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 РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ВЕБОРІЄНТОВАНОЇ СИСТЕМИ ОНЛАЙН-БРОНЮВАННЯ ПОСЛУГ БАРБЕРШОПУ
DEVELOPMENT OF A WEB-BASED ONLINE BOOKING SYSTEM FOR BARBERSHOPS

Виконав(ла): студент(ка) 4 курсу, групи КНТ-122
Спеціальності 121 Інженерія програмного
(код і найменування спеціальності)
забезпечення

Освітня програма (спеціалізація)
Інженерія прогамного забезпечення
КОГАНТІ К.К.
(ПРІЗВИЩЕ та ініціали)
Керівник СТЕПАНЕНКО О.О.
(ПРІЗВИЩЕ та ініціали)
Рецензент ДЬЯЧУК Т.С.

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 121 Інженерія програмного забезпечення
(код і найменування)
Освітня програма (спеціалізація) Інженерія програмного забпезчення
(назва освітньої прогарми (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

КОГАНТІ Кирила Кірановича
(ПРІЗВИЩЕ, ім’я, по батькові)

1. Тема проєкту (роботи) Розроблення програмного забезпечення веборієнтованої системи онлайн-бронювання послуг барбершопу. Development of a Web-Based Online Booking System for Barbershops

керівник проєкту (роботи) к.т.н., СТЕПАНЕНКО Олександр Олексійович ,
(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) 1. Аналіз предметної області. 2. Розробка архітектури програми. 3. Розробка програми. 4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (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	СТЕПАНЕНКО О.О., доцент		
Нормоконтроль	БЄЛОВА А.В., асистент		

7. Дата видачі завдання “ 14 ” квітня 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів курсового проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	2–3 тижні	Розділ 1
3	Розробка архітектури програми.	4 тиждень	Розділ 2
4	Розробка програми.	5 тиждень	Розділ 3
5	Тестування та експериментальне дослідження	6 тиждень	Розділ 4
	програми.		
6	Оформлення пояснювальної записки та документів	7 тиждень	Додатки
	до неї.		
7	Нормоконтроль та рецензування.		
8	Захист роботи.	8 тиждень	

Студент(ка) _____ Кирило КОГАНТІ
(підпис) (Ім’я ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Олександр СТЕПАНЕНКО
(підпис) (Ім’я ПРІЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 210 с., 10 табл., 40 рис., 3 дод., 25 джерел.

АВТОМАТИЗАЦІЯ БІЗНЕС-ПРОЦЕСІВ, ВЕБ-ЗАСТОСУНОК, CRM-СИСТЕМА, ОНЛАЙН-БРОНЮВАННЯ, БАРБЕРШОП, УПРАВЛІННЯ ПЕРСОНАЛОМ, КЛІЄНТСЬКА БАЗА, РОЗКЛАД РОБОТИ.

Об’єкт дослідження – процес інженерії програмного забезпечення для управління взаємодією з клієнтами у сфері обслуговування.

Предмет дослідження – методи та інструменти розроблення клієнт-серверних веб-систем для онлайн-запису, управління розкладом та адміністрування бізнес-процесів підприємства.

Мета роботи – підвищення ефективності роботи барбершопу шляхом розроблення комплексної інформаційної системи, що забезпечує автоматизацію процесу запису клієнтів, управління розкладом майстрів, ведення клієнтської бази та надання зручного інтерфейсу для всіх учасників процесу.

Матеріали, методи та технічні засоби: використано методи об’єктно-орієнтованого програмування та архітектурний підхід монорепозиторію. Серверну частину (Backend) реалізовано на фреймворку NestJS із використанням мови TypeScript, Prisma ORM та реляційної бази даних PostgreSQL. Клієнтську частину (Frontend) побудовано на базі фреймворку Next.js (React) з використанням бібліотеки Zustand для управління глобальним станом [1]-[2]. Для забезпечення безпеки реалізовано надійну систему автентифікації на основі JWT-токенів із використанням захищених HTTP-only cookies та механізмом безшовного оновлення сесій (refresh tokens). Документування API виконано за допомогою Swagger (OpenAPI). Для ізоляції середовищ та підготовки до розгортання застосовано технологію

контейнеризації Docker (із налаштуванням багатоетапних збірок та Docker Compose). Розроблення проводилося в середовищі Visual Studio Code на персональному комп’ютері з операційною системою Windows.

Результати. Розроблено повнофункціональний веб-застосунок «Velvet Razor», який об'єднує клієнтський портал та багатофункціональну панель адміністратора. Реалізовано багатокроковий інтерактивний процес онлайн-запису, що динамічно враховує розклад майстрів, їхні вихідні, кастомні перерви та вже існуючі бронювання. Створено модуль адміністрування з інтерактивним відображенням розкладу у вигляді сітки (Google Calendar style grid), системою управління клієнтами (з можливістю блокування або встановлення вимоги обов'язкової передоплати), а також керування послугами та персоналом. Впроваджено функціонал ведення портфоліо майстрів, відстеження статусів записів та складний алгоритм розрахунку вільного часу з використанням механізмів пошуку часових конфліктів у реальному часі.

Висновки. Досягнуто мети роботи: створено надійний інструмент, який автоматизує рутинні операції адміністратора, робить процес запису максимально зручним для клієнтів та повністю виключає можливість накладання записів завдяки глибокій перевірці конфліктів на стороні сервера. Застосунок має адаптивний, сучасний та інтуїтивно зрозумілий дизайн, підтримує чіткий поділ прав доступу (Admin, Client, Barber) та забезпечує високий рівень безпеки конфіденційних даних користувачів. Проєкт повністю підготовлено до розгортання в production-середовищі за допомогою створеної Docker-інфраструктури.

Галузь використання – автоматизація бізнес-процесів у сфері послуг, зокрема для барбершопів, салонів краси, медичних або стоматологічних кабінетів, СТО та інших підприємств, де вимагається попередній запис клієнтів до спеціалістів.

Економічна ефективність. Використання застосунку дозволяє скоротити витрати часу адміністратора на телефонні розмови та ручне ведення журналів запису до 70%. Система мінімізує ризики людського фактору

(помилки в розкладі, втрата клієнтських даних) та суттєво підвищує лояльність клієнтів за рахунок цілодобової доступності сервісу онлайн-бронювання, що в перспективі призводить до збільшення клієнтського потоку та загального прибутку підприємства.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 210 pages, 10 tables, 40 figures, 3 appendixes, 25 sources.

BUSINESS PROCESS AUTOMATION, WEB APPLICATION, CRM SYSTEM, ONLINE BOOKING, BARBERSHOP, PERSONNEL MANAGEMENT, CLIENT BASE, WORK SCHEDULE.

Research object – the process of software engineering for managing client interaction in the service sector.

Research subject – methods and tools for developing client-server web systems for online booking, schedule management, and business process administration.

The purpose of the work – to increase the efficiency of a barbershop's operations by developing a comprehensive information system that ensures the automation of the client booking process, master schedule management, client base maintenance, and provides a user-friendly interface for all process participants.

Materials, methods, and technical means: object-oriented programming methods and a monorepo architectural approach were used. The backend is implemented on the NestJS framework using TypeScript, Prisma ORM, and a PostgreSQL database. The frontend is built using Next.js (React) with the Zustand library for global state management. To ensure security, a robust authentication system based on JWT tokens was implemented using secure HTTP-only cookies and a seamless refresh token mechanism. API documentation was performed using Swagger (OpenAPI). To isolate environments and prepare for deployment, Docker containerization technology was applied (with multi-stage builds and Docker Compose configurations). Development was carried out in the Visual Studio Code environment on a personal computer running Windows.

Results. A full-featured web application “Velvet Razor” was developed, combining a client portal and a multi-functional administrator dashboard. A multi-step interactive online booking process was implemented, dynamically accounting for master schedules, their days off, custom breaks, and already existing appointments. An administration module was created with an interactive Google Calendar-style grid schedule visualization, a client management system (with the ability to block users or set a mandatory prepayment requirement), as well as service and staff management. Functionality for maintaining master portfolios, tracking appointment statuses, and a complex algorithm for calculating available time using real-time schedule conflict detection mechanisms were introduced.

Conclusions. The work’s objective was achieved: a reliable tool was created that automates the administrator's routine operations, makes the booking process as convenient as possible for clients, and completely eliminates the possibility of overlapping appointments thanks to deep server-side conflict validation. The application features a responsive, modern, and intuitive design, supports clear segregation of access rights (Admin, Client, Barber), and ensures a high level of security for users' confidential data. The project is fully prepared for deployment in a production environment using the created Docker infrastructure.

Field of application – business process automation in the service sector, specifically for barbershops, beauty salons, medical or dental clinics, auto repair shops, and other enterprises where prior client appointment booking with specialists is required.

Economic efficiency. The application reduces the time spent by the administrator on phone calls and manual appointment log maintenance by up to 70%. The system minimizes human error risks (schedule mistakes, loss of client data) and significantly increases client loyalty through the 24/7 availability of the online booking service, which ultimately leads to an increase in client flow and overall enterprise profit.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

ACID – Atomicity, Consistency, Isolation, Durability;
API – Application Programming Interface;
CD – Continuous Delivery;
CI – Continuous Integration;
CORS – Cross-Origin Resource Sharing;
CPU – Central Processing Unit;
CRM – Customer Relationship Management;
CSS – Cascading Style Sheets;
DAST – Dynamic Application Security Testing;
DTO – Data Transfer Object;
ES6 – ECMAScript 6;
HTTP – HyperText Transfer Protocol;
HTTPS – HyperText Transfer Protocol Secure;
ID – Identifier;
IDE – Integrated Development Environment;
IP – Internet Protocol;
JSON – JavaScript Object Notation;
JWT – JSON Web Token;
LCP – Largest Contentful Paint;
LTE – Long-Term Evolution;
ORM – Object-Relational Mapping;
OTP – One-Time Password;
PHP – PHP: Hypertext Preprocessor;
PWA – Progressive Web Application;
RBAC – Role-Based Access Control;
REST – Representational State Transfer;
RSC – React Server Components;
SAST – Static Application Security Testing;

SEO – Search Engine Optimization;
SMS – Short Message Service;
SQL – Structured Query Language;
SSD – Solid-State Drive;
TS – TypeScript;
UI – User Interface;
URL – Uniform Resource Locator;
UX – User Experience;
VPS – Virtual Private Server;
WSL2 – Windows Subsystem for Linux 2;
XSS – Cross-Site Scripting;
БД – база даних;
ГБ – гігабайт;
ІТ – інформаційні технології;
МБ (MB) – мегабайт (Megabyte);
ОС – операційна система;
ПЗ – програмне забезпечення;
СУБД – система управління базами даних.

ВСТУП

Стрімка еволюція цифрових технологій диктує нові вимоги до сфери надання послуг, де ключовим фактором успіху бізнесу стає рівень клієнтського сервісу та швидкість адаптації до потреб цифрового суспільства. У сучасному ритмі життя особливої ваги набуває можливість дистанційної взаємодії: від оперативного пошуку вільного часу спеціаліста до миттєвого бронювання послуги без необхідності телефонних дзвінків. Попри велику кількість доступних програмних рішень, чимало підприємств малого та середнього бізнесу (зокрема барбершопи та салони краси) досі відчувають брак гнучких інструментів, які б забезпечували зручний інтерфейс для клієнта і водночас не перевантажували адміністраторів зайвим функціоналом. Актуальність розробки зумовлена необхідністю створення єдиного середовища, де процес бронювання, управління розкладом та ведення клієнтської бази відбуваються безперервно, автоматизовано та без ризику виникнення часових колізій.

Глобальні тенденції у секторі послуг підтверджують, що цифровізація є фундаментальною умовою збереження конкурентоспроможності. Аналітичні звіти ринку електронної комерції та сервісу наголошують: понад 40% онлайн-бронювань здійснюються в неробочий час, і неможливість записатися на послугу ввечері або тривале очікування на відповідь адміністратора стають головними чинниками втрати потенційних клієнтів [3]. Хоча автоматизація систем бронювання не є новою концепцією, ринок постійно потребує інструментів, що поєднують високу надійність обробки даних із простотою впровадження. Існуючі громіздкі CRM-системи часто виявляються занадто складними та фінансово обтяжливими для швидкого старту малих робочих груп, створюючи додатковий бар'єр для цифровізації малого бізнесу.

Проблема дослідження полягає в необхідності створення легкого, швидкодійного та вузькоспеціалізованого інструменту, який забезпечує

зручне управління записами, автоматичний контроль конфліктів у розкладі майстрів та чітку рольову модель доступу. Огляд ринку показує, що хоча подібні задачі вирішуються в таких комплексних системах, як YCLIENTS, Altegio або Booksy, вони часто мають надлишкову складність, містять непотрібні для невеликих команд модулі (складний складський облік, інтеграція з десятками зовнішніх сервісів) та вимагають тривалого навчання персоналу [4]. Провідні дослідження в галузі сучасної веб-розробки (зокрема, документація фреймворків Next.js та NestJS) вказують на перспективність використання мікросервісних або модульних архітектур та суворо типізованих мов програмування, як-от TypeScript, для створення надійних систем обробки паралельних транзакцій (записів).

Об’єктом дослідження виступають процеси автоматизації надання послуг та управління взаємодією з клієнтами у сфері обслуговування (на прикладі барбершопу).

Предметом дослідження є методи та інструменти розроблення клієнт-серверних веб-систем для онлайн-запису, управління розкладом та адміністрування бізнес-процесів підприємства.

Мета роботи полягає у створенні цілісної веб-екосистеми для управління барбершопом, яка забезпечує безперебійний процес онлайн-бронювання через інтерактивні інтерфейси, гнучке управління графіками роботи персоналу та автоматизований облік клієнтів.

Для досягнення цієї мети було окреслено та виконано такі етапи дослідження:

- системний аналіз бізнес-процесів сфери послуг для визначення оптимального набору функцій майбутнього застосунку та вимог до бізнес-логіки;
- проектування архітектури системи на базі REST API з використанням підходу монорепозиторію для оптимізації розробки та перевикористання типів;

- розробка та впровадження модулів багатокрокового онлайн-бронювання, інтерактивної сітки розкладу (канбан-подібного календаря), управління послугами та ієрархічної моделі доступу;
- реалізація складного алгоритму пошуку вільного часу та запобігання конфліктам у розкладі з урахуванням вихідних, кастомних перерв та тривалості послуг;
- проведення комплексного тестування системи на предмет швидкодії, безпеки автентифікації (за допомогою JWT та HTTP-only cookies) та коректності обробки даних при одночасному доступі користувачів.

Методологічну базу дослідження складають принципи об’єктно-орієнтованого проектування, реляційної алгебри та сучасні стандарти безпечної веб-розробки. Технічна реалізація базується на використанні фреймворку Next.js для фронтенд-частини та NestJS для серверної логіки. Роботу з даними організовано за допомогою Prisma ORM та бази даних PostgreSQL, а розгортання системи стандартизовано через контейнеризацію Docker.

Розроблений інструментарій має високу практичну цінність для підприємств, що прагнуть оптимізувати комунікацію з клієнтами та мінімізувати час на ручне адміністрування розкладу. Результати впровадження свідчать про значне підвищення прозорості робочих процесів, усунення ризиків подвійного бронювання та суттєве покращення користувацького досвіду як для клієнтів, так і для персоналу.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Короткий опис досліджуваної предметної області

Організація процесів надання послуг у сфері краси та догляду (зокрема, в барбершопах) – це комплексна діяльність, яка базується на оптимізації робочого часу спеціалістів, моніторингу завантаженості закладу та безперервній взаємодії з клієнтами. Чимало підприємств малого та середнього бізнесу досі намагаються керувати потоком клієнтів через розрізнені месенджери, телефонні дзвінки або паперові журнали (чи статичні електронні таблиці), що неминуче призводить до втрати контексту комунікації, виникнення «подвійних записів» (накладок) та зниження лояльності аудиторії. Це зумовлює гостру потребу у впровадженні спеціалізованих професійних інструментів – CRM-систем та платформ онлайн-бронювання, здатних повністю автоматизувати життєвий цикл обслуговування клієнта.

Ключові поняття, що формують фундамент сучасного управління бізнес-процесами барбершопу:

- бронювання (запис, appointment) – атомарна одиниця робочого процесу, що закріплює певний проміжок часу майстра за конкретним клієнтом для надання обраного переліку послуг;
- розклад (графік роботи, schedule) – структурована система управління робочим часом майстра, що враховує стандартні робочі години, вихідні дні та кастомні перерви (schedule exceptions);
- клієнтська база (client base) – централізоване сховище даних про клієнтів, яке містить їхню контактну інформацію, історію візитів, а також специфічні статуси (наприклад, рівень надійності, вимога передоплати або блокування);
- послуга (service) – стандартизована пропозиція закладу, яка має визначену тривалість і вартість. На рівні виконання послуги закріплюються за певними майстрами, які мають необхідну кваліфікацію для їх надання;

- майстер (barber) – спеціаліст, доступність якого є динамічною величиною, що розраховується на основі його графіка роботи та вже підтверджених бронювань.

На сьогодні галузь автоматизації послуг стикається з кількома критичними викликами. По-перше, це висока динаміка змін, через яку адміністраторам складно підтримувати актуальність розкладу при інтенсивному потоці телефонних дзвінків та скасувань в останню мить. По-друге, існує проблема синхронізації дій між системою самостійного онлайн-запису та діями адміністратора в салоні: будь-яка затримка в оновленні бази даних призводить до конфліктів у розкладі. По-третє, безпека персональних даних та розмежування прав доступу: клієнти не повинні мати доступу до даних інших відвідувачів, а майстри повинні бачити лише свій графік без доступу до глобальної аналітики закладу, яка є прерогативою адміністратора. Зрештою, більшість базових систем фіксують час жорстко, без інтелектуальних алгоритмів, які б враховували різну тривалість послуг або складні графіки змін.

Серед провідних трендів у цій сфері варто відзначити стрімкий перехід бізнесу до моделі самообслуговування клієнтів (цілодобове онлайн-бронювання), впровадження інтерактивних сіток розкладу з підтримкою миттєвого розрахунку вільного часу (на основі бекенд-алгоритмів обробки колізій), а також фокус на хмарних рішеннях (Cloud) та мобільній адаптивності.

Динаміку зростання світового ринку програмного забезпечення для салонів та спа (із розподілом за типом розгортання), а також стійку тенденцію до збільшення лояльності клієнтів завдяки зручності онлайн-бронювання, відображено на рисунках 1.1 та 1.2.

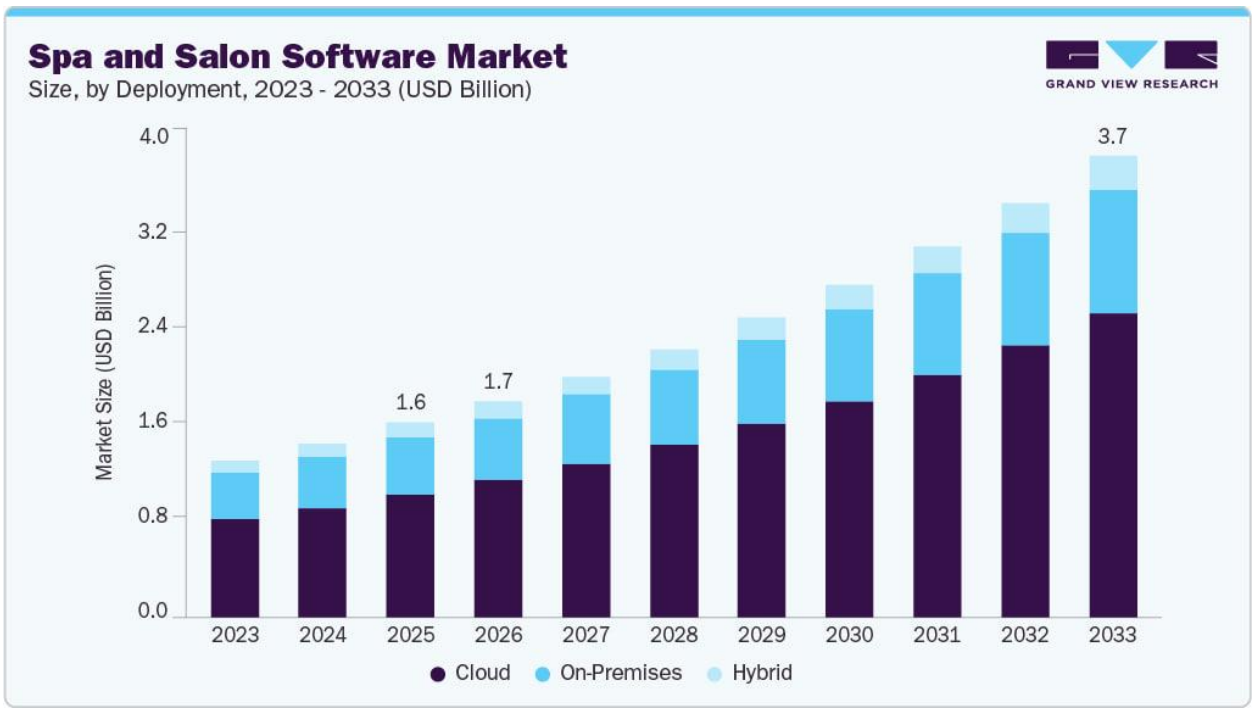

Рисунок 1.1 – Прогноз зростання обсягу світового ринку програмного забезпечення для салонів (у млрд дол. США) за даними аналітичних агентств [5]

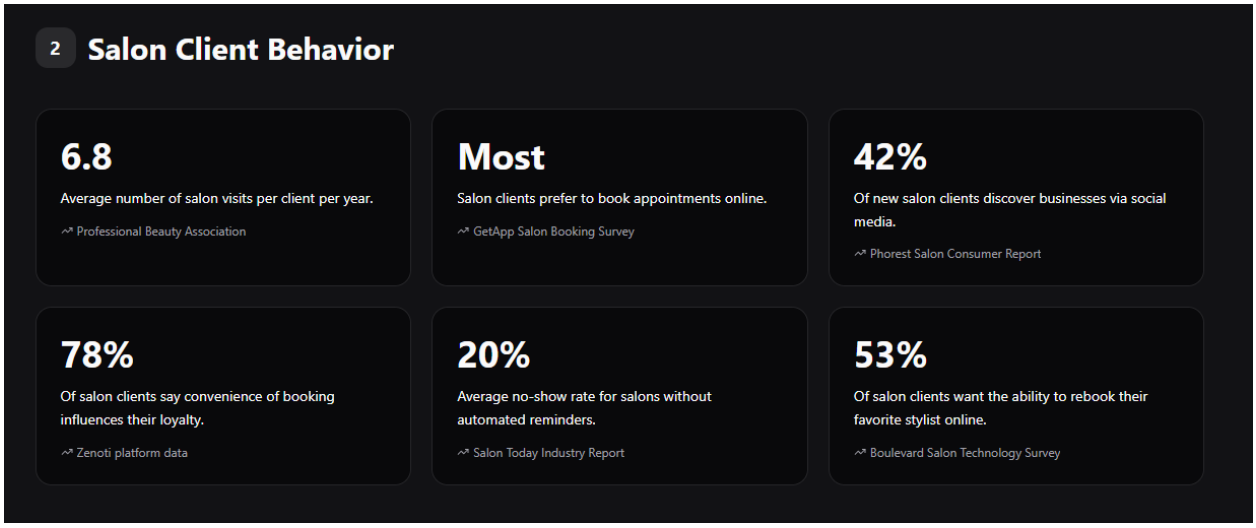

Рисунок 1.2 – Ключові показники поведінки клієнтів салонів та вплив зручності онлайн-бронювання на їхню лояльність [6]

1.2 Огляд існуючих рішень проблеми

Проаналізуємо функціональні можливості трьох відомих систем для автоматизації бізнес-процесів та онлайн-бронювання у сфері краси: Altegio (YCLIENTS), Booksy та Fresha. Кожна з цих платформ пропонує потужний набір інструментів для адміністрування, проте часто вони виявляються занадто громіздкими або фінансово необґрунтованими для невеликих барбершопів, які потребують швидкого, інтуїтивно зрозумілого рішення з акцентом на зручне управління розкладом без зайвого корпоративного функціоналу.

