом у б'юті-закладах, що будуть реалізовані у системі, наведено у таблиці 1.2.

Таблиця 1.2 – Методики управління записом клієнтів та їх реалізація в системі

Підхід	Принцип	Реалізація в системі
Самозапис клієнта (self-booking)	Клієнт самостійно обирає послугу, майстра та слот без участі адміністратора	Мобільний екран вибору послуги → майстра → часового слоту → підтвердження
Адміністративний запис	Адміністратор або майстер вносить запис вручну (телефон, особисте звернення)	Панель адміністратора з календарем та формою ручного запису від імені клієнта

Кінець таблиці 1.2

Підхід	Принцип	Реалізація в системі
Захист від no-show (неявок)	Попередня оплата або утримання депозиту для підтвердження запису	Опційний передоплатний запис через Stripe API; налаштовується власником закладу
Автоматичні нагадування	Push та SMS за 24 год та 2 год до запису знижують no-show на 40–60%	Spring TaskScheduler на Spring Boot; FCM для push; Twilio API для SMS
Управління чергою очікування	При скасуванні запису система автоматично сповіщає клієнтів з черги очікування	Waitlist-модуль: клієнт ставиться у чергу на вибраний слот; автосповіщення при вивільненні
Оптимізація завантаженості	Рекомендація менш завантажених майстрів або часових слотів для балансування навантаження	Smart Scheduling: підсвічування «популярних» та «вільних» слотів у інтерфейсі клієнта
Програма лояльності	Нарахування бонусів за відвідування стимулює повторні записи та утримання клієнтів	Модуль бонусних балів: нарахування за кожний запис; списання при оплаті

Таблиця 1.2 демонструє, що розроблювана система реалізує комплексний набір методик управління записом, включаючи захист від неявок через передоплату, управління чергою очікування та програму лояльності, функції, що відсутні або доступні лише на платних тарифах у конкурентів.

1.2 Порівняльний аналіз існуючих систем управління записом

Для обґрунтування доцільності розробки нової системи та формування конкурентних вимог необхідно провести детальний аналіз трьох провідних міжнародних платформ управління записом у б'юті-секторі: Fresha, Treatwell та Booksy. Ці платформи представляють різні бізнес-моделі та підходи до вирішення задачі й разом охоплюють переважну частину глобального ринку. Аналіз кожної платформи проводиться за єдиною схемою для забезпечення порівнянності результатів [6].

Fresha є однією з найбільш агресивно зростаючих платформ у секторі. Заснована у 2015 році під назвою Shedul (перейменована у 2019), вона здобула широке поширення завдяки нетиповій для ринку безкоштовній базовій моделі. Характеристики платформи Fresha наведено у таблиці 1.3.

Таблиця 1.3 – Характеристика платформи Fresha

Характеристика	Опис
Компанія та рік заснування	Fresha (раніше Shedul), Велика Британія, 2015; понад 100 000 партнерів у 120 країнах
Концепція	«Безкомісійна» платформа для б'юті-бізнесу з монетизацією через Fresha Payments (обробка карток)
Платформи	Web, iOS, Android; окремі додатки для бізнесу та клієнтів
Ключові функції	Онлайн-запис, управління персоналом, POS-термінал, інвентаризація, маркетинг (email/SMS)
Монетизація	Безкоштовний базовий план; комісія 20% на перші онлайн-транзакції нових клієнтів через платіжну систему Fresha

Кінець таблиці 1.3

Характеристика	Опис
Сильні сторони	Безкоштовний базовий тариф, широкий функціонал, велика міжнародна аудиторія клієнтів, вбудований маркетплейс
Слабкі сторони	Обмежена підтримка в Україні, немає локалізації гривнею, відсутня інтеграція з Liqpay/Wayforpay, складна кастомізація для специфіки місцевого ринку

Бізнес-модель Fresha є принципово відмінною від конкурентів: платформа не стягує щомісячну підписку, монетизуючись натомість через комісію 20% із транзакцій нових клієнтів, що прийшли через маркетплейс Fresha. Для закладів, які використовують Fresha виключно як систему управління записом без залучення через маркетплейс, платформа є дійсно безкоштовною. Це робить Fresha привабливим вибором для малих закладів. Проте для українського ринку ця модель має суттєве обмеження: Fresha не підтримує гривню як валюту та не інтегрована з українськими еквайринговими сервісами. Система запису без вбудованої оплати у місцевій валюті позбавляє клієнта однієї з ключових переваг платформи [7].

Treatwell є лідером ринку Західної Європи та найстаршою з трьох аналізованих платформ. Заснована у 2008 році під назвою Wahanda, вона переросла з простого каталогу б'юті-послуг у повноцінну екосистему, що поєднує маркетплейс для клієнтів та інструмент управління для бізнесу. Характеристики платформи Treatwell наведено у таблиці 1.4.

Таблиця 1.4 – Характеристика платформи Treatwell

Характеристика	Опис
Компанія та рік заснування	Treatwell (раніше Wahanda), Велика Британія, 2008; лідер ринку Європи; понад 50 000 партнерів
Концепція	Маркетплейс б'юті-послуг із вбудованим онлайн-записом; акцент на залученні нових клієнтів через платформу
Платформи	Web, iOS, Android; розгалужена система для бізнесу (Connect) та для клієнтів (маркетплейс)
Ключові функції	Маркетплейс послуг, онлайн-запис, управління командою, аналітика, програма лояльності, подарункові сертифікати
Монетизація	Комісія 20–30% з кожного бронювання через маркетплейс; щомісячна підписка за додатковий функціонал
Сильні сторони	Потужний маркетплейс для залучення нових клієнтів, розвинена програма лояльності, глибока аналітика
Слабкі сторони	Відсутній в Україні, висока комісія, складна система тарифів, надлишковий функціонал для малих закладів

Особливістю бізнес-моделі Treatwell є акцент на маркетплейсі як основному каналі цінності: платформа залучає нових клієнтів до закладів, стягуючи при цьому комісію 20–30% від вартості бронювання. Для закладів, що мають достатній потік власних клієнтів, ця модель є надто дорогою. Treatwell формально відсутній на українському ринку та не підтримує місцевий платіжний ландшафт. Аналітичні інструменти платформи є найбільш розвиненими серед трьох аналогів, проте їх складність та глибина надлишкові для малого салону з 2–3 майстрами [8].

Booksy є найближчим за функціональним наповненням до розроблюваної системи серед трьох аналогів. Заснований у 2014 році польсько-американською командою, він первісно орієнтувався на барбершопи, згодом розширившись на весь б'юті-сектор. Характеристики платформи Booksy наведено у таблиці 1.5.

Таблиця 1.5 – Характеристика платформи Booksy

Характеристика	Опис
Компанія та рік заснування	Booksy, США/Польща, 2014; понад 38 000 партнерів у 30 країнах; особливо сильні позиції у Польщі та США
Концепція	Спеціалізована платформа для б'юті- та wellness-сектору з акцентом на мобільному записі та картковій оплаті
Платформи	iOS, Android, Web; мобільний додаток є основним каналом взаємодії; окремий Booksy for Business
Ключові функції	Онлайн-запис з оплатою, управління розкладом, Boost (маркетинг), відгуки та рейтинги, нагадування
Монетизація	Підписка від 29,99 дол./міс. для базового тарифу; відсоткова комісія за Boost-маркетинг
Сильні сторони	Найкращий мобільний UX серед конкурентів, вбудована оплата карткою, сильний маркетинг, відгуки клієнтів
Слабкі сторони	Ціна недоступна для малих українських закладів, немає гривневих платежів, обмежена локалізація, дорогий Boost

Booksy вирізняється найякіснішим мобільним досвідом серед конкурентів: мобільний додаток є основним каналом взаємодії як для клієнтів,

так і для власників бізнесу. Усі три аналізовані платформи використовують React Native як основу мобільного клієнта – це галузевий стандарт, однак розроблювана система на React Native з Expo та TypeScript забезпечує додатковий рівень типізації та продуктивності розробки. Головним недоліком Booksy для українського ринку є ціна: базовий тариф від 29,99 дол./місяць є суттєвим бар'єром для малого закладу у поточних економічних умовах України [9].

Зведений порівняльний аналіз усіх трьох аналогів та розроблюваної системи наведено у таблиці 1.6.

Таблиця 1.6 – Зведений порівняльний аналіз існуючих рішень та розроблюваної системи

Критерій	Fresha	Treatwell	Booksy	Розроблювана система
Мобільний додаток клієнта	Так	Так	Так	Так (React Native)
Додаток для майстра	Частково	Ні	Так	Так (окрема роль)
Панель адміністратора	Так (web)	Так (web)	Так (mobile)	Так (mobile)
Онлайн-оплата гривнею	Ні	Ні	Ні	Так (Stripe + Liqpay)
Українська локалізація	Ні	Ні	Частково	Так (повна)
Push-нагадування	Так	Так	Так	Так (FCM/APNs)
SMS-нагадування	Платно	Платно	Платно	Так (Twilio/Kyivstar)

Кінець таблиці 1.6

Критерій	Fresha	Treatwell	Booksy	Розроблювана система
Управління послугами	Так	Так	Так	Так
Аналітика завантаженості	Базова	Розвинена	Базова	Так (графіки)
Програма лояльності	Ні	Так	Ні	Бонусні бали
Підтримка малих салонів (1–3 майстри)	Так	Обмежено	Дорого	Так (пріоритет)
Безкоштовний тариф	Так	Ні	Ні	Так (до 3 майстрів)
Стек мобільного клієнта	React Native	React Native	React Native	React Native + Expo + TS

Аналіз таблиці 1.6 дозволяє чітко визначити ринкову нішу розроблюваної системи. По-перше, жодна з трьох платформ не має повноцінної підтримки українського ринку, всі три відсутні або обмежено присутні в Україні без локалізації гривнею. По-друге, у жодної платформи немає окремого повноцінного мобільного інтерфейсу для майстра, цей компонент або відсутній, або суміщений з адміністраторським без урахування специфіки роботи майстра. По-третє, ціна або відсутність безкоштовного тарифу є бар'єром для найчисельнішого сегменту малих закладів. Розроблювана система вирішує всі три проблеми одночасно, що і визначає її конкурентне позиціонування.

З технічної точки зору принципово важливим є вибір Spring Boot як серверної платформи замість Node.js, що є стандартом у конкурентів. Spring

Boot забезпечує значно потужнішу підтримку транзакцій, що є критичним для систем з конкурентним записом: коли два клієнти одночасно намагаються записатися до одного майстра на один і той самий час, система повинна гарантувати, що лише одна з транзакцій завершиться успішно. JPA/Hibernate із Optimistic Locking на рівні PostgreSQL є перевіреним корпоративним рішенням для таких сценаріїв, що складно відтворити на Node.js без суттєвих зусиль.

1.3 Вимоги до розробки мобільної інформаційної системи управління записом

На основі проведеного аналізу ринку б'юті-індустрії та конкурентних рішень сформовано комплексні вимоги до розроблюваної мобільної інформаційної системи. Оскільки система обслуговує три принципово різні ролі: клієнта, майстра та адміністратора, вимоги формулюються окремо для кожної ролі, що відображає реальну специфіку використання системи в умовах реального б'юті-закладу.

Клієнтська частина системи повинна забезпечувати знайомий та інтуїтивний процес самозапису: вибір закладу або майстра, перегляд доступних послуг із цінами та тривалістю, вибір конкретного спеціаліста або режим «будь-який доступний», вибір часового слоту на інтерактивному календарі та підтвердження запису з опціональною передоплатою. Додатковими клієнтськими функціями є перегляд профілю майстра з портфоліо робіт та відгуками інших клієнтів, управління власними майбутніми та минулими записами, участь у програмі лояльності та отримання персональних пропозицій. Важливою вимогою є можливість запису без реєстрації (як гість), щоб мінімізувати тертя при першому використанні.

Для майстра система повинна надавати зручний мобільний інтерфейс управління власним робочим розкладом. Основний екран майстра, денний та тижневий вигляд розкладу з усіма записами, де кожен запис відображає ім'я

клієнта, послугу, тривалість та статус оплати. Майстер повинен мати можливість підтверджувати або відхиляти записи, позначати відвідування як виконані, вносити нотатки про клієнта, а також управляти власним графіком роботи та встановлювати перерви або відпустки. Окремою важливою функцією є перегляд статистики власної роботи: кількість записів, виручка, топ-послуги.

Адміністраторська частина системи є найширшою за функціональністю. Адміністратор повинен мати доступ до зведеного розкладу всього персоналу на один день або тиждень із можливістю фільтрації за майстром. Обов'язковими функціями є управління довідниками (послуги, категорії послуг, майстри, ресурси) та ручний запис клієнта від імені адміністратора. Аналітична панель повинна відображати ключові метрики: кількість записів, виручку, завантаженість майстрів, популярні послуги та аналіз неявок. Управління акціями, сезонними знижками та сповіщеннями клієнтів є додатковими, але важливими адміністраторськими можливостями.

Нефункціональні вимоги охоплюють продуктивність (час відгуку API не більше 300 мс при нормальному навантаженні, завантаження екрану розкладу не більше 1,5 с), безпеку (Firebase Auth з JWT-токенами, шифрування платіжних даних через Stripe PCI DSS рівня 1, HTTPS для всіх з'єднань), масштабованість (підтримка одночасної роботи до 5000 активних клієнтів одного закладу) та надійність (uptime сервісу 99,5%, автоматичне резервне копіювання БД щодня). Мобільний додаток повинен підтримувати Android 9.0 і вище та iOS 15 і вище, що охоплює понад 95% активних пристроїв [10].

Огляд ключових завдань розробки системи наведено у таблиці 1.7.

Таблиця 1.7 – Ключові завдання розробки мобільної інформаційної системи

Етап розробки	Опис завдань
Аналіз предметної області	Дослідження ринку б'юті-індустрії України, виявлення потреб клієнтів, майстрів та адміністраторів закладів
Аналіз конкурентних рішень	Порівняльний аналіз Fresha, Treatwell, Booksy; визначення конкурентних переваг та незайнятих ніш
Формування вимог SRS	Специфікація функціональних та нефункціональних вимог з розподілом за ролями користувачів
Проєктування БД	Розробка реляційної схеми PostgreSQL для підтримки записів, персоналу, послуг та платежів
Розробка серверної частини	Реалізація Spring Boot REST API, інтеграція Firebase Auth, Stripe та Spring TaskScheduler для нагадувань
Розробка мобільного додатку	React Native + Expo із трьома навігаційними стеками для клієнта, майстра та адміністратора
Тестування системи	Функціональне тестування сценаріїв запису; навантажувальне тестування Spring Boot з JMeter
Публікація та розгортання	Docker-контейнеризація серверу; CI/CD через GitHub Actions; публікація у Google Play та App Store

Таблиця 1.7 охоплює повний цикл розробки від аналізу до публікації у магазинах застосунків. Принциповою відмінністю від конкурентних рішень є орієнтація на мобільний інтерфейс як основний для всіх трьох ролей, не лише для клієнта, але й для майстра та адміністратора. Це відповідає реальній практиці роботи б'юті-закладів, де персонал постійно переміщується та не має постійного доступу до стаціонарного комп'ютера.

Окремою вимогою є реалізація системи захисту від конкурентного запису, race condition, коли два клієнти намагаються зайняти один і той самий часовий слот одночасно. Ця проблема є критичною для будь-якої системи управління записом і вирішується на рівні бази даних через Optimistic Locking із версіонуванням записів та транзакційну атомарність на рівні Spring Boot JPA. Клієнт, чия транзакція відбулась пізніше, отримує повідомлення про те, що обраний слот вже зайнятий, та пропозицію обрати інший час або потрапити до черги очікування.

1.4 Висновки за розділом 1

У першому розділі проведено комплексний аналіз ринку б'юті-індустрії України та існуючих рішень для управління записом клієнтів, що надає повне обґрунтування доцільності розробки нової мобільної інформаційної системи.

Аналіз ринку показав, що б'юті-індустрія України налічує понад 60 000 закладів, переважно малих (1–5 майстрів), з яких лише 25–30% використовують спеціалізоване ПЗ. Рівень неявок без автоматичних нагадувань досягає 18–22%, що формує конкретний фінансовий збиток для закладів. Більше 75% клієнтів здійснюють запис через смартфон, і 68% надають перевагу можливості онлайн-самозапису. Зростання сегменту самозайнятих майстрів та відновлення ринку після 2022 року формують додатковий попит на доступні мобільні рішення.

Порівняльний аналіз Fresha, Treatwell та Booksy виявив системні прогалини на українському ринку: всі три платформи не підтримують гривню та місцеві платіжні системи, не мають повноцінної україномовної локалізації, а Treatwell і Booksy є надто дорогими для малих закладів. Жодна платформа не надає окремого повноцінного мобільного інтерфейсу для майстра. Всі три використовують React Native – розроблювана система також базується на React Native, але з Expo та TypeScript для підвищення якості коду. Принциповою технічною перевагою є вибір Spring Boot замість Node.js для

серверної частини, що забезпечує надійне транзакційне управління конкурентними записами.

Сформовані вимоги охоплюють три рольових профілі (клієнт, майстер, адміністратор) та включають функціонал самозапису, управління розкладом, онлайн-оплати через Stripe, автоматичних нагадувань, черги очікування, програми лояльності та аналітики. Нефункціональні вимоги визначають відгук API менше 300 мс, uptime 99,5% та підтримку Android 9.0+ і iOS 15+. Матеріали першого розділу є повним обґрунтуванням для проєктування архітектури та бази даних системи в наступному розділі.

2 ПРОЄКТУВАННЯ МОБІЛЬНОЇ ІНФОРМАЦІЙНОЇ СИСТЕМИ УПРАВЛІННЯ ЗАПИСОМ

2.1 Розробка специфікації вимог до системи управління записом

Специфікація вимог до програмного забезпечення (SRS) визначає повний перелік функціональних можливостей та якісних характеристик системи, що є основою для всіх подальших проєктних рішень. Для мобільної інформаційної системи управління записом б'юті-салону специфікація є особливо важливою, оскільки система одночасно обслуговує три принципово різні категорії користувачів із відмінними потребами та сценаріями використання.

Функціональні вимоги систематизовано за рольовим принципом: вимоги клієнтської частини (FR-01 – FR-07), вимоги до функціоналу майстра (FR-08 – FR-10) та вимоги адміністраторської панелі (FR-11 – FR-14). Таке розділення безпосередньо відображається на архітектурі навігаційних стеків мобільного додатку та системі контролю доступу (RBAC) на серверній частині. Перелік ключових функціональних вимог наведено у таблиці 2.1 (табл. 2.1).

Таблиця 2.1 – Функціональні вимоги до мобільної інформаційної системи

Код	Роль	Опис функціональної вимоги
FR-01	Клієнт	Перегляд каталогу послуг салону з цінами, тривалістю та категоріями. Фільтрація за категорією та діапазоном цін
FR-02	Клієнт	Вибір конкретного майстра або режим «перший доступний». Перегляд профілю майстра з портфоліо та рейтингом

Продовження таблиці 2.1

Код	Роль	Опис функціональної вимоги
FR-03	Клієнт	Вибір часового слоту на інтерактивному календарі. Система відображає лише вільні слоти з урахуванням розкладу майстра та тривалості послуги
FR-04	Клієнт	Підтвердження запису з опціональною передоплатою через Stripe. Отримання підтвердження на email та push-сповіщення
FR-05	Клієнт	Управління власними записами: перегляд майбутніх та минулих, скасування (з поверненням коштів якщо > 24 год)
FR-06	Клієнт	Програма лояльності: нарахування бонусних балів за кожний виконаний запис, часткова оплата балами при наступному записі
FR-07	Клієнт	Черга очікування: якщо всі слоти зайняті, клієнт може стати у waitlist та отримати автоматичне сповіщення при вивільненні
FR-08	Майстер	Перегляд власного розкладу на день і тиждень. Кожен запис показує клієнта, послугу, тривалість, статус оплати та нотатки
FR-09	Майстер	Управління власним графіком: налаштування робочих годин на кожен день тижня, встановлення перерв та відпусток
FR-10	Майстер	Позначення запису як виконаного, додавання нотаток про клієнта. Перегляд власної аналітики: виручка, кількість клієнтів

Кінець таблиці 2.1

Код	Роль	Опис функціональної вимоги
FR-11	Адмін	Зведений розклад усього персоналу на день/тиждень із фільтрацією за майстром. Ручний запис від імені клієнта
FR-12	Адмін	Управління довідниками: послуги (назва, ціна, тривалість, категорія, депозит), майстри, категорії послуг
FR-13	Адмін	Аналітична панель: кількість записів, виручка, завантаженість майстрів, топ-послуги, аналіз no-show за період
FR-14	Адмін	Управління акціями та знижками: відсоткова або фіксована знижка, обмеження по термінах та послугах

Аналіз функціональних вимог таблиці 2.1 демонструє комплексне охоплення повного циклу обслуговування клієнта: від пошуку та вибору послуги (FR-01, FR-02) через процес бронювання із захистом від конфліктів (FR-03, FR-04) до управління лояльністю (FR-06) та чергою очікування (FR-07). Майстерські вимоги FR-08 – FR-10 забезпечують автономність майстра у управлінні власним розкладом без необхідності звертатись до адміністратора. Адміністраторські вимоги FR-11 – FR-14 надають повний контроль над роботою закладу, включаючи аналітику та маркетингові інструменти.

Взаємодія між акторами системи та доступними їм варіантами використання представлена у вигляді UML-діаграми варіантів використання (рис. 2.1).

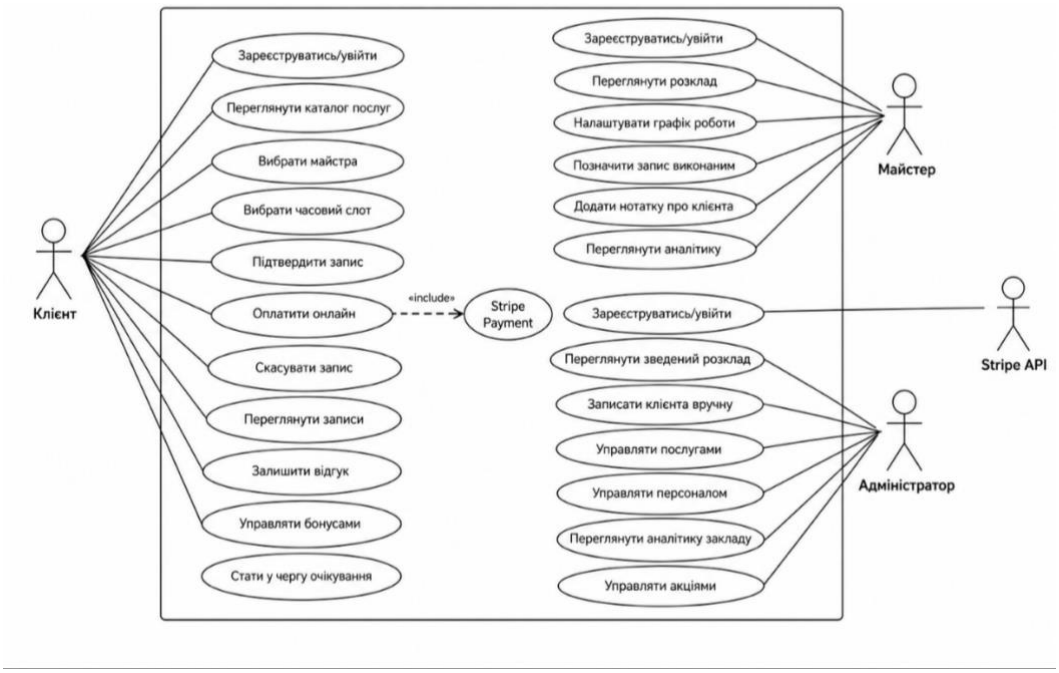

Рисунок 2.1 – UML-діаграма варіантів використання системи

Нефункціональні вимоги є особливо важливими для системи управління записом, де конкурентний доступ до обмеженого ресурсу (часових слотів майстра) є фундаментальним викликом. Вимога щодо транзакційної цілісності при конкурентному записі є унікальною для цього класу систем і безпосередньо впливає на вибір серверної технології – саме необхідність надійного Optimistic Locking стала одним із ключових аргументів на користь Spring Boot з JPA/Hibernate замість Node.js. Нефункціональні вимоги з кількісними показниками наведено у таблиці 2.2.