1.2.1 Застосунок Altegio (YCLIENTS)

Altegio (на ринку Східної Європи також відомий як YCLIENTS) – це одна з найпопулярніших хмарних CRM-систем, спеціально розроблених для підприємств сфери послуг. Платформа забезпечує ведення електронного журналу записів, управління клієнтською базою, складський та фінансовий облік, а також надає віджет для онлайн-бронювання на сайті або в соціальних мережах [7].

Переваги:

- потужний модуль складського та фінансового обліку;
- велика кількість готових інтеграцій (IP-телефонія, платіжні системи, SMS-розсилки);
- деталізована аналітика та звіти щодо утримання клієнтів.

Недоліки:

- надлишкова складність інтерфейсу для малих закладів (перевантаженість непотрібними модулями);
- висока вартість щомісячної підписки, яка зростає залежно від кількості співробітників;

– складний процес початкового налаштування та навчання персоналу;

– обмежені можливості візуальної кастомізації віджета запису під специфічний бренд закладу.

Зовнішній вигляд електронного журналу записів (календаря) сервісу Altegio зображено на рисунку 1.3.

Рисунок 1.3 – Інтерфейс електронного журналу записів сервісу Altegio [7]

1.2.2 Застосунок Booksy

Booksy – це глобальна платформа для бронювання послуг у сфері краси та здоров'я, яка робить сильний акцент на моделі маркетплейсу (B2C). Сервіс не лише надає інструменти для управління розкладом барбершопу, але й активно просуває заклад серед користувачів свого популярного мобільного застосунку, допомагаючи залучати нових клієнтів [8].

Переваги:

– велика вбудована база користувачів (ефект маркетплейсу, що генерує новий трафік);

– зручний та сучасний мобільний застосунок для кінцевого споживача (клієнта);

– вбудовані автоматизовані маркетингові інструменти для утримання клієнтів.

Недоліки:

– високі комісійні збори за кожного нового клієнта, залученого через маркетплейс платформи;

– веб-інтерфейс адміністратора менш гнучкий порівняно зі спеціалізованими корпоративними CRM;

– сильна прив'язка до екосистеми (vendor lock-in), що ускладнює міграцію клієнтської бази на інші платформи;

– обмежені можливості налаштування складної логіки тривалості нестандартних послуг.

Зовнішній вигляд електронного журналу записів (календаря) сервісу Booksy зображено на рисунку 1.4.

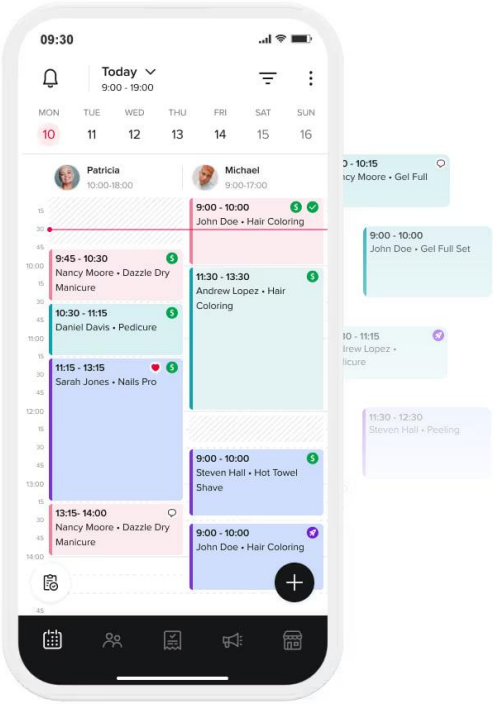

Рисунок 1.4 – Інтерфейс електронного журналу записів сервісу Booksy [8]

1.2.3 Застосунок Fresha

Fresha – популярна платформа для управління салонами краси. Її ключова відмінність – відсутність абонентської плати: базовий функціонал надається безкоштовно, а монетизація відбувається за рахунок транзакційних комісій та відсотка від нових клієнтів, залучених через їхній маркетплейс [9].

Переваги:

- безкоштовний доступ до базового розкладу;
- інтуїтивно зрозумілий інтерфейс;
- вбудована система обробки платежів.

Недоліки:

- високі комісії (понад 20%) за залучення нових клієнтів з маркетплейсу;
- обмежена гнучкість налаштування складних графіків (кастомних перерв);
- жорсткий контроль платформи над клієнтською базою (модель закритої екосистеми).

Зовнішній вигляд електронного журналу записів (календаря) сервісу Fresha зображено на рисунку 1.5.

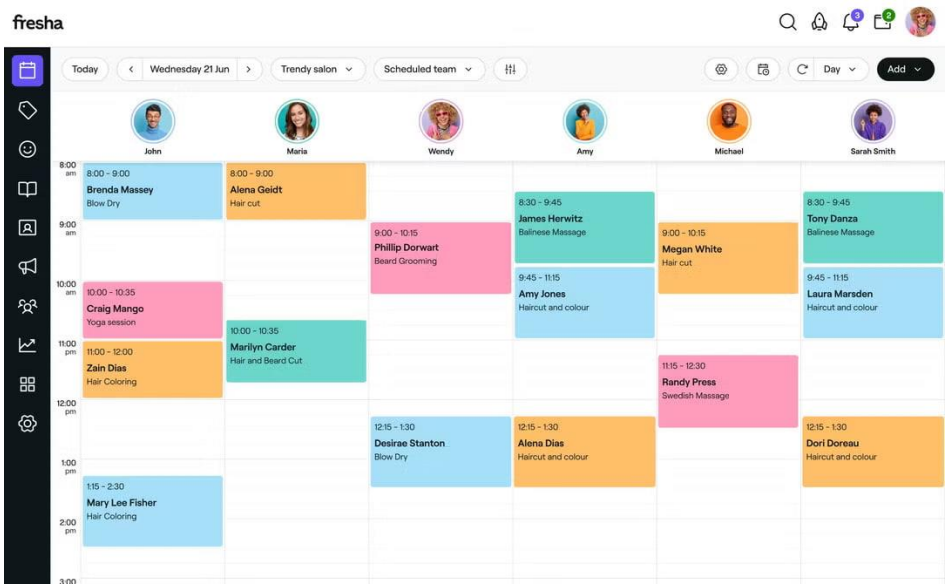

Рисунок 1.5 – Інтерфейс електронного журналу записів сервісу Fresha [9]

1.2.4 Порівняльна таблиця існуючих та розроблюваного програмних продуктів

Для обґрунтування доцільності створення власного програмного забезпечення було проведено порівняльний аналіз розглянутих платформ (Altegio, Booksy, Fresha) та розроблюваного застосунку «Velvet Razor». Аналіз базується на ключових критеріях, які є критичними для невеликих та середніх барбершопів.

Результати порівняльного аналізу наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльна характеристика існуючих та розроблюваного застосунків

Критерій	Altegio	Booksy	Fresha	Velvet Razor
Веб-інтерфейс та мобільна адаптивність	+	+	+	+
Відсутність щомісячної абонентської плати	-	-	+	+
Відсутність комісій за залучених клієнтів	+	-	-	+
Гнучке управління кастомними перервами	+	-	-	+
Відсутність надлишкового корпоративного функціоналу	-	-	+	+
Повний контроль над клієнтською базою	+	-	-	+
Кастомізація інтерфейсу під бренд закладу	-	-	-	+

1.3 Опис основних вимог до програми

1.3.1 Визначення функціональних вимог

Функціональні вимоги визначають конкретні дії, які користувач може виконувати в системі, та її поведінку у відповідь на запити. Для вебзастосунку «Velvet Razor» встановлено такі вимоги:

а) керування доступом, безпекою та профілями:

1) реєстрація нових клієнтів та авторизація в системі з використанням номера телефону та пароля;

2) використання захищених токенів (JWT) та механізму HTTP-only cookies для безпечної і безшовної підтримки сесій (з підтримкою refresh-токенів);

3) суворе розмежування прав доступу на основі рольової моделі (адміністратор системи, майстер/барбер, клієнт);

4) можливість редагування профілю, управління візуальним портфоліо для майстрів та перегляд персональної історії візитів для клієнтів;

б) клієнтський портал та процес онлайн-бронювання:

1) покроковий інтерактивний інтерфейс для самостійного запису на послуги без участі адміністратора;

2) динамічний вибір бажаного набору послуг (з автоматичним розрахунком загальної тривалості та сумарної вартості) та конкретного спеціаліста;

3) інтелектуальна генерація вільних часових слотів "на льоту", яка враховує базовий розклад майстра, його вихідні, кастомні перерви та вже заброньовані іншими клієнтами інтервали;

4) відстеження статусів власних бронювань (очікує підтвердження, підтверджено, скасовано, завершено) та можливість управління ними зі сторони клієнта;

в) модуль управління розкладом та записами (Журнал):

1) візуалізація потоку клієнтів у форматі інтерактивної календарної сітки з абсолютним позиціонуванням блоків залежно від тривалості процедур;

2) створення, редагування, ручне підтвердження та скасування записів адміністратором;

3) можливість створення «тіньових» записів (shadow clients) для клієнтів, які записуються по телефону, без необхідності їх попередньої реєстрації в базі даних;

4) гнучке налаштування графіка роботи персоналу через механізм виключень (додавання позапланових вихідних днів, скорочення або подовження робочих годин на конкретну дату);

г) адміністрування клієнтської бази та персоналу:

1) перегляд повного списку зареєстрованих клієнтів з можливістю пошуку, сортування та перегляду їхньої індивідуальної статистики відвідувань;

2) управління статусом довіри клієнтів: можливість блокування порушників (чорний список) або встановлення прапорця щодо обов'язкової передоплати послуг;

3) управління каталогом послуг: додавання нових позицій, зміна їхньої вартості, тривалості та прив'язка до майстрів;

4) адміністрування персоналу: створення облікових записів майстрів, призначення їм рівнів кваліфікації (наприклад, Junior, Senior, Top Barber) та розподіл за фізичними локаціями барбершопу;

д) алгоритмічне запобігання конфліктам та цілісність даних:

1) автоматична серверна валідація кожного нового бронювання на предмет часових колізій (накладань) із вже існуючими записами;

2) динамічне фільтрування доступного часу майстра з урахуванням тривалості обраних послуг (неможливість записатися на 2-годинну послугу, якщо до кінця робочого дня або початку наступного запису залишається 1 година);

3) захист від конкурентного бронювання (concurrency control) під час спроб одночасного запису кількох клієнтів на один і той самий вільний слот.

1.3.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якісні атрибути системи, такі як надійність, продуктивність, безпека та масштабованість. Для застосунку «Velvet Razor» встановлено такі вимоги:

а) безпека та захист інформації:

1) надійне хешування паролів користувачів перед збереженням у базу даних (за допомогою алгоритму Argon2);

2) жорстка валідація всіх вхідних даних на серверному рівні (з використанням DTO та бібліотеки Zod) для запобігання вразливостям типу SQL-ін'єкцій та XSS-атак (відповідно до переліку критичних загроз OWASP Top Ten) [10];

3) ізоляція даних на рівні бази: клієнти мають доступ виключно до своєї історії записів, майстри бачать лише власний графік, тоді як повний доступ до аналітики та персональних даних надається тільки ролі адміністратора;

б) вимоги до продуктивності:

1) час відгуку API серверної частини на стандартні запити (отримання розкладу, створення запису) не повинен перевищувати 300–400 мс для забезпечення плавності інтерфейсу;

2) оптимізація фронтенд-частини через використання механізмів фреймворку Next.js (комбінація Server-Side Rendering та статичної генерації) для швидкого початкового завантаження сторінок та покращення SEO [11];

3) оптимізація клієнтського стану (state management) за допомогою бібліотеки Zustand для мінімізації зайвих рендерів компонентів інтерфейсу під час роботи з великими сітками розкладу;

в) користувацький інтерфейс та зручність (UI/UX):

1) адаптивна верстка (mobile-first), що забезпечує однаково зручне бронювання зі смартфона для клієнта та адміністрування розкладу з десктопа або планшета для персоналу;

2) використання скелетних завантажувачів (skeleton loaders) для візуального згладжування затримок при отриманні даних із сервера;

3) побудова сучасного, естетичного та консистентного інтерфейсу за допомогою фреймворку Tailwind CSS [12];

г) надійність, сумісність та розгортання:

1) забезпечення абсолютної цілісності бази даних на рівні зовнішніх ключів та обмежень (constraints) через Prisma ORM (наприклад, неможливість видалити послугу, якщо на неї існують активні записи);

2) повна контейнеризація серверної, клієнтської частини та бази даних за допомогою Docker та Docker Compose для швидкого, надійного розгортання та гарантованої стабільності в production-середовищі;

3) кросбраузерна сумісність: коректна робота в актуальних версіях популярних веб-оглядачів (Google Chrome, Safari, Mozilla Firefox);

д) локалізація та підтримка:

1) інтерфейс застосунку має бути повністю локалізований українською мовою з урахуванням місцевих форматів відображення дат та номерів телефонів;

2) інтуїтивно зрозумілі текстові підказки та сповіщення про помилки (наприклад, при спробі записатися на вже зайнятий час).

1.4 Постановка завдання роботи

Метою даної роботи є розроблення комплексної кросплатформеної вебсистеми «Velvet Razor» для автоматизації бізнес-процесів, зручного онлайн-бронювання послуг та ефективного управління персоналом у сфері краси (на прикладі барбершопу).

Основні завдання розробки:

– проєктування архітектури. Вивчити та реалізувати модульну структуру застосунку (на базі монорепозиторію) з використанням фреймворків NestJS (backend) та Next.js (frontend), а також системи управління базами даних PostgreSQL;

– реалізація серверної частини. Розробка RESTful API для керування профілями користувачів, послугами, графіком майстрів та потоком бронювань; налаштування безпечної взаємодії з базою даних через об'єктно-реляційне відображення Prisma ORM;

– впровадження алгоритмів розкладу. Розробка складної серверної бізнес-логіки для обчислення доступних часових слотів у реальному часі, обробки кастомних перерв (schedule exceptions) та запобігання конфліктам записів (conflict detection);

– розробка клієнтського інтерфейсу. Створення динамічного та адаптивного фронтенду з високим рівнем інтерактивності (багатокроковий майстер запису, інтерактивна сітка календаря для адміністраторів) з використанням Tailwind CSS та управлінням станом через Zustand;

– забезпечення безпеки. Впровадження захищених механізмів автентифікації на основі JWT (з використанням HTTP-only cookies) та суворої рольової моделі (Admin, Barber, Client);

– тестування та розгортання. Перевірка коректності обробки паралельних бронювань, оптимізація швидкодії та розгортання ізольованого середовища виконання через багатоетапну збірку (multi-stage builds) у Docker Compose.

Цілі розробки:

– створити надійний та безвідмовний інструмент для автоматизації рутинної роботи адміністраторів та майстрів закладу;

– зробити процес запису максимально швидким і зручним для клієнтів, мінімізувавши кількість телефонних дзвінків;

– гарантувати абсолютну конфіденційність клієнтської бази та розмежувати інформаційне поле між різними рівнями персоналу;

– надати адміністрації зручний візуальний інструмент для глобального моніторингу та управління завантаженістю барбершопу в будь-який момент часу.

1.5 Висновки за розділом 1

Дослідження процесів автоматизації у сфері послуг підтверджує, що ефективне управління розкладом та клієнтською базою залишається складним викликом. Популярні платформи, такі як Altegio, Booksy чи Fresha, пропонують потужний функціонал, проте вони часто перевантажені зайвими корпоративними модулями, вимагають сплати прихованих маркетплейс-комісій або обмежують контроль над базою даних (через vendor lock-in). Головними перешкодами для оптимізації роботи барбершопів є високий поріг входження в CRM-системи, ризик виникнення часових колізій при бронюванні та громіздкість інтерфейсів.

Вебсистема «Velvet Razor» покликана усунути ці недоліки. Застосунок пропонує повністю незалежну екосистему, де акцент зроблено на алгоритми динамічного запобігання конфліктам у розкладі, впровадження інтуїтивної багатокрокової форми бронювання та забезпечення надійного захисту сесій через JWT та HTTP-only cookies. Такий підхід дозволяє адміністраторам зосередитися виключно на якості обслуговування клієнтів.

2 РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМИ

2.1 Структура системи

2.1.1 Основні компоненти системи

Архітектурна побудова системи «Velvet Razor» базується на принципі суворого розділення відповідальності між клієнтською та серверною частинами. На відміну від класичних розділених проєктів, систему реалізовано з використанням архітектурного підходу «монорепозиторій» (Monorepo), керованого інструментом Turborepo. Це забезпечує максимальне перевикористання коду (спільних типів та схем валідації) та зручність масштабування. Проєкт побудовано за багатошаровою моделлю [13].

Клієнтський рівень (Frontend – Next.js):

– компоненти інтерфейсу (UI Components): набір ізольованих візуальних елементів (на основі сучасного CSS-фреймворку Tailwind CSS), що формують інтерфейс користувача (наприклад, інтерактивна сітка розкладу, багатокрокова форма запису клієнта);

– рівень маршрутизації (App Router): сучасна архітектура маршрутизації Next.js із використанням серверних компонентів (React Server Components), що відповідає за навігацію та SEO-оптимізацію контенту;

– управління станом (State Management): використання легкої бібліотеки Zustand для оптимізованого управління складними глобальними станами (наприклад, зберігання поточних вибраних параметрів під час процесу онлайн-бронювання без зайвих перемалювань інтерфейсу).

Серверний рівень (Backend – NestJS):

– контролери (Controllers): відповідають за обробку вхідних HTTP-запитів, строгу валідацію вхідних даних (за допомогою DTO) та повернення відповідей клієнту [14];

– провайдери бізнес-логіки (Services): інкапсулюють основні алгоритми системи, взаємодіють з базою даних, обробляють логіку пошуку вільних часових слотів та перевірку конфліктів у розкладі;

– модульна структура (Modules): логічне групування функціонала (наприклад, AuthModule, AppointmentsModule, ScheduleModule, ServicesModule) для забезпечення високої зв'язності всередині модуля та низької залежності між ними.

Спільний рівень даних (Shared Packages). Пакет спільних схем (@velvet-razor/shared) містить автоматично згенеровані Zod-схеми (за допомогою prisma-zod-generator), що забезпечують єдине джерело істини (Single Source of Truth) для наскрізної типізації та валідації даних як на клієнтському, так і на серверному рівнях [15].

Рівень роботи з даними (Persistence Layer):

– ORM-шар (Prisma): забезпечує безпечну, типізовану взаємодію з базою даних, автоматичну генерацію міграцій та мапінг TypeScript-моделей на таблиці бази даних;

– база даних (PostgreSQL): надійне реляційне сховище для гарантованого збереження інформації про клієнтів, послуги, графіки майстрів та історію записів.

Спеціалізовані сервіси (Core Services):

– сервіс автентифікації (AuthService): реалізує механізми перевірки прав доступу, криптографічного хешування паролів за алгоритмом Argon2id та генерації JWT-токенів із захистом сесій через HTTP-only cookies [16]-[17];

– сервіс управління записами (AppointmentsService): ядро бізнес-логіки системи, що містить математичні алгоритми валідації часових колізій та динамічного розрахунку загальної тривалості складених послуг;

– сервіс розкладу (ScheduleService): відповідає за управління індивідуальними графіками майстрів та обробку кастомних перерв чи додаткових вихідних днів (schedule exceptions);

– сервіс взаємодії з БД (PrismaService): централізований вузол для керування пулом підключень до бази даних та обробки транзакцій.

Зовнішні інтеграції та інфраструктура. Docker-середовище забезпечує контейнеризацію бази даних, бекенду та фронтенду (із застосуванням

багатоетапних збірок – multi-stage builds) для уніфікації процесів розгортання та гарантування абсолютної стабільності роботи системи в ізольованих середовищах незалежно від хост-машини [18].

2.1.2 Взаємодія компонентів системи

Внутрішні комунікації в межах платформи «Velvet Razor» базуються на чіткому розподілі потоків даних між клієнтським застосунком, серверним API та сховищем даних, із суворим контролем типів на кожному етапі.

Клієнтський інтерфейс – Серверний API (REST):

– користувач ініціює дії через UI-компоненти (багатокрокова форма запису для клієнтів або інтерактивна сітка календаря для адміністраторів);

– дані з форм попередньо валідуються на фронтенді за допомогою спільних Zod-схем (із пакета @velvet-razor/shared);

– валідовані запити передаються до NestJS-контролерів через HTTP-протокол (REST API) з додаванням JWT-токенів у заголовки або HTTP-only cookies для автентифікації;

– отримані відповіді кешуються та синхронізуються завдяки бібліотеці TanStack Query, а глобальний стан UI управляється за допомогою Zustand, що ініціює миттєве перемальовування інтерфейсу без перезавантаження сторінки [2], [19].

Бізнес-логіка (Services) – База даних (Prisma):

– сервіси (наприклад, AppointmentsService) обробляють запит, застосовуючи складну бізнес-логіку (перевірка графіків майстрів, виявлення кастомних вихідних днів);

– сервіси звертаються до Prisma ORM для маніпуляції даними (читання, запис, оновлення);

– Prisma генерує оптимізовані SQL-запити до PostgreSQL, забезпечуючи транзакційну безпеку, та повертає строго типізовані об’єкти (TypeScript models);

– дані трансформуються сервісами у необхідний DTO-формат (Data Transfer Object) і відправляються клієнту.

Обробка колізій та конкурентних бронювань (Concurrency Control):

– під час ініціації бронювання послуги сервер не просто виконує запис у базу, а ініціює алгоритм динамічної перевірки перетинів часу (conflict detection);

– система аналізує поточні активні записи майстра та його вихідні на цільову дату;

– у разі спроби конкурентного запису (коли два клієнти одночасно намагаються забронювати один і той самий вільний слот) база даних та бізнес-логіка гарантують, що запис відбудеться лише для першого валідованого запиту, а другий клієнт отримає відповідне сповіщення про зміну розкладу.

2.2 Функціонування системи

2.2.1 Основні процеси системи

Робота застосунку складається з декількох взаємопов'язаних процесів, що забезпечують повний життєвий цикл управління розкладом та обслуговування клієнтів закладу.

Процедура доступу та ідентифікації:

– користувач вводить облікові дані (телефон та пароль), які перевіряються сервісом автентифікації на бекенді;

– система генерує захищений JWT-токен доступу та Refresh-токен (який безпечно зберігається у HTTP-only cookie для захисту від крадіжки сесії);

– доступ до сторінок адміністративної панелі та захищених API-ендпоінтів жорстко регулюється серверними механізмами (Guards) на основі ролей (Admin, Barber, Client).

Організація бази персоналу та клієнтів:

– адміністратор створює профілі нових майстрів, закріплює їх за конкретними локаціями (філіями) барбершопу та призначає рівні кваліфікації;

– база клієнтів формується динамічно: або під час самостійної реєстрації користувача, або при створенні «тіньових» записів (shadow clients) адміністратором по телефону;

– система дозволяє адміністрації управляти «статусом довіри» клієнтів, блокуючи небажаних відвідувачів або активуючи вимогу обов'язкової передоплати.