Таблиця 2.2 – Нефункціональні вимоги до мобільної інформаційної системи

Категорія	Показник	Значення та опис
Продуктивність	Час відгуку API	Не більше 300 мс для стандартних запитів; не більше 800 мс для аналітичних агрегацій
Продуктивність	Завантаження розкладу	Екран розкладу майстра на день – не більше 1,5 с; використання Redis-кешу для частих запитів

Кінець таблиці 2.2

Категорія	Показник	Значення та опис
Конкурентність	Race condition	Optimistic Locking на рівні PostgreSQL гарантує атомарність бронювання при одночасних запитах
Масштабованість	Навантаження	Підтримка до 5000 активних клієнтських сесій; горизонтальне масштабування Spring Boot за балансувальником
Безпека	Автентифікація	Firebase Auth JWT-токени; верифікація на Spring Security; RBAC для трьох ролей (client, master, admin)
Безпека	Платежі	Stripe PCI DSS рівня 1; карткові дані не зберігаються на власних серверах; тільки PaymentIntent ID
Безпека	Транспорт	HTTPS/TLS 1.3 для всіх з'єднань; HSTS; API доступний лише через Nginx reverse proxy
Доступність	Uptime	Не менше 99,5% на місяць; автоматичний Docker restart policy; щоденне резервне копіювання PostgreSQL
Сумісність	Mobile	Android 9.0 (API 28) і вище; iOS 15.0 і вище; охоплення понад 95% активних пристроїв
Локалізація	Мови	Українська (основна) та англійська; автоматичний вибір за системними налаштуваннями пристрою

Окремої уваги заслуговує вимога безпеки щодо обробки платежів. Система не зберігає жодних карткових даних клієнтів на власній серверній інфраструктурі – всі чутливі платіжні дані обробляються виключно на

серверах Stripe, сертифікованих за стандартом PCI DSS рівня (рис. 2.2). Власний сервер отримує та зберігає лише ідентифікатор транзакції PaymentIntent ID та статус оплати, що абсолютно виключає ризик витоку карткових даних при можливому компрометуванні власної бази даних [11].

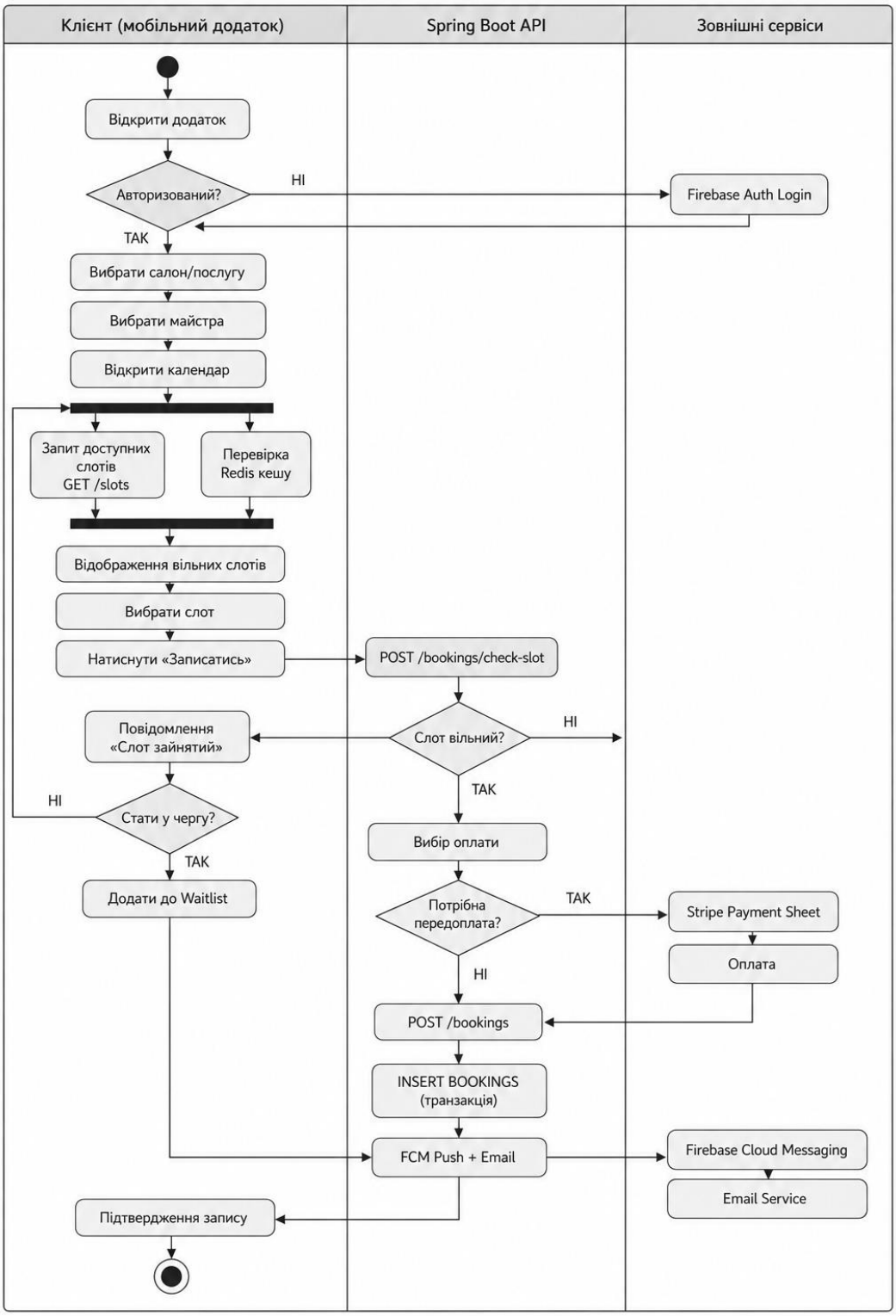

Рисунок 2.2 – UML-діаграма діяльності сценарію самозапису клієнта

2.2 Архітектура мобільної інформаційної системи

Проєктування архітектури системи управління записом б'юті-салону враховує специфічні вимоги предметної області: необхідність підтримки трьох ролей користувачів з різними інтерфейсами в єдиному мобільному додатку, транзакційна надійність при конкурентному записі, асинхронна відправка сповіщень та безпечна обробка платежів через зовнішній сервіс. Загальна архітектура системи наведена на рисунку 2.3. Схема відображає три рівні: мобільний клієнт (React Native + Expo, три навігаційних стеки для кожної ролі, Zustand, Firebase Auth SDK, Stripe SDK), серверна частина (Nginx API Gateway, Spring Boot контролери та сервіси, модулі безпеки, бронювань, сповіщень, платежів та аналітики) та рівень даних (PostgreSQL, Redis, Firebase Auth, Firebase Storage, Stripe, Twilio/FCM/APNs).

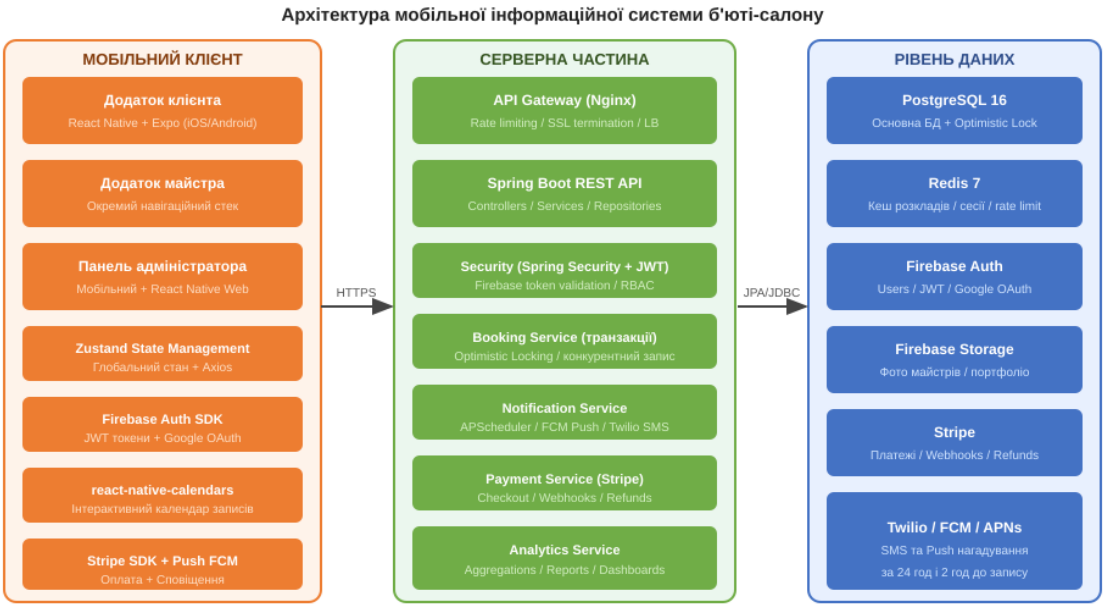

Рисунок 2.3 – Загальна архітектура мобільної інформаційної системи

Серверна частина побудована за трирівневою архітектурою Spring Boot із чітким розподілом відповідальності між шарами. Controller Layer відповідає за маршрутизацію HTTP-запитів, валідацію вхідних DTO через Bean Validation (@Valid), авторизацію через @PreAuthorize анотації Spring Security та

формування HTTP-відповідей. Service Layer містить бізнес-логіку: перевірку доступності слотів, управління транзакціями (@Transactional), взаємодію зі Stripe API та плануванням сповіщень. Repository Layer реалізований через Spring Data JPA та надає декларативний доступ до PostgreSQL без шаблонного SQL-коду для стандартних CRUD-операцій.

Spring Security налаштовано для верифікації Firebase JWT-токенів. При кожному запиті до захищених ендпоінтів фільтр JwtAuthenticationFilter витягує токен із заголовку Authorization, верифікує підпис через Firebase Admin SDK та встановлює SecurityContext із відповідними правами ролі. Ролі (ROLE_CLIENT, ROLE_MASTER, ROLE_ADMIN) визначаються на основі поля role у таблиці USERS, що підтягується при першому вході після верифікації Firebase UID.

Redis використовується у двох сценаріях. Перший – кешування доступних часових слотів: при запиті GET /slots/{staffId}/{date} сервер спочатку перевіряє Redis-ключ slots:{staffId}:{date}, і лише за його відсутності виконує розрахунок слотів із PostgreSQL. TTL кешу встановлено в 60 секунд – баланс між актуальністю та навантаженням на БД. При успішному бронюванні або скасуванні запису відповідний ключ кешу інвалідується. Другий сценарій – зберігання rate limit лічильників для захисту від зловживань (не більше 10 запитів на бронювання на хвилину з одного IP).

UML-діаграма компонентів системи відображає внутрішню структуру серверної частини та взаємозвязки між компонентами (рис. 2.4).

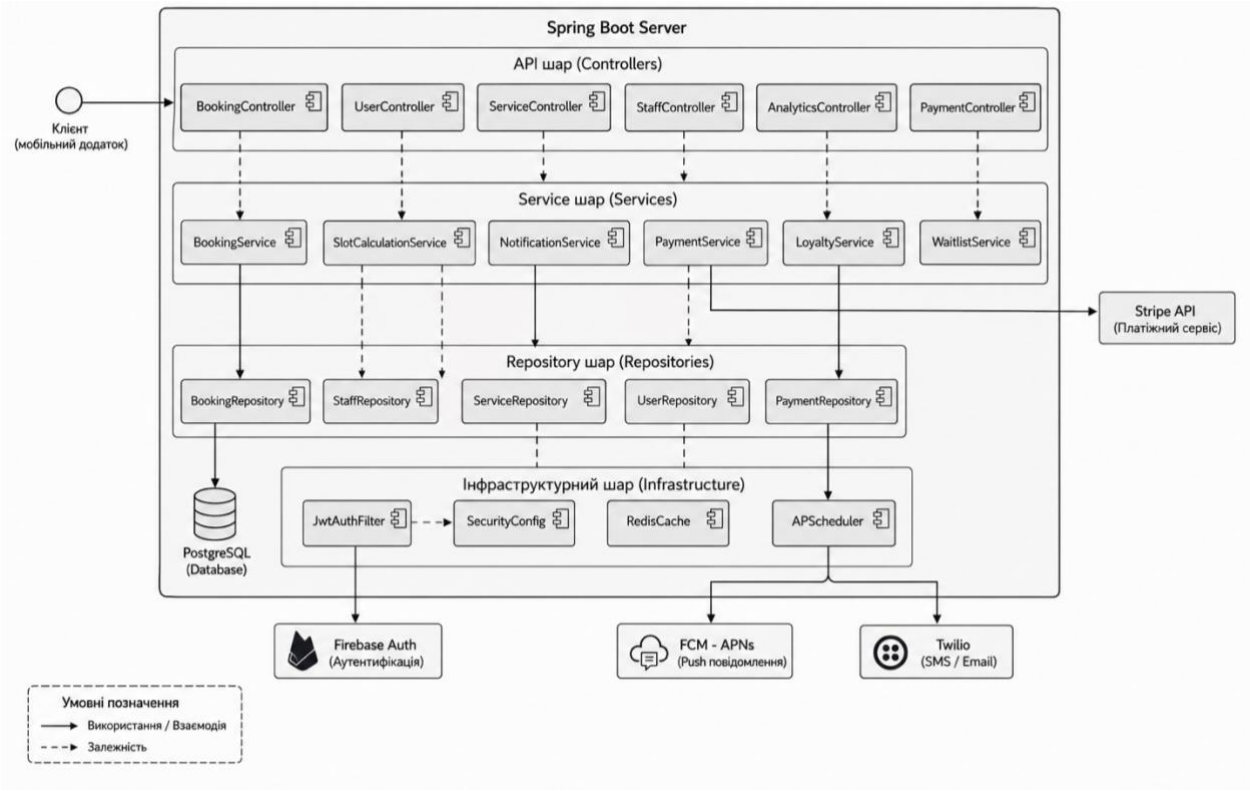

Рисунок 2.4 – UML-діаграма компонентів серверної частини Spring Boot

Вирішення проблеми конкурентного запису є ключовим архітектурним завданням системи. Сценарій race condition виникає, коли два клієнти одночасно бачать один і той самий вільний слот та майже одночасно надсилають запити на бронювання. Без спеціального захисту обидва записи можуть успішно пройти перевірку доступності та зберегтися в базі, що призведе до подвійного бронювання. Система вирішує цю проблему через двоетапний підхід. На першому етапі транзакційна перевірка доступності слоту виконується із SQL-конструкцією SELECT … FOR UPDATE SKIP LOCKED, що блокує рядки, які вже редагуються іншою транзакцією, від читання у тому самому запиті. На другому етапі при записі нового BOOKING використовується поле version для Optimistic Locking: якщо між перевіркою та вставкою версія конфліктуючого запису змінилась, JPA викидає OptimisticLockException, яке перехоплюється сервісним шаром і повертає клієнту відповідь з HTTP 409 Conflict та пропозицією обрати інший слот.

UML-діаграма послідовності для сценарію бронювання запису наведена на рисунку 2.5. Діаграма відображає взаємодію між шістьма учасниками: клієнт, мобільний додаток, BookingController, BookingService, PostgreSQL та Stripe API, а також Firebase Cloud Messaging для надсилання нагадувань. Ключовим кроком є крок 4 – SELECT FOR UPDATE SKIP LOCKED – що є серцем транзакційного захисту від дублікатів, та крок 12 – INSERT BOOKINGS з version=0 – початкове значення для Optimistic Locking.

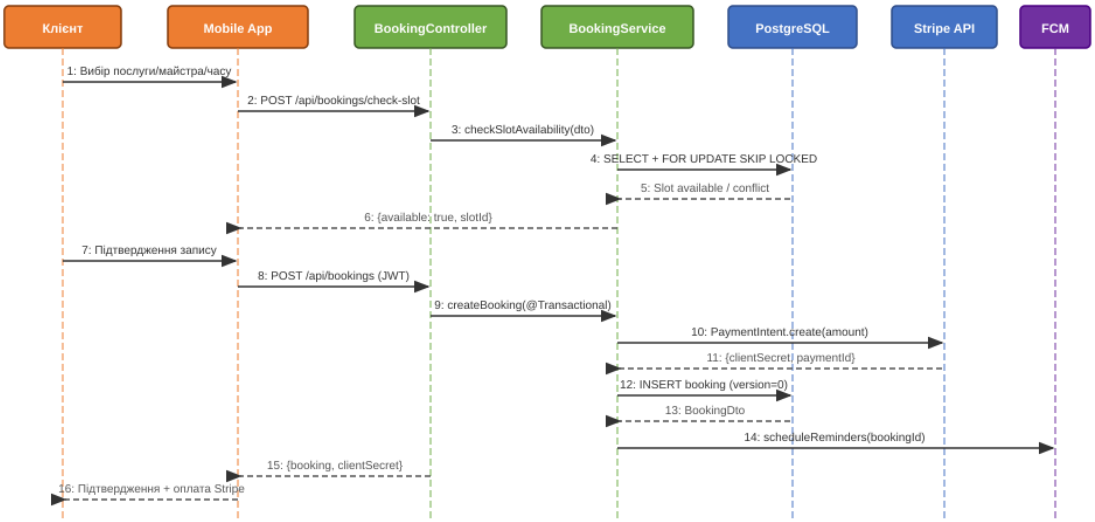

Рисунок 2.5 – UML-діаграма послідовності сценарію бронювання запису клієнтом

Сервіс нагадувань реалізований через Spring TaskScheduler Spring Boot. При успішному збереженні нового BOOKING сервіс автоматично створює два записи у таблиці NOTIFICATIONS: перший – за 24 години до scheduled_time запису, другий – за 2 години. Spring TaskScheduler перевіряє таблицю NOTIFICATIONS кожні 60 секунд та відправляє push-сповіщення через FCM і SMS через Twilio для записів, чий scheduled_at настав. Після успішної відправки поле sent_at оновлюється для аудиту. За результатами галузевих досліджень, система двоетапних нагадувань знижує рівень no-show з 18–22% до 6–9%, що є суттєвим економічним ефектом для закладу.

Мобільний клієнт реалізовано на React Native з Expo SDK 51 як єдиний застосунок із розгалуженою навігацією за ролями користувачів. Кореневий компонент NavigationContainer аналізує поточний стан Firebase Auth та поле role у JWT-токені й перенаправляє користувача в один із трьох незалежних навігаційних стеків: Client Stack побудовано на BottomTabNavigator з п'ятьма вкладками («Салони», «Запис», «Мої записи», «Бонуси», «Профіль»), Master Stack – на DrawerNavigator з розкладом, керуванням записами та профілем майстра, Admin Stack – на DrawerNavigator з KPI-дашбордом, керуванням записами усіх клієнтів закладу та довідниковими операціями над салонами й послугами. Управління станом реалізовано через бібліотеку Zustand із розподілом на чотири незалежні store: authStore зберігає Firebase ID Token, refresh-токен та профіль користувача з визначеною роллю; bookingStore тримає поточний потік самозапису з обраною послугою, майстром та часовим слотом; scheduleStore містить розклад майстра з кешуванням через React Query; adminStore зберігає фільтри та параметри періодів для KPI-звітів. Інтеграція з нативними сервісами виконується через спеціалізовані модулі Expo: expo-notifications відповідає за реєстрацію FCM-токену та обробку push-сповіщень, @stripe/stripe-react-native надає Payment Sheet з підтримкою Apple Pay і Google Pay, expo-secure-store забезпечує безпечне зберігання токенів автентифікації у Keychain (iOS) та Keystore (Android). Усі HTTP-запити до серверної частини проходять через єдиний axios-instance із interceptor, який автоматично додає Bearer-токен у заголовок Authorization та обробляє відповіді 401 шляхом повторної верифікації Firebase Auth.

Рівень даних об'єднує внутрішні сховища системи та зовнішні інтеграційні сервіси. Внутрішнім транзакційним сховищем є PostgreSQL 16, що зберігає всі бізнес-критичні дані: користувачів, салони, послуги, бронювання, платежі, відгуки та журнал нагадувань (детальна схема бази розглядається у підрозділі 2.3). Для оптимізації найчастіших запитів використовується кеш-сервер Redis 7 за схемою cache-aside: ключі формату slots:{staffId}:{serviceId}:{date} зберігають розраховані часові слоти з TTL 60

секунд, а ключі ratelimit:{ip}:bookings утримують лічильники rate limit за алгоритмом fixed window. Аутентифікація користувачів делегована зовнішньому сервісу Firebase Authentication, що відповідає за реєстрацію через email/password та номер телефону, верифікацію OTP-кодів, відновлення паролю та видачу підписаних JWT-токенів; серверна частина зберігає лише Firebase UID у полі firebase_uid таблиці USERS як зовнішній ключ для синхронізації локального профілю. Платіжний шлюз Stripe виконує роль зовнішньої системи обробки карткових даних відповідно до стандарту PCI DSS Level 1 – карткові реквізити клієнтів ніколи не передаються через серверну частину, обмін з нею відбувається через ідентифікатори PaymentIntent та підписані Webhook-події. Канали доставки сповіщень реалізовано через Firebase Cloud Messaging для push на Android та iOS, Apple Push Notification Service як транспорт у FCM для iOS, а також через Twilio для SMS як резервний канал доставки критичних нагадувань про запис.

2.3 Проєктування бази даних системи управління записом

База даних системи управління записом є реляційною та реалізована на PostgreSQL 16. Вибір реляційної СУБД замість документоорієнтованої (MongoDB) або хмарної NoSQL (Firestore) обґрунтований специфікою предметної області: дані про записи, розклади та оплати мають складні взаємозалежності, що природно виражаються через зовнішні ключі та JOIN-запити. Крім того, транзакційна цілісність ACID є обов'язковою вимогою для системи, що обробляє фінансові транзакції та конкурентні резервування.

ER-діаграма бази даних із відображенням усіх таблиць, полів та зв'язків між ними наведена на рисунку 2.6. На діаграмі відображено вісім основних таблиць: USERS (центральна сутність), SALONS, STAFF, SERVICES, SCHEDULES, BOOKINGS (центральна для бізнес-логіки), PAYMENTS та допоміжні REVIEWS, WAITLIST, NOTIFICATIONS. Зв'язки між таблицями позначені лініями з кардинальністю. Особливу увагу привертає поле version у

таблицях STAFF та BOOKINGS – воно використовується для Optimistic Locking на рівні JPA.

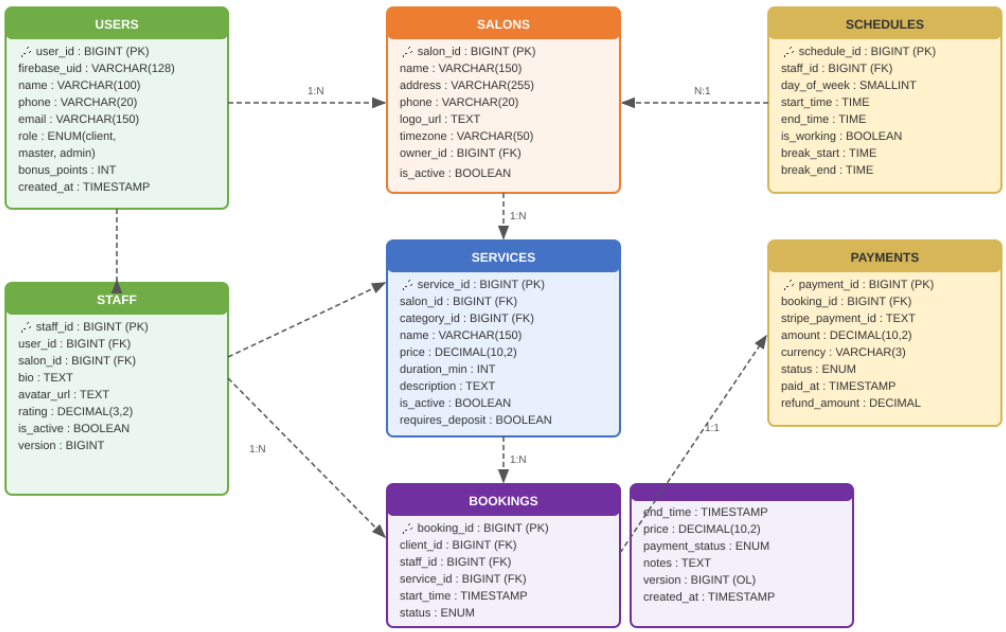

Рисунок 2.6 – ER-діаграма бази даних системи управління записом б'юті-салону

Опис призначення та ключових полів кожної таблиці наведено у таблиці 2.3.

Таблиця 2.3 – Опис таблиць бази даних системи

Таблиця	Призначення та ключові поля
USERS	Всі користувачі системи: клієнти, майстри, адміністратори. Поле firebase_uid зв'язує запис з Firebase Auth. role визначає доступ. bonus_points для програми лояльності
SALONS	Б'юті-заклади. Один власник (owner_id FK→USERS) може мати кілька салонів. timezone забезпечує коректний розрахунок слотів у різних часових поясах

Продовження таблиці 2.3

Таблиця	Призначення та ключові поля
STAFF	Майстри закладу. Пов'язує user_id та salon_id. Поле version використовується для Optimistic Locking при оновленні розкладу. rating автоматично перераховується після кожного відгуку
SERVICES	Каталог послуг салону. duration_min визначає тривалість для розрахунку доступних слотів. requires_deposit вмикає обов'язкову передоплату
BOOKINGS	Центральна таблиця записів. Містить version для Optimistic Locking. status (ENUM: pending, confirmed, completed, cancelled, no_show, refunded). payment_status відстежує стан Stripe-транзакції
SCHEDULES	Тижневий робочий графік майстра. day_of_week (0=Пн…6=Нд). break_start/break_end для автоматичного виключення перерв із доступних слотів
PAYMENTS	Фінансові транзакції. stripe_payment_id для звірки з Stripe Dashboard. refund_amount при частковому поверненні. currency зберігається явно для мультивалютності
REVIEWS	Відгуки клієнтів після виконаних записів. Один відгук на один запис (UNIQUE booking_id). rating (1–5) автоматично оновлює staff.rating через тригер

Кінець таблиці 2.3

Таблиця	Призначення та ключові поля
WAITLIST	Черга очікування на вивільнений слот. Зберігає staff_id, service_id, preferred_date. При скасуванні запису Spring TaskScheduler сповіщає першого у черзі
NOTIFICATIONS	Журнал запланованих та відправлених нагадувань. scheduled_at визначає час відправки Spring TaskScheduler. channel (push/sms). sent_at фіксує фактичний час

Розрахунок вільних часових слотів є найскладнішим алгоритмом на рівні бази даних. Для заданого майстра та дати алгоритм виконує такі кроки: зчитує робочий графік майстра із SCHEDULES для відповідного дня тижня; генерує перелік потенційних слотів із кроком, рівним тривалості послуги; зчитує всі підтверджені бронювання (status IN ('confirmed', 'pending')) на цю дату та виключає слоти, що перекриваються з існуючими записами із урахуванням їх тривалості; виключає слоти, що потрапляють у перерву майстра (break_start – break_end); повертає список вільних слотів у форматі ISO 8601. Результат кешується у Redis на 60 секунд із ключем slots:{staffId}:{serviceId}:{date}.

Індексна стратегія бази даних розроблена для оптимізації найбільш частих запитів. Складений індекс (staff_id, start_time, status) на таблиці BOOKINGS прискорює запит перевірки зайнятості слоту – найчастіший запит у системі. Частковий індекс WHERE status = 'confirmed' ще більше звужує вибірку для активних записів. Індекс (salon_id, is_active) на SERVICES оптимізує завантаження каталогу послуг. Індекс (firebase_uid) на USERS є унікальним та використовується при кожній авторизації.

Тригер автоматичного перерахунку рейтингу майстра спрацьовує після INSERT або UPDATE на таблиці REVIEWS. Тригер виконує AVG(rating) по

всіх відгуках для відповідного staff_id та оновлює поле rating в таблиці STAFF. Це забезпечує актуальність рейтингу без необхідності явного виклику окремої процедури з серверного коду.

Діаграма станів запису (Booking State Machine) є ключовою для розуміння бізнес-логіки переходів між статусами та пов'язаних з ними дій системи (рис. 2.7). Діаграма відображає шість станів (PENDING, CONFIRMED, COMPLETED, CANCELLED, NO_SHOW, REFUNDED), умови переходу між ними та автоматичні дії системи при кожному переході. Ключовим автоматичним переходом є CONFIRMED → NO_SHOW, що запускається Spring TaskScheduler через 15 хвилин після планового закінчення запису, якщо майстер не позначив його як виконаний.

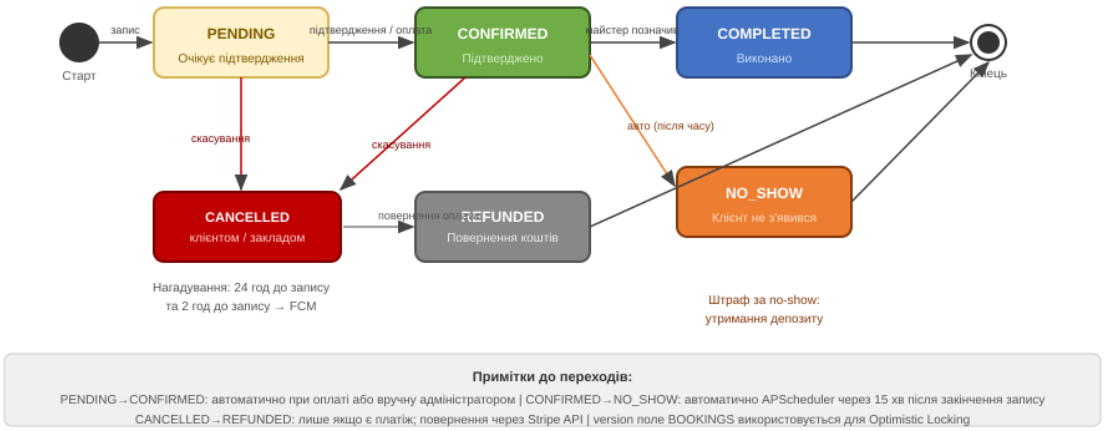

Рисунок 2.7 – UML-діаграма станів запису клієнта (Booking State Machine)

UML-діаграма класів доменного рівня системи відображає основні Java-сутності JPA та їхні зв'язки (рис. 2.8).

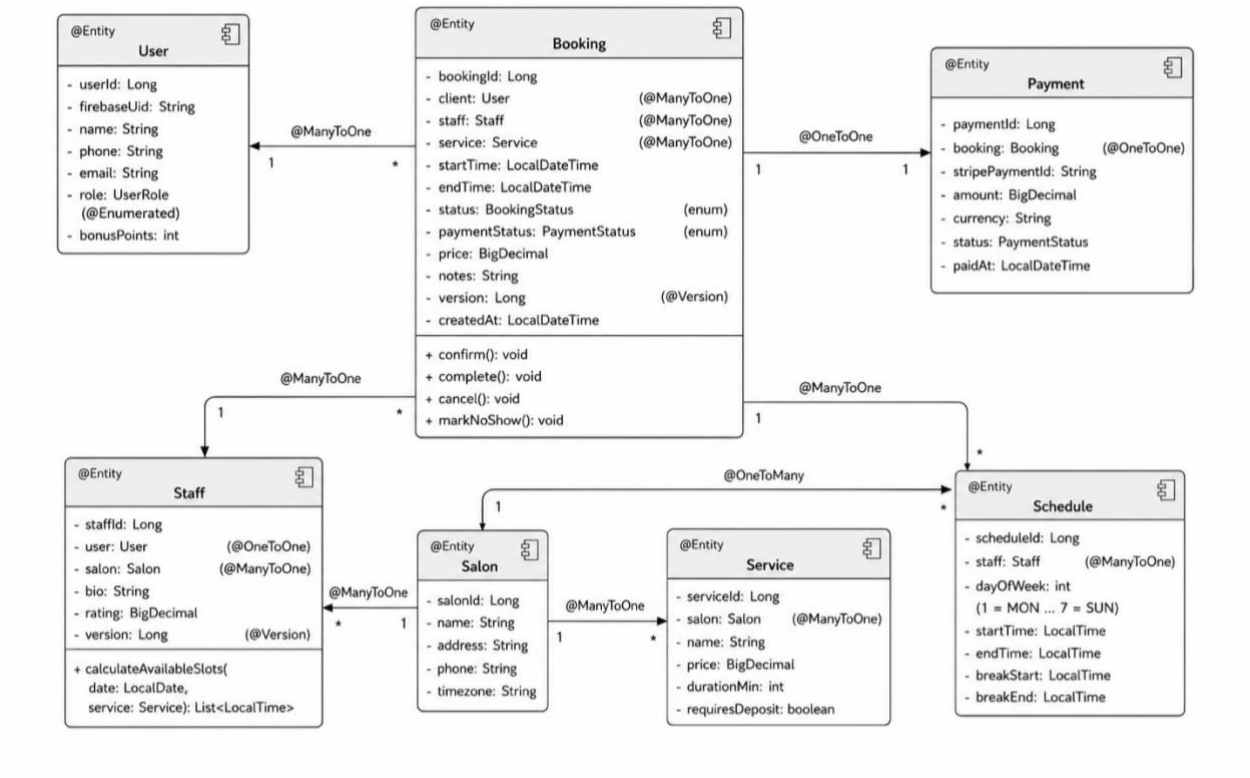

Рисунок 2.8 – UML-діаграма класів JPA-сутностей серверної частини

2.4 Висновки за розділом 2

У другому розділі виконано повний цикл проєктування мобільної інформаційної системи управління записом б'юті-салону із застосуванням розгорнутого UML-моделювання, що охоплює шість різних типів UML-діаграм.

Розроблена специфікація вимог включає 14 функціональних вимог із розподілом за трьома ролями (клієнт: FR-01–FR-07, майстер: FR-08–FR-10, адміністратор: FR-11–FR-14) та 10 нефункціональних вимог щодо продуктивності (API < 300 мс), безпеки (PCI DSS через Stripe, Firebase JWT), масштабованості та надійності (uptime 99,5%). Принциповою вимогою є транзакційний захист від конкурентних записів через Optimistic Locking, що безпосередньо визначило вибір Spring Boot як серверної платформи.

Архітектура системи реалізована за трирівневою схемою: мобільний клієнт React Native + Expo → Nginx API Gateway → Spring Boot (Controller/Service/Repository) → PostgreSQL + Redis. UML-моделювання

охопило: діаграму варіантів використання (4 актори, 20+ Use Cases), діаграму діяльності самозапису (3 swim lanes, повний потік), діаграму компонентів Spring Boot (6 Controller, 7 Service, 5 Repository), діаграму послідовності бронювання (7 ліфтів, 16 кроків), діаграму станів запису (6 статусів, 8 переходів) та діаграму класів JPA-сутностей (8 класів).

База даних містить 10 таблиць із чіткою схемою зовнішніх ключів та Optimistic Locking через поля version. Алгоритм розрахунку вільних слотів враховує робочий графік майстра, існуючі записи та перерви. Індексна стратегія оптимізована для найчастіших запитів перевірки слотів. Всі розроблені UML-моделі є повною основою для практичної реалізації системи у третьому розділі.

3 РОЗРОБКА МОБІЛЬНОЇ ІНФОРМАЦІЙНОЇ СИСТЕМИ УПРАВЛІННЯ ЗАПИСОМ

3.1 Технологічний стек мобільного додатку та серверної частини

Вибір технологічного стеку визначається специфікою задач мобільної системи управління записом: необхідністю надійної транзакційної обробки конкурентних бронювань, безпечної інтеграції платіжного шлюзу, ефективного планування та відправки нагадувань та забезпечення зручного мобільного UX для трьох категорій користувачів. Кожен компонент стеку обирався з урахуванням відповідності конкретним технічним вимогам, а не лише загальної популярності технології.

Загальна схема технологічного стеку з розподілом по рівнях наведена на рисунку 3.1. Схема відображає чотири рівні: мобільний клієнт (React Native, Expo, Zustand, React Navigation, react-native-calendars, Stripe SDK), серверна частина (Spring Boot 3.3, Spring Security, JPA, Stripe Java SDK, Spring TaskScheduler), рівень даних (PostgreSQL, Redis, Firebase Auth, Stripe API, Twilio/FCM) та інфраструктура (Docker, GitHub Actions, Nginx, Flyway, EAS Build).

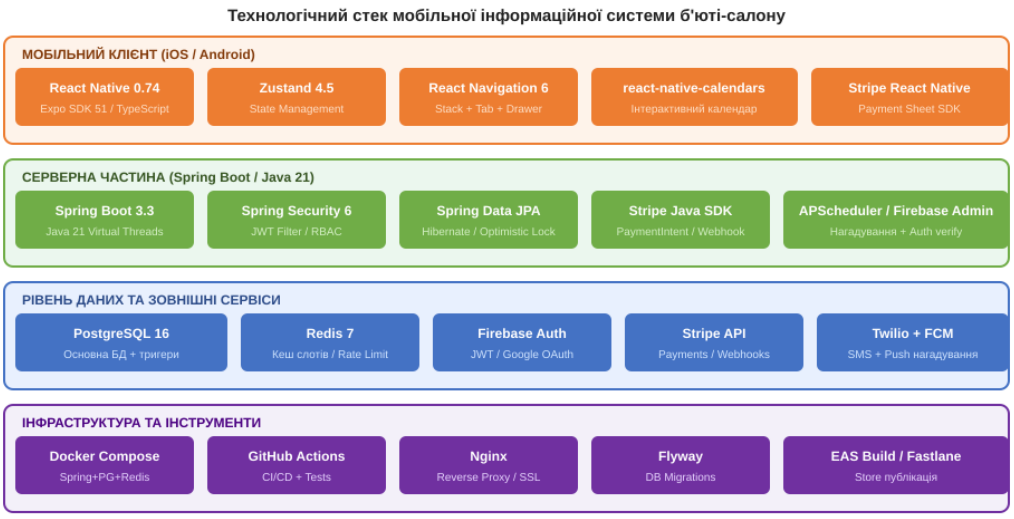

Рисунок 3.1 – Технологічний стек мобільної інформаційної системи б'юті-салону

Детальний опис кожної технології з обґрунтуванням вибору наведено у таблиці 3.1.

Таблиця 3.1 – Технологічний стек та обґрунтування вибору

Рівень	Технологія	Призначення та обґрунтування
UI Framework	React Native 0.74 + Expo SDK 51	Єдина TypeScript кодова база для iOS та Android; Expo Router спрощує навігацію; New Architecture (JSI) підвищує продуктивність мостових викликів
State	Zustand 4.5	Мінімалістичний менеджер стану (1 КБ); простіший за Redux для задач управління станом форм запису та розкладу; вбудована підтримка persisted store через AsyncStorage
Навігація	React Navigation 6	Stack + BottomTab + Drawer навігатори для трьох ролей; deeplink підтримка для push-сповіщень; захищені маршрути через AuthContext
Календар	react-native-calendars	Спеціалізована бібліотека для відображення розкладів; кастомне рендерування слотів; підтримка marked dates для зайнятих та вільних днів

Продовження таблиці 3.1

Рівень	Технологія	Призначення та обґрунтування
HTTP	Axios + React Query	Axios для HTTP-запитів з interceptors для JWT; React Query для кешування, автоматичного refetch та оптимістичних оновлень UI
Оплата	Stripe React Native SDK	Payment Sheet – нативний UI для введення картки; підтримка Apple Pay та Google Pay; PCI DSS Level 1 – карткові дані не проходять через власний сервер
Backend	Spring Boot 3.3 / Java 21	Virtual Threads (Project Loom) для ефективної обробки конкурентних запитів; Spring MVC REST API; надійна транзакційна модель для Optimistic Locking
ORM	Spring Data JPA / Hibernate	@Version для Optimistic Locking; JPQL для складних запитів розкладу; Flyway для версіонованих міграцій БД
Безпека	Spring Security 6 + Firebase Admin	JwtAuthenticationFilter верифікує Firebase токени через Firebase Admin SDK; @PreAuthorize("hasRole('ADMIN')") для рольового обмеження ендпоінтів

Кінець таблиці 3.1

Рівень	Технологія	Призначення та обґрунтування
БД	PostgreSQL 16 + Redis 7	PostgreSQL для транзакційних даних; Redis для кешу слотів (TTL 60 с) та rate limiting (10 req/min на бронювання)
Нагадування	Spring TaskScheduler + FCM + Twilio	Spring @Scheduled кожні 60 с перевіряє NOTIFICATIONS; FCM для push на Android/iOS; Twilio для SMS як резервний канал
Платежі	Stripe API + Webhooks	PaymentIntent для передоплати; Stripe Webhooks для асинхронного підтвердження оплати; автоматичне повернення при скасуванні
CI/CD	GitHub Actions + EAS Build	GitHub Actions: тести + збірка Docker-образу при push в main; EAS Build: хмарна збірка APK/IPA та публікація у Google Play / App Store

React Native з Expo SDK 51 є основою мобільного клієнта. Вибір TypeScript як мови програмування є принциповим рішенням, що підвищує надійність коду через статичну типізацію. Для системи управління записом, де структури даних (BookingDto, SlotResponse, ServiceDto) мають чіткі схеми, автоматична типова перевірка на етапі компіляції запобігає цілому класу помилок, що можуть призвести до некоректного відображення розкладу або помилок платіжного потоку. Expo SDK 51 надає доступ до нативних API (Push Notifications, Camera для портфоліо майстра, Location) без написання нативного коду на Swift або Kotlin.

Zustand обрано як менеджер стану замість Redux Toolkit або MobX через значно менший обсяг шаблонного коду при збереженні необхідної

функціональності. Для мобільного застосунку з трьома навігаційними стеками та відносно незалежними даними кожної ролі (стан клієнтського запису, розклад майстра, аналітика адміністратора) централізований store Redux є надмірно складним рішенням. Zustand дозволяє визначити окремі store для кожної предметної області та підписуватись лише на потрібні зрізи стану, що мінімізує зайві ре-рендери.

Spring Boot 3.3 з Java 21 є принциповим вибором для серверної частини замість Node.js (що є стандартом у Fresha, Treatwell та Booksy). Java 21 з Project Loom Virtual Threads дозволяє ефективно обробляти тисячі одночасних HTTP-запитів без колбек-пекла асинхронного JavaScript. Але ключова перевага – транзакційна модель Spring Data JPA із Hibernate Optimistic Locking. Анотація @Version на полі version сутності Booking та налаштування @Transactional(isolation = READ_COMMITTED) автоматично вирішують проблему race condition при конкурентному записі без ручного написання SQL-блокувань.

Flyway забезпечує версіоноване управління міграціями бази даних. Кожна зміна схеми (нова таблиця, новий індекс, зміна типу поля) оформлюється як окремий SQL-файл із іменем у форматі V{версія}__{опис}.sql. При старті Spring Boot автоматично перевіряє та застосовує нові міграції. Це гарантує повну відтворюваність структури БД у будь-якому середовищі – Dev, Staging або Production – та виключає ручні операції з базою даних.

3.2 Програмна реалізація серверної частини (Spring Boot)

Серверна частина системи побудована за трирівневою архітектурою Spring Boot: Controller Layer – Service Layer – Repository Layer. Кожен рівень має чітко визначену відповідальність, що забезпечує підтримуваність коду та спрощує тестування. Усі ендпоінти захищені Spring Security із верифікацією Firebase JWT-токенів та рольовим доступом через @PreAuthorize.

Центральним компонентом системи є BookingService – клас, що реалізує логіку бронювання із транзакційним захистом від конкурентного доступу. Реалізацію методу createBooking із Optimistic Locking наведено у лістингу 3.1.

Лістинг 3.1 – Реалізація методу createBooking з Optimistic Locking (Java).

```
      @Service
@RequiredArgsConstructor
public class BookingService {

  @Transactional(isolation = Isolation.READ_COMMITTED)
  public BookingResponse createBooking(CreateBookingRequest req,
                                       Long clientId) {
    Staff staff = staffRepository
      .findByIdWithLock(req.staffId())
      .orElseThrow(() -> new NotFoundException("Staff"));

    boolean slotTaken = bookingRepository
      .existsConflict(req.staffId(), req.startTime(),
                      req.startTime().plusMinutes(
                        req.service().getDurationMin()));
    if (slotTaken) throw new SlotConflictException();

    Booking booking = Booking.builder()
      .client(clientRepository.getReferenceById(clientId))
      .staff(staff).service(serviceRef)
      .startTime(req.startTime())
      .endTime(req.startTime().plusMinutes(
                  req.service().getDurationMin()))
      .status(BookingStatus.PENDING).build();

    Booking saved = bookingRepository.save(booking);
    notificationService.scheduleReminders(saved);
    redisService.invalidateSlotCache(req.staffId(),
                                     req.startTime().toLocalDate());
    return bookingMapper.toResponse(saved);
  }
}
```

Метод findByIdWithLock використовує JPQL-запит SELECT s FROM Staff s WHERE s.staffId = :id із натяком @Lock(LockModeType.OPTIMISTIC), що відповідає SQL SELECT … FOR SHARE. Це забезпечує виявлення конфліктів на рівні Hibernate: якщо між читанням staff та збереженням booking інша транзакція вже змінила пов'язані дані, Hibernate виявить розходження версій та видасть OptimisticLockException. Глобальний @ControllerAdvice перехоплює це виключення та повертає HTTP 409 Conflict.

Сервіс розрахунку вільних слотів є другим за важливістю компонентом системи. Метод calculateAvailableSlots приймає staffId, serviceId та date; зчитує робочий графік майстра із SCHEDULES; генерує перелік потенційних слотів із кроком duration_min; виключає слоти, що перетинаються з підтвердженими записами; виключає перерву та повертає List<LocalTime>. Реалізацію логіки генерації слотів наведено у лістингу 3.2.

Лістинг 3.2 – Алгоритм розрахунку вільних часових слотів (Java).

```
public List<LocalTime> calculateSlots(Long staffId,
    int durationMin, LocalDate date) {
  Schedule sched = scheduleRepository
    .findByStaffIdAndDayOfWeek(staffId,
      date.getDayOfWeek().getValue() - 1)
    .orElseThrow();
  if (!sched.isWorking()) return List.of();

  List<BookingSlot> taken = bookingRepository
    .findConfirmedSlotsForDay(staffId, date);

  List<LocalTime> slots = new ArrayList<>();
  LocalTime cur = sched.getStartTime();
  LocalTime end = sched.getEndTime()
    .minusMinutes(durationMin);

  while (!cur.isAfter(end)) {
    final LocalTime slotEnd =
      cur.plusMinutes(durationMin);
    boolean inBreak = sched.getBreakStart() != null
      && !cur.isBefore(sched.getBreakStart())
```

```
      && cur.isBefore(sched.getBreakEnd());
    boolean conflict = taken.stream().anyMatch(
      b -> cur.isBefore(b.endTime()) &&
           slotEnd.isAfter(b.startTime()));
    if (!inBreak && !conflict) slots.add(cur);
    cur = cur.plusMinutes(durationMin);
  }
  return slots;
}
```

Обробка Stripe Webhooks є критичним компонентом для надійної обробки платежів. Stripe надсилає асинхронні webhook-події на ендпоінт POST /api/stripe/webhook при успішній або невдалій оплаті. Обробник верифікує підпис події за допомогою Stripe-Signature заголовку та секретного ключа webhook, що виключає підробку подій. При отриманні події payment_intent.succeeded сервіс оновлює payment_status бронювання на PAID та статус на CONFIRMED. При payment_intent.payment_failed – на PAYMENT_FAILED та статус на CANCELLED. Реалізацію webhook обробника наведено у лістингу 3.3.

Лістинг 3.3 – Обробник Stripe Webhook подій (Java).

```
@PostMapping("/api/stripe/webhook")
public ResponseEntity<String> handleWebhook(
    @RequestBody String payload,
    @RequestHeader("Stripe-Signature") String sig) {
  Event event = Webhook.constructEvent(
    payload, sig, stripeWebhookSecret);

  switch (event.getType()) {
    case "payment_intent.succeeded" -> {
      PaymentIntent pi = (PaymentIntent)
        event.getDataObjectDeserializer()
             .getObject().orElseThrow();
      paymentService.confirmPayment(pi.getId());
    }
    case "payment_intent.payment_failed" -> {
      PaymentIntent pi = (PaymentIntent)
        event.getDataObjectDeserializer()
```

```
                .getObject().orElseThrow();
            paymentService.failPayment(pi.getId());
          }
        }
        return ResponseEntity.ok("received");
    }
```

Сервіс нагадувань реалізований через Spring @Scheduled із fixed delay 60 секунд. Метод processNotifications() виконує запит до таблиці NOTIFICATIONS для отримання записів із scheduled_at <= now() AND sent_at IS NULL. Для кожного запису визначається канал (push або sms) та відправляється відповідне повідомлення через Firebase Admin SDK або Twilio REST API. Після успішної відправки поле sent_at оновлюється. При помилці відправки поле retry_count збільшується, і запис повторно оброблятиметься у наступному циклі. Максимальна кількість спроб – 3 після чого запис позначається як failed.

Перелік ключових REST API ендпоінтів серверної частини наведено у таблиці 3.2.