Процес самостійного онлайн-запису клієнтів:

– клієнт ініціює багатокроковий процес бронювання: обирає перелік необхідних послуг, улюбленого майстра;

– алгоритми бекенду динамічно розраховують сумарну тривалість обраних послуг та генерують список вільних часових слотів, враховуючи робочі години майстра та вже зайнятий час;

– після підтвердження часу дані валідуються на сервері, запис фіксується в БД, а користувач отримує можливість відстежувати його статус у своєму профілі.

Управління розкладом та журналом записів:

– адміністратор керує потоком клієнтів через інтерактивну календарну сітку (журнал), де візити відображаються як блоки з абсолютним позиціонуванням залежно від часу початку та тривалості;

– система безперервно відстежує розклад на предмет колізій: неможливо створити або перенести запис на слот, який перекривається з іншим візитом або кастомною перервою майстра (schedule exception);

– адміністратор може оперативно змінювати статуси записів (наприклад, позначати як «завершено» або «скасовано») чи створювати нові бронювання безпосередньо з інтерфейсу сітки.

Управління портфоліо та послугами:

– адміністратор має повний доступ до редагування каталогу послуг (зміна вартості, тривалості, прив’язка до майстрів);

– майстри (Barbers) можуть завантажувати фотографії виконаних робіт до свого особистого портфоліо;

– система управління медіафайлами автоматично обробляє зображення та прив'язує їх до профілів відповідних спеціалістів для демонстрації клієнтам.

2.2.2 Логіка роботи системи

Функціонування платформи «Velvet Razor» базується на архітектурних засадах, що гарантують стабільність, цілісність даних та зручність користування:

– принцип динамічного розрахунку розкладу: система не зберігає в базі даних статичні "вільні слоти". Натомість доступний час генерується «на льоту» шляхом складного алгоритмічного накладання базового графіка майстра, його кастомних вихідних (або змінених годин роботи) та вже заброньованих іншими клієнтами інтервалів. Тривалість вікна автоматично підлаштовується під сумарний час усіх обраних клієнтом послуг;

– принцип превентивного вирішення конфліктів (Concurrency Control): бекенд використовує строгу транзакційну перевірку для унеможливлення подвійних бронювань (накладок). У разі спроби одночасного запису двох клієнтів на один і той самий слот, система безпечно відхилить одну з транзакцій, зберігши консистентність розкладу;

– принцип безпеки та конфіденційності: доступ до інформації суворо обмежений механізмами автентифікації на базі JWT-токенів, які зберігаються у захищених HTTP-only cookies для запобігання XSS-атакам. Паролі піддаються незворотному криптографічному хешуванню перед записом у БД;

– принцип суворого рольового розмежування (RBAC): архітектура передбачає чітку ієрархію прав (Admin, Barber, Client). Адміністратор має повний контроль над закладом, майстер бачить лише власний графік та

клієнтів, а звичайний користувач отримує доступ виключно до особистої історії візитів [20];

– принцип ефективного реактивного керування станом (State management): використання інструменту Zustand на стороні клієнта дозволяє кешувати вибір користувача на кожному етапі багатокрокового бронювання (вибір локації, майстра, послуги). Це мінімізує кількість зайвих запитів до сервера та забезпечує абсолютну «плавність» роботи інтерфейсу.

2.3 Вибір мови програмування та інструментарію для розробки програми

2.3.1 Мова програмування

Для розробки системи «Velvet Razor» обрано TypeScript як єдину мову для клієнтської (Next.js) та серверної (NestJS) частин [21]. Таке рішення (Full-stack TypeScript) обґрунтоване низкою стратегічних переваг:

– наскрізна типізація (End-to-end type safety) та універсальність: використання однієї мови в межах монорепозиторію дозволяє ділити спільний пакет типів та Zod-схем валідації. Типи бази даних (від Prisma) автоматично передаються безпосередньо у фронтенд-компоненти, повністю виключаючи ризик неузгодженості API між клієнтом та сервером;

– надійність: сувора статична типізація виявляє логічні помилки та відсутні параметри ще на етапі написання коду. Це є критично важливим фактором при розробці складних алгоритмів перевірки часових колізій та розрахунку вартості послуг;

– продуктивність та асинхронність: високоефективна подієва архітектура Node.js гарантує швидку обробку паралельних запитів на бронювання без блокування потоку виконання (відгук до 0,3 с). Водночас Next.js забезпечує миттєве завантаження інтерфейсу завдяки серверним компонентам (RSC);

– екосистема: доступ до найбільшого у світі реєстру пакетів (npm) дозволяє легко інтегрувати потужні інструменти (Prisma ORM, Tailwind CSS, Zustand) і сфокусуватися виключно на написанні бізнес-логіки.

Поряд із суттєвими перевагами існують певні обмеження та виклики під час розробки:

– високий поріг входження: для ефективної роботи потрібне глибоке розуміння архітектурних патернів (наприклад, Dependency Injection у NestJS) та принципів функціонування монорепозиторіїв (Turborepo);

– час компіляції: перевірка типів компілятором TypeScript та подальша транспіляція коду у звичайний JavaScript вимагають додаткових апаратних ресурсів і часу під час збірки проєкту (CI/CD);

– об’єм коду: необхідність попередньо описувати інтерфейси, DTO та типи збільшує початковий обсяг написаного коду порівняно з розробкою на динамічно типізованому JavaScript.

Порівняльна характеристика популярних мов програмування для створення вебплатформ наведена в таблиці 2.1.

Таблиця 2.1 – Порівняльний аналіз інструментаріїв розробки

Критерій	TypeScript (Next.js/NestJS)	Python (Django/FastAPI)	PHP (Laravel)
Типізація	Строга (статична)	Динамічна / Опціональна	Динамічна / Опціональна
Швидкість Full-stack розробки	Дуже висока (спільний код)	Середня (різні мови)	Середня (різні мови)
Наскрізна перевірка типів (E2E)	Відмінна (через монорепозиторій)	Ускладнена	Ускладнена
Синтаксис	Сучасний, С-подібний	Дуже простий, лаконічний	Класичний серверний
Екосистема	Найбільша у світі (npm)	Велика	Велика
Масштабованість архітектури	Висока (мікросервіси/модулі)	Середня	Середня
Відгук інтерфейсу	< 0,4 с	0,5 – 0,8 с	0,5 – 0,9 с

2.3.2 Середовище розробки

Visual Studio Code (VS Code) від корпорації Microsoft обрано основним інтегрованим середовищем розробки (IDE) системи завдяки його бездоганній швидкодії, гнучкості та нативній сумісності з обраним стеком технологій. VS Code виступає як ідеально збалансоване Full-stack рішення, що поєднує легкість сучасного текстового редактора з потужністю повноцінної IDE.

Ключові чинники вибору та вбудовані інструменти, що забезпечують високу продуктивність розробки:

– нативна підтримка TypeScript та IntelliSense: оскільки мова та редактор розроблені однією компанією, VS Code забезпечує миттєву перевірку типів та розумне автодоповнення коду з урахуванням експортованих модулів монорепозиторію;

– екосистема розширень та робота з БД: наявність спеціалізованих офіційних плагінів для роботи з класами Tailwind CSS та синтаксисом Prisma (забезпечує підсвічування, автоматичне форматування та візуалізацію зв'язків у файлі schema.prisma);

– інтегрований термінал та Debugging: зручне керування робочими скриптами (наприклад, одночасний запуск NestJS-сервера та Next.js-клієнта через команду turbo dev в одному вікні) та наявність потужного дебаггера для покрокового аналізу логіки Node.js у реальному часі;

– інструменти якості коду (ESLint/Prettier) та контроль версій: автоматичне підтримання єдиного стилю написання коду в усій системі та вбудовані інструменти інтеграції з Git для вирішення конфліктів і фіксації змін;

– продуктивність та кастомізація: низьке споживання системних ресурсів порівняно з іншими IDE (навіть при роботі з об'ємними проєктами), а також можливість гнучкої персоналізації робочого простору і створення власних сніпетів (snippets).

Порівняння популярних середовищ для веброзробки представлено в таблиці 2.2.

Таблиця 2.2 – Порівняльна характеристика інструментів розробки

Критерій	Visual Studio Code	WebStorm (JetBrains)	Sublime Text
Підтримка TypeScript	Повна (нативна)	Повна (IDE-рівень)	Базова (через плагіни)
Споживання ресурсів	Низьке	Високе	Дуже низьке
Інтегрований термінал	Вбудований	Вбудований	Відсутній (через плагіни)
Вартість ліцензії	Безкоштовно (Open Source)	Платно (комерційна)	Умовно-безкоштовно
Екосистема плагінів	Найбільша на ринку	Замінена вбудованими функціями	Обмежена
Підтримка монорепозиторіїв	Відмінна	Відмінна	Слабка

Visual Studio Code є об'єктивно найоптимальнішим вибором для реалізації архітектурно складного проєкту «Velvet Razor», оскільки він пропонує бездоганну підтримку обраного технологічного стеку (TypeScript, Next.js, Prisma) при мінімальних вимогах до апаратного забезпечення комп'ютера розробника.

2.3.3 База даних

Для стабільного функціонування платформи «Velvet Razor», що передбачає складні багаторівневі зв’язки між графіками майстрів, каталогом

послуг та фінансовою історією клієнтів, було обрано реляційну модель збереження даних. Основним технологічним стеком для роботи з інформацією стали PostgreSQL як серверна база даних та Prisma як інструмент об’єктно-реляційного відображення (ORM) [22]-[23].

2.3.3.1 Реляційна база даних PostgreSQL

PostgreSQL є найдосконалішою системою керування реляційними базами даних з відкритим кодом, яка забезпечує абсолютну надійність та цілісність інформації. У проєкті «Velvet Razor» вона виступає централізованим сховищем (Single Source of Truth).

Ключові переваги:

– надійність та ACID-відповідність: гарантує строгу консистентність виконання транзакцій, що є критичним фактором при одночасному бронюванні одного часового слоту різними клієнтами (запобігання подвійним записам);

– підтримка складних зв’язків: реляційна архітектура ідеально підходить для реалізації специфічних доменних структур (Клієнт – Запис – Список послуг – Майстер – Кастомні перерви);

– висока продуктивність: здатність швидко обробляти складні запити з об'єднаннями (JOIN) під час алгоритмічного пошуку вільного часу в розкладі барбершопу.

Обмеження: потребує налаштованої серверної інфраструктури (у нашому випадку вирішено за допомогою контейнеризації Docker) та постійного мережевого з'єднання. Проте ці вимоги цілком виправдовуються безперебійною роботою системи та можливістю її легкого масштабування при збільшенні кількості клієнтів.

2.3.3.2 Об’єктно-реляційне відображення (Prisma ORM)

Prisma використовується як сучасний, суворо типізований шар абстракції між кодом бізнес-логіки (NestJS) та базою даних PostgreSQL.

Переваги використання Prisma:

– типобезпека (Type-safety): автоматична генерація TypeScript-типів на основі схеми БД schema.prisma повністю виключає помилки невідповідності типів даних у бекенді;

– декларативні міграції: спрощує процес еволюції структури бази даних (наприклад, додавання нових полів до профілю клієнта), дозволяючи надійно відстежувати зміни в системі Git;

– синергія екосистеми: використання генератора prisma-zod-generator дозволяє автоматично створювати схеми валідації на основі моделей бази даних, які потім використовуються як на сервері, так і на клієнті для валідації форм.

Для порівняльного аналізу було розглянуто альтернативні варіанти, такі як NoSQL рішення (MongoDB) та легковагові локальні бази (SQLite). Хоча MongoDB пропонує гнучкість безсхемного підходу, вона значно поступається у надійності при транзакційному забезпеченні складних реляційних зв’язків, які є основою систем бронювання. SQLite, своєю чергою, не підходить для багатокористувацької вебсистеми через обмеження у конкурентному доступі на запис. Таким чином, комбінація PostgreSQL та Prisma визначена як єдине правильне архітектурне рішення.

Порівняння різних систем керування даними наведено в таблиці 2.3.

Таблиця 2.3 – Порівняльний аналіз систем керування базами даних

Критерій	PostgreSQL (з Prisma)	MongoDB	SQLite
1	2	3	4
Тип моделі	Реляційна (SQL)	Документо-орієнтована (NoSQL)	Реляційна (файлова)

Кінець табліці 2.3

1	2	3	4
Цілісність даних	Дуже висока (ACID)	Базова (eventual consistency)	Висока (локальна)
Транзакційна безпека бронювань	Гарантована	Ускладнена	Обмежена (блокування файлу)
Масштабованість	Висока (вертикальна/горизонтальна)	Дуже висока	Низька
Типізація у коді	Сувора (через Prisma Client)	Відсутня (динамічна)	Базова
Складність зв'язків (JOINs)	Будь-яка складність	Вкрай ускладнена	Обмежена продуктивністю

Використання PostgreSQL у парі з Prisma ORM забезпечує платформі «Velvet Razor» необхідний рівень транзакційної надійності, безпрецедентну швидкість розробки завдяки автоматизації типів та гарантує математичну цілісність даних під час онлайн-запису клієнтів.

2.4 Вимоги до технічних і програмних засобів

2.4.1 Необхідні технічні засобів для розробки та експлуатації

Для успішної побудови, тестування та подальшої експлуатації платформи «Velvet Razor» необхідний наступний комплекс технічних засобів:

– робоча станція розробника: комп'ютер з достатньою обчислювальною потужністю для одночасного запуску серверної (NestJS) та клієнтської (Next.js) частин застосунку в режимі монорепозиторію, а також віртуалізації бази даних;

– серверна інфраструктура: хостинг або хмарний VPS-сервер із підтримкою технології Docker для розгортання PostgreSQL та виконання серверного коду на базі Node.js в ізольованих контейнерах;

– клієнтські пристрої: будь-які пристрої з сучасними веббраузерами (смартфони, планшети, ноутбуки, десктопи) для взаємодії з системою онлайн-бронювання та перевірки адаптивності інтерфейсу (mobile-first);

– мережеве з’єднання: стабільний канал зв'язку для взаємодії клієнтської частини з REST API сервером та гарантованого підтвердження транзакцій запису.

2.4.2 Вимоги до апаратного та програмного забезпечення

Апаратні параметри для комфортної розробки (мінімальні):

– процесор: багатоядерний процесор рівня Intel Core i5 (8-го покоління або новіше), AMD Ryzen 5 або Apple Silicon (M1/M2/M3) для швидкої транспіляції TypeScript-коду та обробки збірок інструментом Turborepo;

– оперативна пам'ять: не менше 8 Гб (наполегливо рекомендовано 16 Гб) для забезпечення стабільної роботи Docker-контейнерів, локального сервера бази даних та декількох паралельно запущених екземплярів Node.js;

– дисковий простір: від 5 Гб вільного місця на SSD-накопичувачі для встановлення середовища Node.js, локального кешу пакетів (node_modules у монорепозиторії) та зберігання Docker-образів.

Програмне забезпечення:

– операційна система: macOS (12.0+), Windows (10/11) з обов'язковою підтримкою підсистеми WSL2 (Windows Subsystem for Linux) або сучасні дистрибутиви Linux (Debian/Ubuntu);

– середовище розробки: Visual Studio Code з актуальними офіційними розширеннями для TypeScript, Tailwind CSS та Prisma;

– середовище виконання: Node.js версії 18.x (або новішої) та менеджер пакетів npm;

– система керування базами даних: PostgreSQL версії 14.x або вище;

– додаткові інструменти: Docker Engine та інструмент оркестрації Docker Compose для багатоетапної збірки (multi-stage build) та уніфікації середовища розгортання в production.

2.5 Висновки за розділом 2

У межах другого розділу було детально спроєктовано архітектуру вебсистеми «Velvet Razor» та визначено механізми взаємодії її ключових модулів. Обрана модель Full-stack розробки на базі мови TypeScript з використанням архітектурного підходу монорепозиторію дозволила створити цілісну, суворо типізовану екосистему, де Next.js відповідає за швидкий інтерактивний інтерфейс, а NestJS – за стабільну серверну логіку та обробку алгоритмів розкладу.

Використання Visual Studio Code як основного інструменту розробки обґрунтовано його нативною підтримкою обраного стеку, що значно прискорює цикл написання та налагодження коду. Впровадження реляційної бази даних PostgreSQL у поєднанні з Prisma ORM забезпечило необхідну для систем бронювання транзакційну надійність збереження інформації та високу швидкість виконання складних об'єднань при пошуку вільних слотів. Реалізація динамічного розрахунку вільного часу майстрів та алгоритмів запобігання конфліктам (concurrency control) стала ключовим елементом для забезпечення безвідмовної роботи онлайн-запису. Запропонована багатошарова архітектура повністю відповідає сучасним індустріальним вимогам щодо продуктивності, безпеки та масштабованості платформ для малого бізнесу.

3 РОЗРОБКА ПРОГРАМИ

3.1 Архітектура системи

Архітектура вебсистеми «Velvet Razor» побудована на базі сучасної клієнт-серверної моделі з використанням передового архітектурного підходу «монорепозиторій» (Monorepo), керованого інструментом Turborepo. Такий підхід забезпечує не лише чітке розділення бізнес-логіки, представлення даних та взаємодії з базою даних, але й дозволяє глибоко інтегрувати фронтенд та бекенд через спільні контракти даних. Це сприяє високій продуктивності, абсолютній типобезпеці, масштабованості та значній зручності при подальшій підтримці системи, усуваючи ризики розсинхронізації між клієнтом та сервером.

Основні компоненти архітектури логічно поділено на кілька взаємопов'язаних рівнів.

Data Layer (Рівень даних / Модель) – відповідає за структуру збереження інформації та безпечну транзакційну взаємодію з реляційною базою даних (PostgreSQL):

– класи моделей даних, задекларовані через схему Prisma (користувачі: User/Master/Client, бронювання: Appointment, послуги: Service, локації: Location, виключення графіка: ScheduleException);

– рівень доступу до даних (Prisma Client), що забезпечує виконання оптимізованих SQL-запитів, обробку конкурентних блокувань та суворе дотримання цілісності даних на рівні зовнішніх ключів.

Shared Layer (Рівень спільних даних) – унікальний архітектурний прошарок монорепозиторію, що виключає дублювання коду. Він містить автоматично згенеровані Zod-схеми валідації та TypeScript-інтерфейси (на базі моделей БД), які імпортуються як у клієнтські форми, так і в серверні DTO для наскрізної перевірки даних (End-to-end type safety).

Presentation Layer (Рівень представлення / View) – відповідає за візуалізацію інтерфейсу та зручну (реактивну) взаємодію з різними типами користувачів (фронтенд на базі Next.js):

– сторінки та макети маршрутизатора App Router (панель адміністратора Admin Dashboard, сторінка онлайн-запису Booking Wizard, особистий кабінет клієнта);

– складні інтерактивні клієнтські компоненти (абсолютно позиційована сітка розкладу CalendarGrid, багатокрокові модальні вікна AppointmentModal, компоненти вибору послуг);

– UI-система на базі Tailwind CSS для забезпечення єдиного, адаптивного та сучасного корпоративного стилю.

Logic Layer (Рівень управління / Controller) – з’єднує інтерфейс із бізнес-даними та обробляє мережеві запити:

– контролери бекенду NestJS, що приймають REST HTTP-запити, проводять авторизацію через Guards та валідують вхідні дані (AuthController, AppointmentsController, ScheduleController);

– клієнтські провайдери стану (Zustand stores) та спеціалізовані React-хуки для управління локальним станом на стороні фронтенду (наприклад, збереження проміжних даних під час багатокрокового бронювання без зайвих звернень до сервера).

Services (Сервіси бізнес-логіки) – ядро системи, де реалізується специфічна бізнес-функціональність для всіх модулів застосунку:

– AuthService – багаторівнева автентифікація користувачів (JWT, HTTP-only cookies) та суворе розмежування прав доступу (RBAC);

– PrismaService – централізована робота з базою даних та управління пулом з'єднань;

– AppointmentsService – складне управління життєвим циклом записів, перевірка часових колізій (conflict detection), розрахунок загальної вартості та тривалості збірних послуг;

– ScheduleService – адміністрування індивідуальних робочих графіків майстрів та генерація доступних часових слотів з урахуванням кастомних вихідних;

– UsersService – управління клієнтською базою, майстрами та їхніми специфічними статусами (вимога передоплати, блокування);

– ServicesService – управління каталогом доступних процедур та створення нових категорій послуг.

Схема взаємодії компонентів системи зображена на рисунку 3.1.

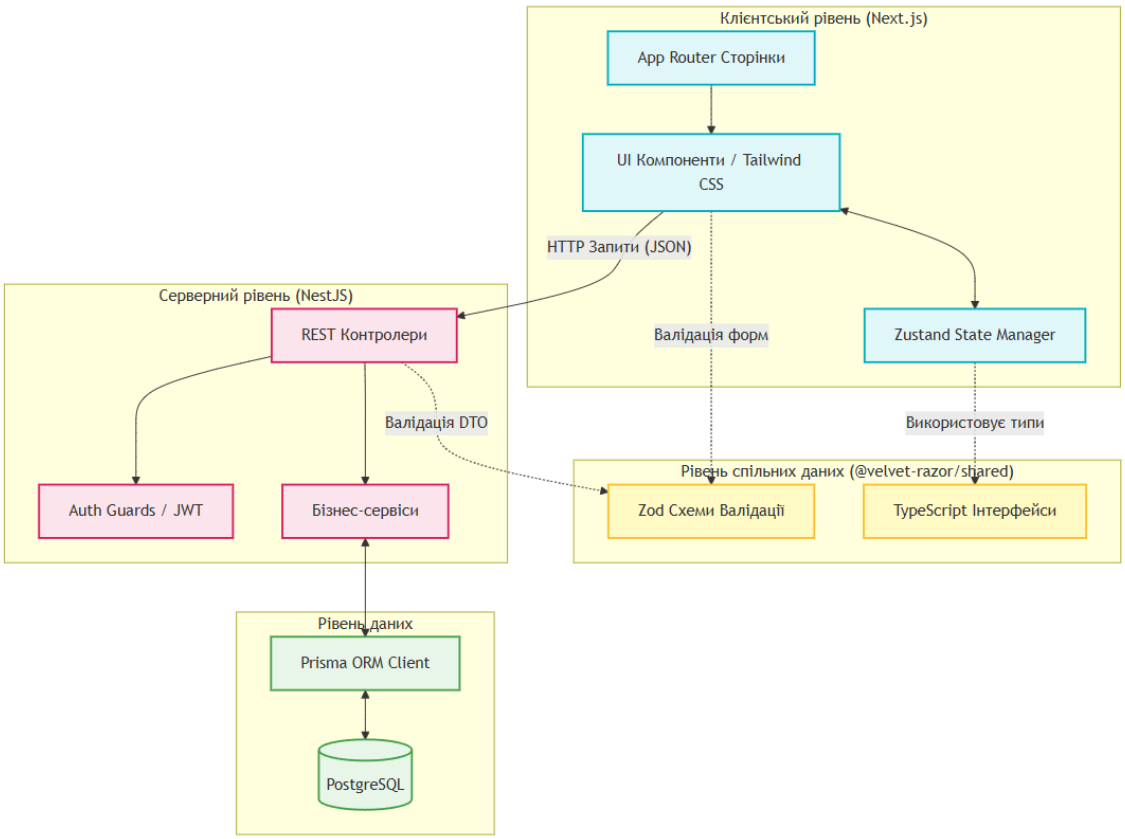

Рисунок 3.1 – Архітектурна схема взаємодії компонентів системи «Velvet Razor»

3.1.1 Структурна схема системи

Розробка проєкту «Velvet Razor» здійснювалася за принципом глибокої модульної декомпозиції, реалізованої на рівні монорепозиторію.