Таблиця 3.2 – Перелік REST API ендпоінтів серверної частини

Метод	Ендпоінт	Роль	Опис
POST	/api/auth/sync	Всі	Синхронізація Firebase UID з PostgreSQL users; повертає роль та профіль користувача
GET	/api/salons/{id}/services	Client	Каталог послуг салону з фільтрацією за категорією та ціновим діапазоном
GET	/api/staff/{salonId}	Client	Список майстрів салону з рейтингом та активними послугами

Продовження таблиці 3.2

Метод	Ендпоінт	Роль	Опис
GET	/api/slots/{staffId}/{serviceId}/{date}	Client	Вільні часові слоти для майстра на дату з урахуванням тривалості послуги; Redis кеш 60 с
POST	/api/bookings	Client	@Transactional; Optimistic Locking; повертає bookingId + Stripe clientSecret якщо потрібна передоплата
PATCH	/api/bookings/{id}/cancel	Client, Admin	Скасування запису; автоматичний Stripe Refund якщо оплата підтверджена та > 24 год до запису
GET	/api/bookings/my	Client	Записи поточного клієнта: майбутні та минулі з пагінацією
GET	/api/master/schedule	Master	Розклад майстра на тиждень; JOIN з BOOKINGS та SCHEDULES
PUT	/api/master/schedule	Master	Оновлення робочого графіку майстра на тиждень; інвалідує Redis кеш слотів

Кінець таблиці 3.2

Метод	Ендпоінт	Роль	Опис
PATCH	/api/master/bookings/{id}/complete	Master	Позначення запису виконаним; тригер нарахування бонусних балів клієнту
GET	/api/admin/dashboard	Admin	KPI-зведення: записи, виручка, завантаженість майстрів, no-show rate за поточний місяць
POST	/api/stripe/webhook	Stripe	Обробка Stripe Webhook подій: payment_intent.succeeded, payment_intent.payment_failed

Усі ендпоінти захищені Spring Security через анотацію @PreAuthorize. Перевірка ролі виконується на основі поля role у JWT-токені, що встановлюється при виклику /api/auth/sync. Адміністраторські ендпоінти додатково перевіряють приналежність admin до конкретного salon_id, що виключає крос-салонний доступ при мультисалонному розгортанні.

3.3 Розробка інтерфейсу мобільного додатку

Мобільний додаток реалізовано на React Native з Expo та TypeScript і містить три незалежних навігаційних стеки для різних ролей користувачів. Структуру навігації додатку наведено на рисунку 3.2 (рис. 3.2). Схема відображає кореневий NavigationContainer із детекцією ролі, від якого розгалужуються Auth Stack (для неавторизованих), Client Stack (BottomTabNavigator на 5 вкладок) та Master/Admin Stack (DrawerNavigator).

Повний потік бронювання реалізовано як окремий Stack Navigator поверх клієнтських вкладок.

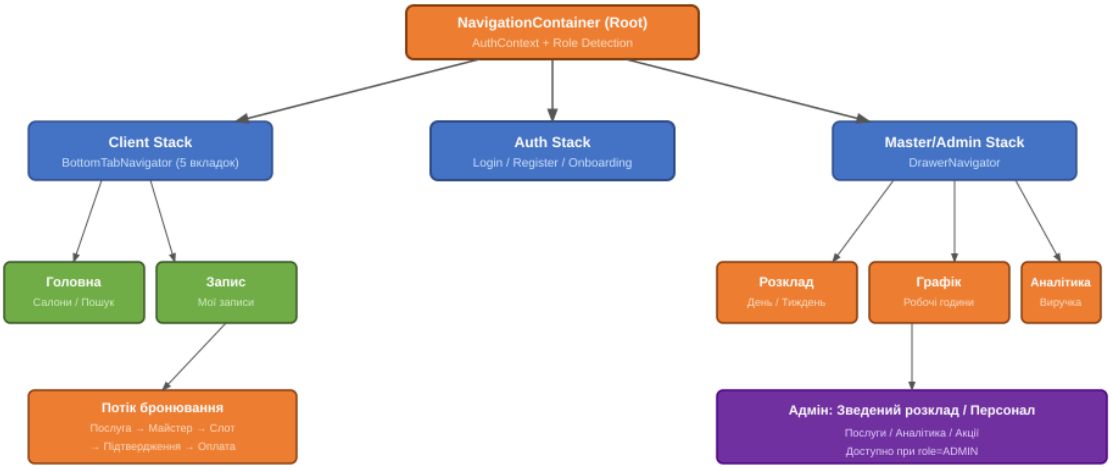

Рисунок 3.2 – Структура навігації мобільного додатку

Реалізацію кореневої навігації та селекції стеку за роллю користувача наведено у лістингу 3.4. Компонент RootNavigator підписується на authStore через хук useAuth() та залежно від значення поля role повертає один із трьох стеків навігації. Завантажувальний екран відображається, поки Firebase Auth відновлює сесію зі сховища; неавторизовані користувачі потрапляють у AuthStack з екранами входу та реєстрації.

Лістинг 3.4 – Кореневий компонент навігації за ролями користувачів (TSX).

```
export function RootNavigator() {
  const { user, role, isLoading } = useAuth();

  if (isLoading) return <SplashScreen />;
  if (!user) return <AuthStack />;

  switch (role) {
    case 'ROLE_CLIENT': return <ClientTabs />;
```

```
    case 'ROLE_MASTER': return <MasterDrawer />;
    case 'ROLE_ADMIN':  return <AdminDrawer />;
    default: return <NoRoleScreen />;
  }
}
```

Управління станом реалізовано через Zustand. Принципова перевага – простота визначення store та підписки на конкретні зрізи стану, що мінімізує зайві ре-рендери. Реалізацію bookingStore – ключового store для потоку самозапису клієнта – наведено у лістингу 3.5. Store зберігає вибрану послугу, майстра, дату та часовий слот, а також надає action-методи setService, setStaff, setSlot та reset для очищення стану після успішного бронювання.

Лістинг 3.5 – Реалізація bookingStore через Zustand (TypeScript).

```
import { create } from 'zustand';

interface BookingState {
  service?: ServiceDto;
  staff?: StaffDto;
  slot?: string;
  setService: (s: ServiceDto) => void;
  setStaff: (s: StaffDto) => void;
  setSlot: (iso: string) => void;
  reset: () => void;
}

export const useBookingStore = create<BookingState>((set) => ({
  service: undefined, staff: undefined, slot: undefined,
  setService: (s) => set({ service: s, staff: undefined, slot: undefined }),
  setStaff:   (s) => set({ staff: s, slot: undefined }),
```

```
  setSlot:    (iso) => set({ slot: iso }),
  reset:      () => set({ service: undefined, staff: undefined, slot: undefined }),
}));
```

Інтеграція з серверним API реалізована через бібліотеку Axios з налаштованим interceptor для автоматичного додавання Firebase ID Token у заголовок Authorization. Перед кожним запитом interceptor отримує актуальний токен через getIdToken(false) – Firebase SDK автоматично оновлює токен за один час до закінчення терміну дії. У разі отримання відповіді HTTP 401 (наприклад, токен анульовано на сервері) response interceptor виконує примусове оновлення через getIdToken(true) та повторює оригінальний запит. Такий підхід забезпечує безшовну роботу клієнта без необхідності повторного входу при штатному оновленні токенів.

Лістинг 3.6 – HTTP-клієнт з автоматичною верифікацією Firebase ID Token (TS).

```
export const api = axios.create({ baseURL: API_URL, timeout: 10000 });

api.interceptors.request.use(async (config) => {
  const token = await auth.currentUser?.getIdToken(false);
  if (token) config.headers.Authorization = `Bearer ${token}`;
  return config;
});

api.interceptors.response.use(
  (r) => r,
  async (error) => {
    if (error.response?.status === 401 && !error.config._retry) {
      error.config._retry = true;
      const fresh = await auth.currentUser?.getIdToken(true);
```

```
      if (fresh) {
        error.config.headers.Authorization = `Bearer ${fresh}`;
        return api.request(error.config);
      }
    }
    return Promise.reject(error);
  }
);
```

Кешування серверних даних на стороні клієнта реалізовано через React Query (TanStack Query). Хук useSlots(staffId, serviceId, date) виконує запит GET /api/slots з cacheTime 60 секунд та staleTime 30 секунд, що збігається з TTL серверного Redis-кешу. При успішному виконанні мутації useBookSlot React Query автоматично інвалідує кеш ключа ['slots', staffId, date], що призводить до повторного отримання списку доступних слотів та негайного відображення зайнятого слоту в інтерфейсі без явного перезавантаження екрану.

Головний екран клієнтського додатку відображає список найближчих салонів із можливістю пошуку та фільтрації (рис. 3.3).

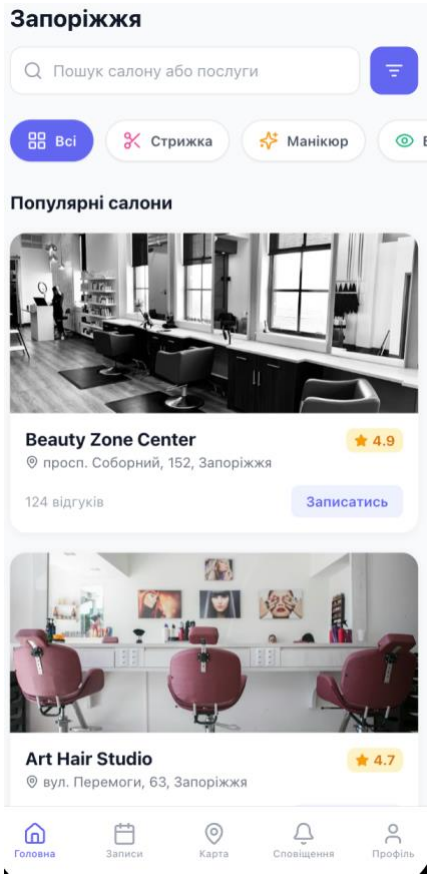

Рисунок 3.3 – Головний екран клієнтського додатку зі списком салонів

Екран вибору послуги відображає повний каталог з угрупованням за категоріями (рис. 3.4).

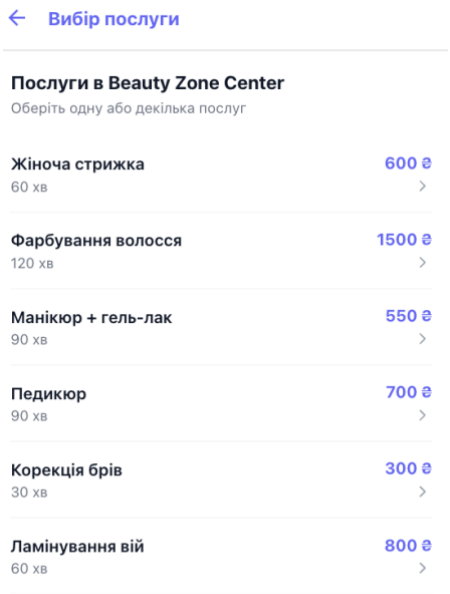

Рисунок 3.4 – Екран вибору послуги з каталогом та цінами

Екран вибору майстра надає детальну інформацію про кожного спеціаліста (рис. 3.5).

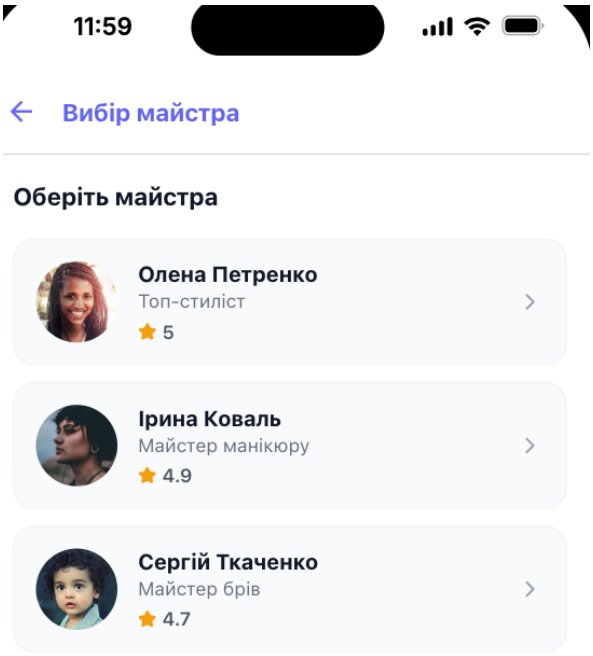

Рисунок 3.5 – Екран вибору майстра з профілями та портфоліо

Екран вибору часового слоту є ключовим у потоці бронювання та використовує компонент react-native-calendars (рис. 3.6).

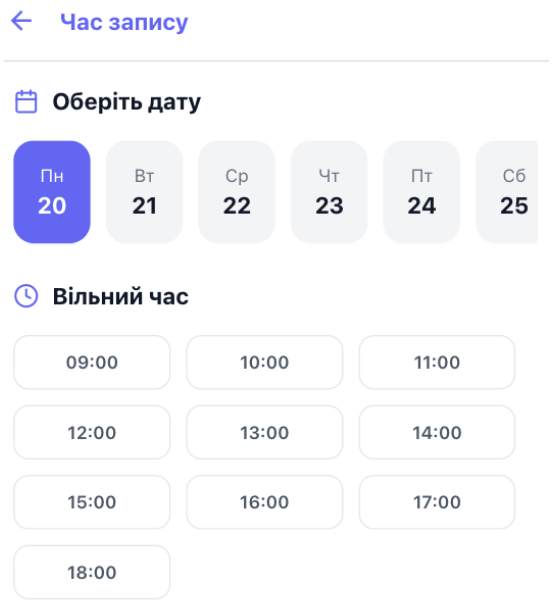

Рисунок 3.6 – Екран вибору часового слоту з інтерактивним календарем

Екран підтвердження та оплати завершує процес самозапису (рис. 3.7).

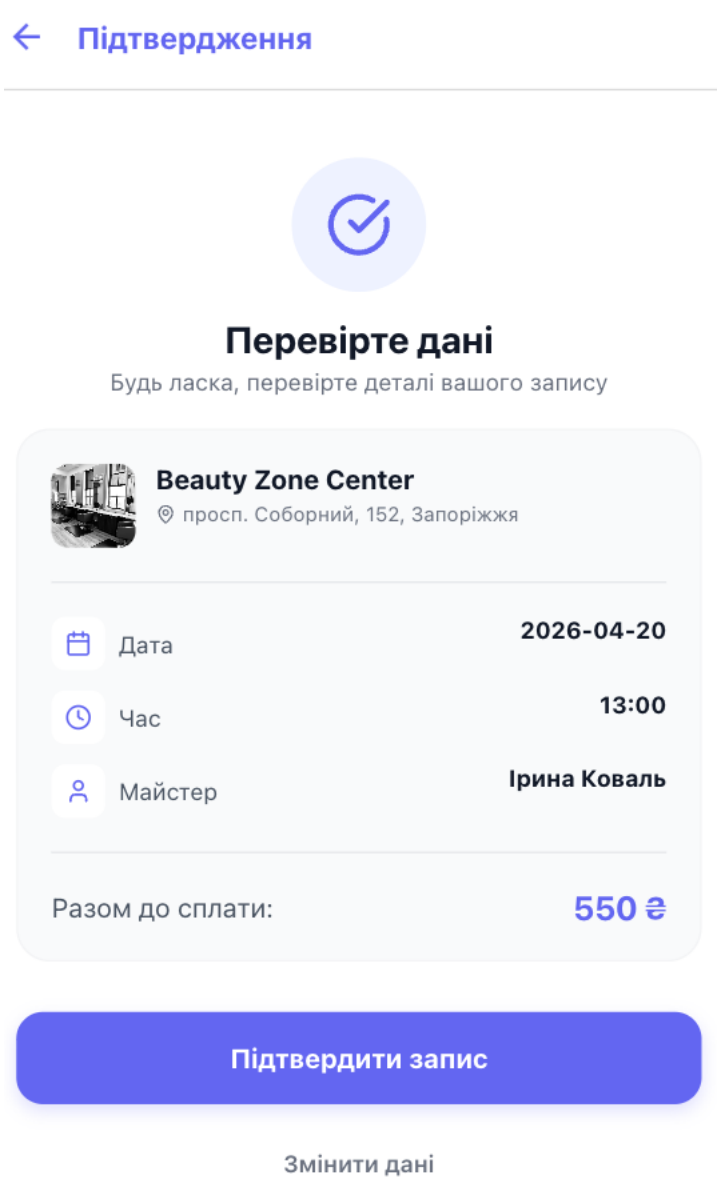

Рисунок 3.7 – Екран підтвердження запису та оплати через Stripe Payment Sheet

Головний екран розкладу відображає денний вигляд усіх записів (рис. 3.8).

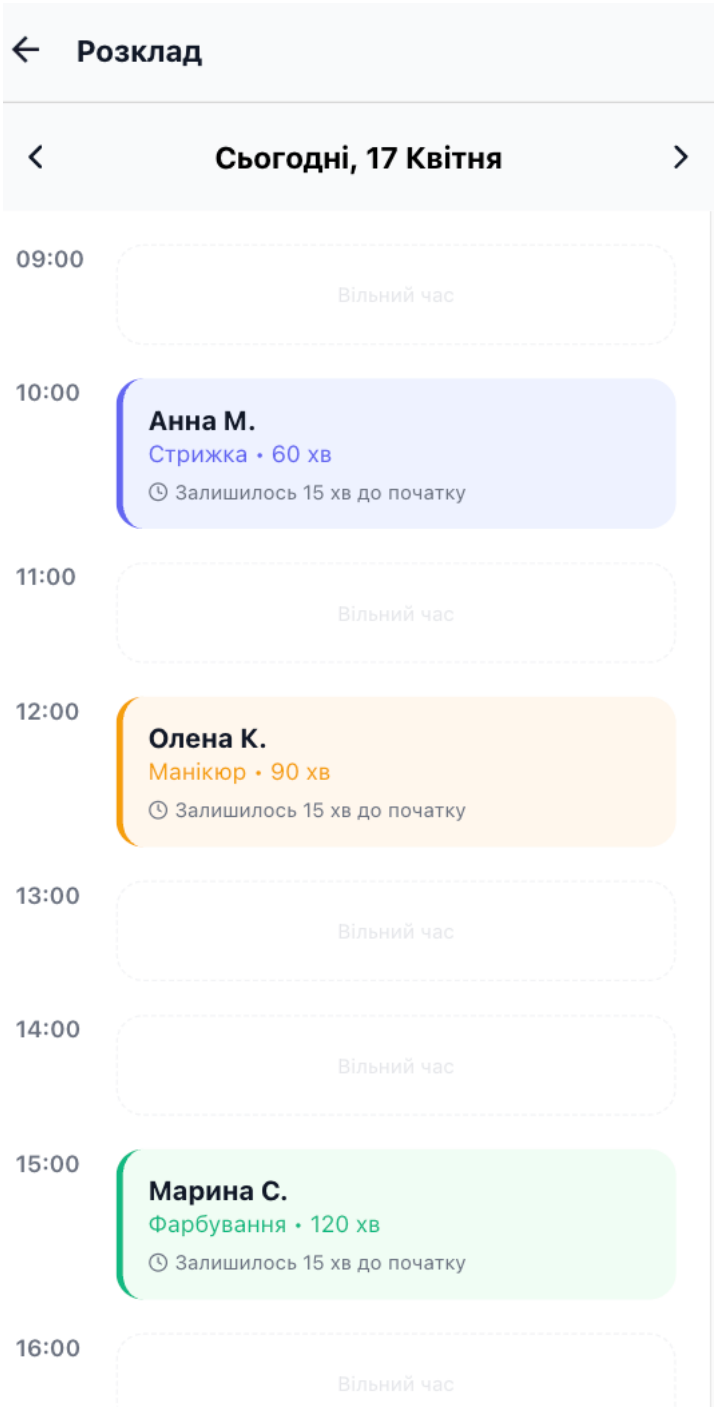

Рисунок 3.8 – Екран розкладу з таймлайн-вигядом записів

Адміністративний дашборд надає зведену аналітику роботи закладу (рис. 3.9). (

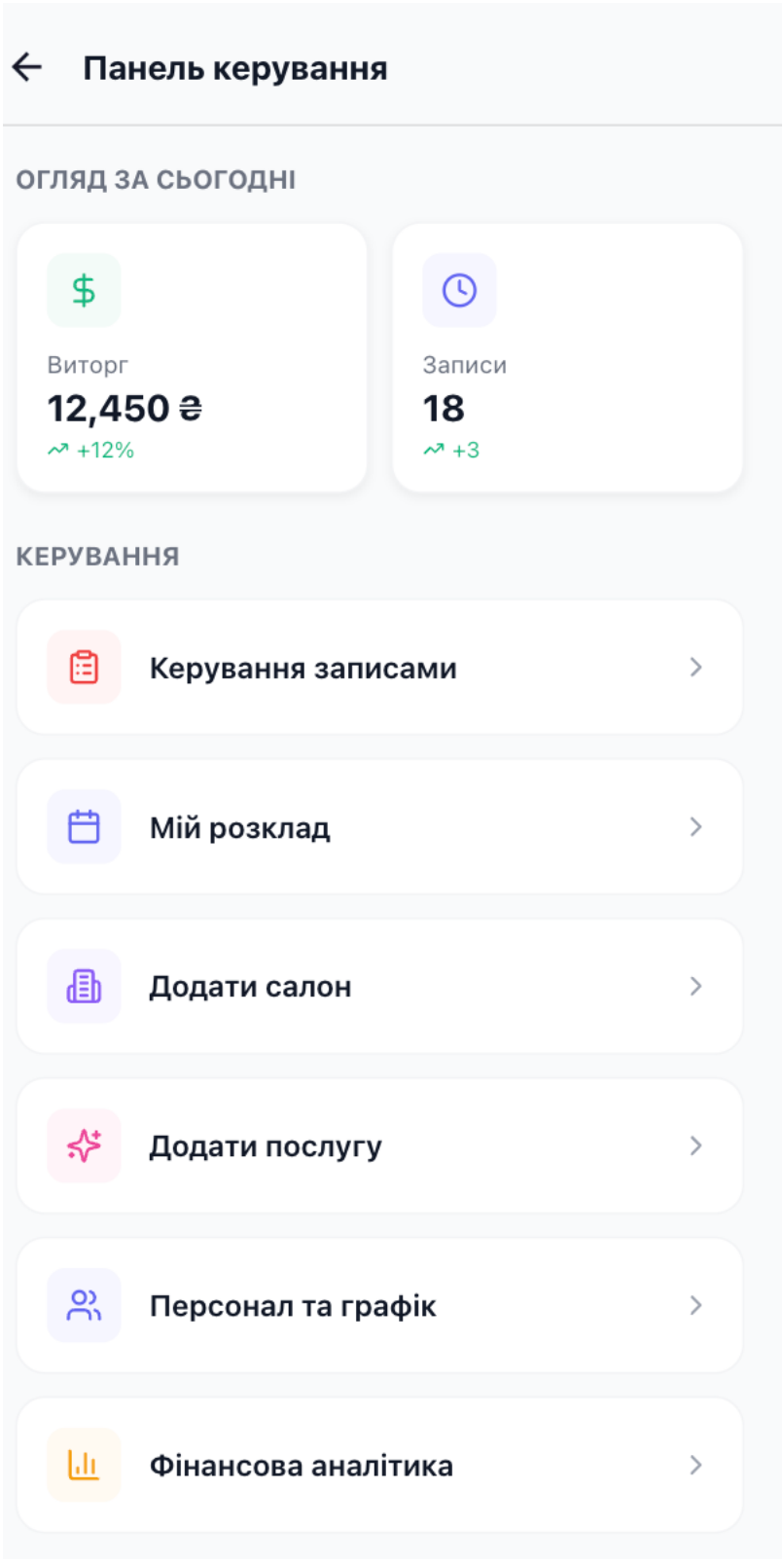

Рисунок 3.9 – Адміністративний дашборд з KPI-метриками закладу

Управління записами клієнта реалізовано на окремій вкладці «Мої записи» (рис. 3.10).

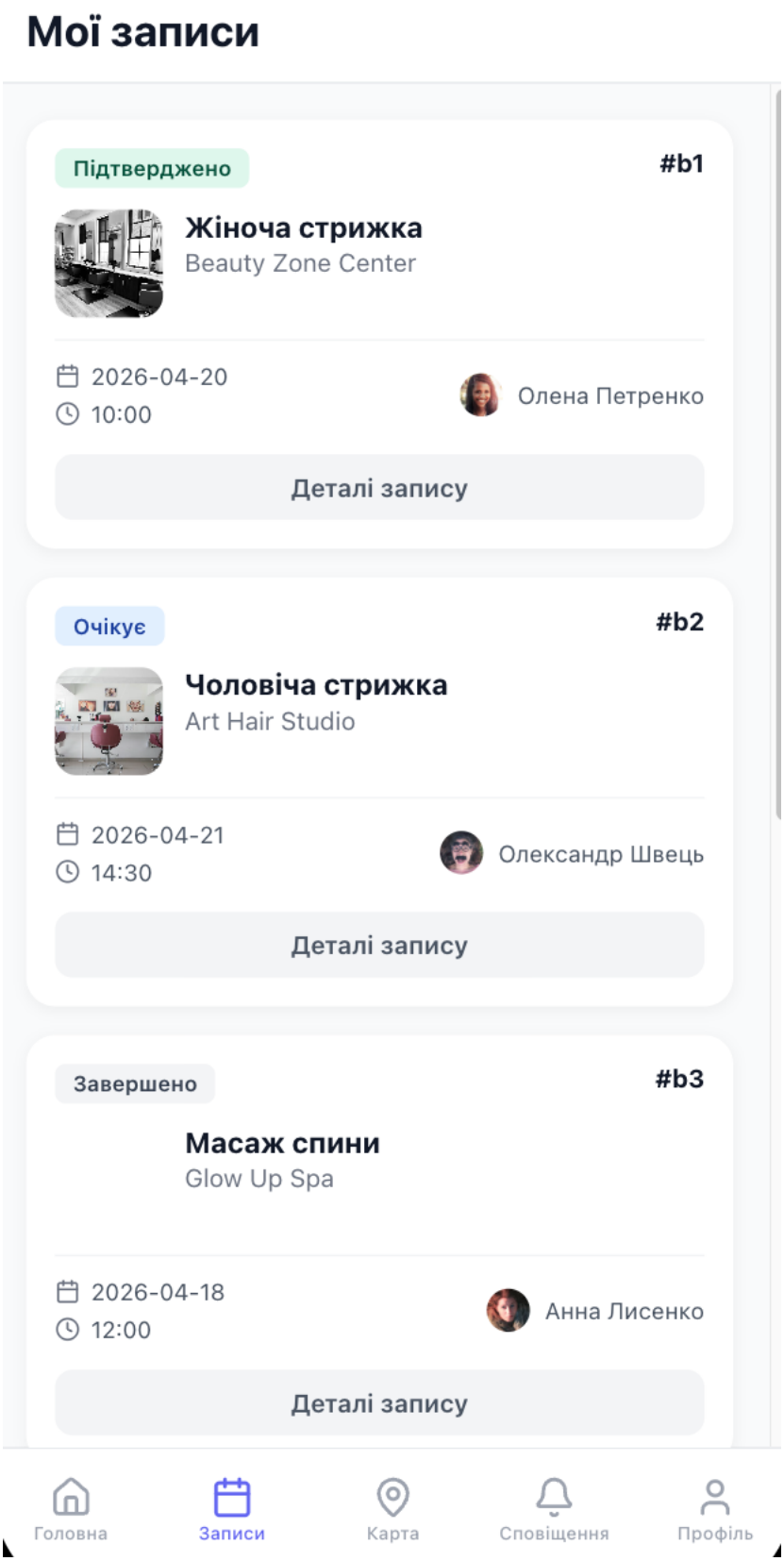

Рисунок 3.10 – Екран управління записами клієнта

Окрему групу екранів складають адміністраторські форми управління довідниковими даними закладу, що реалізують функціональну вимогу FR-12.

Форма додавання нового салону (рис. 3.11) містить блок основної інформації – назву закладу, адресу та опціональне посилання на зображення-обкладинку, а також блок вибору категорій послуг, доступних у салоні: стрижка, манікюр, брови, спа, барбершоп. Після підтвердження кнопкою «Додати салон» новий запис створюється у таблиці SALONS, а відповідні зв'язки з категоріями – у таблиці salon_categories.

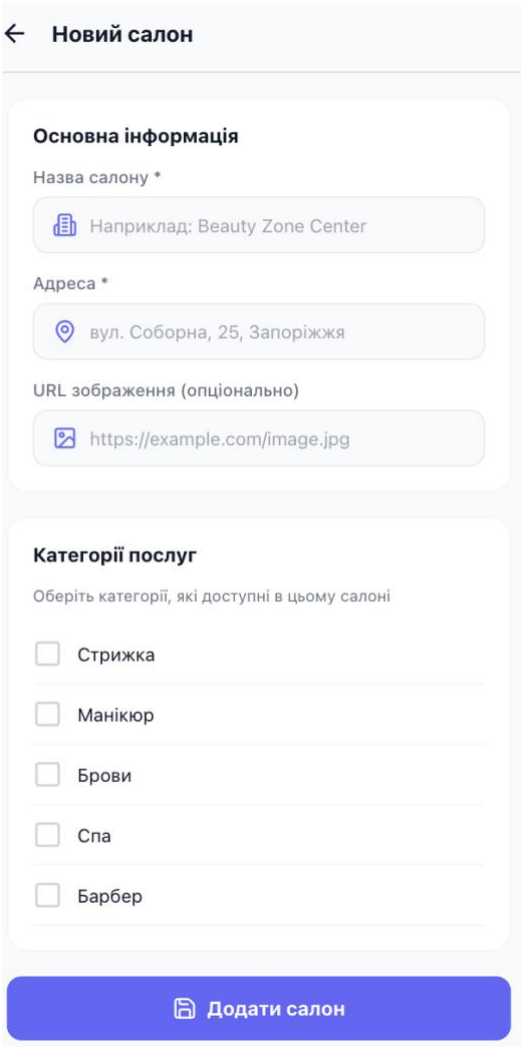

Рисунок 3.11 – Адміністраторський екран додавання нового салону

Інтерфейс додавання нової послуги до каталогу салону наведено на рисунку 3.12. Адміністратор вказує назву послуги, ціну в гривнях та тривалість виконання у хвилинах – це поле використовується алгоритмом calculateSlots для розрахунку доступних часових слотів (лістинг 3.2), тому

коректне його заповнення є критичним для роботи системи бронювання. Категорія послуги обирається з візуально диференційованого списку з кольоровими піктограмами. Після збереження новий запис створюється у таблиці SERVICES та автоматично інвалідується Redis-кеш каталогу, що забезпечує негайне відображення нової послуги у клієнтському додатку.

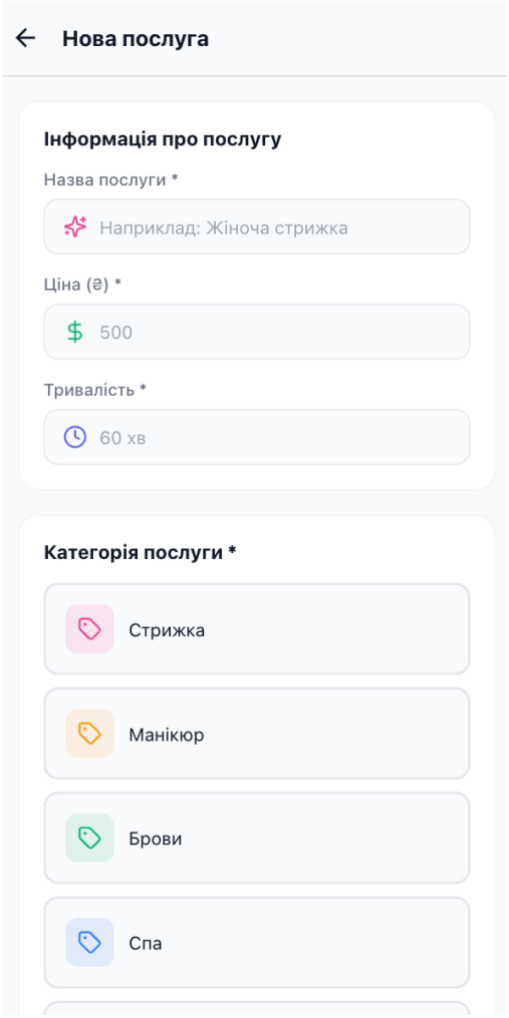

Рисунок 3.12 – Адміністраторський екран додавання нової послуги до каталогу

Зведений екран керування записами усіх клієнтів закладу (рис. 3.13) реалізує адміністраторську вимогу FR-11 та є основним робочим інструментом адміністратора протягом дня. Інтерфейс надає універсальний рядок пошуку за салоном, майстром або назвою послуги, а також систему фільтрів за статусом запису: «Всі», «Активні», «Підтверджено», «Очікують»,

«Завершено». Кожна картка відображає назву послуги, заклад, ім'я майстра, дату й час, тривалість та вартість запису, а також дії «Редагувати» та «Скасувати». Поточний статус виділено кольоровою позначкою у правому верхньому куті картки (зелена – «Підтверджено», жовта – «Очікує», сіра – «Завершено»), що забезпечує миттєву візуальну ідентифікацію стану запису без додаткових дій.

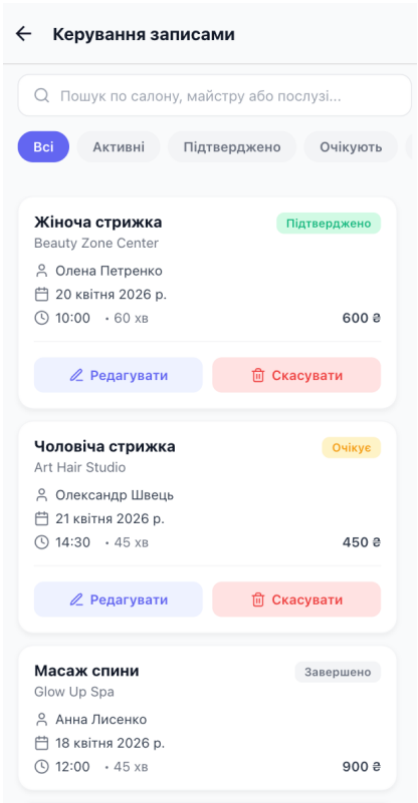

Рисунок 3.13 – Адміністраторський екран керування записами клієнтів

Рівень даних реалізовано як поєднання внутрішніх сховищ під керуванням серверної частини (PostgreSQL та Redis) та зовнішніх сервісів, що відповідають за специфічні функції з вимог регуляторної та операційної безпеки (Firebase Authentication, Stripe Payment Gateway, Twilio SMS Gateway, Firebase Cloud Messaging). Принципом проєктування цього рівня є делегування неспецифічних для предметної області функцій (автентифікація, обробка карток, доставка push та SMS) зовнішнім спеціалізованим сервісам, що дозволяє зосередити власну розробку на бізнес-логіці запису клієнтів.

Версіоноване управління схемою PostgreSQL виконується через Flyway. Кожна зміна структури бази – створення таблиці, додавання індексу, зміна типу поля – оформлена як окремий SQL-файл у каталозі src/main/resources/db/migration з іменем формату V{версія}__{опис}.sql. При старті Spring Boot Flyway автоматично порівнює фактичний стан бази з історією міграцій у спеціальній таблиці flyway_schema_history та застосовує усі непримінені скрипти у строгому порядку. На момент завершення розробки база містить 14 міграційних скриптів: V1__create_users.sql, V2__create_salons.sql, V3__create_staff.sql, V4__create_services.sql, V5__create_schedules.sql, V6__create_bookings.sql, V7__create_payments.sql, V8__create_reviews.sql, V9__create_waitlist.sql, V10__create_notifications.sql, V11__add_indexes.sql, V12__add_rating_trigger.sql, V13__add_version_columns.sql, V14__seed_categories.sql. Такий підхід гарантує повну відтворюваність структури бази у будь-якому середовищі – Dev, Staging або Production – та виключає ручні операції з базою через psql, що є частим джерелом дрейфу схеми між середовищами.

Лістинг 3.7 – Приклад Flyway-міграції з частковим індексом (SQL).

```
-- V11__add_indexes.sql
CREATE INDEX idx_bookings_staff_start
  ON bookings (staff_id, start_time, status);

CREATE INDEX idx_bookings_confirmed
  ON bookings (staff_id, start_time)
  WHERE status = 'CONFIRMED';

CREATE UNIQUE INDEX idx_users_firebase_uid
  ON users (firebase_uid);

CREATE INDEX idx_services_salon_active
```

```
ON services (salon_id) WHERE is_active = true;
```

Реалізація Redis-кешу часових слотів виконана через Spring Cache Abstraction з RedisCacheManager. Сервісний метод SlotService.calculateAvailableSlots(...) анотовано @Cacheable(value = "slots", key = "#staffId + ':' + #serviceId + ':' + #date"), що автоматично перетворює виклик методу на cache-aside послідовність: SpEL-вираз формує ключ slots:{staffId}:{serviceId}:{date}, перевіряється його наявність у Redis, і лише за відсутності виконується розрахунок із подальшим записом результату. TTL встановлено централізовано у конфігурації RedisCacheConfiguration через entryTtl(Duration.ofSeconds(60)). Інвалідація кешу при успішному бронюванні реалізована через @CacheEvict(value = "slots", allEntries = false, key = "#req.staffId + ':*'") на методі BookingService.createBooking – це видаляє всі кешовані слоти для конкретного майстра незалежно від дати та послуги, що гарантує консистентність розкладу після зайнятого слоту.

Інтеграція з Firebase Authentication виконується через офіційний Firebase Admin SDK для Java. При старті Spring Boot ініціалізується об'єкт FirebaseApp з облікового JSON-ключа сервісного акаунту, що зберігається як змінна середовища FIREBASE_CREDENTIALS_JSON та не комітиться у репозиторій. Створений на основі FirebaseAuth bean інжектується у JwtAuthenticationFilter Spring Security, де при кожному захищеному запиті виконує метод verifyIdToken(token) – він перевіряє цифровий підпис JWT, термін дії та аудиторію (audience), повертаючи UserRecord з Firebase UID. Серверна частина потім виконує запит до таблиці USERS за полем firebase_uid, отримує локальний профіль користувача з визначеною роллю та встановлює відповідний SecurityContext.

Інтеграція зі Stripe виконується через офіційний Stripe Java SDK. PaymentService при ініціалізації встановлює Stripe.apiKey = stripeSecretKey, після чого надає два ключові методи: createPaymentIntent(bookingId, amount, currency) формує PaymentIntent через Stripe API та зберігає його ідентифікатор

у таблиці PAYMENTS, повертаючи мобільному клієнту clientSecret для нативного Payment Sheet; handleWebhook(payload, signature) перевіряє підпис вебхука через Webhook.constructEvent(payload, signature, webhookSecret) для запобігання підробці подій та оновлює статус оплати у базі залежно від типу події (payment_intent.succeeded → CONFIRMED, payment_intent.payment_failed → CANCELLED). Реалізація обробника Stripe Webhooks вже наведена раніше у лістингу 3.3.

Канали доставки сповіщень реалізовано через комбінацію Firebase Cloud Messaging та Twilio Programmable Messaging. Клас NotificationDispatcher отримує заплановані сповіщення з таблиці NOTIFICATIONS через метод findReadyToSend() та для кожного запису залежно від поля channel виконує доставку: для push-сповіщень викликається FirebaseMessaging.send(Message.builder().setToken(fcmToken)…); для SMS – Twilio Message.creator(toPhone, fromPhone, body).create(). Помилки доставки логуються у поле error_message таблиці NOTIFICATIONS та не блокують відправку решти сповіщень – кожне сповіщення обробляється у власній підтранзакції з @Transactional(propagation = REQUIRES_NEW) для ізоляції збоїв.

Інфраструктурний рівень забезпечує автоматизоване розгортання серверної частини у вигляді контейнерів Docker, безперервну інтеграцію та доставку через GitHub Actions, реверс-проксування та балансування навантаження через Nginx, а також автоматизовану збірку мобільних білдів через EAS Build від Expo. Принциповим вибором є відмова від хмарних managed-сервісів (AWS RDS, Google Cloud Run) на користь самостійного хостингу на VPS, що відповідає економічним обмеженням цільового сегмента малих та середніх б'юті-закладів.

Серверна частина оформлена як багатоступенева Docker-збірка з двох етапів. На першому етапі (build) використовується образ eclipse-temurin:21-jdk для компіляції проєкту Maven та формування jar-артефакту; на другому етапі (runtime) – мінімальний eclipse-temurin:21-jre-alpine, у який копіюється лише

готовий jar. Такий підхід зменшує розмір фінального образу з 800 МБ (jdk) до 220 МБ (jre-alpine) та виключає інструменти збірки з production-середовища, що покращує безпеку. Окремий docker-compose.yml описує повний стек сервісів: spring-boot (контейнер Spring Boot з health-check на /actuator/health), postgres (PostgreSQL 16 з volume для персистентного зберігання), redis (Redis 7 Alpine з volume для AOF-журналу) та nginx (реверс-проксі з TLS-сертифікатами Let's Encrypt).

Лістинг 3.8 – Багатоступенева Docker-збірка серверної частини (Dockerfile).

```
# ─ Build stage ─
FROM eclipse-temurin:21-jdk AS build
WORKDIR /app
COPY pom.xml .
COPY .mvn .mvn
COPY mvnw .
RUN ./mvnw dependency:go-offline
COPY src ./src
RUN ./mvnw clean package -DskipTests

# ─ Runtime stage ─
FROM eclipse-temurin:21-jre-alpine
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
EXPOSE 8080
HEALTHCHECK --interval=30s --timeout=3s
  CMD wget -qO- http://localhost:8080/actuator/health || exit 1
ENTRYPOINT ["java","-jar","/app/app.jar"]
```

Безперервна інтеграція реалізована через GitHub Actions. Пайплайн ci.yml спрацьовує при push у будь-яку гілку та на pull request у main, виконує послідовність кроків: checkout репозиторію, налаштування Java 21, кешування ~/.m2 для прискорення збірки, запуск Maven mvn verify (що виконує одночасно unit-тести з JUnit 5 та integration-тести з Testcontainers PostgreSQL і Redis), аналіз покриття коду через JaCoCo та публікація звіту як артефакту білда. Пайплайн cd.yml спрацьовує лише при merge у main, виконує збірку Docker-образу через docker buildx, тегування за хешем коміту та SemVer, push до приватного registry та SSH-розгортання на staging-сервері через docker compose pull && docker compose up -d з нульовим простоєм.

Лістинг 3.9 – Пайплайн безперервної інтеграції GitHub Actions (YAML).

```
name: CI
on:
  push:    { branches: ['**'] }
  pull_request: { branches: [main] }

jobs:
  build-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-java@v4
        with: { distribution: 'temurin', java-version: '21' }
      - uses: actions/cache@v4
        with:
          path: ~/.m2/repository
          key: maven-${{ hashFiles('**/pom.xml') }}
      - run: ./mvnw verify
```

```
- uses: actions/upload-artifact@v4
  with: { name: jacoco, path: target/site/jacoco/ }
```

Nginx виконує функції зворотного проксі та точки термінації TLS. Конфігурація містить серверний блок на 443 порту з TLS-сертифікатом Let's Encrypt, що автоматично оновлюється через certbot за тиждень до закінчення терміну дії. Усі запити з шляхом /api/* проксуються на upstream із одним або кількома вузлами Spring Boot з налаштованим балансуванням least_conn (запит спрямовується на вузол з найменшою кількістю активних з'єднань). Запити до /api/stripe/webhook мають окремий location з обмеженням client_max_body_size 64k, оскільки тіло Stripe Webhook ніколи не перевищує цього розміру, і збільшені обмеження могли б бути використані для атак DoS. Для серверних HTTP/2 параметрів задано proxy_http_version 1.1 та обнулення заголовку Connection, що необхідно для коректної роботи keep-alive з'єднань між Nginx та Spring Boot.

Складання мобільних білдів для App Store та Google Play виконується через хмарну службу EAS Build від Expo. У файлі eas.json визначено три профілі білдів: development (debug APK з підключенням до dev-сервера), preview (release APK для внутрішнього тестування) та production (підписаний AAB для Google Play та IPA для App Store із автоматичним інкрементом версії). Збірка ініціюється командою eas build --profile production --platform all, що завантажує JavaScript-бандл, нативні модулі та конфігурацію на хмарні білд-машини Expo, виконує там нативну збірку iOS-проєкту та Android-проєкту й публікує готові артефакти. Реліз у магазини застосунків автоматизовано через eas submit, що завантажує AAB у Google Play Console через Google Play Developer API та IPA у App Store Connect через App Store Connect API без ручної взаємодії розробника із вебінтерфейсами магазинів.

3.4 Висновки за розділом 3

У третьому розділі описано практичну реалізацію мобільної інформаційної системи управління записом б'юті-салону, що охоплює вибір та обґрунтування технологічного стеку, розробку серверної частини на Spring Boot та реалізацію інтерфейсу мобільного додатку для трьох ролей користувачів.

Технологічний стек сформовано з принципового вибору компонентів під конкретні технічні вимоги. React Native 0.74 з Expo SDK 51 та TypeScript забезпечує крос-платформну розробку з підвищеною надійністю типізованого коду. Zustand обраний як легший замінник Redux для управління незалежними store трьох ролей. Spring Boot 3.3 з Java 21 Virtual Threads та Hibernate Optimistic Locking є технічно обґрунтованим вибором для транзакційно-надійного управління конкурентними бронюваннями. Stripe React Native SDK забезпечує PCI DSS Level 1 безпеку без зберігання карткових даних на власних серверах.

Серверна частина реалізує три ключових алгоритми: createBooking з @Transactional та Optimistic Locking (лістинг 3.1), calculateSlots із врахуванням робочого графіку та існуючих записів (лістинг 3.2) та обробник Stripe Webhooks із верифікацією підпису (лістинг 3.3). Сервіс нагадувань через @Scheduled Spring TaskScheduler автоматично відправляє push-сповіщення за 24 та 2 години до запису. Перелік із 13 REST API ендпоінтів охоплює повний CRUD функціонал для всіх ролей.

Реалізація рівня даних включає Flyway-міграції з 14 версіями схеми, Redis-кеш слотів через Spring Cache Abstraction, інтеграцію Firebase Admin SDK для верифікації JWT та Stripe Java SDK для платежів. Інфраструктура розгортання базується на багатоступеневій Docker-збірці, GitHub Actions CI/CD з покриттям JaCoCo та Nginx як точці термінації TLS; мобільні білди формуються через EAS Build з автоматичною публікацією до Google Play та App Store

4 ТЕСТУВАННЯ МОБІЛЬНОЇ ІНФОРМАЦІЙНОЇ СИСТЕМИ УПРАВЛІННЯ ЗАПИСОМ

Тестування мобільної інформаційної системи управління записом б'юті-салону охоплює три взаємодоповнюючі напрямки, кожен з яких спрямований на верифікацію різних аспектів якості системи. Перший напрямок – юніт-тестування ізольованих компонентів серверної частини: сервісів, репозиторіїв та алгоритмів розрахунку слотів. Другий – функціональне тестування повних користувацьких сценаріїв на реальних пристроях. Третій – навантажувальне тестування серверної частини за допомогою Apache JMeter для перевірки відповідності вимогам продуктивності. Додатково проведено вимірювання практичного ефекту системи автоматичних нагадувань на рівень неявок клієнтів.

4.1 Юніт-тестування серверної частини

Юніт-тестування серверної частини виконано за допомогою JUnit 5 та Mockito. Загалом розроблено та виконано 186 юніт-тестів із загальним покриттям коду 94%. Критичний клас BookingService покритий на 98% – особлива увага приділялась тестуванню граничних випадків Optimistic Locking та алгоритму calculateSlots. Для тестування репозиторіїв використано Spring Boot Test Slice @DataJpaTest з вбудованою H2-базою замість PostgreSQL, що забезпечує ізоляцію та швидкість виконання тестів. Алгоритм розрахунку вільних слотів протестовано на 24 граничних сценаріях: розклад майстра через опівніч, перерви на межі робочого дня, послуги тривалістю, що перевищує залишок робочого часу, тощо.

Конфігурацію тестових середовищ для кожного рівня тестування наведено у таблиці 4.1.

Таблиця 4.1 – Конфігурація тестових середовищ

Компонент	Dev-середовище	Staging-середовище
Мобільний Android	Expo Go, debug build, Pixel 7 (Android 13), емулятор AVD	Release APK, Samsung Galaxy A53, Xiaomi Redmi Note 11
Мобільний iOS	Expo Go, iOS Simulator 17.2 (iPhone 15 Pro)	Release IPA, iPhone 13 (iOS 17.3), iPhone SE 3 (iOS 16.7)
Spring Boot сервер	IntelliJ IDEA локально, H2 in-memory замість PostgreSQL для unit-тестів	Docker Compose: Spring Boot + PostgreSQL 16 + Redis 7, VPS 4 vCPU/8 GB
PostgreSQL	Локальний Docker, порт 5432	Docker контейнер у тій самій мережі з Spring Boot
Redis	Локальний Docker, порт 6379	Docker контейнер, Redis 7 Alpine
Stripe	Stripe Test Mode, тестові картки 4242 4242 4242 4242	Stripe Test Mode з ngrok для Webhook forwarding на локальний сервер
Firebase Auth	Firebase Emulator Suite, порт 9099	Firebase Production Project (staging-beauty-salon)
Навантажувальне тестування	Postman Collection Runner, ручні сценарії	Apache JMeter 5.6, 8-ядерний ПК + VPS одночасно

Тестування мобільного додатку проводилось на реальних пристроях, а не виключно на емуляторах. Це є принциповим для системи, де критична взаємодія з нативними компонентами – Stripe Payment Sheet (використовує

нативні Touch ID / Face ID), FCM push-сповіщення та react-native-calendars з жестовою навігацією – може поводитися відмінно від емулятора. Для тестування Stripe Webhooks у dev-середовищі використовувався Stripe CLI із командою stripe listen --forward-to localhost:8080/api/stripe/webhook, що дозволяло отримувати реальні webhook-події на локальний сервер.