Програмний код логічно та фізично розподілено на ізольовані блоки (застосунки та спільні пакети) відповідно до їхнього функціонального призначення.

Такий архітектурний підхід дозволяє забезпечити надзвичайно низьку залежність між компонентами системи (low coupling) та високу спеціалізацію кожного модуля (high cohesion). Зокрема, винесення бази даних та типів валідації в окремі незалежні пакети значно спрощує процес тестування, підтримки та подальшого масштабування застосунку під нові бізнес-вимоги, оскільки зміна логіки в одному модулі мінімально впливає на інші.

Структурну схему системи, що відображає загальну ієрархію монорепозиторію та внутрішній склад основних модулів клієнтської і серверної частин, зображено на рисунку 3.2.

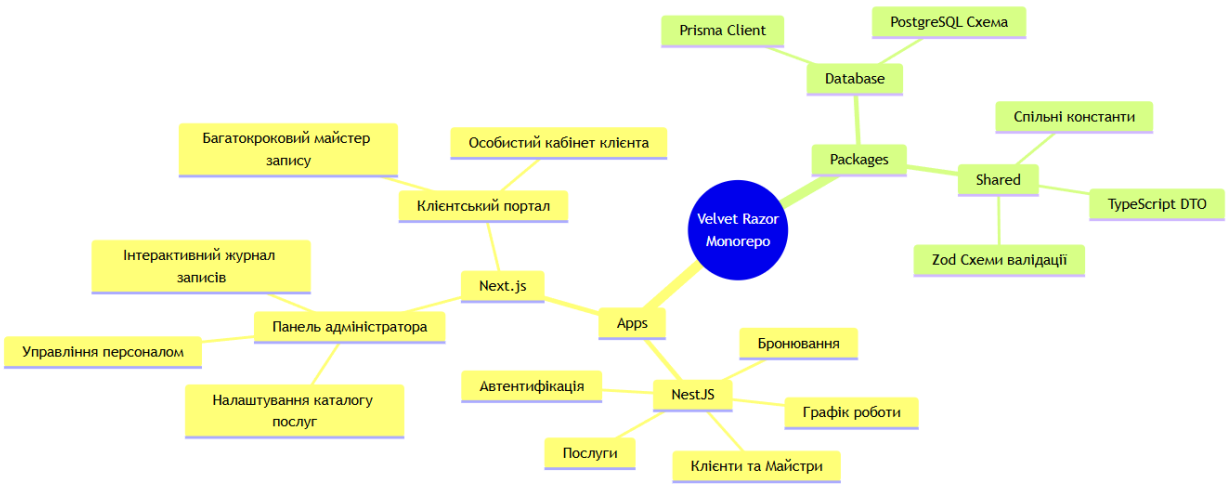

Рисунок 3.2 – Структурна схема монорепозиторію та модулів системи «Velvet Razor»

3.1.2 Організація монорепозиторію та розгортання

Для забезпечення високої швидкості розробки та абсолютної узгодженості типів між клієнтською та серверною частинами, у проєкті «Velvet Razor» застосовано архітектуру монорепозиторію, оркестрація якого здійснюється інструментом Turborepo. Це рішення дозволяє об'єднати

незалежні застосунки (фронтенд і бекенд) та спільні бібліотеки (база даних, схеми валідації) в єдину кодову базу, забезпечуючи при цьому інтелектуальне кешування завдань збірки (build caching).

Ключовим елементом такої організації є розроблений пакет спільного коду (@velvet-razor/shared), що містить автоматично згенеровані Zod-схеми на основі моделей Prisma. Завдяки цьому будь-яка зміна структури бази даних (наприклад, додавання нового поля до профілю майстра) миттєво оновлює вимоги до валідації як у NestJS-контролерах, так і в React-формах клієнта, унеможливлюючи виникнення помилок інтеграції.

Процес розгортання системи (Deployment) повністю стандартизовано за допомогою технологій контейнеризації Docker. Для створення образів застосовано метод багатоетапної збірки (Multi-stage builds), що дозволяє суттєво зменшити розмір кінцевих контейнерів, залишаючи в них виключно скомпільований JavaScript-код та необхідні production-залежності, без важких інструментів розробки.

Управління інфраструктурою здійснюється через docker-compose, який контролює життєвий цикл та внутрішню мережу чотирьох основних сервісів:

– postgres: сервер бази даних PostgreSQL;

– migrate: спеціалізований тимчасовий контейнер, який відповідає за безпечне застосування міграцій Prisma (db push) та наповнення бази початковими даними (seed) до моменту запуску основних застосунків;

– backend: серверна частина системи (NestJS), що обробляє бізнес-логіку;

– frontend: клієнтська частина (Next.js), що віддає інтерфейс користувачам.

Схему оркестрації контейнерів під час розгортання системи зображено на рисунку 3.3.

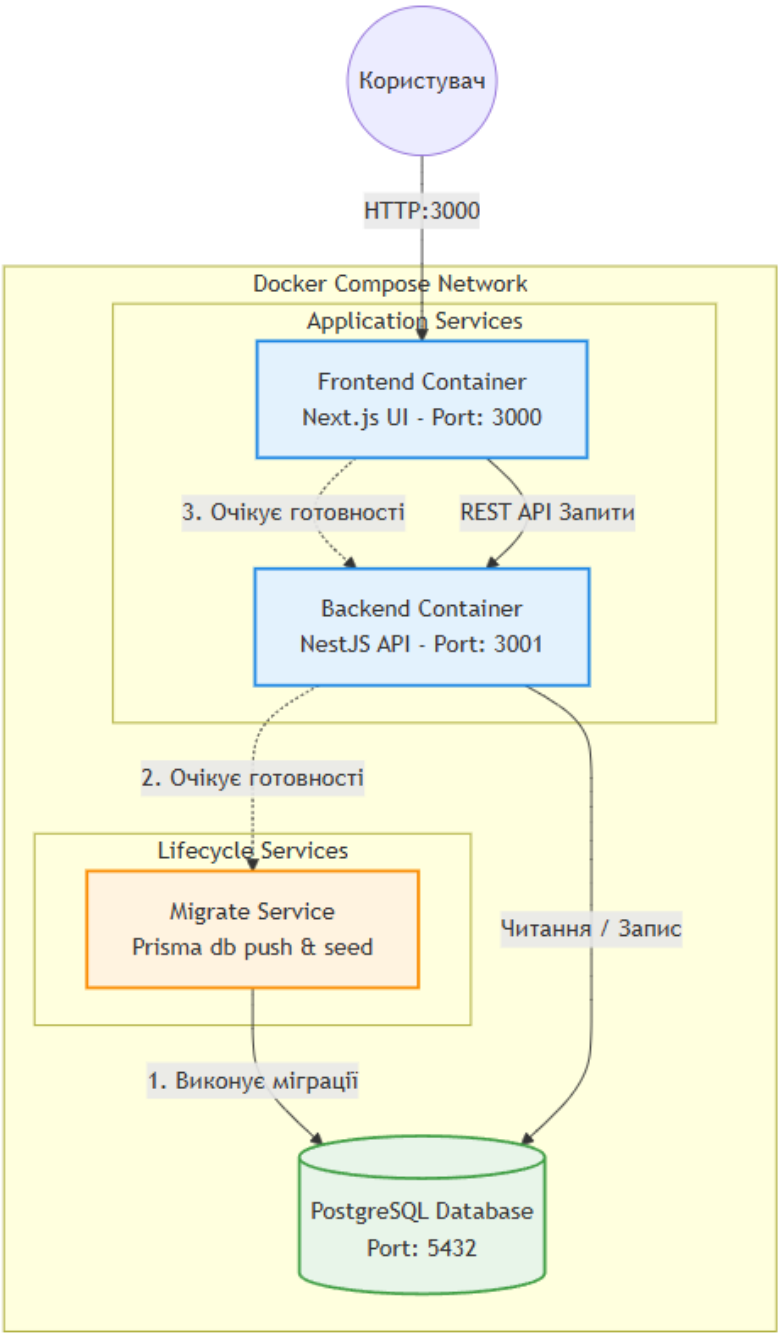

Рисунок 3.3 – Модель розгортання та взаємодії контейнерів системи «Velvet Razor»

3.2 Функціонування системи

3.2.1 Функціональна схема

Функціонування платформи «Velvet Razor» базується на послідовній взаємодії низки інтелектуальних алгоритмів, що забезпечують безперебійне

управління життєвим циклом записів та оперативне адміністрування робочого процесу барбершопу. Система підтримує стабільну роботу в інтенсивному багатокористувацькому режимі, суворо гарантуючи транзакційну цілісність даних та унеможливлюючи подвійні бронювання (накладки) завдяки вбудованим механізмам алгоритмічної перевірки колізій. Функціональні модулі платформи охоплюють повний спектр процесів: від первинної автентифікації та багаторівневого розмежування прав доступу до надскладних операцій із динамічного розрахунку доступних часових слотів та гнучкого управління графіками персоналу. Загальну функціональну схему розробленої системи представлено на рисунку 3.4.

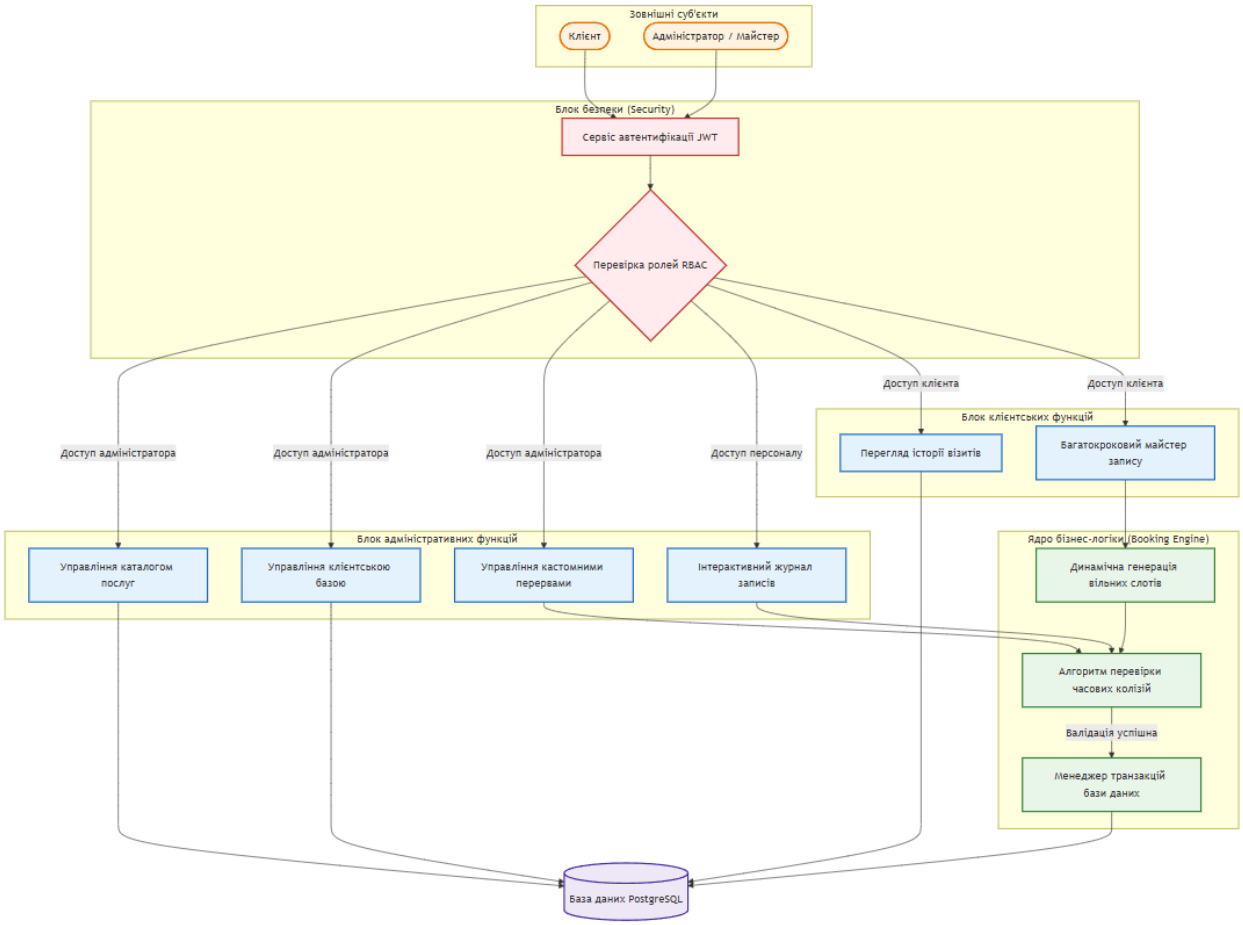

Рисунок 3.4 – Функціональна схема застосунку «Velvet Razor»

3.2.2 Опис процесу обміну даними у системі

Архітектура потоків даних у «Velvet Razor» реалізована на засадах REST-взаємодії. Обмін інформацією відбувається за циклом, що гарантує транзакційну цілісність даних на сервері та їх актуальність у клієнтському інтерфейсі (рис. 3.5):

– ініціація дії: через інтерфейс (Next.js) користувач виконує цільову операцію (створення бронювання, коригування розкладу тощо);

– обробка запиту та валідація: дані передаються на бекенд (NestJS), де проходять сувору перевірку типів (через спільні Zod-схеми) та перевірку прав доступу (JWT Guards);

– персистентність: після алгоритмічної перевірки (наприклад, на відсутність часових колізій), дані транзакційно зберігаються в базі PostgreSQL за допомогою Prisma ORM;

– реактивне оновлення: після успішної відповіді сервера спрацьовує менеджер стану (Zustand), автоматично оновлюючи інтерфейс користувача без перезавантаження сторінки.

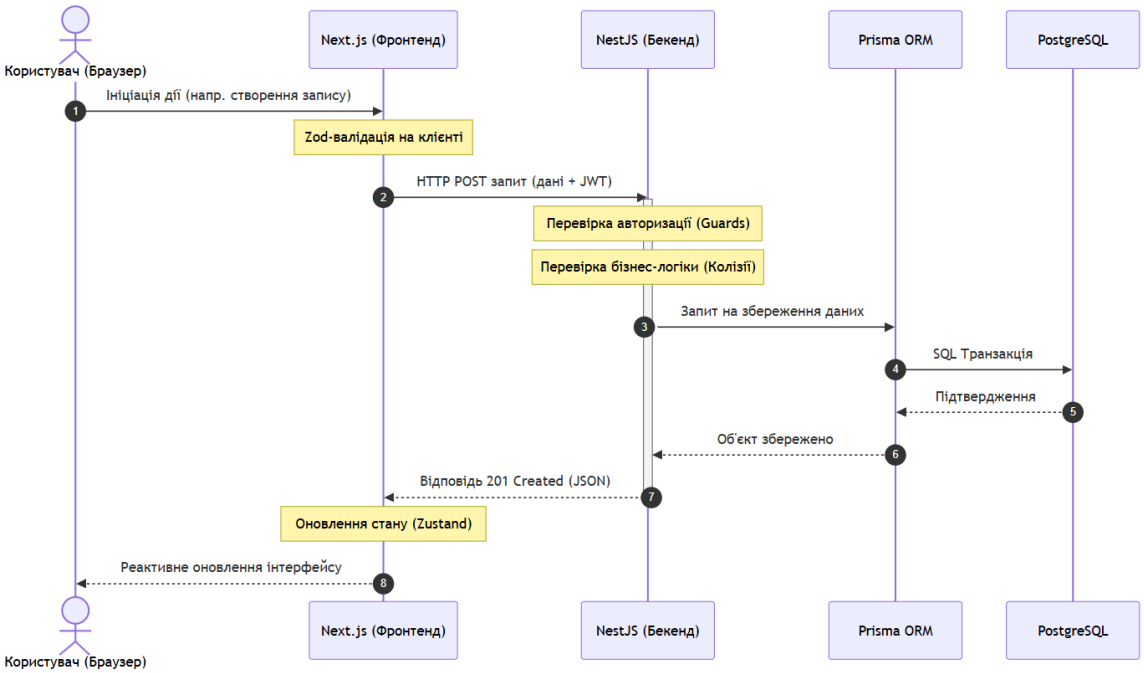

Рисунок 3.5 – Схема процесу обміну даними у системі

3.3 Проєктування бази даних

3.3.1 Структура бази даних

База даних вебзастосунку «Velvet Razor» побудована на реляційній моделі та реалізована за допомогою СУБД PostgreSQL. Для взаємодії з даними використовується сучасний інструмент Prisma ORM, що забезпечує наскрізну типізацію запитів та значно спрощує механізм декларативних міграцій. Архітектура бази даних розроблена з урахуванням необхідності надшвидкого алгоритмічного пошуку вільних слотів (через використання складених індексів на дати та ідентифікатори майстрів) та забезпечення суворої транзакційної цілісності при конкурентному бронюванні.

Основними сутностями системи є користувачі (User), спеціалізовані профілі майстрів (Barber), каталог послуг (Service), локації (Outlet), самі бронювання (Appointment), параметри робочого графіка (WorkSchedule, ScheduleException) та аудит змін (BookingHistory). Кожна сутність відповідає за конкретний аспект бізнес-логіки: від безпечної автентифікації до відстеження історії зміни статусів (наприклад, скасування чи неявки).

3.3.2 Взаємозв'язки між таблицями

Взаємозв'язки в системі побудовані на основі доменної логіки управління розкладом та складними послугами:

– User – Barber: зв'язок один-до-одного (1:1), де профіль майстра (Barber) розширює базові авторизаційні дані користувача специфічними полями (рівень майстерності, портфоліо);

– Outlet – Barber: зв'язок один-до-багатьох (1:N), що закріплює майстрів за конкретними філіями (локаціями) закладу;

– Barber – Service: складний зв'язок багато-до-багатьох (N:M) через проміжну таблицю BarberService, що дозволяє кастомізувати тривалість та ціну однієї й тієї ж послуги для кожного майстра індивідуально;

– User – Appointment: зв'язок один-до-багатьох (1:N), що фіксує всю історію візитів конкретного клієнта;

– Barber – Appointment: зв'язок один-до-багатьох (1:N) для формування журналу записів майстра та обчислення його зайнятості;

– Appointment – Service: зв'язок багато-до-багатьох (N:M) через AppointmentService, оскільки один візит клієнта може включати набір з декількох різних послуг (наприклад, стрижка + борода);

– Appointment – BookingHistory: зв'язок один-до-багатьох (1:N) для строгого аудиту зміни статусів запису (від створення до підтвердження чи скасування), включаючи фіксацію того, хто саме змінив статус;

– Barber – WorkSchedule / ScheduleException: зв'язки один-до-багатьох (1:N) для гнучкого управління базовим графіком роботи (по днях тижня) та кастомними перервами чи відпустками.

Схему взаємозв'язків між таблицями зображено за допомогою діаграми на рисунку 3.6.

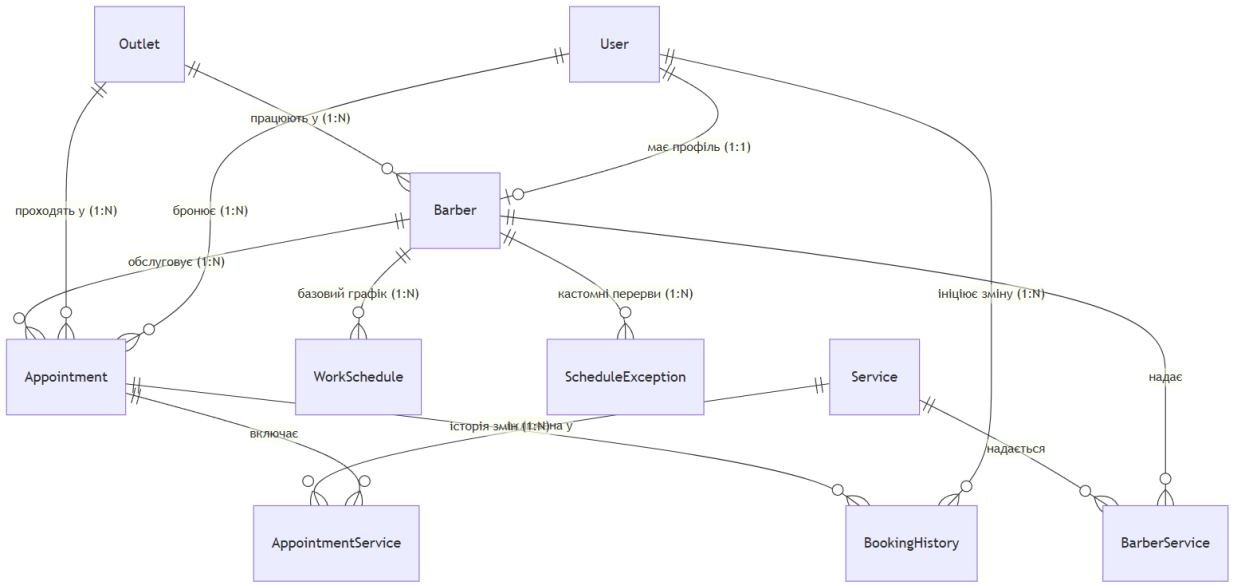

Рисунок 3.6 – Схема взаємозв'язків (ER-діаграма) між таблицями бази даних «Velvet Razor»

3.4 Висновки за розділом 3

У результаті виконання третього розділу детально описано процес проєктування та технічної реалізації багатокористувацької платформи управління барбершопом «Velvet Razor». Розроблено ключові алгоритми системи: механізми безпечної автентифікації, математичні алгоритми розрахунку доступних часових слотів, методи транзакційного запобігання колізіям під час конкурентного бронювання та логіку позиційованого рендерингу адміністративного журналу записів.

Архітектура комплексу побудована на базі клієнт-серверної моделі в монорепозиторії з чітким розподілом обов'язків. Серверна частина реалізована на NestJS із застосуванням модульного підходу, а клієнтська – на Next.js (React) із використанням Zustand для реактивного управління станом. Така структура забезпечує високу швидкість обробки даних, масштабованість системи та зручність її подальшого супроводу.

Розроблено та реалізовано сім основних логічних модулів системи.

Модуль реєстрації та автентифікації (AuthModule) – забезпечує безпечний доступ через механізми JWT-автентифікації та хешування паролів за допомогою Argon2. Екран входу автоматично перенаправляє користувача до робочого простору залежно від ролі. Фрагменти реалізації серверної та клієнтської частин процесу автентифікації наведено у Додатку Б.1.

Модуль управління бронюваннями (AppointmentsModule) – ядро системи, що дозволяє клієнтам обирати послуги та час візиту з проведенням серверної перевірки часових колізій. Математичний алгоритм пошуку вільних часових слотів з урахуванням буферного часу між послугами та кастомних вихідних майстра наведено у Додатку Б.2. Логіку транзакційного збереження запису в базі даних (з рівнем ізоляції Serializable) для унеможливлення подвійного бронювання наведено у Додатку Б.3.

Модуль управління графіком (ScheduleModule) – дозволяє майстрам та адміністраторам налаштовувати робочі дні та встановлювати кастомні

перерви (Schedule Exceptions). Модуль клієнтської бази та персоналу (UsersModule) дозволяє керувати профілями, призначати рівні майстерності та встановлювати специфічні статуси довіри клієнтам. Модуль управління каталогом послуг (ServicesModule) відповідає за налаштування вартості, базової тривалості та буферного часу послуг.