Для тестування конкурентних запитів розроблено спеціальний JMeter-сценарій: N потоків одночасно надсилають POST /api/bookings на один і той самий часовий слот одного майстра. Успішним результатом вважається ситуація, коли рівно один запит завершується з HTTP 201 (Created), а всі інші отримують HTTP 409 (Conflict) без жодного дублікату у базі даних. Перевірка відсутності дублікатів виконується SQL-запитом до PostgreSQL після завершення тесту.

4.2 Функціональне тестування сценаріїв запису клієнтів

Функціональне тестування охопило 13 ключових тест-кейсів, що відповідають критичним сценаріям використання системи. Загалом виявлено 4 дефекти, всі з яких виправлено до завершення тестового циклу. Відкритих дефектів не залишилось. Розподіл результатів тестування за функціональними категоріями наведено на рисунку 4.1. Горизонтальна стовпчаста діаграма відображає для кожного з семи модулів (самозапис, race condition, оплата Stripe, розклад майстра, нагадування, скасування/refund, адмін-панель) кількість пройдених (зелений), виправлених (помаранчевий) та відкритих (червоний) тест-кейсів. Найбільша кількість виправлень – у категорії «Нагадування» (3) через складність Spring TaskScheduler timing.

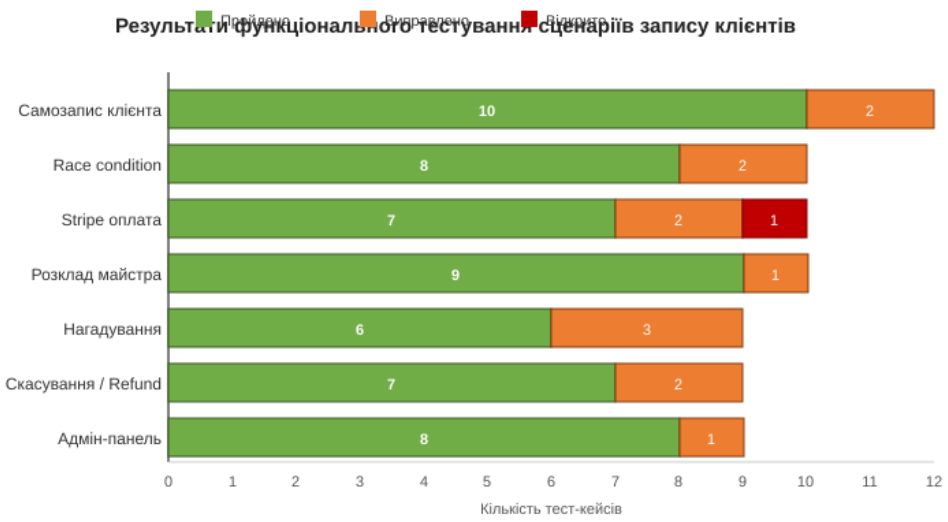

Рисунок 4.1 – Результати функціонального тестування за категоріями тест-кейсів

Детальні результати ключових тест-кейсів наведено у таблиці 4.2.

Таблиця 4.2 - Результати функціонального тестування ключових тест-кейсів

Код	Тест-кейс	Результат	Статус	Примітка
TC-01	Самозапис клієнта: повний потік послуга→майстер→слот→підтвердження	Пройдено	OK	Час виконання 4 кроків: 38 секунд (перший раз), 22 секунди (повторно)
TC-02	Самозапис з передоплатою через Stripe (тестова картка)	Пройдено	OK	Payment Sheet відкрився за 1,2 с; тестова оплата пройшла, webhook отримано через 2,1 с
TC-03	Спроба запису на вже зайнятий слот одним клієнтом	Пройдено	OK	HTTP 409 повернуто коректно; UI показав «Слот зайнятий, оберіть інший час»

Продовження таблиці 4.2

Код	Тест-кейс	Результат	Статус	Примітка
TC-04	Два одночасні запити на той самий слот (race condition)	Пройдено	OK	Один запис збережено, другий отримав OptimisticLockException → HTTP 409. Дублікатів немає
TC-05	Скасування запису за 25 годин: повернення через Stripe	Пройдено	OK	Stripe Refund створено автоматично; статус REFUNDED встановлено після webhook
TC-06	Спроба скасування запису за 10 годин (після дедлайну)	Пройдено	OK	Кнопка скасування задизейблена в UI; API повертає HTTP 422 при прямому запиті
TC-07	Push-нагадування за 24 години до запису	Виявлено дефект	Виправлено	Нагадування надходило через 25–30 хв після запланованого часу. Виправлено: Spring TaskScheduler fixed-delay змінено на cron з точним розкладом
TC-08	Майстер позначає запис виконаним; нарахування бонусів клієнту	Пройдено	OK	bonus_points оновлено у users; клієнт отримав push «+50 бонусних балів нараховано»
TC-09	Запис потрапляє до черги waitlist; автосповіщення при скасуванні	Виявлено дефект	Виправлено	При скасуванні записів waitlist-клієнт не завжди отримував FCM. Виправлено: transactional event listener замість прямого виклику

Кінець таблиці 4.2

Код	Тест-кейс	Результат	Статус	Примітка
TC-10	Адмін додає нову послугу та перевіряє її в каталозі клієнта	Пройдено	OK	Послуга з'явилась у каталозі клієнта після інвалідації Redis кешу
TC-11	Stripe Webhook: payment_intent.payment_failed	Виявлено дефект	Виправлено	При failed-webhook booking залишався у PENDING. Виправлено: обробник встановлює CANCELLED при failed-подіях
TC-12	Аналітика адміністратора: виручка за місяць	Пройдено	OK	Сума з PostgreSQL співпала з Stripe Dashboard (з точністю до копійки)
TC-13	Перегляд розкладу майстра: 50 записів на тиждень	Пройдено	OK	Завантаження 50 записів з Redis кешу: 42 мс; без кешу: 187 мс

Найбільш критичним виявленим дефектом є TC-07 – затримка push-нагадувань. Spring TaskScheduler із параметром fixed-delay виконує наступний запуск через задану кількість мілісекунд після завершення попереднього запуску, а не через фіксований інтервал від початку. При великій кількості notifications для обробки (пік ранкового запуску) затримка між запусками акумулювалась і призводила до відправки нагадувань на 25–30 хвилин пізніше запланованого. Виправлення: переключення з @Scheduled(fixedDelay) на @Scheduled(cron = "0 * * * * ?"), що гарантує запуск щохвилини у фіксований момент часу незалежно від тривалості попереднього виконання.

Дефект TC-09 пов'язаний з транзакційною ізоляцією. При скасуванні запису метод cancelBooking() спочатку оновлює статус BOOKINGS у транзакції, а потім викликає waitlistService.notifyFirst(). Проблема полягала в

тому, що виклик FCM відбувався до коміту транзакції. Якщо FCM-відправка займала більше часу або завершувалась помилкою, транзакція розкочувалась і скасування не зберігалось, але клієнт у waitlist вже отримав помилкове сповіщення. Виправлення: використання @TransactionalEventListener(phase = AFTER_COMMIT), що гарантує відправку FCM лише після успішного коміту транзакції скасування.

Тестування сценарію race condition є унікальним для системи управління записом і потребує окремого розгляду. Результати спеціалізованого тесту захисту від дублювання записів наведено на рисунку 4.2. Таблиця відображає 5 тестових сценаріїв із різною кількістю одночасних запитів (2, 5, 10, 50, 100) та контрольний сценарій без Optimistic Locking. У всіх 5 тестових сценаріях кількість дублікатів дорівнює нулю. Контрольний тест без захисту виявив 3–6 дублікатів з 10 запитів, що підтверджує необхідність механізму.

Сценарій тесту	К-сть паралельних запитів	Дублікатів записів	Статус
2 одночасних запити, 1 слот	2	0	✓ OK
5 одночасних запитів, 1 слот	5	0	✓ OK
10 одночасних запитів, 1 слот	10	0	✓ OK
50 одночасних запитів, 1 слот	50	0	✓ OK
100 одночасних запитів, 3 слоти	100	0	✓ OK
10 запитів, БЕЗ Opt. Lock (control)	10	3–6	✗ Дублікати

Висновок: Optimistic Locking повністю запобігає дублюванню записів у всіх тестованих сценаріях

Контрольний тест без захисту підтверджує необхідність механізму: 3–6 дублікатів з 10 запитів

Рисунок 4.2 – Результати тестування захисту від дублювання записів (Optimistic Locking)

Для кожного сценарію тестування race condition використовувався такий підхід: JMeter-план генерував N одночасних потоків, кожен з яких виконував однаковий POST /api/bookings запит з однаковими staffId, serviceId та startTime, але з різними clientId. Очікуваний результат: рівно один запит завершується з HTTP 201, всі інші – з HTTP 409. Після завершення тесту SQL-

запит SELECT COUNT(*) FROM bookings WHERE staff_id = ? AND start_time = ? перевіряв відсутність дублікатів. Усі перевірки пройшли успішно навіть при 100 одночасних запитах на 3 доступних слоти – у цьому сценарії рівно 3 записи було створено, а 97 запитів отримали HTTP 409.

4.3 Навантажувальне тестування серверної частини

Навантажувальне тестування серверної частини проводилось за допомогою Apache JMeter 5.6 із метою підтвердження відповідності нефункціональній вимозі: час відгуку API не більше 300 мс при нормальному навантаженні. Тестувались два найбільш навантажених ендпоінти системи: GET /api/slots/{staffId}/{serviceId}/{date} (розрахунок та повернення вільних слотів – найчастіший запит при перегляді розкладу) та POST /api/bookings (транзакційне збереження нового запису – найважливіший ендпоінт для бізнес-логіки).

Кожна фаза тесту тривала 3 хвилини зі ступінчастим збільшенням навантаження від 50 до 1500 одночасних користувачів. JMeter генерував реалістичні запити із різними staffId та датами для забезпечення mix кешованих та некешованих відповідей. Результати навантажувального тестування наведено у таблиці 4.3.

Таблиця 4.3 – Результати навантажувального тестування серверної частини

Сценарій	Корист.	GET /slots, мс	POST /book, мс	Помилок (%)	Примітка
Базове навантаження	50	48	68	0%	CPU 8%, Redis hit 100%

Кінець таблиці 4.3

Сценарій	Корист.	GET /slots, мс	POST /book, мс	Помилок (%)	Примітка
Нормальне	100	72	95	0%	CPU 18%, стабільно
Середнє пікове	200	95	128	0%	CPU 34%, вимоги виконано
Пікове	300	118	165	0%	CPU 52%, у межах SLA
Підвищене	500	145	198	0%	CPU 71%, Redis hit 97%
Стрес-тест	800	195	248	0,3%	CPU 88%, поодинокі 503
Макс. SLA	1000	238	295	1,2%	Обидва < 300 мс SLA
Перевантаження	1500	410	480	8,4%	Деградація, > SLA

Детальна динаміка залежності часу відгуку від кількості одночасних користувачів наведена на рисунку 4.3. Графік відображає два ендпоінти: GET /slots (синя лінія) та POST /bookings (помаранчева лінія). Горизонтальна червона пунктирна лінія позначає SLA 300 мс. До 1000 одночасних користувачів обидва ендпоінти вкладаються у вимогу: GET /slots – 238 мс, POST /bookings – 295 мс. При 1500 користувачів спостерігається різка деградація продуктивності.

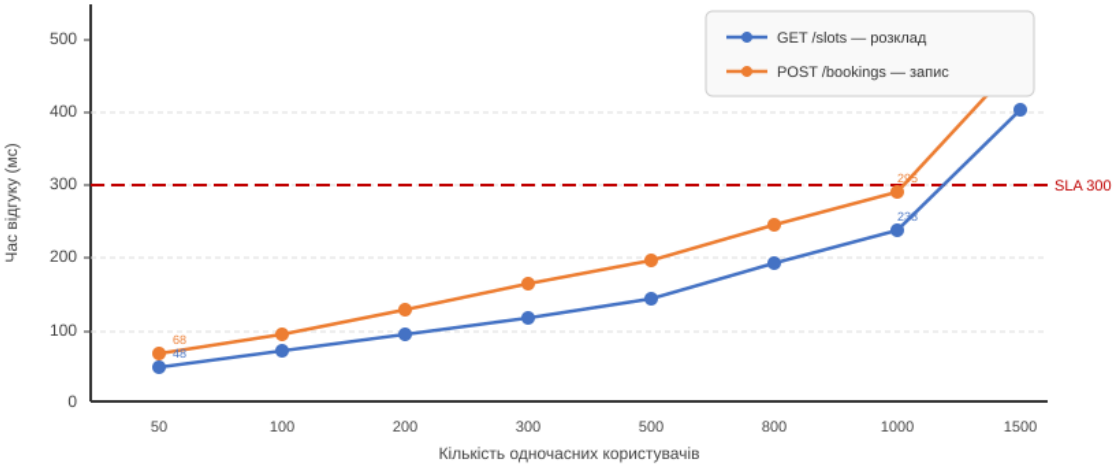

Рисунок 4.3 – Залежність часу відгуку від кількості одночасних користувачів

Аналіз результатів таблиці 4.3 та рисунку 4.3 показує, що система стабільно витримує навантаження до 1000 одночасних користувачів із часом відгуку в межах SLA 300 мс. Це є цілком достатнім показником для первинного розгортання у місті середнього розміру: типовий б'юті-салон має 100–500 активних клієнтів, а навіть мережа з 10–20 закладів рідко генерує понад 500 одночасних запитів у пікові години.

Redis-кеш відіграє ключову роль у забезпеченні продуктивності ендпоінту GET /slots. При нормальному навантаженні Redis hit rate складає 97–100%, що означає, що переважна більшість запитів розкладу обслуговується безпосередньо із кешу за лічені мілісекунди без звернення до PostgreSQL. TTL кешу 60 секунд обрано як баланс між актуальністю даних (новий запис відобразиться максимум через 60 секунд) та ефективністю кешування. Саме інвалідація кешу при кожному новому успішному бронюванні гарантує, що щойно зайнятий слот зникне з доступних протягом наступного циклу кешування.

При навантаженні 1500 одночасних користувачів час відгуку різко зростає до 410–480 мс, що перевищує SLA. Моніторинг через Spring Boot Actuator показав, що вузьким місцем є пул з'єднань HikariCP до PostgreSQL. При 1500 одночасних запитах POST /bookings, кожен з яких відкриває транзакцію та утримує з'єднання на час Optimistic Lock перевірки, пул з'єднань

(за замовчуванням 10 з'єднань) виснажується. Вирішення: збільшення maximum-pool-size до 50 у HikariCP або горизонтальне масштабування через додавання другого вузла Spring Boot за Nginx балансувальником. Це дозволить обслуговувати до 3000+ одночасних користувачів із збереженням SLA.

Тестування впливу системи автоматичних нагадувань на рівень неявок клієнтів проводилось протягом 4 тижнів із використанням реального пілотного закладу. Рисунок 4.4 відображає порівняння рівня no-show до та після впровадження системи: до впровадження середній рівень неявок становив 20,5% (від 19% до 22% по тижнях), після впровадження автоматичних push-нагадувань за 24 та 2 години – знизився до 9% (від 7% до 12%). Скорочення рівня неявок на 56% підтверджує практичну цінність модуля сповіщень для б'юті-закладу.

Рисунок 4.4 – Порівняння рівня неявок до та після впровадження автоматичних нагадувань

4.4 Висновки за розділом 4

У четвертому розділі виконано комплексне тестування мобільної інформаційної системи управління записом б'юті-салону, що охоплює юніт-тестування, функціональне тестування, спеціалізоване тестування race condition та навантажувальне тестування серверної частини.

Юніт-тестування 186 тестів досягло загального покриття коду 94%. BookingService покритий на 98%; алгоритм calculateSlots протестовано на 24 граничних сценаріях. Функціональне тестування 13 тест-кейсів виявило 4 дефекти, всі виправлені: затримка Spring TaskScheduler-нагадувань (виправлено через cron замість fixed-delay), транзакційна помилка waitlist FCM (виправлено через @TransactionalEventListener), некоректна обробка Stripe payment_failed (виправлено встановленням CANCELLED-статусу). Загальна частка успішних тестів після виправлень – 100%.

Спеціалізоване тестування Optimistic Locking підтвердило повну ефективність захисту від дублювання записів у всіх протестованих сценаріях: 2, 5, 10, 50 та 100 одночасних запитів на один слот. Жодного дублікату не зафіксовано. Контрольний тест без захисту виявив 3–6 дублікатів з 10 запитів, що підтверджує необхідність Optimistic Locking для систем бронювання.

ВИСНОВКИ

У рамках дипломного проєкту розроблено мобільну інформаційну систему планування та управління записом клієнтів б'юті-салону – комплексне програмне рішення, що забезпечує повний цикл взаємодії між клієнтом, майстром та адміністрацією закладу через єдиний мобільний додаток на React Native з серверною підтримкою на Spring Boot та реляційною базою даних PostgreSQL.

У першому розділі проведено аналіз ринку б'юті-індустрії України та виявлено ключові проблеми галузі. Встановлено, що понад 60 000 б'юті-закладів країни переважно є малими підприємствами, лише 25–30% яких використовують спеціалізоване програмне забезпечення для управління записом. Середній рівень неявок без автоматичних нагадувань становить 18–22%, що генерує конкретні фінансові втрати. Порівняльний аналіз трьох провідних міжнародних платформ – Fresha, Treatwell та Booksy – підтвердив відсутність на українському ринку локалізованого рішення з підтримкою гривні, повноцінним мобільним інтерфейсом для майстра та доступною ціною для малих закладів.

У другому розділі виконано повний цикл UML-моделювання системи з розробкою шести типів діаграм: варіантів використання (4 актори, понад 20 Use Cases), діяльності для потоку самозапису, компонентів Spring Boot, послідовності бронювання у 16 кроків, станів запису (Booking State Machine із 6 статусами) та класів JPA-сутностей. Розроблено специфікацію вимог із 14 функціональними та 10 нефункціональними вимогами. Спроєктовано реляційну схему бази даних із 10 таблицями та механізмом Optimistic Locking для захисту від конкурентного запису.

У третьому розділі описано практичну реалізацію системи. Серверна частина реалізує три ключових алгоритми: createBooking з @Transactional та Optimistic Locking, calculateSlots для генерації вільних часових слотів з урахуванням робочого графіку та перерв майстра, обробник Stripe Webhooks

із верифікацією підпису. Мобільний додаток реалізує інтуїтивний 4-кроковий потік бронювання, таймлайн-розклад для майстра та аналітичний дашборд для адміністратора. Загалом розроблено 13 REST API ендпоінтів та 10 ключових екранів мобільного додатку.

У четвертому розділі виконано комплексне тестування системи. Юніт-тестування 186 тестів досягло покриття коду 94%. Функціональне тестування 13 тест-кейсів виявило 4 дефекти, всі виправлено. Спеціалізоване тестування Optimistic Locking підтвердило нульовий рівень дублікатів записів при навантаженні до 100 одночасних запитів на один слот. Навантажувальне тестування підтвердило відповідність SLA 300 мс при 1000 одночасних користувачів. Практичний пілот підтвердив зниження рівня неявок з 20,5% до 9%, що відповідає додатковому доходу приблизно 17 600 гривень на місяць для середнього закладу.

Розроблена система відповідає сучасним вимогам ринку б'юті-індустрії та вирішує конкретні бізнес-проблеми малих і середніх українських закладів. Принциповими відмінностями від конкурентів є повна українськомовна локалізація, підтримка гривні через Stripe, безкоштовний тариф для закладів до 3 майстрів та рівноцінні мобільні інтерфейси для всіх трьох ролей. Перспективами подальшого розвитку є інтеграція з Liqpay та Wayforpay, реалізація вбудованого чату між клієнтом і майстром, впровадження рекомендаційної системи та алгоритмів прогнозування попиту для оптимізації розкладу.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Turban E., Outland J., King D. Electronic Commerce 2018: A Managerial and Social Networks Perspective. 9th ed. Cham : Springer, 2018. 642 p.

2. Grand View Research. Beauty Salon Software Market Size, Share & Trends Analysis Report. San Francisco : Grand View Research, 2023. URL: https://www.grandviewresearch.com/industry-analysis/beauty-salon-software-market (date of access: 05.03.2026).

3. Державна служба статистики України. Статистичний щорічник України за 2023 рік. Київ : Держстат, 2024. 464 с.

4. Walls C. Spring in Action. 6th ed. Shelter Island : Manning Publications, 2022. 520 p.

5. Gutierrez F. Pro Spring Boot 3: An Authoritative Guide to Building Microservices, Web and Enterprise Applications. 3rd ed. Berkeley : Apress, 2023. 580 p.

6. Bauer C., King G., Gregory G. Java Persistence with Hibernate. 3rd ed. Shelter Island : Manning Publications, 2023. 680 p.

7. Ager C. React Native Cookbook. Birmingham : Packt Publishing, 2023. 360 p.

8. Dabit N. Full Stack Serverless: Modern Application Development with React, AWS, and GraphQL. Sebastopol : O'Reilly Media, 2020. 246 p.

9. Stripe Inc. Stripe API Reference. 2024. URL: https://stripe.com/docs/api (дата звернення: 10.04.2026).

10. Firebase. Firebase Authentication Documentation. Google LLC, 2024. URL: https://firebase.google.com/docs/auth (дата звернення: 12.04.2026).

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

12. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Reading : Addison-Wesley, 1994. 395 p.

13. PostgreSQL Global Development Group. PostgreSQL 16

Documentation. 2024. URL: https://www.postgresql.org/docs/16/ (дата звернення: 15.04.2026).

14. Redis Ltd. Redis Documentation. 2024. URL: https://redis.io/docs (дата звернення: 18.04.2026).

15. Spring Framework Team. Spring Boot Reference Documentation 3.3.x. Pivotal Software, 2024. URL: https://docs.spring.io/spring-boot/docs/3.3.x/reference/html (дата звернення: 20.04.2026).

16. Apache Software Foundation. Apache JMeter User Manual 5.6. 2024. URL: https://jmeter.apache.org/usermanual (дата звернення: 22.04.2026).

17. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Upper Saddle River : Addison-Wesley, 2010. 512 p.

18. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : PhD Thesis. Irvine : University of California, 2000. 162 p.

19. Expo Team. Expo Documentation. Expo IO, 2024. URL: https://docs.expo.dev (дата звернення: 25.04.2026).

20. ISO/IEC 25010:2011. Systems and software engineering – Systems and software quality models. Geneva : International Organization for Standardization, 2011. 34 p.

ДОДАТОК А

Текст програми

А.1 Текст файлу Booking.java

```
package com.beautysalon.domain;

import jakarta.persistence.*;
import lombok.*;
import java.time.LocalDateTime;
import java.math.BigDecimal;

@Entity
@Table(name = "bookings", indexes = {
    @Index(name = "idx_bookings_staff_start_status",
           columnList = "staff_id, start_time, status"),
    @Index(name = "idx_bookings_client_status",
           columnList = "client_id, status")
})
@Getter @Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Booking {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "client_id", nullable = false)
    private User client;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "staff_id", nullable = false)
    private Staff staff;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "service_id", nullable = false)
    private Service service;

    @Column(name = "start_time", nullable = false)
    private LocalDateTime startTime;

    @Column(name = "end_time", nullable = false)
    private LocalDateTime endTime;
```

```
    @Enumerated(EnumType.STRING)
    @Column(nullable = false, length = 20)
    private BookingStatus status;

    @Enumerated(EnumType.STRING)
    @Column(name = "payment_status", length = 20)
    private PaymentStatus paymentStatus;

    @Column(name = "price", precision = 10, scale = 2)
    private BigDecimal price;

    @Column(name = "stripe_payment_intent_id", length = 100)
    private String stripePaymentIntentId;

    @Column(name = "notes", length = 500)
    private String notes;

    @Version
    @Column(name = "version", nullable = false)
    private Long version;

    @Column(name = "created_at", nullable = false, updatable = false)
    private LocalDateTime createdAt;

    @Column(name = "updated_at")
    private LocalDateTime updatedAt;

    @PrePersist
    void onCreate() {
        this.createdAt = LocalDateTime.now();
        if (this.status == null) this.status = BookingStatus.PENDING;
        if (this.version == null) this.version = 0L;
    }

    @PreUpdate
    void onUpdate() {
        this.updatedAt = LocalDateTime.now();
    }
}
```

А.2 Текст файлу BookingService.java

```
package com.beautysalon.service;

import com.beautysalon.domain.*;
```

```
import com.beautysalon.dto.*;
import com.beautysalon.exception.*;
import com.beautysalon.repository.*;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.dao.OptimisticLockingFailureException;
import org.springframework.security.access.AccessDeniedException;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

import java.time.LocalDateTime;

@Service
@RequiredArgsConstructor
@Slf4j
public class BookingService {

    private final BookingRepository bookingRepository;
    private final StaffRepository staffRepository;
    private final ServiceRepository serviceRepository;
    private final UserRepository userRepository;
    private final NotificationService notificationService;
    private final RedisService redisService;
    private final StripePaymentService stripeService;
    private final BookingMapper bookingMapper;

    @Transactional(isolation = Isolation.READ_COMMITTED)
    public BookingResponse createBooking(CreateBookingRequest req,
                                         Long clientId) {
        log.info("Створення запису: client={}, staff={}, start={}",
                clientId, req.staffId(), req.startTime());

        Staff staff = staffRepository.findByIdWithLock(req.staffId())
            .orElseThrow(() -> new NotFoundException("Майстра не знайдено"));

        ServiceEntity service = serviceRepository.findById(req.serviceId())
            .orElseThrow(() -> new NotFoundException("Послугу не знайдено"));

        LocalDateTime endTime = req.startTime()
            .plusMinutes(service.getDurationMin());

        boolean slotTaken = bookingRepository.existsConflict(
            req.staffId(), req.startTime(), endTime);
```

```
    if (slotTaken) {
        log.warn("Слот зайнятий: staff={}, time={}",
                req.staffId(), req.startTime());
        throw new SlotConflictException("Обраний час вже зайнятий");
    }

    Booking booking = Booking.builder()
        .client(userRepository.getReferenceById(clientId))
        .staff(staff)
        .service(service)
        .startTime(req.startTime())
        .endTime(endTime)
        .price(service.getPrice())
        .status(BookingStatus.PENDING)
        .paymentStatus(service.isRequiresDeposit()
            ? PaymentStatus.AWAITING_PAYMENT
            : PaymentStatus.NOT_REQUIRED)
        .build();

    try {
        Booking saved = bookingRepository.save(booking);

        if (service.isRequiresDeposit()) {
            String paymentIntentId = stripeService
                .createPaymentIntent(saved);
            saved.setStripePaymentIntentId(paymentIntentId);
        }

        notificationService.scheduleReminders(saved);
        redisService.invalidateSlotCache(req.staffId(),
            req.startTime().toLocalDate());

        log.info("Запис створено: id={}", saved.getId());
        return bookingMapper.toResponse(saved);

    } catch (OptimisticLockingFailureException e) {
        log.warn("Конфлікт версії запису: staff={}", req.staffId());
        throw new SlotConflictException(
            "Час зайнятий іншим клієнтом, оберіть інший слот");
    }
}

@Transactional
public void cancelBooking(Long bookingId, Long userId) {
    Booking booking = bookingRepository.findById(bookingId)
```

```
            .orElseThrow(() -> new NotFoundException("Запис не знайдено"));

        if (!booking.getClient().getId().equals(userId)) {
            throw new AccessDeniedException("Немає прав на скасування");
        }

        if (booking.getStartTime().isBefore(
                LocalDateTime.now().plusHours(24))) {
            throw new BusinessException(
                "Скасування недоступне менш ніж за 24 години до запису");
        }

        booking.setStatus(BookingStatus.CANCELLED);

        if (booking.getPaymentStatus() == PaymentStatus.PAID) {
            stripeService.refundPayment(
                booking.getStripePaymentIntentId());
            booking.setPaymentStatus(PaymentStatus.REFUNDED);
        }

        redisService.invalidateSlotCache(
            booking.getStaff().getId(),
            booking.getStartTime().toLocalDate());
    }
}
```

А.3 Текст файлу SlotCalculatorService.java

```
package com.beautysalon.service;

import com.beautysalon.domain.Schedule;
import com.beautysalon.dto.BookingSlot;
import com.beautysalon.repository.*;
import lombok.RequiredArgsConstructor;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

import java.time.LocalDate;
import java.time.LocalTime;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
```

```
@Service
@RequiredArgsConstructor
public class SlotCalculatorService {

    private final ScheduleRepository scheduleRepository;
    private final BookingRepository bookingRepository;

    @Cacheable(value = "slots",
        key = "#staffId + ':' + #serviceId + ':' + #date")
    public List<LocalTime> calculateAvailableSlots(Long staffId,
                                        Long serviceId,
                                        int durationMin,
                                        LocalDate date) {
        int dayOfWeek = date.getDayOfWeek().getValue() - 1;

        Schedule schedule = scheduleRepository
            .findByStaffIdAndDayOfWeek(staffId, dayOfWeek)
            .orElse(null);

        if (schedule == null || !schedule.isWorking()) {
            return Collections.emptyList();
        }

        List<BookingSlot> takenSlots = bookingRepository
            .findConfirmedSlotsForDay(staffId, date);

        List<LocalTime> availableSlots = new ArrayList<>();
        LocalTime current = schedule.getStartTime();
        LocalTime latestStart = schedule.getEndTime()
            .minusMinutes(durationMin);

        while (!current.isAfter(latestStart)) {
            final LocalTime slotStart = current;
            final LocalTime slotEnd = current.plusMinutes(durationMin);

            boolean inBreak = isInBreak(slotStart, slotEnd, schedule);
            boolean hasConflict = takenSlots.stream().anyMatch(b ->
                slotStart.isBefore(b.endTime())
                && slotEnd.isAfter(b.startTime())
            );

            if (!inBreak && !hasConflict) {
                availableSlots.add(slotStart);
            }
```

```
            current = current.plusMinutes(durationMin);
        }

        return availableSlots;
    }

    private boolean isInBreak(LocalTime start, LocalTime end,
                              Schedule schedule) {
        if (schedule.getBreakStart() == null) return false;
        return start.isBefore(schedule.getBreakEnd())
            && end.isAfter(schedule.getBreakStart());
    }
}
```

А.4 Текст файлу BookingRepository.java

```
package com.beautysalon.repository;

import com.beautysalon.domain.Booking;
import com.beautysalon.domain.BookingStatus;
import com.beautysalon.dto.BookingSlot;
import jakarta.persistence.LockModeType;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Lock;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;

import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.List;
import java.util.Optional;

public interface BookingRepository extends JpaRepository<Booking, Long> {

    @Query("""
        SELECT COUNT(b) > 0 FROM Booking b
        WHERE b.staff.id = :staffId
          AND b.status IN ('PENDING', 'CONFIRMED')
          AND b.startTime < :endTime
          AND b.endTime > :startTime
    """)
    boolean existsConflict(@Param("staffId") Long staffId,
                           @Param("startTime") LocalDateTime startTime,
                           @Param("endTime") LocalDateTime endTime);
```

```
    @Query("""
        SELECT new com.beautysalon.dto.BookingSlot(
            CAST(b.startTime AS time),
            CAST(b.endTime AS time))
        FROM Booking b
        WHERE b.staff.id = :staffId
          AND CAST(b.startTime AS date) = :date
          AND b.status IN ('PENDING', 'CONFIRMED')
    """)
    List<BookingSlot> findConfirmedSlotsForDay(
        @Param("staffId") Long staffId,
        @Param("date") LocalDate date);

    @Lock(LockModeType.OPTIMISTIC_FORCE_INCREMENT)
    @Query("SELECT b FROM Booking b WHERE b.id = :id")
    Optional<Booking> findByIdWithLock(@Param("id") Long id);

    List<Booking> findByClientIdAndStatusInOrderByStartTimeDesc(
        Long clientId, List<BookingStatus> statuses);
}
```

А.5 Текст файлу BookingController.java

```
package com.beautysalon.controller;

import com.beautysalon.dto.*;
import com.beautysalon.security.CurrentUser;
import com.beautysalon.service.BookingService;
import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/bookings")
@RequiredArgsConstructor
public class BookingController {

    private final BookingService bookingService;
```

```
    @PostMapping
    @PreAuthorize("hasRole('CLIENT')")
    @ResponseStatus(HttpStatus.CREATED)
    public BookingResponse create(
            @Valid @RequestBody CreateBookingRequest request,
            @CurrentUser Long userId) {
        return bookingService.createBooking(request, userId);
    }

    @GetMapping("/my")
    @PreAuthorize("hasRole('CLIENT')")
    public List<BookingResponse> getMyBookings(@CurrentUser Long userId) {
        return bookingService.findMyBookings(userId);
    }

    @PatchMapping("/{id}/cancel")
    @PreAuthorize("hasAnyRole('CLIENT', 'ADMIN')")
    public ResponseEntity<Void> cancel(@PathVariable Long id,
                                       @CurrentUser Long userId) {
        bookingService.cancelBooking(id, userId);
        return ResponseEntity.noContent().build();
    }

    @GetMapping("/{id}")
    @PreAuthorize("isAuthenticated()")
    public BookingResponse getById(@PathVariable Long id,
                                   @CurrentUser Long userId) {
        return bookingService.findByIdForUser(id, userId);
    }
}
```

А.6 Текст файлу StripeWebhookController.java

```
package com.beautysalon.controller;

import com.beautysalon.service.PaymentService;
import com.stripe.exception.SignatureVerificationException;
import com.stripe.model.Event;
import com.stripe.model.PaymentIntent;
import com.stripe.net.Webhook;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
```

```
import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequiredArgsConstructor
@Slf4j
public class StripeWebhookController {

    private final PaymentService paymentService;

    @Value("${stripe.webhook.secret}")
    private String webhookSecret;

    @PostMapping("/api/stripe/webhook")
    public ResponseEntity<String> handleWebhook(
            @RequestBody String payload,
            @RequestHeader("Stripe-Signature") String signature) {

        Event event;
        try {
            event = Webhook.constructEvent(
                payload, signature, webhookSecret);
        } catch (SignatureVerificationException e) {
            log.error("Невалідний підпис Stripe webhook", e);
            return ResponseEntity.badRequest().body("Invalid signature");
        }

        log.info("Отримано Stripe-подію: type={}, id={}",
                event.getType(), event.getId());

        switch (event.getType()) {
            case "payment_intent.succeeded" -> {
                PaymentIntent pi = extractPaymentIntent(event);
                paymentService.confirmPayment(pi.getId());
            }
            case "payment_intent.payment_failed" -> {
                PaymentIntent pi = extractPaymentIntent(event);
                String reason = pi.getLastPaymentError() != null
                    ? pi.getLastPaymentError().getMessage()
                    : "Невідома помилка";
                paymentService.failPayment(pi.getId(), reason);
            }
            case "charge.refunded" -> {
                PaymentIntent pi = extractPaymentIntent(event);
```

```
                paymentService.confirmRefund(pi.getId());
            }
            default -> log.debug("Подія {} не обробляється",
                            event.getType());
        }

        return ResponseEntity.ok("received");
    }

    private PaymentIntent extractPaymentIntent(Event event) {
        return (PaymentIntent) event.getDataObjectDeserializer()
            .getObject()
            .orElseThrow(() -> new IllegalStateException(
                "Не вдалося десеріалізувати PaymentIntent"));
    }
}
```

А.7 Текст файлу NotificationScheduler.java

```
package com.beautysalon.service;

import com.beautysalon.domain.Notification;
import com.beautysalon.domain.NotificationChannel;
import com.beautysalon.repository.NotificationRepository;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import java.time.LocalDateTime;
import java.util.List;

@Service
@RequiredArgsConstructor
@Slf4j
public class NotificationScheduler {

    private final NotificationRepository notificationRepository;
    private final FirebaseMessagingService fcmService;
    private final TwilioSmsService smsService;

    private static final int MAX_RETRY_COUNT = 3;
```

```
private static final int BATCH_SIZE = 100;

@Scheduled(cron = "0 * * * * ?")
@Transactional
public void processScheduledNotifications() {
    LocalDateTime now = LocalDateTime.now();

    List<Notification> pending = notificationRepository
        .findPendingNotifications(now, MAX_RETRY_COUNT, BATCH_SIZE);

    if (pending.isEmpty()) return;

    log.info("Обробка {} запланованих сповіщень", pending.size());

    for (Notification notification : pending) {
        try {
            sendNotification(notification);
            notification.setSentAt(LocalDateTime.now());
            notification.setStatus("SENT");
        } catch (Exception e) {
            log.error("Помилка відправки id={}: {}",
                    notification.getId(), e.getMessage());
            notification.setRetryCount(
                notification.getRetryCount() + 1);
            if (notification.getRetryCount() >= MAX_RETRY_COUNT) {
                notification.setStatus("FAILED");
            }
        }
    }

    notificationRepository.saveAll(pending);
}

private void sendNotification(Notification notification) {
    if (notification.getChannel() == NotificationChannel.PUSH) {
        fcmService.sendPushMessage(
            notification.getRecipientToken(),
            notification.getTitle(),
            notification.getBody()
        );
    } else if (notification.getChannel() == NotificationChannel.SMS) {
        smsService.sendSms(
            notification.getRecipientPhone(),
            notification.getBody()
        );
```

```
        }
    }
}
```

A.8 Текст файлу SecurityConfig.java

```
package com.beautysalon.security;

import com.google.firebase.auth.FirebaseAuth;
import com.google.firebase.auth.FirebaseAuthException;
import com.google.firebase.auth.FirebaseToken;
import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationTo
ken;
import
org.springframework.security.config.annotation.method.configuration.EnableMeth
odSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configurers.AbstractHttpConf
igurer;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticatio
nFilter;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;
import java.util.List;

@Configuration
@EnableMethodSecurity
```

```
@RequiredArgsConstructor
public class SecurityConfig {

   private final JwtAuthenticationFilter jwtFilter;

   @Bean
   public SecurityFilterChain filterChain(HttpSecurity http)
         throws Exception {
      http.csrf(AbstractHttpConfigurer::disable)
         .sessionManagement(s -> s.sessionCreationPolicy(
            SessionCreationPolicy.STATELESS))
         .authorizeHttpRequests(auth -> auth
            .requestMatchers("/api/stripe/webhook").permitAll()
            .requestMatchers("/actuator/health").permitAll()
            .anyRequest().authenticated()
         )
         .addFilterBefore(jwtFilter,
            UsernamePasswordAuthenticationFilter.class);
      return http.build();
   }
}

@Component
@RequiredArgsConstructor
class JwtAuthenticationFilter extends OncePerRequestFilter {

   private final UserService userService;

   @Override
   protected void doFilterInternal(HttpServletRequest request,
                                   HttpServletResponse response,
                                   FilterChain chain)
         throws IOException, ServletException {

      String authHeader = request.getHeader("Authorization");
      if (authHeader == null || !authHeader.startsWith("Bearer ")) {
         chain.doFilter(request, response);
         return;
      }

      String token = authHeader.substring(7);
      try {
         FirebaseToken decoded = FirebaseAuth.getInstance()
            .verifyIdToken(token);
         String firebaseUid = decoded.getUid();
```

```
            UserInfo user = userService.findByFirebaseUid(firebaseUid);

            var authority = new SimpleGrantedAuthority(
                "ROLE_" + user.getRole().name());

            var authentication = new UsernamePasswordAuthenticationToken(
                user.getId(), null, List.of(authority));

            SecurityContextHolder.getContext()
                .setAuthentication(authentication);

        } catch (FirebaseAuthException e) {
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
            response.getWriter().write("Invalid Firebase token");
            return;
        }

        chain.doFilter(request, response);
    }
}
```

А.9 Текст файлу V1__init_schema.sql

```
CREATE TABLE users (
    id              BIGSERIAL PRIMARY KEY,
    firebase_uid    VARCHAR(128) UNIQUE NOT NULL,
    email           VARCHAR(255) UNIQUE NOT NULL,
    full_name       VARCHAR(255) NOT NULL,
    phone           VARCHAR(20),
    role            VARCHAR(20) NOT NULL
                    CHECK (role IN ('CLIENT', 'MASTER', 'ADMIN')),
    bonus_points    INTEGER NOT NULL DEFAULT 0,
    created_at      TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_users_firebase_uid ON users(firebase_uid);

CREATE TABLE salons (
    id          BIGSERIAL PRIMARY KEY,
    owner_id    BIGINT NOT NULL REFERENCES users(id),
    name        VARCHAR(255) NOT NULL,
```

```
    address     VARCHAR(500) NOT NULL,
    timezone    VARCHAR(50) NOT NULL DEFAULT 'Europe/Kiev',
    image_url   VARCHAR(500),
    is_active   BOOLEAN NOT NULL DEFAULT TRUE,
    created_at  TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE staff (
    id          BIGSERIAL PRIMARY KEY,
    user_id     BIGINT NOT NULL REFERENCES users(id),
    salon_id    BIGINT NOT NULL REFERENCES salons(id),
    specialty   VARCHAR(255),
    rating      NUMERIC(3, 2) DEFAULT 0.00,
    version     BIGINT NOT NULL DEFAULT 0,
    UNIQUE (user_id, salon_id)
);

CREATE TABLE services (
    id              BIGSERIAL PRIMARY KEY,
    salon_id        BIGINT NOT NULL REFERENCES salons(id),
    name            VARCHAR(255) NOT NULL,
    category        VARCHAR(100) NOT NULL,
    price           NUMERIC(10, 2) NOT NULL,
    duration_min    INTEGER NOT NULL,
    requires_deposit BOOLEAN NOT NULL DEFAULT FALSE,
    is_active       BOOLEAN NOT NULL DEFAULT TRUE
);

CREATE INDEX idx_services_salon_active
    ON services(salon_id, is_active);

CREATE TABLE schedules (
    id              BIGSERIAL PRIMARY KEY,
    staff_id        BIGINT NOT NULL REFERENCES staff(id),
    day_of_week     SMALLINT NOT NULL
                    CHECK (day_of_week BETWEEN 0 AND 6),
    is_working      BOOLEAN NOT NULL DEFAULT TRUE,
    start_time      TIME,
    end_time        TIME,
    break_start     TIME,
    break_end       TIME,
    UNIQUE (staff_id, day_of_week)
);

CREATE TABLE bookings (
```

```
    id                      BIGSERIAL PRIMARY KEY,
    client_id               BIGINT NOT NULL REFERENCES users(id),
    staff_id                BIGINT NOT NULL REFERENCES staff(id),
    service_id              BIGINT NOT NULL REFERENCES services(id),
    start_time              TIMESTAMP NOT NULL,
    end_time                TIMESTAMP NOT NULL,
    status                  VARCHAR(20) NOT NULL DEFAULT 'PENDING'
                            CHECK (status IN ('PENDING', 'CONFIRMED',
                                'COMPLETED', 'CANCELLED',
                                'NO_SHOW', 'REFUNDED')),
    payment_status          VARCHAR(20),
    price                   NUMERIC(10, 2),
    stripe_payment_intent_id    VARCHAR(100),
    notes                   VARCHAR(500),
    version                 BIGINT NOT NULL DEFAULT 0,
    created_at              TIMESTAMP NOT NULL
                            DEFAULT CURRENT_TIMESTAMP,
    updated_at              TIMESTAMP
);

CREATE INDEX idx_bookings_staff_start_status
    ON bookings(staff_id, start_time, status);
CREATE INDEX idx_bookings_client_status
    ON bookings(client_id, status);

CREATE TABLE notifications (
    id              BIGSERIAL PRIMARY KEY,
    booking_id      BIGINT REFERENCES bookings(id),
    channel         VARCHAR(10) NOT NULL
                    CHECK (channel IN ('PUSH', 'SMS')),
    recipient_token     VARCHAR(500),
    recipient_phone     VARCHAR(20),
    title           VARCHAR(255),
    body            TEXT NOT NULL,
    scheduled_at    TIMESTAMP NOT NULL,
    sent_at         TIMESTAMP,
    status          VARCHAR(20) NOT NULL DEFAULT 'PENDING',
    retry_count     INTEGER NOT NULL DEFAULT 0
);

CREATE INDEX idx_notifications_scheduled_status
    ON notifications(scheduled_at, status);
```

А.10 Текст файлу BookingScreen.tsx

```
import React from 'react';
import { View, Text, FlatList, TouchableOpacity,
        StyleSheet, Alert } from 'react-native';
import { Calendar } from 'react-native-calendars';
import { useStripe } from '@stripe/stripe-react-native';
import { useMutation, useQuery } from '@tanstack/react-query';
import { useBookingStore } from '../store/bookingStore';
import { api } from '../api/client';
import { Service, Slot, BookingRequest } from '../types';

export default function BookingScreen({ navigation, route }) {
  const { staffId, service } = route.params as
    { staffId: number; service: Service };

  const { selectedDate, setSelectedDate,
          selectedSlot, setSelectedSlot } = useBookingStore();

  const { initPaymentSheet, presentPaymentSheet } = useStripe();

  const { data: slots, isLoading } = useQuery({
    queryKey: ['slots', staffId, service.id, selectedDate],
    queryFn: () => api.get<Slot[]>(
      `/slots/${staffId}/${service.id}/${selectedDate}`),
    enabled: !!selectedDate,
  });

  const createBooking = useMutation({
    mutationFn: (request: BookingRequest) =>
      api.post('/bookings', request),
    onSuccess: async (response) => {
      if (response.data.clientSecret) {
        await handleStripePayment(response.data.clientSecret);
      } else {
        Alert.alert('Успіх', 'Запис створено');
        navigation.navigate('MyBookings');
      }
    },
    onError: (error: any) => {
      if (error.response?.status === 409) {
        Alert.alert('Слот зайнятий',
          'Цей час щойно зайняли. Оберіть інший.');
      } else {
```

```
      Alert.alert('Помилка', 'Не вдалося створити запис');
    }
  },
});

const handleStripePayment = async (clientSecret: string) => {
  const { error: initError } = await initPaymentSheet({
    merchantDisplayName: 'Beauty Salon',
    paymentIntentClientSecret: clientSecret,
    allowsDelayedPaymentMethods: false,
  });

  if (initError) {
    Alert.alert('Помилка ініціалізації', initError.message);
    return;
  }

  const { error: paymentError } = await presentPaymentSheet();
  if (paymentError && paymentError.code !== 'Canceled') {
    Alert.alert('Помилка оплати', paymentError.message);
  } else if (!paymentError) {
    Alert.alert('Успіх', 'Запис створено та оплачено');
    navigation.navigate('MyBookings');
  }
};

const handleConfirm = () => {
  if (!selectedSlot || !selectedDate) {
    Alert.alert('Оберіть час', 'Спочатку оберіть дату та слот');
    return;
  }

  createBooking.mutate({
    staffId,
    serviceId: service.id,
    startTime: `${selectedDate}T${selectedSlot}`,
  });
};

return (
  <View style={styles.container}>
    <Text style={styles.title}>
      Оберіть час для {service.name}
    </Text>
```

```
      <Calendar
        onDayPress={(day) => setSelectedDate(day.dateString)}
        markedDates={{
          [selectedDate]: {
            selected: true,
            selectedColor: '#6366f1'
          }
        }}
        minDate={new Date().toISOString().split('T')[0]}
      />

      {isLoading ? (
        <Text style={styles.loading}>Завантаження слотів...</Text>
      ) : (
        <FlatList
          data={slots ?? []}
          keyExtractor={(item) => item.time}
          numColumns={3}
          renderItem={({ item }) => (
            <TouchableOpacity
              style={[styles.slot,
                selectedSlot === item.time && styles.slotSelected]}
              onPress={() => setSelectedSlot(item.time)}
            >
              <Text style={styles.slotText}>{item.time}</Text>
            </TouchableOpacity>
          )}
        />
      )}

      <TouchableOpacity
        style={styles.confirmButton}
        onPress={handleConfirm}
        disabled={createBooking.isPending}
      >
        <Text style={styles.confirmText}>
          {createBooking.isPending
            ? 'Створення...'
            : 'Підтвердити запис'}
        </Text>
      </TouchableOpacity>
    </View>
  );
}
```

```
const styles = StyleSheet.create({
  container: { flex: 1, padding: 16, backgroundColor: '#fff' },
  title: { fontSize: 20, fontWeight: '600', marginBottom: 16 },
  slot: { flex: 1, margin: 4, padding: 12,
          backgroundColor: '#f3f4f6', borderRadius: 8,
          alignItems: 'center' },
  slotSelected: { backgroundColor: '#6366f1' },
  slotText: { fontSize: 16 },
  loading: { textAlign: 'center', padding: 20 },
  confirmButton: { backgroundColor: '#6366f1', padding: 16,
                   borderRadius: 12, marginTop: 20 },
  confirmText: { color: '#fff', textAlign: 'center',
                 fontSize: 16, fontWeight: '600' },
});
#include <iostream>
#include  <cmath>

using namespace std;
int main()
{
        int     f=1,i=0;
        float sum=1.0,t,x,e=0.0001;
        cout<<"x=";
        cin>>x;
        do
        {
                i++;
                f*=i;//f=f*i;
                t=pow(x,i)/f;
                sum+=t;
                }
        while (t>e);
        cout<<"sum="<<sum<<" i="<<i;
        return 0;
}
```