Модуль інтерактивного журналу (Admin Dashboard) – адміністративний інтерфейс для управління потоком клієнтів. Він рендерить записи у вигляді блоків із розрахунком їхньої абсолютної позиції по висоті залежно від часу. Алгоритм абсолютного позиціонування карток у сітці календаря наведено у Додатку Б.4.

Модуль наскрізної типізації (Shared Zod) – унікальний прошарок монорепозиторію, що автоматично генерує схеми валідації на основі моделей БД для клієнта та сервера. Будь-яка зміна структури бази даних миттєво оновлює вимоги до валідації як у NestJS-контролерах, так і в React-формах клієнта.

Особливу увагу приділено проєктуванню бази даних на основі PostgreSQL. Визначено структуру таблиць, складні зв'язки «один-до-багатьох» та «багато-до-багатьох» (для збірних послуг), а також впроваджено систему складених індексів для оптимізації пошукових запитів за датами та майстрами. Безпека системи гарантується використанням захищених HttpOnly-cookies, наскрізною валідацією даних через Zod-DTO на рівні API та суворим розмежуванням прав доступу на основі ролей (RBAC).

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Опис застосування програми

Вебзастосунок «Velvet Razor» – це спеціалізована CRM-система та платформа для онлайн-бронювання послуг, розроблена для радикальної оптимізації робочих процесів сучасних барбершопів. Програма дозволяє автоматизувати ведення розкладу майстрів, гнучко управляти каталогом процедур, відстежувати статус візитів та надавати клієнтам зручний багатокроковий інтерфейс для самостійного запису з мобільних пристроїв чи десктопів.

Застосунок охоплює ключові аспекти управління салоном: від високорівневого налаштування базових робочих графіків (включаючи кастомні перерви та відпустки) до детального адміністрування щоденного потоку відвідувачів через інтерактивну календарну сітку (журнал). Основний функціонал включає управління профілями майстрів, ведення бази клієнтів (з можливістю встановлення статусу довіри), алгоритмічне запобігання накладкам у розкладі та безпечну автентифікацію.

Цільова аудиторія застосунку є гібридною (B2B та B2C) і охоплює: власників барбершопів, адміністраторів, індивідуальних майстрів (Barbers), для яких критично важливою є прозорість процесів та цифровізація обліку, а також безпосередньо клієнтів закладу, які потребують швидкого та зручного інструменту для запису. Завдяки інтуїтивно зрозумілому та суворо розділеному за ролями інтерфейсу, система є доступною як для досвідчених менеджерів салонів, так і для пересічних користувачів, які не мають технічних навичок.

4.1.1 Огляд планової цільової аудиторії

Для оцінки потенційного ринку та користувацької бази застосунку «Velvet Razor» було проведено аналіз сучасних трендів цифрової трансформації у сфері надання послуг (зокрема б'юті-індустрії). Нижче наведено дві прогнозовані структури аудиторії: за віковими категоріями кінцевих споживачів (клієнтів) та за типами користувачів (організаційною структурою) всередині платформи. Ці дані базуються на маркетингових звітах про використання CRM-рішень у сфері обслуговування та специфіці українського ринку барбершопів, який демонструє високий рівень адаптації до цифрових інструментів.

4.1.1.1 Розподіл за віком

Прогнозований розподіл базується на статистичних даних щодо поведінки споживачів у сфері сучасних послуг (зокрема барбершопів) та цифровізації малого бізнесу. Згідно з дослідженнями, основне ядро користувачів систем онлайн-бронювання зосереджено в сегменті молоді та людей середнього віку, які цінують свій час [24].

Група 18–25 років (40%): Найбільш активний клієнтський сегмент (B2C), а також молоді майстри-початківці. Вони принципово уникають телефонних дзвінків для запису, прагнуть максимально цифровізувати рутину та надзвичайно цінують швидкість роботи інтерфейсу, зручність із мобільних пристроїв та сучасний дизайн.

Група 26–35 років (35%): Ключова категорія, що складається як із регулярних клієнтів зі стабільним доходом, так і з досвідчених барберів та менеджерів салонів (B2B). Їхній інтерес до застосунку зумовлений потребою в надійності, зручному управлінні власним часом та перегляді історії візитів.

Група 36–45 років (15%): Дорослі клієнти та безпосередньо власники бізнесу (інвестори). Вони формують суворі вимоги до надійності системи, безпеки фінансових даних та наявності прозорого адміністративного журналу.

Група 46+ років (10%): Найбільш консервативна аудиторія. Частина цих клієнтів може не користуватися онлайн-інтерфейсом самостійно, тому для них передбачено функціонал «тіньових клієнтів», яких адміністратор додає в систему вручну.

4.1.1.2 Розподіл за організаційною структурою

Прогнозована структура базується на глобальних трендах споживання нішевих SaaS-продуктів (Software as a Service), де основний драйвер ринку — малий та мікробізнес, що шукає дешевшу та швидшу альтернативу перевантаженим корпоративним CRM-системам [5].

Індивідуальні майстри / Орендарі крісел (45%): Цей сегмент є домінуючим. Сучасні майстри, які працюють на себе, гостро потребують автономного, легкого інструменту для ведення власного розкладу та клієнтської бази без сплати високих комісій маркетплейсам [25].

Локальні барбершопи / Малий бізнес (35%): Класичні салони з 3-6 робочими місцями. Для них критично важливою є координація роботи кількох майстрів, ведення єдиного інтерактивного журналу адміністратором та контроль за статусами записів (передоплата, неявка).

Мережеві барбершопи / Середній бізнес (15%): Організації, що мають декілька філій (локацій). Впровадження платформи дозволяє їм централізувати базу клієнтів та керувати розподілом майстрів між різними точками.

Салони краси широкого профілю (5%): Сегмент «пілотних проєктів», де «Velvet Razor» може використовуватися як зручна альтернатива для тестування автоматизації в суміжних б'юті-сферах завдяки гнучкому конструктору послуг [3].

4.1.2 Візуалізація планованої цільової аудиторії

На основі проведеного вище аналізу нижче представлено графічне відображення структури потенційних користувачів системи «Velvet Razor» за віковими показниками клієнтів та форматом професійної діяльності B2B-аудиторії (рис. 4.1 та 4.2).

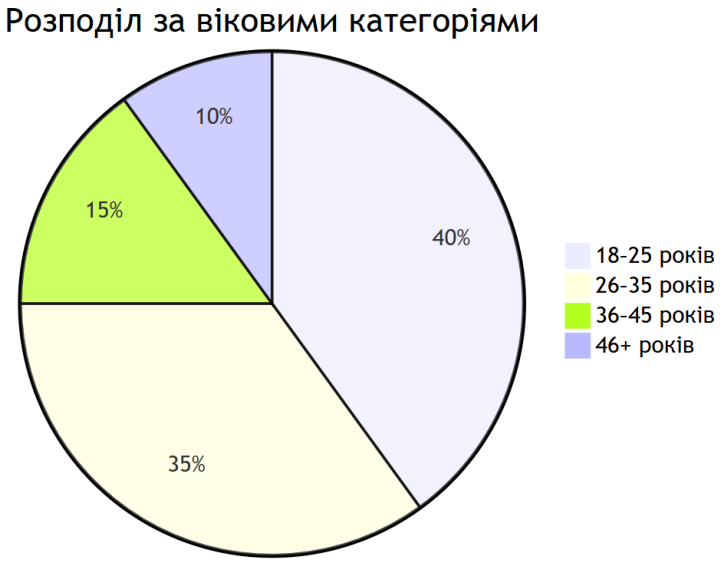

Рисунок 4.1 – Розподіл аудиторії «Velvet Razor» за віковими категоріями

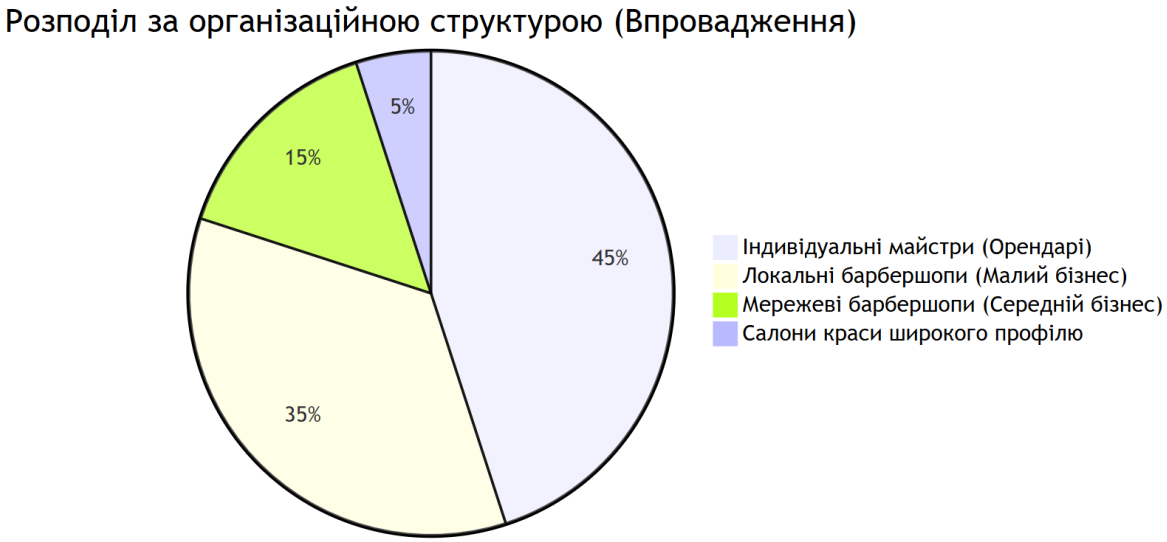

Рисунок 4.2 – Розподіл за типом організаційної структури

4.2 Інструкція по роботі з програмою

4.2.1 Встановлення програми

Оскільки «Velvet Razor» є кросплатформним вебзастосунком (на базі хмарної SaaS-архітектури), він не потребує класичного встановлення на локальний пристрій, проте для його стабільної та повноцінної роботи необхідно дотримуватися технічних вимог:

а) технічні вимоги до середовища:

1) актуальна версія браузера (Google Chrome 90+, Safari 14+, Firefox 88+);

2) мінімальна роздільна здатність екрана для клієнтів (B2C) – 320 пікселів (смартфони), для адміністраторів (B2B) рекомендовано від 1024 пікселів для комфортної роботи з інтерактивною сіткою розкладу;

3) стабільне підключення до мережі Інтернет (для алгоритмічної перевірки вільних слотів);

4) увімкнена підтримка JavaScript та Cookies (зокрема, підтримка HTTP-only cookies для безпечної автентифікації);

б) доступ через веб-інтерфейс:

1) перейдіть за відповідним URL-посиланням на розгорнутий застосунок;

2) додайте сторінку на головний екран смартфона (як PWA) або в закладки десктопного браузера для оперативного доступу до робочого простору;

в) необхідні дозволи браузера:

1) дозвіл на використання локального сховища (LocalStorage) для збереження налаштувань інтерфейсу та кешу Zustand;

2) дозвіл на доступ до файлової системи пристрою (для завантаження фотографій майстрів у портфоліо та зміни аватарів профілів).

Схему процесу реєстрації та входу в систему зображено на рисунку 4.3.

Рисунок 4.3 – Схема процесу авторизації в системі

4.2.2 Запуск та початкове налаштування програми

Після відкриття вебзастосунку користувачу необхідно виконати базові кроки для автентифікації та конфігурації профілю:

а) перший запуск та реєстрація:

1) відкрийте головну сторінку застосунку. Для здійснення запису або доступу до адміністрування система потребує авторизації;

2) для входу або створення нового акаунту використовується номер телефону. Якщо такий номер вводиться вперше, користувач має пройти швидку реєстрацію, вказавши повне ім'я (Full Name) та надійний пароль або здійснити реєстрацію за OTP;

3) після реєстрації ви за замовчуванням отримуєте базову роль клієнта (CLIENT);

б) налаштування профілю клієнта:

1) система автоматично ініціалізує ваш особистий кабінет, використовуючи надані дані;

2) в інтерфейсі кабінету ви одразу отримуєте доступ до перегляду майбутніх записів, історії минулих візитів та можливості управління поточними бронюваннями;

в) початкове налаштування для персоналу (майстрів та адміністраторів):

1) призначення ролі: на відміну від публічної реєстрації, підвищення прав до рівня майстра (BARBER) або адміністратора (ADMIN) здійснюється виключно діючим адміністратором через контрольну панелі (Admin Dashboard), де співробітник закріплюється за конкретною філією (Outlet);

2) конфігурація портфоліо майстра: після отримання ролі BARBER, у вас з'являється доступ до розширених налаштувань. Необхідно заповнити короткий опис (Bio), додати посилання на фотографії робіт та завантажити аватар;

3) формування розкладу: перед тим як система почне приймати записи до нового майстра, адміністратор зобов'язаний налаштувати його базовий робочий графік (Work Schedule) по днях тижня та прив’язати послуги.

Інтерфейси сторінок авторизації, реєстрації та особистого кабінету (налаштувань профілю), які зустрічають користувача під час першого запуску застосунку, наведено на рисунках 4.4, 4.5 та 4.6 відповідно.

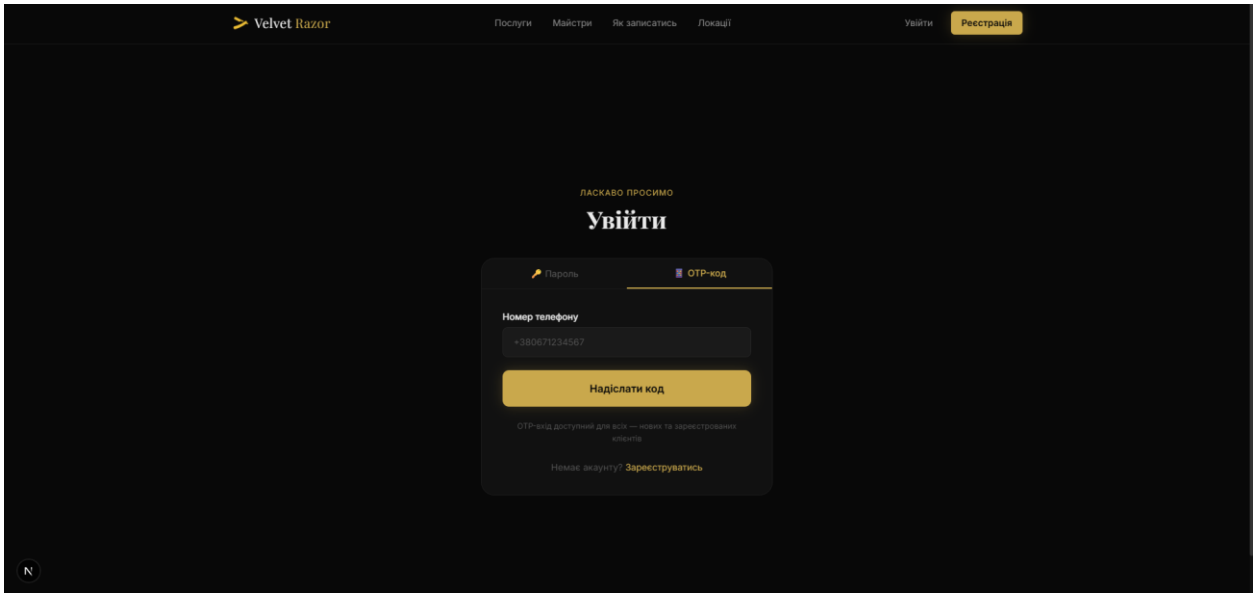

Рисунок 4.4 – Сторінка авторизації за номером телефону та OTP

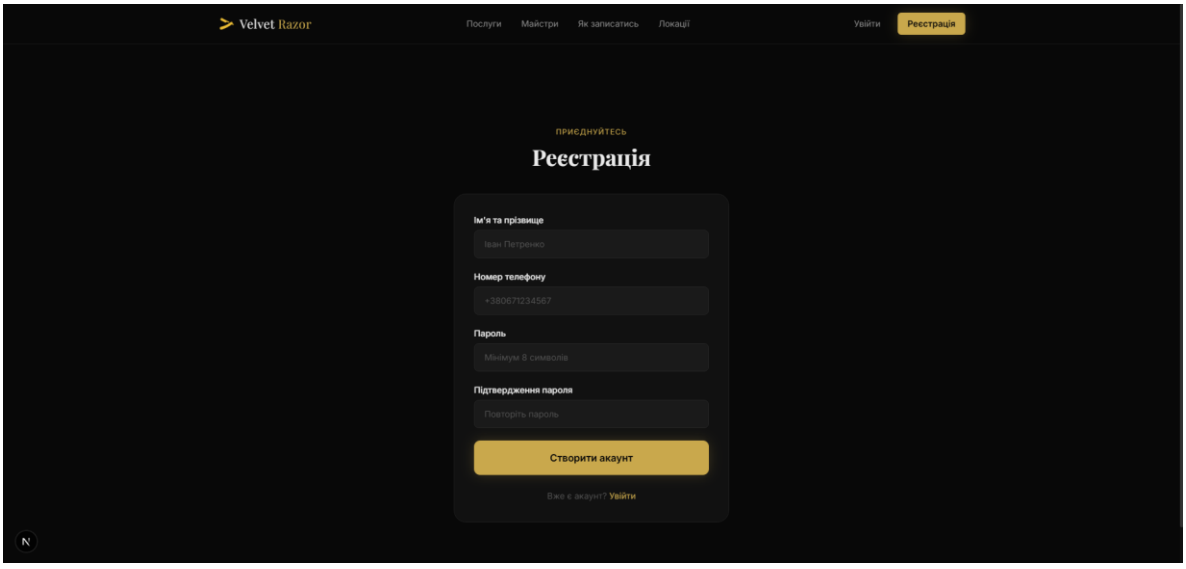

Рисунок 4.5 – Сторінка реєстрації нового користувача

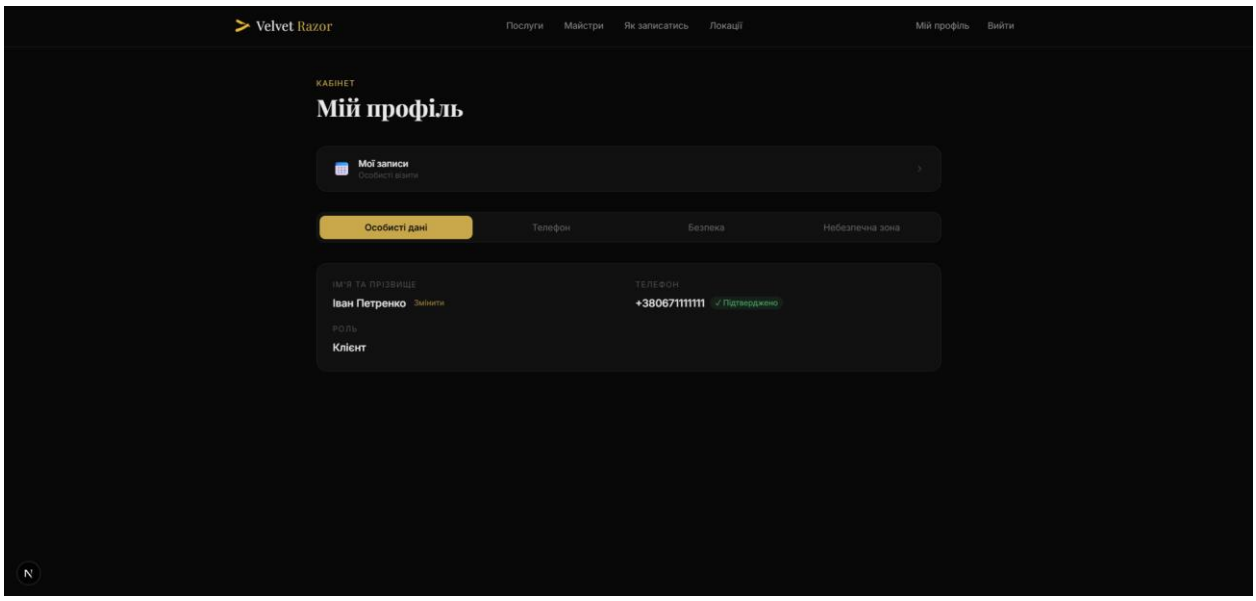

Рисунок 4.6 – Сторінка налаштувань особистого профілю (кабінет)

4.2.3 Методика роботи з програмою

Процес взаємодії з системою «Velvet Razor» базується на трьох ключових операційних циклах (рис. 4.7):

а) управління розкладом та каталогом (для персоналу):

1) створення та редагування каталогу послуг із визначенням їхньої базової ціни, тривалості, буферного часу на прибирання та прив’язка послуг до майстрів;

2) налаштування базового робочого графіка майстрів (Work Schedule) по днях тижня;

3) гнучке управління відхиленнями: додавання кастомних перерв, лікарняних або вихідних днів (Schedule Exceptions) для конкретного майстра;

б) робота із бронюваннями (для клієнтів та адміністраторів):

1) здійснення онлайн-запису клієнтами через багатокроковий інтерфейс із динамічним розрахунком вільних слотів;

2) оформлення складних записів, що включають декілька різних послуг одночасно;

3) оперативна зміна статусів запису (PENDING, CONFIRMED, COMPLETED, NO_SHOW, CANCELLED) адміністратором або майстром;

в) координація та моніторинг (для управління):

1) моніторинг щоденного завантаження салону через інтерактивний адміністративний журнал, де записи абсолютно позиційовані у часовій сітці;

2) відстеження детальної історії змін (Booking History) кожного запису у БД для вирішення можливих спірних ситуацій;

3) управління рівнем довіри клієнтської бази (наприклад, блокування акаунту або активація вимоги обов'язкової передоплати для клієнтів із частими неявками).

Для максимальної продуктивності салону рекомендується своєчасно оновлювати статуси візитів (щоб звільняти час для інших клієнтів у разі скасування) та підтримувати актуальність робочих графіків майстрів. Це дозволяє алгоритмам системи ефективно максимізувати кількість доступних для запису годин (рис. 4.7).

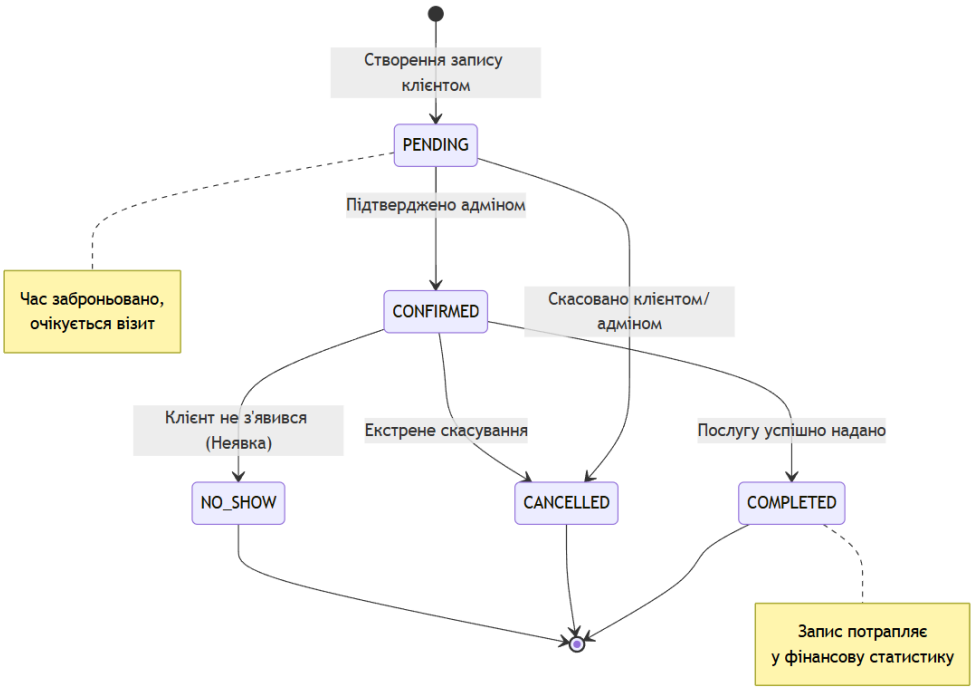

Рисунок 4.7 – Схема життєвого циклу онлайн-бронювання в системі

4.3 Приклад роботи користувача з програмою

Розглянемо типовий комплексний сценарій використання системи «Velvet Razor» на прикладі взаємодії адміністратора салону, майстра (барбера) та кінцевого клієнта. Процес логічно розпочинається з ініціалізації робочого простору адміністратором.

Адміністратор авторизується в системі та переходить до панелі керування (Admin Dashboard). Він реєструє нового співробітника, призначаючи йому рівень доступу BARBER (рис. 4.8). Після цього адміністратор налаштовує робочий профіль майстра: прив'язує до нього послуги, які той надає (з можливістю індивідуального коригування ціни та тривалості), та формує його щотижневий робочий графік (рис. 4.9).

З боку клієнта процес виглядає максимально інтуїтивно. Клієнт відкриває вебзастосунок та запускає багатокроковий майстер онлайн-бронювання (Booking Wizard). Він послідовно обирає необхідні послуги (наприклад, «Чоловіча стрижка» та «Оформлення бороди»), бажаного фахівця і переходить до календаря (рис. 4.10). Серверний алгоритм системи динамічно розраховує доступні часові слоти з урахуванням загальної тривалості всіх обраних послуг та графіка майстра. Клієнт обирає зручний час і підтверджує запис (рис. 4.11).

Одразу після підтвердження адміністратор та барбер бачать новий запис у своєму робочому просторі — в інтерактивному адміністративному журналі, де картка клієнта абсолютно позиційована на часовій сітці відповідно до години початку та тривалості процедури (рис. 4.12).

Після того як клієнт відвідав салон і отримав послугу, адміністратор відкриває картку запису в журналі та одним кліком змінює його статус на COMPLETED. У свою чергу, клієнт у будь-який момент може відкрити свій особистий кабінет, щоб переглянути історію минулих візитів, перевірити статус майбутніх бронювань (рис. 4.13).

Основні вікна інтерфейсу, що супроводжують описаний сценарій взаємодії, наведено на рисунках нижче.

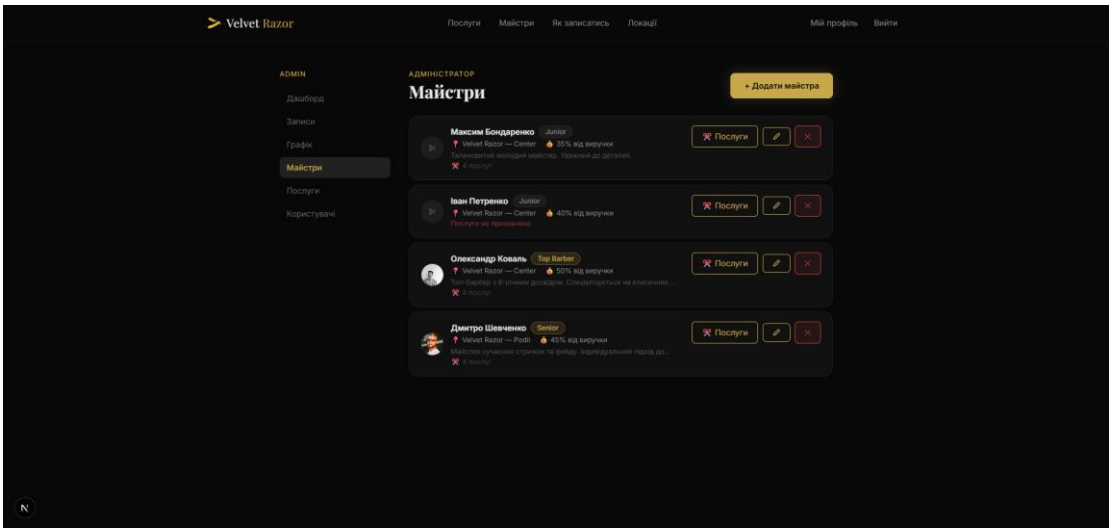

Рисунок 4.8 – Профіль адміністратора та налаштування ролей персоналу

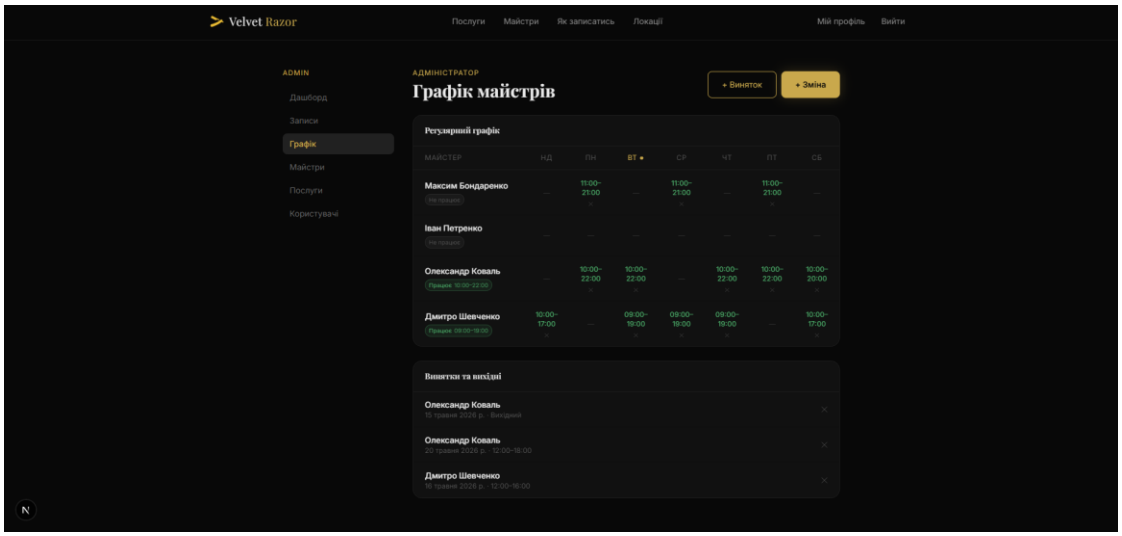

Рисунок 4.9 – Налаштування графіка роботи та послуг майстра

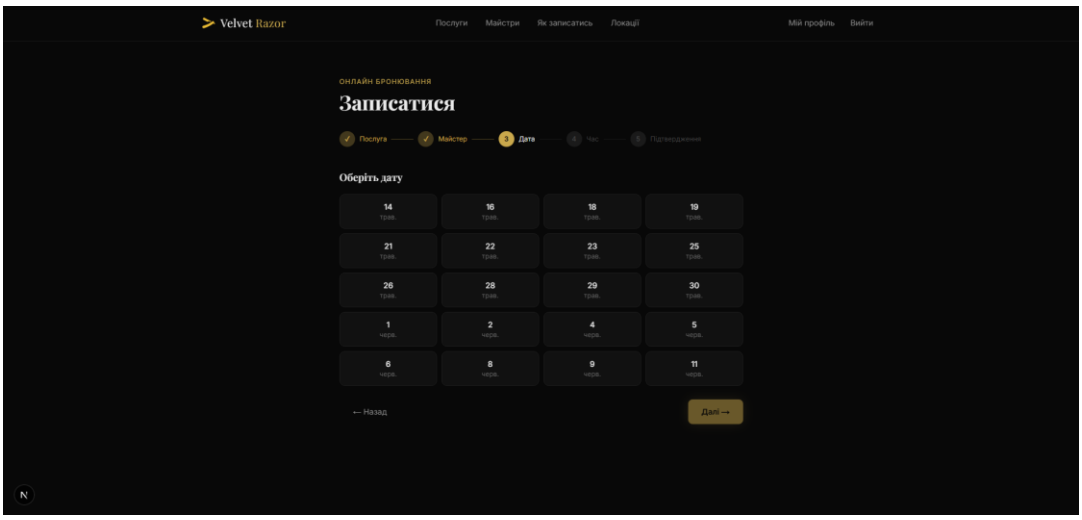

Рисунок 4.10 – Інтерфейс багатокрокового майстра онлайн-бронювання

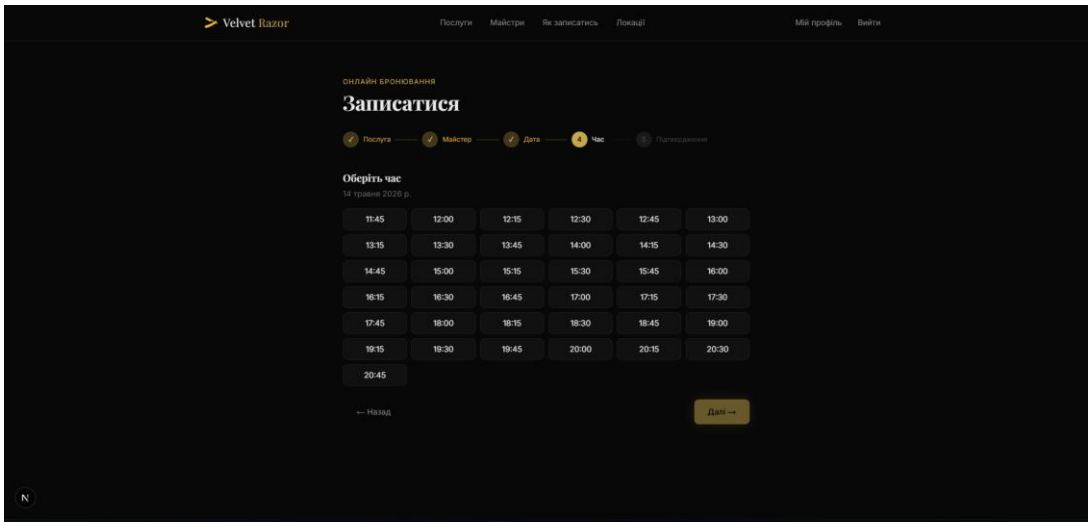

Рисунок 4.11 – Алгоритмічний розрахунок та вибір вільних часових слотів

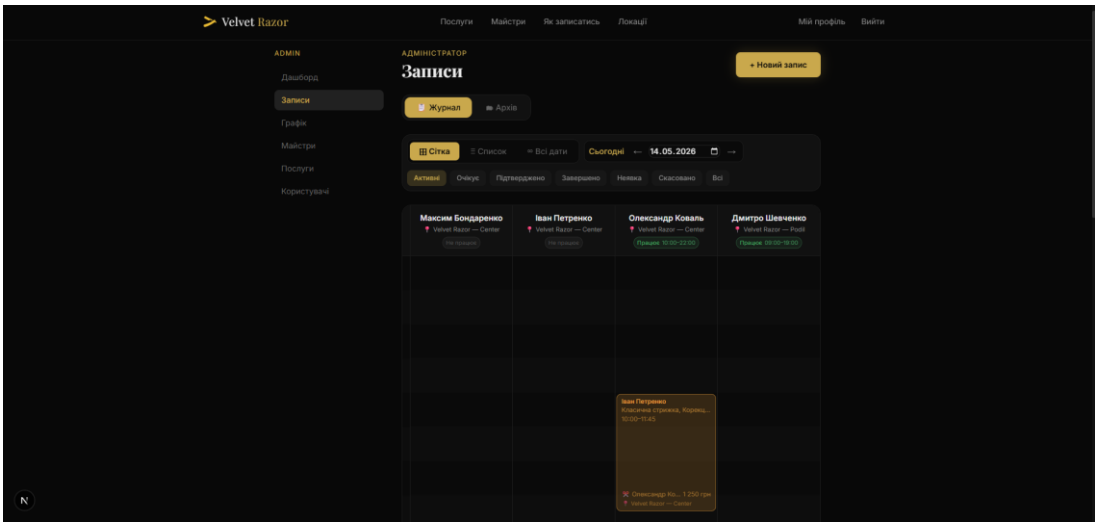

Рисунок 4.12 – Інтерактивний адміністративний журнал записів (Grid)

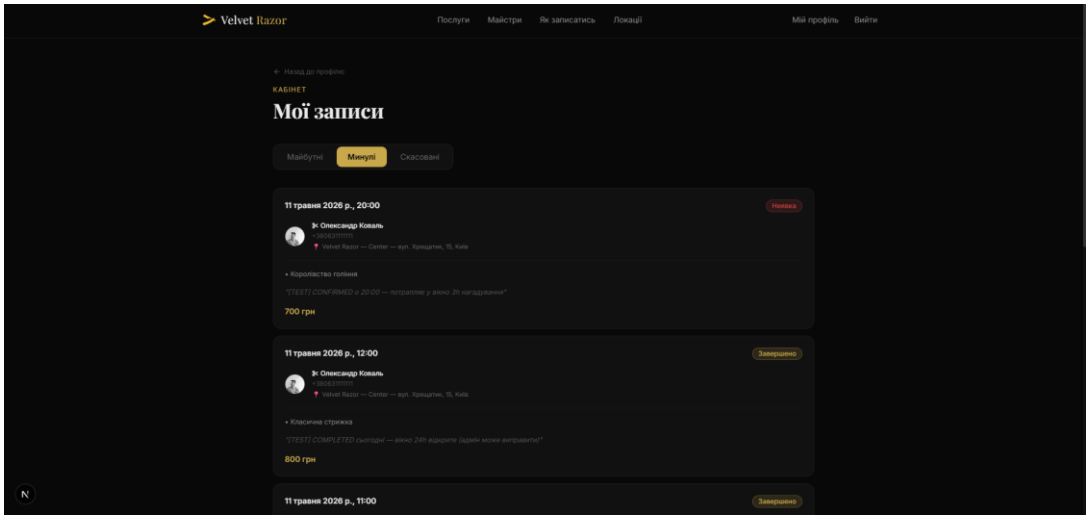

Рисунок 4.13 – Особистий кабінет клієнта та історія його візитів

4.4 Тестування роботи програми

4.4.1 Функціональне тестування

Для забезпечення максимальної надійності та коректності роботи всіх модулів системи «Velvet Razor» було проведено комплексне функціональне тестування, яке включало такі методи:

а) модульне тестування (Unit Testing):

1) перевірка ключових сервісів бекенду (зокрема AppointmentsService та ScheduleService);

2) тестування математичної бізнес-логіки динамічного розрахунку вільних часових слотів та алгоритмів запобігання накладкам (колізіям);

3) наскрізна валідація вхідних даних на основі згенерованих Zod-схем (DTO-моделей);

б) інтеграційне тестування:

1) перевірка транзакційної взаємодії між REST API та базою даних PostgreSQL через Prisma ORM;

2) тестування механізмів безпеки та автентифікації на основі JWT-токенів із використанням захищених HttpOnly-cookies;

3) валідація процесів збереження комплексних бронювань (коли клієнт обирає декілька послуг одночасно);

в) тестування користувацького інтерфейсу (UI Testing):

1) перевірка адаптивності компонентів багатокрокового майстра бронювання (Booking Wizard) на екранах мобільних пристроїв;

2) тестування навігації між розділами управління персоналом, послугами та клієнтською базою в панелі адміністратора;

3) перевірка коректності абсолютного позиціонування карток записів на інтерактивній календарній сітці (Grid) залежно від часу.

Результати функціонального тестування основних модулів системи наведено в таблиці 4.1.

Таблиця 4.1 – Результати функціонального тестування застосунку «Velvet Razor»

Функціональний модуль	Тест–кейси	Результат	Виправлені помилки
Автентифікація та профілі	12	Успішно	1
Управління розкладом (Schedule)	15	Успішно	2
Алгоритм пошуку вільних слотів	22	Успішно	4
Транзакційне онлайн-бронювання	14	Успішно	1
Інтерактивний журнал (Grid UI)	9	Успішно	3

4.4.2 Аналіз функціонального тестування

Детальна оцінка результатів проведеного тестування підтвердила стабільність основних компонентів платформи «Velvet Razor». Загальний показник надійності системи повністю відповідає встановленим технічним вимогам для комерційного програмного забезпечення.

Результати функціонального тестування (рис. 4.14):

– загальна кількість виконаних тест-кейсів: 72;

– успішно пройдено з першої спроби: 61 (~85%);
– виявлено технічних невідповідностей: 11;
– успішно виправлено дефектів: 11.

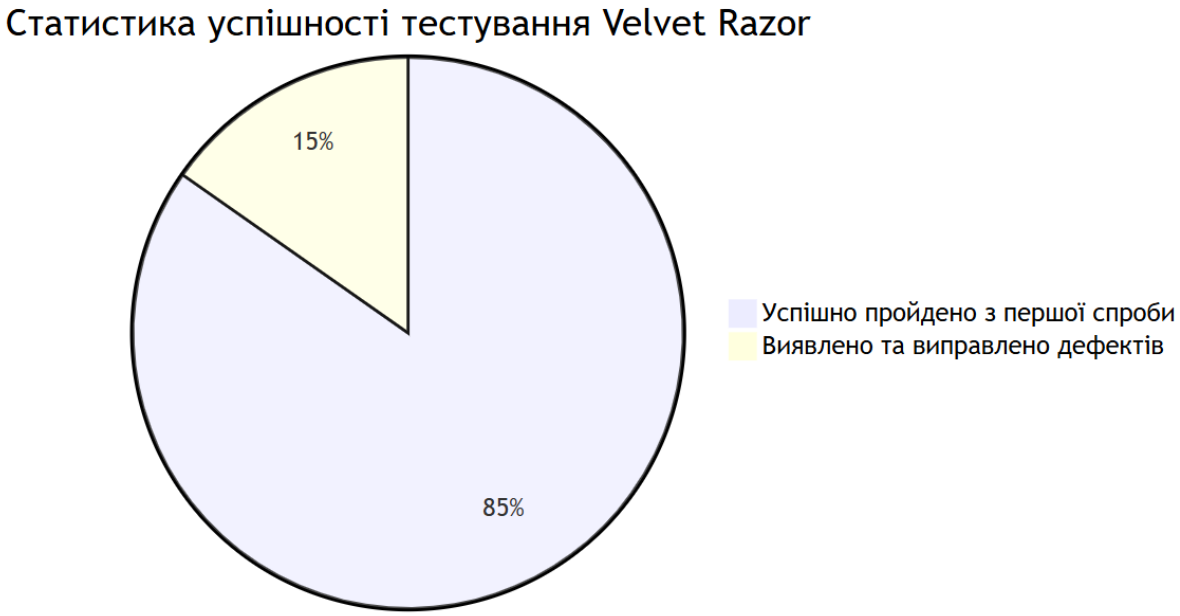

Рисунок 4.14 – Статистика успішності тестування модулів «Velvet Razor»

Основні виправлені дефекти:

– дублювання бронювань (Double Booking): усунуто класичну помилку конкурентного доступу (Race Condition), коли двоє користувачів практично одночасно намагалися забронювати один і той самий часовий слот, шляхом впровадження жорстких транзакційних перевірок у базі даних;

– розрахунок доступного часу: виправлено логіку динамічного розрахунку буферного часу, коли перерва на прибирання робочого місця після послуги не враховувалася при відображенні наступного доступного вікна для клієнта;

– персистентність автентифікації: ліквідовано збій, за якого при різкому оновленні сторінки (F5) локальний стан Zustand "блимав" як неавторизований до моменту повної валідації HttpOnly cookie сервером;

– візуалізація адміністративного журналу: виправлено некоректне візуальне накладання блоків (карток) в інтерактивній сітці (Grid), що

виникало, коли сумарна тривалість декількох послідовних послуг не була кратна 15 хвилинам.

Деталізація найбільш критичних помилок, виявлених та успішно усунених під час розробки та тестування, наведена в таблиці 4.2.

Таблиця 4.2 – Аналіз виявлених та усунених помилок у системі «Velvet Razor»

Тип помилки (Модуль)	Пріоритет	Час виявлення	Час виправлення	Статус
Конкурентний запис (Double Booking)	Критичний	05.04.2026	05.04.2026	Виправлено
Алгоритм буферного часу послуг	Високий	10.04.2026	11.04.2026	Виправлено
Персистентність стану (HttpOnly)	Середній	15.04.2026	16.04.2026	Виправлено
Абсолютне позиціонування (Grid)	Низький	20.04.2026	20.04.2026	Виправлено

4.4.3 Тестування продуктивності

Метою тестування продуктивності було визначення стабільності роботи платформи «Velvet Razor» під високим навантаженням та оцінка ефективності використання системних ресурсів як клієнтською (браузер), так і серверною (NestJS) частинами.

Методи оцінки продуктивності системи:

а) навантажувальне тестування (Load Testing):

1) моделювання конкурентного доступу: перевірка здатності бекенду обробляти масові одночасні запити (50+ користувачів) на

бронювання одного й того ж часового слоту без втрати транзакційної цілісності;

2) стрес-тест математичного ядра: оцінка швидкодії алгоритмів динамічного розрахунку вільних вікон під час масового звернення до графіка найбільш популярних майстрів;

3) перевірка оптимізації бази даних: аналіз швидкості виконання складних запитів до PostgreSQL (через Prisma) при вибірці зв'язаних сутностей (Appointments, Services, ScheduleExceptions) за допомогою впроваджених складених індексів;

б) аналіз споживання системних ресурсів:

1) використання пам'яті (Heap Memory): моніторинг об’єму оперативної пам’яті, що використовується браузером при завантаженні та рендерингу складних інтерфейсів (зокрема, щільної сітки адміністративного журналу з десятками карток);

2) навантаження на процесор (CPU): оцінка інтенсивності обчислень на стороні клієнта під час динамічного абсолютного позиціонування карток візитів на часовій шкалі;

3) мережева затримка (Latency): вимірювання часу відгуку системи на дії користувача (пошук слотів, підтвердження бронювання) за умов різної якості інтернет-з’єднання (наприклад, 3G мережі).

Графік динаміки використання оперативної пам’яті клієнтською частиною застосунку під час активного робочого сеансу (з моменту авторизації адміністратора до повного рендерингу тижневого інтерактивного журналу записів) зображено на рисунку 4.15.