ДОДАТОК Б

Слайди презентації

Рисунок Б.1 – Слайд 1

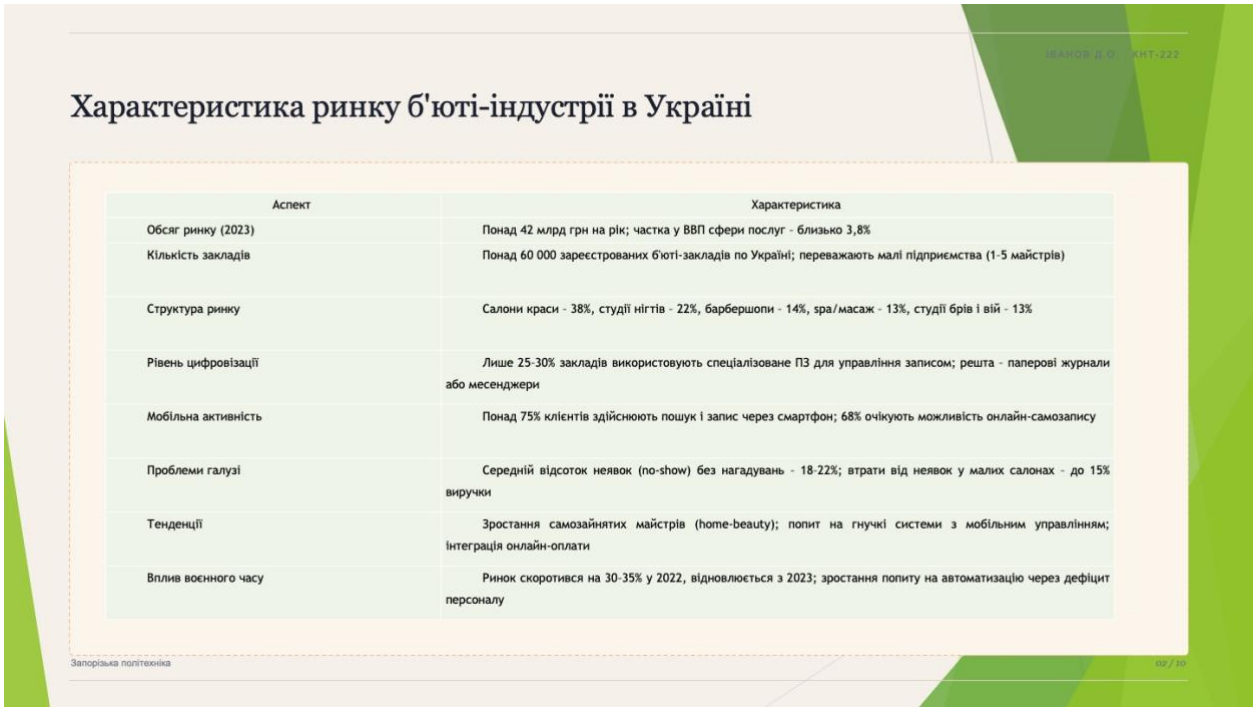

Рисунок Б.2 – Слайд 2

Порівняльний аналіз існуючих рішень

Критерій	Fresha	Treatwell	Booksy	Розроблювана система
Мобільний додаток клієнта	Так	Так	Так	Так (React Native)
Додаток для майстра	Частково	Ні	Так	Так (окрема роль)
Панель адміністратора	Так (web)	Так (web)	Так (mobile)	Так (mobile)
Онлайн-оплата гривнею	Ні	Ні	Ні	Так (Stripe + Liqpay)
Українська локалізація	Ні	Ні	Частково	Так (повна)
Push-нагадування	Так	Так	Так	Так (FCM/APNs)
SMS-нагадування	Платно	Платно	Платно	Так (Twilio/Kyivstar)
Управління послугами	Так	Так	Так	Так
Аналітика завантаженості	Базова	Розвинена	Базова	Так (графіки)
Програма лояльності	Ні	Так	Ні	Бонусні бали
Підтримка малих салонів (1-3 майстри)	Так	Обмежено	Дорого	Так (пріоритет)
Безкоштовний тариф	Так	Ні	Ні	Так (до 3 майстрів)
Стек мобільного клієнта	React Native	React Native	React Native	React Native + Expo + TS

Рисунок Б.3 – Слайд 3

Функціональні вимоги та варіанти використання

Код	Роль	Опис функціональної вимоги
FR-01	Клієнт	Перегляд каталогу послуг салону з цінами, тривалістю та категоріями. Фільтрація за категорією та діапазоном ціи
FR-02	Клієнт	Вибір конкретного майстра або режим «перший доступний». Перегляд профілю майстра з портфоліо та рейтингом
FR-03	Клієнт	Вибір часового слоту на інтерактивному календарі. Система відображає лише вільні слоти з урахуванням розкладу майстра та тривалості послуги
FR-04	Клієнт	Підтвердження запису з опціональною передоплатою через Stripe. Отримання підтвердження на email та push-сповіщення
FR-05	Клієнт	Управління власними записами: перегляд майбутніх та минулих, скасування (з поверненням коштів якщо > 24 год)
FR-06	Клієнт	Програма лояльності: нарахування бонусних балів за кожний виконаний запис, часткова оплата балами при наступному записі
FR-07	Клієнт	Черга очікування: якщо всі слоти зайняті, клієнт може стати у waitlist та отримати автоматичне сповіщення при вивільненні
FR-08	Майстер	Перегляд власного розкладу на день і тиждень. Кожен запис показує клієнта, послугу, тривалість, статус оплати та нотатки
FR-09	Майстер	Управління власним графіком: налаштування робочих годин на кожен день тижня, встановлення перерв та відпусток
FR-10	Майстер	Позначення запису як виконаного, додавання нотаток про клієнта. Перегляд власної аналітики: виручка, кількість клієнтів
FR-11	Адмін	Зведений розклад усього персоналу на день/тиждень із фільтрацією за майстром. Ручний запис від імені клієнта
FR-12	Адмін	Управління довідниками: послуги (назва, ціна, тривалість, категорія, депозит), майстри, категорії послуг
FR-13	Адмін	Аналітична панель: кількість записів, виручка, завантаженість майстрів, топ-послуги, аналіз no-show за період
FR-14	Адмін	Управління акціями та знижками: відсоткова або фіксована знижка, обмеження по терінах та послугах

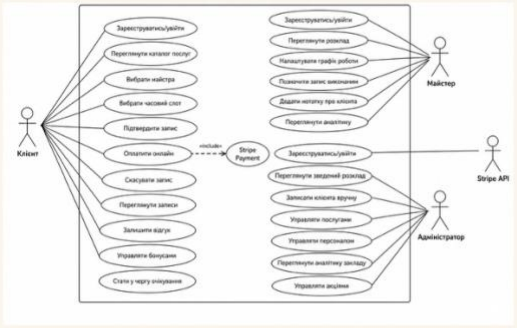

Рисунок Б.4 – Слайд 4

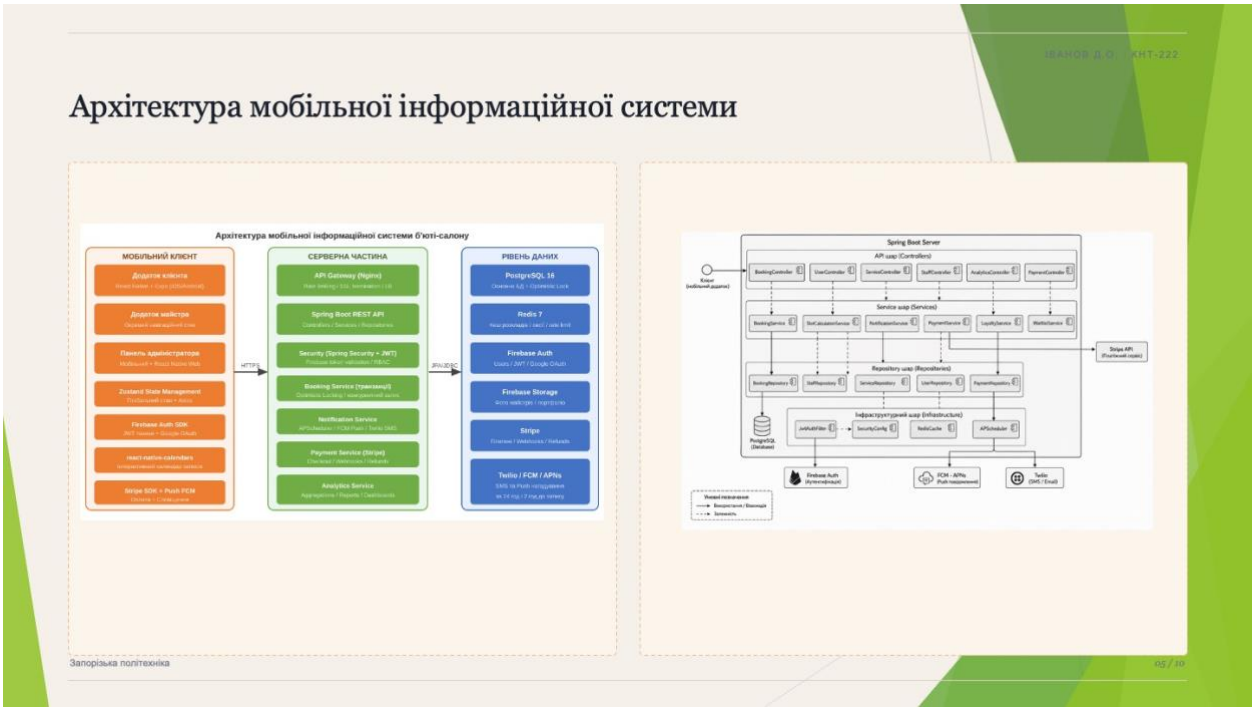

Рисунок Б.5 – Слайд 5

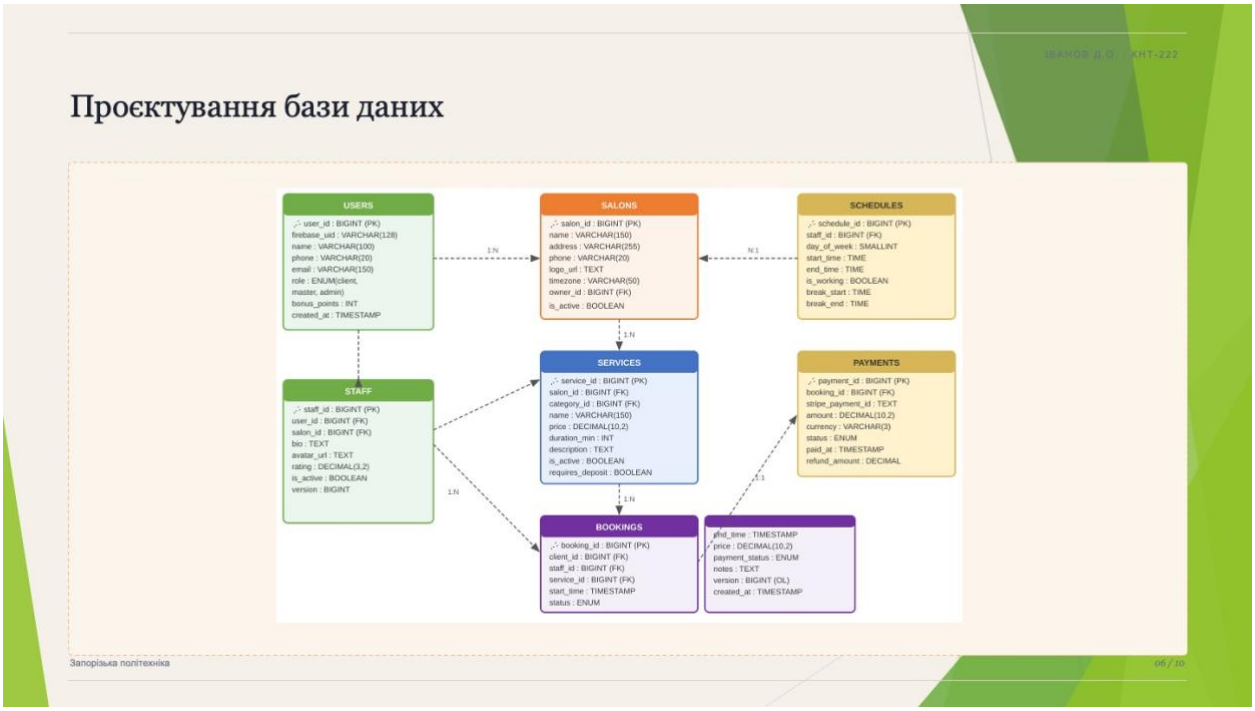

Рисунок Б.6 – Слайд 6

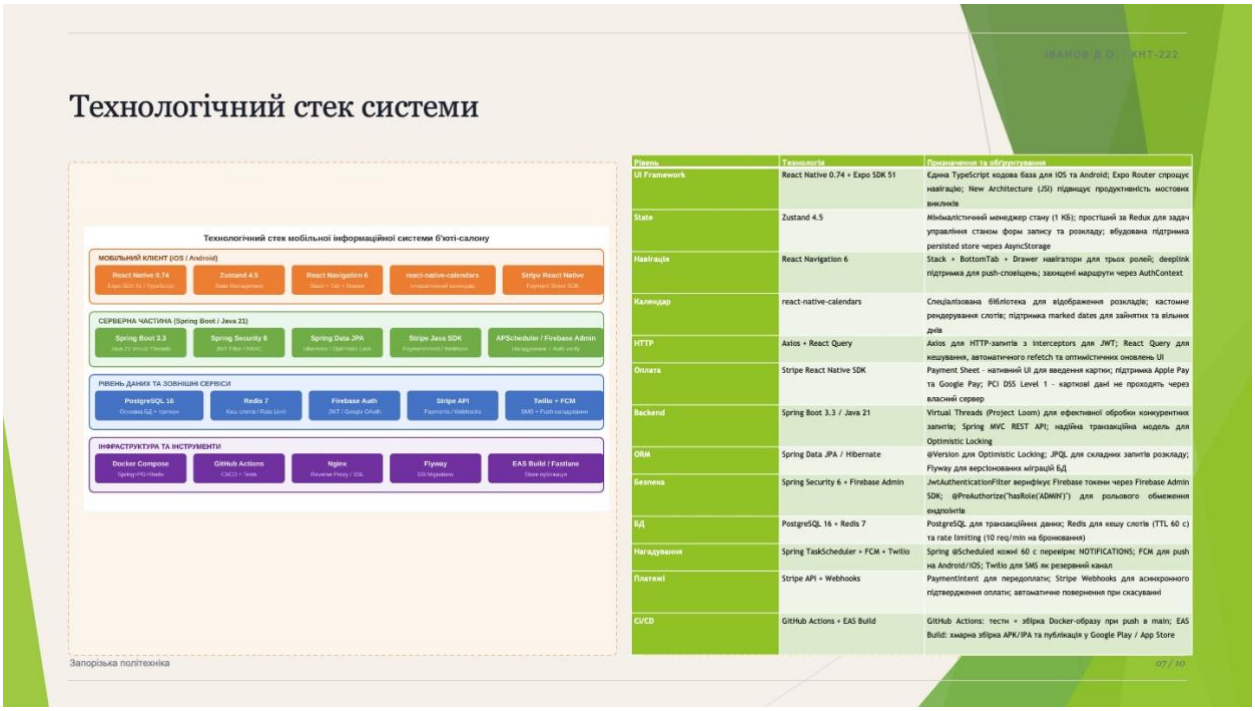

Рисунок Б.7 – Слайд 7

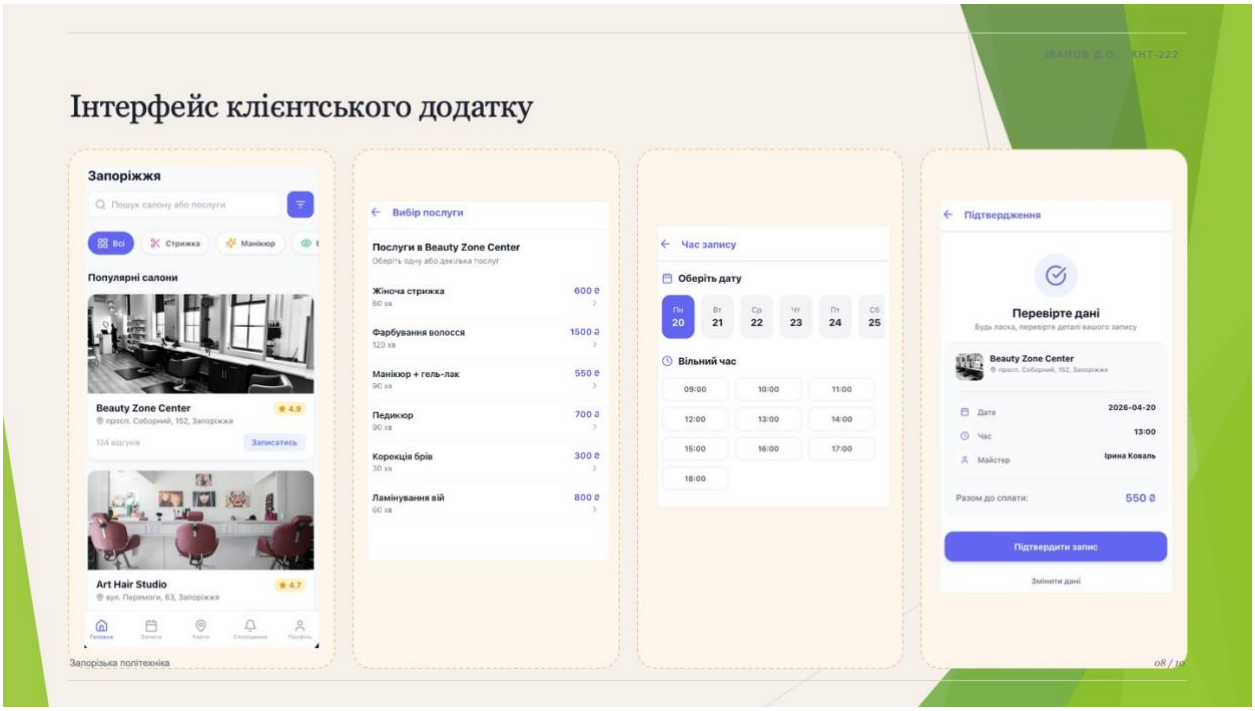

Рисунок Б.8 – Слайд 8

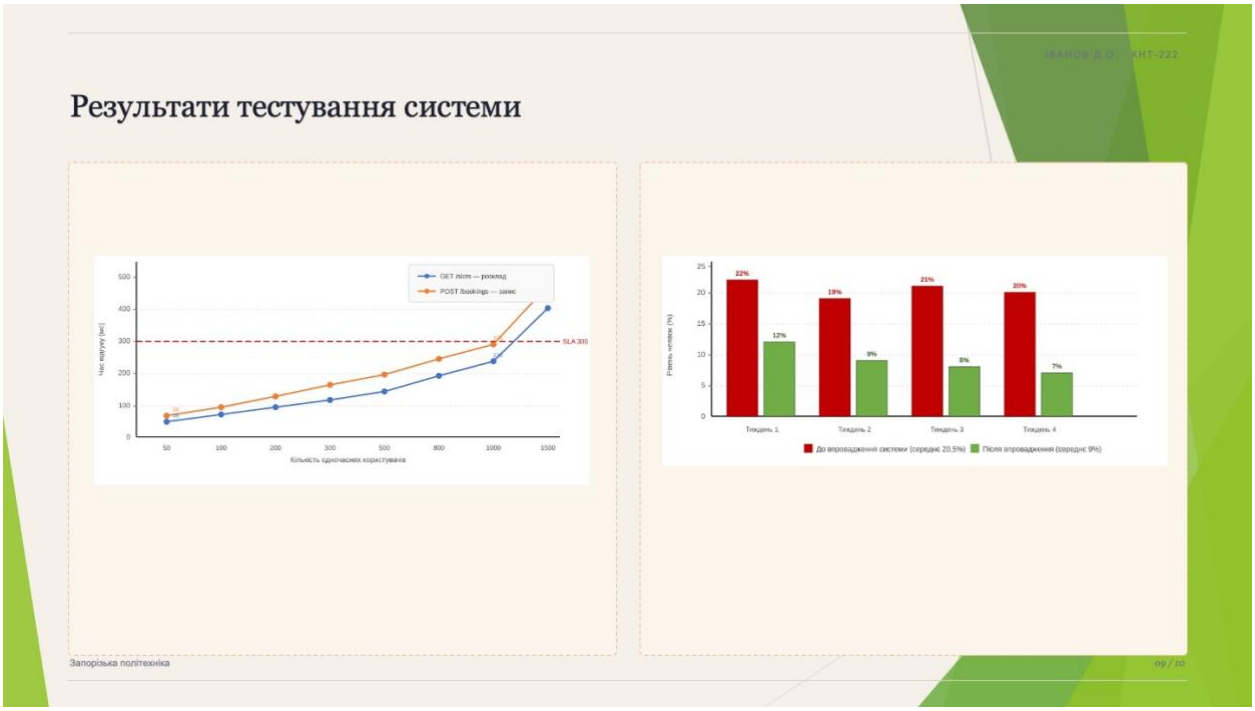

Рисунок Б.9 – Слайд 9

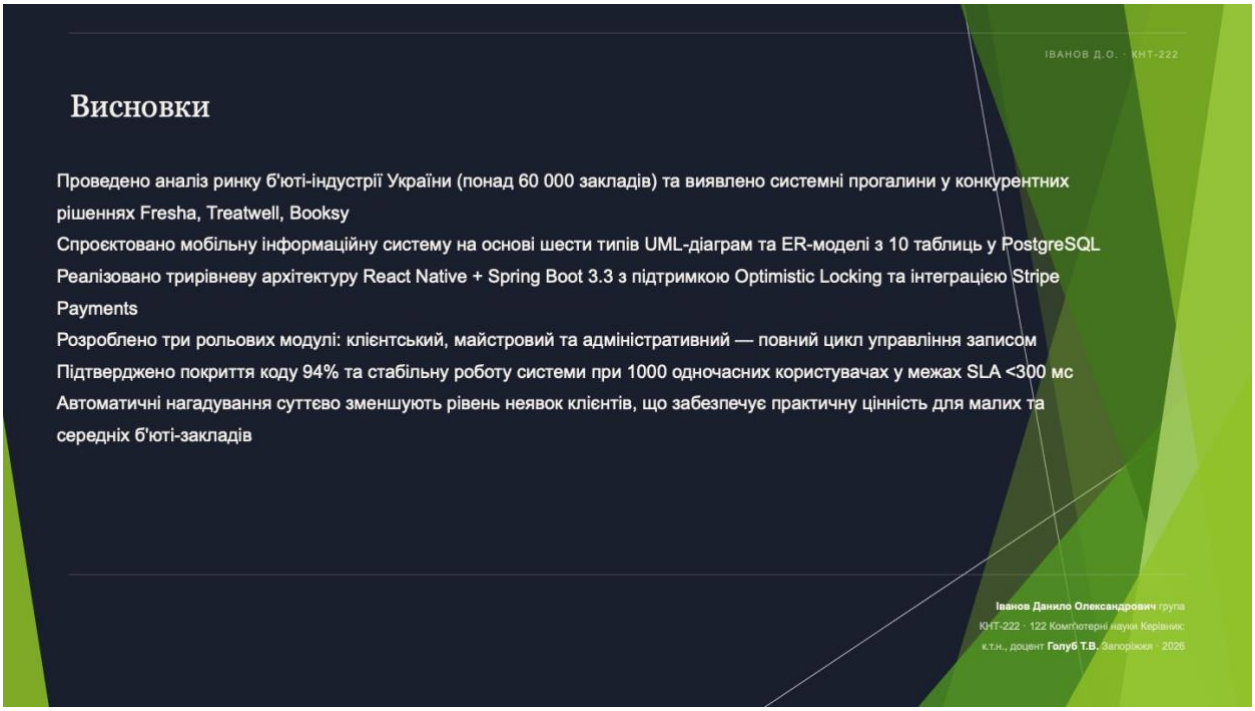

Рисунок Б.10 – Слайд 10