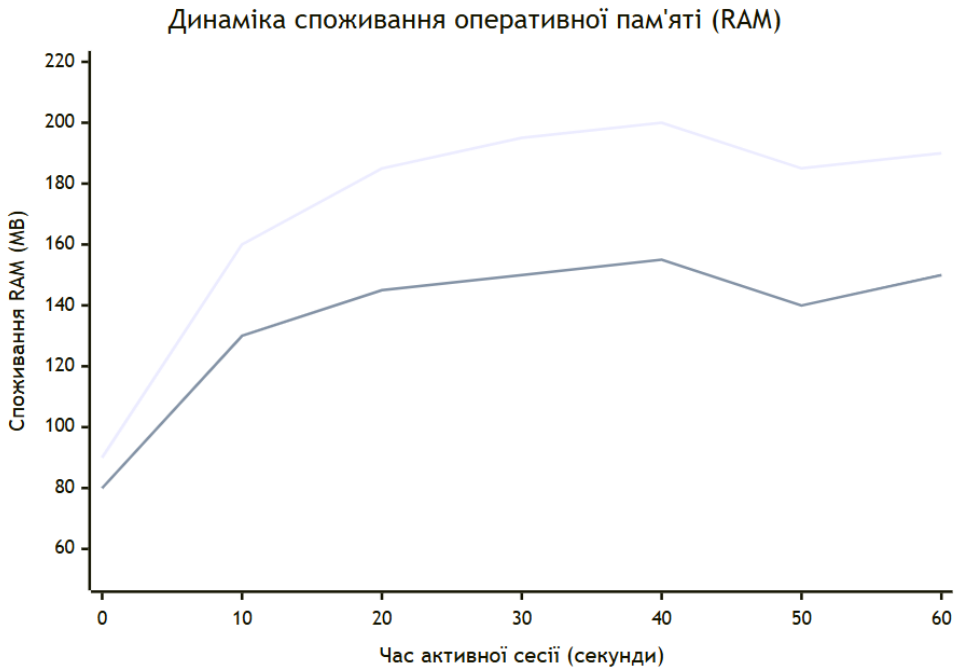

Рисунок 4.15 – Графік використання пам'яті під час роботи інтерактивного журналу

4.4.4 Аналіз продуктивності

Комплексне оцінювання продуктивності вебплатформи «Velvet Razor» проводилося на різних типах пристроїв (Desktop та Mobile) у сучасних браузерах (Chrome, Safari, Firefox). Результати демонструють високу швидкість завантаження інтерфейсу (завдяки серверному рендерингу Next.js) та стабільну роботу математичних алгоритмів (рис. 4.16).

Час завантаження та повної ініціалізації візуального інтерфейсу (показник LCP – Largest Contentful Paint):

– робочі станції (Desktop): 0.8–1.5 секунди;
– мобільні пристрої (смартфони): 1.2–2.2 секунди.

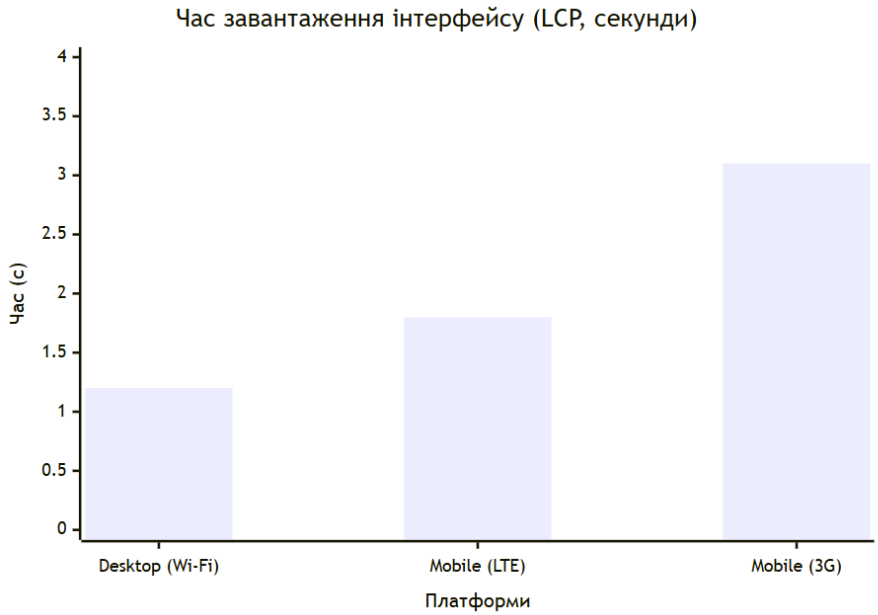

Рисунок 4.16 – Графік часу завантаження інтерфейсу на різних платформах

Показники споживання системних ресурсів при різних сценаріях використання застосунку наведено в таблиці 4.3.

Таблиця 4.3 – Споживання ресурсів клієнтською частиною «Velvet Razor»

Режим роботи	Використання пам'яті	CPU (Main Thread)	Вплив на батарею
Активний рендеринг журналу (Grid)	~180–200 MB	10-14%	~3.5%/год
Багатокроковий майстер бронювання	~145–155 MB	4–6%	~1.5%/год
Активний рендеринг журналу (Адмін)	~80–90 MB	< 1%	< 0.5%/год

Швидкість виконання ключових операцій:

– динамічний розрахунок вільних слотів майстра: <0.3 секунди;

– транзакційне збереження нового комплексного запису: <0.5 секунди;

– зміна статусу візиту адміністратором (відгук інтерфейсу): <0.2 секунди.

Аналіз цих показників переконливо свідчить про те, що архітектура на базі фреймворку Next.js у поєднанні зі спільними Zod-схемами валідації та оптимізованими запитами до реляційної бази даних (через Prisma) забезпечує практично миттєвий відгук системи. Це є критично важливим фактором для утримання клієнтів на етапі онлайн-запису та комфортної роботи персоналу салону.

4.4.5 Тестування безпеки

Для гарантування конфіденційності персональних даних клієнтів (включаючи номери телефонів та історію відвідувань) і захисту бізнесу від несанкціонованого доступу було проведено багаторівневе тестування безпеки платформи «Velvet Razor».

Основні напрямки тестування безпеки:

а) тестування автентифікації та сесій:

1) перевірка механізму HttpOnly-cookies: валідація неможливості викрадення JWT-токенів через XSS-атаки (міжсайтовий скриптинг) завдяки відмові від їх зберігання у вразливому LocalStorage браузера;

2) надійність паролів: тестування сучасного криптографічного алгоритму хешування Argon2, який визнаний значно стійкішим до брутфорс-атак у порівнянні з класичними алгоритмами (такими як Bcrypt);

3) механізм виходу (Logout): перевірка повного анулювання сесії користувача на сервері та безумовного очищення файлів cookie на клієнті;

б) тестування захисту даних та API:

1) наскрізна валідація: тестування спільних Zod-схем на стійкість до некоректних, неповних або навмисно сформованих шкідливих HTTP-запитів;

2) захист від ін'єкцій: перевірка стійкості бази даних PostgreSQL до SQL-ін'єкцій завдяки використанню абстракцій Prisma ORM;

3) захист бізнес-логіки: тестування транзакцій на стійкість до спроб масового бронювання («заспамлювання» слотів) з метою штучного блокування розкладу майстра;

в) тестування розмежування прав доступу (RBAC):

1) перевірка системних ролей: жорсткий контроль доступу до адміністративного журналу виключно для ролей ADMIN та BARBER;

2) ізоляція клієнтських даних: валідація неможливості доступу клієнта (роль CLIENT) до історії візитів, статусу передоплат або персональних даних будь-яких інших користувачів;

3) ізоляція прав майстрів: перевірка обмежень для барберів, які можуть керувати лише власним портфоліо та відстежувати свій розклад, але не мають доступу до зміни базових цін на послуги (Service) та відсотка власної ставки (Income Share).

Результати проведеного тестування безпеки наведено в таблиці 4.4.

Таблиця 4.4 – Результати тестування безпеки «Velvet Razor»

Категорія тестування	Кількість тестів	Виявлені вразливості	Усунені вразливості
Автентифікація та HttpOnly-cookies	8	1	1
Валідація Zod та цілісність даних	12	2	2
Рольова модель (RBAC ізоляція)	7	2	2

4.4.6 Аналіз безпеки

За результатами проведеного тестування було виконано комплексний аналіз безпеки застосунку «Velvet Razor», який підтвердив високу ефективність впроваджених механізмів захисту бізнес-даних.

Особлива увага приділялася надійності сучасного алгоритму Argon2, що використовується для криптографічного хешування паролів, та безпеці передачі даних через протокол HTTPS. Аналіз автентифікації на основі JWT (JSON Web Tokens) показав, що токени мають обмежений термін дії та надійно зберігаються у HttpOnly-cookies, що математично виключає можливість їх перехоплення через шкідливі клієнтські скрипти. Перевірка механізму розмежування прав (RBAC) підтвердила, що користувачі мають доступ виключно до тих ресурсів, на які вони мають право: клієнти бачать лише власну історію, майстри – свій розклад, а адміністратори – всю систему.

Результати автоматизованого сканування та тестування на проникнення наведено в таблиці 4.5.

Таблиця 4.5 – Результати сканування безпеки «Velvet Razor»

Метод перевірки	Кількість знайдених вразливостей	Критичні дефекти	Статус усунення
Статичний аналіз коду (SAST)	3	0	Виправлено
Динамічний аналіз (DAST)	2	0	Виправлено
Penetration testing (ручний аналіз)	5	1	Виправлено

Основні результати аналізу та виправлення:

– критична вразливість: було виявлено теоретичну можливість скасування запису іншого користувача шляхом прямої підміни ID бронювання

в HTTP-запиті. Проблему повністю усунено шляхом впровадження додаткової суворої перевірки "власника" ресурсу (порівняння clientId з ID авторизованого токена) на рівні NestJS-контролерів;

– статичний аналіз: виявлено використання застарілих npm-залежностей у монорепозиторії (у package.json), які були оперативно оновлені до стабільних версій із вбудованими патчами безпеки;

– динамічний аналіз: виправлено занадто м'яку політику CORS та додано механізм Rate Limiting (обмеження частоти запитів), що посилило захист сервера від автоматизованого перебору паролів (Brute-force) та спаму фейковими бронюваннями.

4.4.7 Аналіз користувацького досвіду

Для оцінки ергономічності та загальної ефективності інтерфейсу системи «Velvet Razor» було проведено апробацію застосунку групою альфа-тестувальників (команда з 5 осіб, що імітували ролі адміністраторів салону та клієнтів) після завершення повного циклу налаштування та тестового бронювання послуг (рис. 4.17).

Показники задоволеності інтерфейсом (UI/UX):

– високий рівень (дуже задоволені): 50% (відзначили плавність анімацій багатокрокового майстра та загальну швидкість роботи);

– задовільний рівень (задоволені): 35% (позитивно оцінили зручність адміністративного журналу);

– нейтральний рівень: 15% (висловили пропозиції щодо майбутнього розширення функціонала онлайн-оплати).

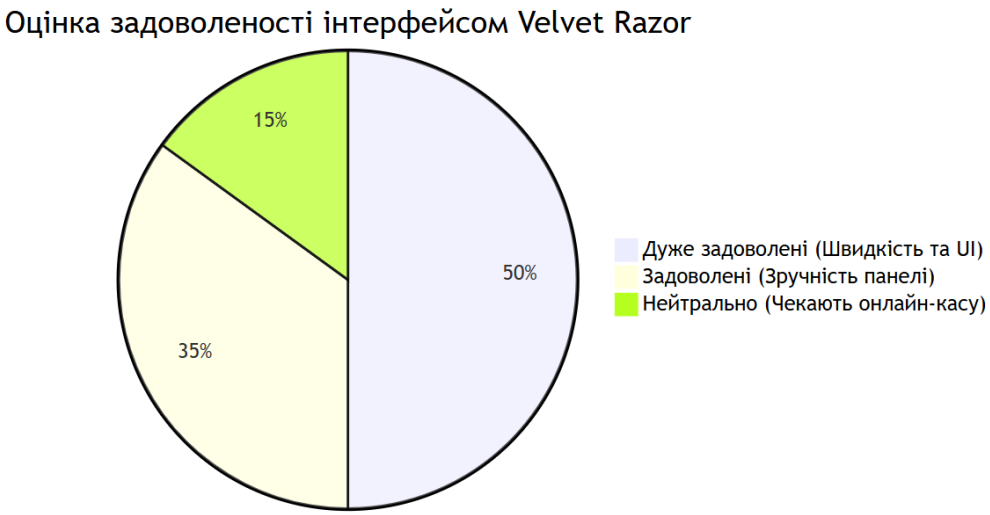

Рисунок 4.17 – Оцінка задоволеності інтерфейсом системи «Velvet Razor»

Результати аналізу частоти використання ключових модулів та їхньої суб'єктивної оцінки користувачами наведено в таблиці 4.6.

Таблиця 4.6 – Оцінка функціональних модулів «Velvet Razor»

Функціональний модуль	Частота звернень	Середня оцінка користувачів (max 5)
Багатокроковий майстер бронювання (Client UI)	98%	4.9/5
Інтерактивний адміністративний журнал (Grid)	85%	4.8/5
Управління розкладом майстрів та послугами	60%	4.7/5
Особистий кабінет клієнта (Історія візитів)	35%	4.5/5

Тестувальники відзначили надзвичайно високу ефективність багатокрокового процесу бронювання, що дозволяє клієнтам записатися без зайвих роздумів завдяки поступовому виведенню інформації (вибір послуги,

майстра, часу). З боку персоналу найвищий рейтинг отримала функція адміністративного журналу (Grid) через візуальну чистоту абсолютного позиціонування карток та зручність миттєвої зміни статусів візитів.

4.5 Рекомендації щодо використання

На основі результатів апробації та аналізу відгуків було сформовано перелік рекомендацій для максимально ефективного та безболісного впровадження системи «Velvet Razor» у щоденні робочі процеси барбершопу.

Для оптимального налаштування робочого простору (з боку адміністраторів) рекомендується:

– суворо розмежувати ролі співробітників (Admin, Barber) для забезпечення безпеки клієнтської бази та конфіденційності загальних фінансових налаштувань;

– відповідально підійти до конфігурації каталогу послуг: обов'язково вказувати реальний час на виконання кожної процедури та закладати достатній буферний час (Buffer Time) на прибирання робочого місця між клієнтами;

– своєчасно вносити зміни до графіка майстрів (використовуючи функціонал Schedule Exceptions) у разі їхніх відпусток, лікарняних або запізнень, щоб алгоритм унеможливив запис клієнтів на цей час.

Для ефективного використання функціоналу платформи варто:

– систематично та одразу оновлювати статуси візитів (наприклад, переводити у COMPLETED або CANCELLED), щоб система завжди мала актуальну картину зайнятості та могла звільнити слот для інших клієнтів;

– активно використовувати інтерактивний адміністративний журнал (Grid) як основний інструмент моніторингу потоку клієнтів на екрані ресепшена;

– використовувати внутрішні інструменти контролю довіри: позначати недобросовісних клієнтів марнерами блокування або вимогою обов'язкової передоплати (requiresPrepayment) при наступних записах.

Щоденна робота з «Velvet Razor» з боку персоналу передбачає регулярну перевірку нових онлайн-бронювань у журналі та підготовку робочого місця відповідно до обраних клієнтом послуг. Такий цифровий підхід дозволяє підтримувати високий темп роботи салону без відволікання майстрів на телефонні дзвінки.

Для запобігання організаційним помилкам та технічним невідповідностям слід:

– уникати використання фіктивних номерів телефонів при швидкому створенні "тіньових" записів, оскільки унікальність цього поля в базі даних не дозволить створити більше одного клієнта з певним номером;

– періодично проводити ревізію списку персоналу, переводячи в архівний стан тих майстрів (isArchived), які припинили співпрацю із закладом, щоб автоматично закрити їм доступ до платформи.

4.6 Плани з розвитку

На основі аналізу отриманих відгуків, технічного тестування та дослідження ринку CRM-систем для б'юті-індустрії було визначено стратегію подальшого розвитку платформи «Velvet Razor» у кількох ключових напрямках.

У короткостроковій перспективі передбачається впровадження повноцінної інтеграції зі сторонніми комунікаційними шлюзами (SMS / Viber / Telegram) для автоматичного розсилання нагадувань клієнтам про майбутні візити та безпечного підтвердження номерів телефонів через OTP-паролі. Також високим пріоритетом є реалізація модуля безпечної онлайн-оплати (інтеграція з платіжними провайдерами на кшталт LiqPay або Stripe), що дозволить повністю автоматизувати бізнес-логіку стягнення обов'язкових

передоплат для клієнтів із відповідним статусом. Окрім цього, планується розширення адміністративного дашборда складними аналітичними звітами (фінансові графіки рентабельності філії, індекс щільності записів та відсоток утримання клієнтів окремими майстрами).

У довгострокових планах основна увага зосереджена на розробці нативного мобільного застосунку (на базі React Native), який суттєво розширить можливості поточної вебверсії та забезпечить інтеграцію з нативними календарями та Push-сповіщеннями смартфонів. Також планується розробка гнучкої підсистеми програми лояльності (нарахування кешбеку, реферальні посилання, промокоди) для стимулювання повторних візитів. Стратегічним же вектором розширення проєкту є еволюція платформи у повноцінний Multi-tenant SaaS-продукт, що дозволить абсолютно незалежним мережам барбершопів та салонів краси автоматично реєструватися на платформі, створювати власний робочий простір і кастомізувати дизайн онлайн-запису під свій унікальний бренд без будь-якого втручання з боку розробників.

ВИСНОВКИ

З метою підвищення рівня автоматизації та загальної продуктивності робочих процесів у сфері надання послуг (зокрема, б'юті-індустрії), у роботі було успішно вирішено задачу створення комплексної CRM-системи та вебплатформи онлайн-бронювання «Velvet Razor». Отримані результати підтверджують ефективність розробленого програмного інструменту, який є надійним, швидким та інтуїтивно зрозумілим рішенням для координації розкладу майстрів та управління щоденним потоком клієнтів.

У ході дослідження проведено аналіз існуючих бізнес-процесів управління салонами та ринку відповідного програмного забезпечення. Встановлено, що популярні маркетплейси та монолітні CRM-системи часто є надмірно перевантаженими для малого бізнесу або пропонують застарілий інтерфейс для кінцевого споживача. Це обґрунтувало необхідність розробки нової нішевої системи з акцентом на мобільну адаптивність, швидкість роботи та гнучкість налаштування послуг, що забезпечує низький поріг входження.

Архітектуру платформи побудовано за сучасним принципом монорепозиторію з використанням фреймворку Next.js для клієнтської частини та NestJS для серверної. Застосування Prisma ORM у поєднанні з реляційною базою даних PostgreSQL дозволило створити оптимізовану модель збереження даних, а впровадження спільного пакета Zod-схем забезпечило наскрізну типізацію. Використання бібліотеки Zustand гарантувало миттєве реактивне оновлення клієнтського інтерфейсу після кожної дії без необхідності перезавантаження сторінок.

Програмно реалізовано модулі автентифікації користувачів, багатокрокового онлайн-бронювання, управління каталогом послуг, конфігурації робочих графіків та рольового доступу (RBAC). Унікальною особливістю «Velvet Razor» є поєднання складних серверних алгоритмів динамічного розрахунку вільних слотів із візуальною чистотою інтерактивного адміністративного журналу (Grid), де картки візитів

абсолютно позиційовані у часі. Додатково впроваджено функціонал транзакційного запобігання колізіям під час конкурентного бронювання.

Під час тестування працездатності, продуктивності та безпеки система продемонструвала високі показники надійності. Успішне проходження 85% тест-кейсів та оптимізований час завантаження інтерфейсу (в межах 0.8–2.2 секунди) підтверджують технічну досконалість обраних рішень. Виявлені на етапі розробки критичні дефекти, пов’язані з алгоритмом розрахунку буферного часу та персистентністю стану, були оперативно усунені, що гарантує стабільну роботу платформи в експлуатаційному режимі.

Безпека персональних і комерційних даних реалізована через багаторівневу систему захисту, що включає відмову від LocalStorage на користь захищених HttpOnly-cookies для зберігання JWT-токенів, надійне криптографічне хешування паролів сучасним алгоритмом Argon2 та суворе розмежування прав на рівні API-контролерів. Аналіз безпеки підтвердив стійкість платформи до SQL-ін'єкцій та унеможливив підміну ідентифікаторів записів.

На основі аналізу трендів цифровізації малого бізнесу визначено, що ключову частку B2B-користувачів системи складатимуть індивідуальні майстри-орендарі (45%), локальні малі барбершопи (35%) та середні мережі (15%). Це дозволяє чітко сфокусувати подальший розвиток продукту на вирішенні проблем мікробізнесу.

Сформовано дорожню карту вдосконалення «Velvet Razor», яка передбачає впровадження інтеграції з SMS/Viber/Telegram для надсилання сповіщень, розробку модуля безпечної онлайн-оплати (через LiqPay/Stripe) та створення повноцінного нативного мобільного застосунку. Це значно підвищить конкурентоспроможність платформи. Розроблений вебзастосунок вже зараз успішно автоматизує рутину барбершопу, позбавляє адміністраторів паперових журналів та закладає потужний фундамент для масштабування у повноцінний Multi-tenant SaaS-продукт.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. React: The library for web and native user interfaces. Meta Open Source. URL: https://react.dev (date of access: 03.04.2026).

2. Zustand: A small, fast and scalable bearbones state-management solution. pmndrs. GitHub. URL: https://github.com/pmndrs/zustand (date of access: 03.04.2026).

3. 25+ Incredible Online Booking Statistics [2023]: Facts, Trends, And More. Zippia. URL: https://www.zippia.com/advice/appointment-scheduling-statistics (date of access: 10.04.2026).

4. Salon Software Comparison & Reviews. Capterra. URL: https://www.capterra.com/salon-software (date of access: 16.04.2026).

5. Spa and Salon Software Market Size, Share & Trends Analysis Report, 2023–2033. Grand View Research. URL: https://www.grandviewresearch.com/industry-analysis/spa-salon-software-market-report (date of access: 16.04.2026).

6. Salon Industry Statistics and Consumer Trends. Scheduling Kit. URL: https://schedulingkit.com/statistics/salon-industry-statistics (date of access: 24.04.2026).

7. Офіційний сайт платформи автоматизації бізнесу Altegio. Altegio. URL: https://altegio.com (date of access: 24.04.2026).

8. Booksy for Business: Appointment Scheduling Software. Booksy. URL: https://booksy.com/biz/en-us (date of access: 24.04.2026).

9. Free salon software for beauty and wellness. Fresha. URL: https://www.fresha.com/for-business (date of access: 24.04.2026).

10. OWASP Top Ten: The Ten Most Critical Web Application Security Risks. OWASP Foundation. URL: https://owasp.org/www-project-top-ten (date of access: 01.05.2026).

11. Next.js: The React Framework for the Web. Vercel. URL: https://nextjs.org/docs/app/building-your-application/optimizing (date of access: 02.05.2026).

12. Tailwind CSS: A utility-first CSS framework for rapid UI development. Tailwind Labs. URL: https://tailwindcss.com/docs (date of access: 03.05.2026).

13. Turborepo: High-performance build system for JavaScript and TypeScript codebases. Vercel. URL: https://turborepo.dev/docs (date of access: 04.05.2026).

14. NestJS Framework: A progressive Node.js framework for building efficient and scalable server-side applications. Trilon LLC. URL: https://docs.nestjs.com/security/authentication (date of access: 05.05.2026).

15. Zod: TypeScript-first schema validation with static type inference. Colin McDonnell. URL: https://zod.dev (date of access: 06.05.2026).

16. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT). RFC 7519. *Internet Engineering Task Force (IETF)*. URL: https://www.rfc-editor.org/rfc/rfc7519 (date of access: 07.05.2026).

17. Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications. RFC 9106 / A. Biryukov et al. *Internet Engineering Task Force (IETF)*. URL: https://www.rfc-editor.org/rfc/rfc9106 (date of access: 07.05.2026).

18. Multi-stage builds. Docker Documentation. Docker, Inc. URL: https://docs.docker.com/build/building/multi-stage (date of access: 08.05.2026).

19. TanStack Query: Powerful asynchronous state management for TypeScript/JavaScript. TanStack. URL: https://tanstack.com/query/latest/docs (date of access: 09.05.2026).

20. Proposed NIST Standard for Role-Based Access Control / D. Ferraiolo et al. National Institute of Standards and Technology (NIST). URL: https://csrc.nist.gov/projects/role-based-access-control (date of access: 10.05.2026).

21. TypeScript: Typed JavaScript at Any Scale. Documentation. Microsoft Corporation. URL: https://www.typescriptlang.org/docs (date of access: 10.05.2026).

22. PostgreSQL: The World's Most Advanced Open Source Relational Database. PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs (date of access: 11.05.2026).

23. Prisma: Next-generation Node.js and TypeScript ORM. Prisma Data, Inc. URL: https://www.prisma.io/docs (date of access: 11.05.2026).

24. Online booking is on the rise: Current statistics and trends. Calenso. URL: https://calenso.com/en/blog/online-buchen-nimmt-zu (date of access: 12.05.2026).

25. The Digital Transformation of SMEs: Policy Highlights. OECD Publishing. URL: https://www.oecd.org/content/dam/oecd/en/publications/support-materials/2021/02/the-digital-transformation-of-smes_ec3163f5/Digital-Transformation-of-SMEs-policy-highlights-en.pdf (date of access: 12.05.2026).

ДОДАТОК А

Технічне завдання

Вступ

Вебзастосунок «Velvet Razor» призначений для автоматизації процесів управління розкладом та координації роботи у сфері надання послуг (зокрема, в барбершопах та салонах краси). Він розроблений для використання адміністраторами, індивідуальними майстрами (барберами) та кінцевими клієнтами, які потребують зручного інструменту для онлайн-бронювання, відстеження графіка роботи та оперативного управління щоденним потоком відвідувачів. Область застосування охоплює малий та середній бізнес у б'юті-індустрії, а об’єктом використання є кросплатформний веб-інтерфейс, адаптований як для десктопних моніторів (для зручної роботи адміністраторів із сіткою), так і для екранів мобільних пристроїв (для швидкого запису клієнтами).

А.1 Підстави до розробки

Підставою для розробки є завдання на дипломну роботу, видане 14 квітня 2026 року. Найменування теми розробки – «Розроблення програмного забезпечення веборієнтованої системи онлайн-бронювання послуг барбершопу».

А.2 Призначення розробки

Застосунок розроблений для повної автоматизації життєвого циклу обслуговування клієнта: від моменту самостійного вибору вільних годин у смартфоні до фіксації успішного завершення візиту в журналі салону. Система забезпечує складний математичний розрахунок вільних слотів з урахуванням тривалості послуг, гарантує цілісність даних та цілком унеможливлює накладки у розкладі завдяки транзакційним механізмам бази даних. Експлуатаційне призначення продукту полягає у цифровізації рутини

адміністраторів, відмові від ведення паперових журналів та забезпеченні клієнтів сучасним інструментом для швидкого бронювання 24/7.

А.3 Вимоги до програми чи програмного виробу

А.3.1 Вимоги до фунціоналних характеристик

Вимоги до функціональних характеристик Вебплатформа «Velvet Razor» розроблена для вирішення широкого спектра завдань, пов’язаних із координацією роботи персоналу та менеджментом клієнтської бази. Система забезпечує інтуїтивний процес реєстрації та автентифікації користувачів (за номером телефону) через захищені канали, використовуючи JWT-токени в HttpOnly-cookies, що гарантує надійний доступ до профілів. Застосунок дозволяє чітко розподіляти повноваження в межах філії за допомогою системи ролей: адміністратор (ADMIN), майстер (BARBER) та звичайний користувач (CLIENT).

Керування розкладом та послугами є фундаментальною функцією «Velvet Razor». Адміністратори мають можливість гнучко налаштовувати каталог послуг (вказуючи ціни, тривалість та буферний час на прибирання), а також формувати базові робочі графіки майстрів із можливістю додавання специфічних перерв або відпусток (Schedule Exceptions). Для клієнтів реалізовано багатокроковий майстер бронювання, який алгоритмічно відфільтровує зайняті години та пропонує виключно доступні для запису слоти.

Застосунок пропонує потужні інструменти для щоденного моніторингу завантаженості закладу. Адміністратори можуть відстежувати стан записів (від очікування до завершення чи неявки) за допомогою інтерактивної календарної сітки (Grid), де кожне бронювання відображається у вигляді абсолютно позиційованої картки, довжина якої строго відповідає тривалості процедури на шкалі часу. Це дозволяє оперативно оцінювати «вікна» у розкладі та керувати статусами візитів.

Особливу увагу приділено безпеці та цілісності інформації. Доступ до фінансової інформації та загальної бази клієнтів суворо обмежений ролями учасників, а вся взаємодія з сервером жорстко валідується через спільні Zod-схеми. Вхідними даними для системи є контактна інформація користувачів, параметри послуг, графіки роботи та деталі конкретних бронювань. На основі цих показників формуються вихідні результати: персональні розклади майстрів, візуалізація загального потоку в адміністративному журналі та деталізована історія клієнтських візитів.

А.3.2 Вимоги до надійності

Для забезпечення безперебійного функціонування платформи необхідно дотримуватися ряду технічних вимог. Оскільки «Velvet Razor» є хмарним клієнт-серверним застосунком, користувачі мають забезпечити підключення до мережі Інтернет для коректного виконання транзакцій бронювання та оновлення адміністративного журналу. Використання сучасного стеку технологій, зокрема фреймворку NestJS для серверної частини та Next.js для клієнтського інтерфейсу, гарантує високу швидкість обробки REST-запитів та абсолютну сумісність із актуальними мобільними й десктопними браузерами. Безпека профілів та комерційних даних підтримується завдяки впровадженню алгоритму хешування Argon2, авторизації через захищені HttpOnly-cookies та жорсткого рольового доступу (RBAC).

Система спроєктована таким чином, щоб ефективно реагувати на непередбачувані ситуації та повністю виключати критичні колізії (наприклад, одночасний запис двох клієнтів на один часовий слот). Це досягається завдяки використанню суворих транзакційних перевірок бази даних на рівні Prisma ORM. При спробі введення некоректних контактних даних або обрання вже недоступного часу спрацьовує надійна наскрізна система валідації на основі спільних Zod-схем, яка миттєво інформує користувача про помилку через

спливаючі сповіщення, категорично запобігаючи порушенню цілісності бази даних.

А.3.3 Умови експлуатації

Для коректної роботи платформи «Velvet Razor» необхідно забезпечити стандартні умови експлуатації мобільної чи комп'ютерної техніки (робота пристроїв у температурному діапазоні від +5°C до +35°C та вологості до 80%). Оскільки застосунок розроблено на базі вебтехнологій, ключовою вимогою є наявність сучасного браузера (Google Chrome, Safari, Firefox або Edge) та доступу до мережі Інтернет для взаємодії з сервером.

Рівень підготовки кінцевого клієнта (B2C) передбачає володіння лише базовими навичками користування смартфоном: проходження покрокових форм та вибір доступних кнопок часу на екрані. Для персоналу салону (адміністраторів та барберів) достатньо базових навичок роботи з веб-інтерфейсами: навігація по меню, заповнення профілів та взаємодія з інтерактивною сіткою розкладу (Grid). Система розроблена за принципами чистого та інтуїтивно зрозумілого дизайну (UI/UX), що зводить до мінімуму час на освоєння інструментарію. Усі оновлення системи відбуваються централізовано на сервері, тому технічна підтримка не потребує залучення IT-персоналу на боці конкретного барбершопу.

А.3.4 Вимоги до складу параметрів технічних засобів

Для забезпечення стабільної та швидкої роботи вебплатформи «Velvet Razor», пристрій користувача має відповідати наступним мінімальним технічним характеристикам:

– операційна система: актуальні мобільні ОС (Android 9.0+ або iOS 14+) для зручного доступу клієнтів та майстрів, а також десктопні системи

Windows 10/11 чи macOS 11+ для повноцінної комфортної роботи адміністраторів;

– програмне забезпечення: сучасний веббраузер із повноцінною підтримкою стандарту ES6+ (настійно рекомендовано використання останніх версій Google Chrome, Safari, Mozilla Firefox або Microsoft Edge);

– оперативна пам’ять: мінімум 2 ГБ для плавної роботи реактивних компонентів інтерфейсу (зокрема, обробки анімацій багатокрокового майстра бронювання та рендерингу адміністративної сітки);

– вільне місце: до 50 МБ дискового простору для кешування статичних ресурсів (шрифтів, зображень портфоліо майстрів та скриптів) у пам'яті браузера;

– мережеве з’єднання: обов’язкове підключення до мережі Інтернет (стабільне 3G/4G, LTE або Wi-Fi з’єднання, від 1 Мбіт/с) для забезпечення миттєвої відправки транзакцій на бронювання та оновлення статусів візитів без таймаутів.

А.3.5 Вимоги до інформаційної й програмної сумісності

Для забезпечення найвищого рівня інформаційної сумісності та гнучкості системи, обмін даними між клієнтським застосунком та сервером здійснюється виключно через REST API у форматі JSON. Програмна реалізація проєкту повністю базується на строгій мові TypeScript та архітектурі монорепозиторію (на базі Turborepo), що гарантує наскрізну статичну типізацію коду, повну мінімізацію помилок під час обміну даними завдяки єдиному пакету Zod-схем та надвисоку надійність продукту.

Як ключові фреймворки обрано Next.js (екосистема React) для розробки клієнтського інтерфейсу та NestJS для побудови потужної серверної архітектури. Таке поєднання дозволило створити кросплатформний вебзастосунок із єдиною кодовою базою, який максимально ефективно працює на пристроях будь-якого типу.

Система збереження та управління даними побудована на дворівневій архітектурі:

– централізоване серверне сховище: використовується реляційна база даних PostgreSQL, взаємодія з якою здійснюється за допомогою абстракцій Prisma ORM. Це забезпечує ідеальну транзакційну цілісність складних зв’язків між профілями майстрів, каталогом послуг, винятками у робочих графіках та історією бронювань;

– клієнтське керування станом: для забезпечення миттєвого відгуку інтерфейсу та уникнення зайвих мережевих запитів активно використовується швидке глобальне сховище на базі бібліотеки Zustand.

Безпека та конфіденційність є найвищими пріоритетами під час розробки застосунку. Доступ до ресурсів захищений за допомогою надійної системи JWT-авторизації, токени якої передаються виключно через захищені від скриптів HttpOnly-cookies. Паролі користувачів проходять жорстке криптографічне хешування за сучасним стандартом Argon2. Вся інформація про записи, фінансові статуси та клієнтську базу строго ізольована на рівні бекенд-контролерів відповідно до присвоєних ролей (RBAC), що гарантує стовідсотковий захист комерційної таємниці барбершопу та персональних даних його клієнтів.

А.3.6 Вимоги до маркування й упакування

Доступ до платформи «Velvet Razor» надається через мережу Інтернет за унікальною веб-адресою, що робить систему доступною миттєво без необхідності завантаження та встановлення з магазинів застосунків (завдяки підтримці сучасних вебстандартів). Розповсюдження продукту може відбуватися за моделлю SaaS (програмне забезпечення як сервіс) або шляхом розгортання безпосередньо на серверній інфраструктурі замовника (Self-hosting) через механізми контейнеризації (Docker).

Оскільки «Velvet Razor» є виключно інтелектуальним цифровим продуктом, будь-які вимоги до фізичного пакування, матеріального маркування чи логістичного транспортування повністю відсутні. Усі етапи передачі прав доступу, інсталяції бази даних та подальшого оновлення функціоналу виконуються дистанційно у цифровому середовищі, що забезпечує максимальну швидкість впровадження системи в щоденні робочі процеси барбершопу.

А.3.7 Вимоги до транспортування й збереження

З огляду на хмарну модель розгортання та виключно цифровий формат постачання системи «Velvet Razor», будь-які стандарти чи вимоги до фізичного транспортування, логістики або спеціальних умов складського зберігання матеріальних об’єктів до продукту не застосовуються. Доступ до інструментарію, а також отримання технічних патчів та нових оновлень функціоналу здійснюється миттєво через глобальну мережеву інфраструктуру. Це повністю виключає необхідність у фізичному пакуванні, маркуванні чи механічному переміщенні копій програмного забезпечення.

А.4 Вимоги до програмної документації

Супровідна документація до платформи «Velvet Razor» включає дане технічне завдання та розширену пояснювальну записку дипломної роботи. Ці матеріали містять вичерпні відомості про прийняті архітектурні рішення (використання монорепозиторію та спільних типів), детальний опис ключових функціональних модулів (таких як модуль багатокрокового онлайн-бронювання, управління розкладом та послугами майстрів, інтерактивний адміністративний журнал), а також чітко регламентовані технічні вимоги до середовища розробки та умов експлуатації кінцевого програмного продукту.

А.5 Техніко–економічні показники

Розроблена вебплатформа має надзвичайно високий потенціал для докорінного покращення процесів управління розкладом та клієнтської взаємодії в індустрії краси. Очікується, що впровадження системи «Velvet Razor» дозволить скоротити час адміністраторів на обробку телефонних дзвінків та ручне ведення паперових журналів щонайменше на 40-50%, делегуючи процес алгоритмічного вибору вільного часу безпосередньо клієнтам через зручний мобільний інтерфейс.

Відкрита базова архітектура робить застосунок доступним для широкого кола користувачів: від індивідуальних майстрів (орендарів крісел) до повноцінних локальних барбершопів. Операційні переваги полягають у можливості професійного ведення клієнтської бази, відстеження статистики візитів та автоматичного запобігання конфліктам у розкладі (Double Booking) без потреби у дороговартісних регулярних підписках на перевантажені корпоративні аналоги. Це сприяє суттєвій економії часових та фінансових ресурсів закладу, дозволяючи майстрам та адміністраторам зосередитися безпосередньо на наданні якісного сервісу клієнтам, а не на організаційній рутині.

А.6 Стадії й етапи розробки

Розробка застосунку включає наступні етапи:

– постановка завдання (1 тиждень);
– аналіз предметної області (1–2 тижні);
– розробка архітектури (1 тиждень);
– розробка програми (1–2 тижні);
– тестування (1 тиждень);
– оформлення документації (1 тиждень).

А.7 Порядок контролю та приймання

Процес підтвердження працездатності платформи «Velvet Razor» включає комплексний багаторівневий цикл тестування. Юніт-тестування (Unit testing) дозволяє гарантувати безпомилкову роботу відокремлених програмних модулів, таких як складні алгоритми динамічного розрахунку вільних слотів та математична логіка валідації робочих графіків. Інтеграційне тестування зосереджене на перевірці надійної транзакційної взаємодії між фронтендом, REST-бекендом та базою даних, зокрема стабільності механізму запобігання подвійному (конкурентному) бронюванню. Окремий етап присвячений тестуванню інтерфейсу (UI/UX), що забезпечує бездоганну інтуїтивність багатокрокового майстра запису, коректність абсолютного позиціонування карток в адміністративному журналі (Grid) та стабільну роботу розроблених компонентів UI.

Працездатність системи суворо перевіряється в актуальних версіях веббраузерів (Google Chrome, Safari, Firefox) як на великих десктопних моніторах, так і на екранах мобільних пристроїв під управлінням ОС Android та iOS. Особлива увага приділяється поведінці застосунку при втраті зв'язку із сервером (виведення інформативних помилок), безпечній валідації сесій через HttpOnly-cookies та коректності гідратації глобального стану за допомогою Zustand. Після успішного завершення всіх ітерацій тестування та підтвердження повної відповідності технічним вимогам, закладеним у специфікації, система передається замовнику для фінального приймання та введення в комерційну експлуатацію.

ДОДАТОК Б

Текст програми

Б.1 Код файлу auth.service.ts

```
import {
  Injectable,
  UnauthorizedException,
  BadRequestException,
  ForbiddenException,
} from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { ConfigService } from '@nestjs/config';
import * as argon2 from 'argon2';
import { randomBytes } from 'crypto';
import type { Response } from 'express';
import { PrismaService } from '../core/database/prisma.service';
import { isDev } from '../common/constants/environment';
import { RegisterDto, LoginDto, ResendCodeDto, VerifyPhoneDto, OtpVerifyDto } from './dto/auth.dto';
import { CodeType } from '@prisma/client';
import { JwtPayload } from '../common/interfaces/jwt-payload.interface';
import { COOKIE_OPTIONS, ACCESS_COOKIE_OPTIONS, REFRESH_COOKIE, ACCESS_COOKIE } from './auth.constants';

@Injectable()
export class AuthService {
  constructor(
    private readonly prisma: PrismaService,
    private readonly jwtService: JwtService,
    private readonly config: ConfigService,
  ) {}
```

```
// ——— Token Generation ———————————————————————————————————————————

  private async generateTokenPair(payload: JwtPayload) {
    const accessToken = await this.jwtService.signAsync(payload, {
      secret: this.config.getOrThrow<string>('JWT_SECRET'),
      expiresIn: '15m',
    });

    const rawRefreshToken = randomBytes(64).toString('hex');
    const expiresAt = new Date();
    expiresAt.setDate(expiresAt.getDate() + 7);

    // Enforce max 5 concurrent sessions per user — delete oldest if exceeded
    const existingTokens = await this.prisma.client.refreshToken.findMany({
      where: { userId: payload.sub },
      orderBy: { expiresAt: 'asc' },
    });

    const MAX_SESSIONS = 5;
    if (existingTokens.length >= MAX_SESSIONS) {
      const toDelete = existingTokens.slice(0, existingTokens.length - MAX_SESSIONS + 1);
      await this.prisma.client.refreshToken.deleteMany({
        where: { id: { in: toDelete.map(t => t.id) } },
      });
    }

    await this.prisma.client.refreshToken.create({
```

```
    data: { token: rawRefreshToken, userId: payload.sub, expiresAt },
   });

   return { accessToken, refreshToken: rawRefreshToken };
  }

  // ── Auth Flows ──────────────────────────────────────────────────────

  /**
   * Registration: does NOT create the User yet.
   * Instead stores { fullName, passwordHash } in VerificationCode.payload.
   * The User is created only after verifyPhone() confirms the code.
   */
  async register(registerDto: RegisterDto) {
   // Check for existing verified user with this phone
   const existingUser = await this.prisma.client.user.findUnique({
    where: { phone: registerDto.phone },
   });
   if (existingUser) {
      throw new BadRequestException('Користувач з таким номером телефону вже існує');
   }

   // Delete any stale pending registration codes for this phone
   await this.prisma.client.verificationCode.deleteMany({
    where: { phone: registerDto.phone, type: CodeType.REGISTRATION },
   });
```

```
    const passwordHash = await argon2.hash(registerDto.password);
    const code = this.generateCode();
    const expiresAt = new Date(Date.now() + 10 * 60_000); // 10 min

    await this.prisma.client.verificationCode.create({
      data: {
        phone: registerDto.phone,
        code,
        type: CodeType.REGISTRATION,
        expiresAt,
        payload: { fullName: registerDto.name, passwordHash },
      },
    });

      if (isDev()) console.log(`[Mock SMS] Код реєстрації для ${registerDto.phone}: ${code}`);

    return {
      message: 'Код підтвердження відправлено. Будь ласка, підтвердіть ваш номер телефону.',
      ...(isDev() && { devCode: code }),
    };
  }

  async login(loginDto: LoginDto) {
    const user = await this.prisma.client.user.findUnique({
      where: { phone: loginDto.phone },
    });
```

```
        // Use constant-time comparison to prevent user enumeration
        if (!user) {
          await argon2.hash('dummy_password_for_timing');
          throw new UnauthorizedException('Невірні облікові дані');
        }

          const isPasswordValid = await argon2.verify(user.passwordHash, loginDto.password);
        if (!isPasswordValid) {
          throw new UnauthorizedException('Невірні облікові дані');
        }

          const payload: JwtPayload = { sub: user.id, phone: user.phone, role: user.role };
                const { accessToken, refreshToken } = await this.generateTokenPair(payload);

        return {
          accessToken,
          refreshToken,
          user: {
            id: user.id,
            phone: user.phone,
            role: user.role,
            isPhoneVerified: user.isPhoneVerified,
          },
        };
      }

      async refresh(rawToken: string) {
```

```
    const stored = await this.prisma.client.refreshToken.findUnique({
      where: { token: rawToken },
      include: { user: true },
    });

    if (!stored || stored.expiresAt < new Date()) {
      if (stored) await this.prisma.client.refreshToken.delete({ where: { id: stored.id } });
      throw new UnauthorizedException('Refresh token недійсний або прострочений');
    }

    // Token rotation: delete old, issue new pair
    await this.prisma.client.refreshToken.delete({ where: { id: stored.id } });

    const payload: JwtPayload = {
      sub: stored.user.id,
      phone: stored.user.phone,
      role: stored.user.role,
    };

    return this.generateTokenPair(payload);
  }

  async logout(userId: string, rawToken?: string) {
    if (rawToken) {
      await this.prisma.client.refreshToken.deleteMany({
        where: { token: rawToken, userId },
      });
    } else {
```

```
    await this.prisma.client.refreshToken.deleteMany({ where: { userId } });
   }
   return { message: 'Вихід виконано успішно' };
  }

  // ── Verification ──────────────────────────────────────────────────────

  /**
   * Unified phone verification handler.
    * - REGISTRATION: creates the User from payload, marks isPhoneVerified=true.
   * - PHONE:        marks isPhoneVerified=true for the existing user.
   * - PHONE_CHANGE: updates the user's phone to the new value stored in payload.
   */
  async verifyPhone(dto: VerifyPhoneDto) {
   const record = await this.prisma.client.verificationCode.findFirst({
    where: {
     phone: dto.phone,
     code: dto.code,
          type: { in: [CodeType.REGISTRATION, CodeType.PHONE, CodeType.PHONE_CHANGE] },
    },
   });

   if (!record || record.expiresAt < new Date()) {
    if (record) await this.prisma.client.verificationCode.delete({ where: { id: record.id } });
```

```
        throw new BadRequestException('Невірний або прострочений код підтвердження');
      }

      await this.prisma.client.verificationCode.delete({ where: { id: record.id } });

      if (record.type === CodeType.REGISTRATION) {
        // Create the actual user from the stored payload
        const p = record.payload as { fullName: string; passwordHash: string };
        await this.prisma.client.user.create({
          data: {
            fullName: p.fullName,
            phone: dto.phone,
            passwordHash: p.passwordHash,
            isPhoneVerified: true,
          },
        });
        return { message: 'Реєстрація успішна! Тепер ви можете увійти.' };
      }

      if (record.type === CodeType.PHONE_CHANGE) {
        // Activate the new phone number for the existing user
        const p = record.payload as { newPhone: string };
        await this.prisma.client.user.update({
          where: { id: record.userId! },
          data: { phone: p.newPhone, isPhoneVerified: true },
        });
        return { message: 'Номер телефону успішно змінено!' };
      }
```

```
        // CodeType.PHONE — simple re-verification for existing unverified user
        await this.prisma.client.user.update({
          where: { id: record.userId! },
          data: { isPhoneVerified: true },
        });
        return { message: 'Номер телефону успішно підтверджено' };
      }

      /**
       * Initiates a phone number change for an authenticated user.
       * Does NOT update the phone immediately — sends a PHONE_CHANGE code to
       * the new number. The phone is only updated after verifyPhone() confirms it.
       */
      async requestPhoneChange(userId: string, newPhone: string) {
        // Ensure new phone isn't already taken by another account
        const existing = await this.prisma.client.user.findUnique({ where: { phone: newPhone } });
        if (existing && existing.id !== userId) {
          throw new BadRequestException('Цей номер вже використовується іншим акаунтом');
        }

        // Delete any stale PHONE_CHANGE codes for this user
        await this.prisma.client.verificationCode.deleteMany({
          where: { userId, type: CodeType.PHONE_CHANGE },
        });
```

```
    const code = this.generateCode();
    const expiresAt = new Date(Date.now() + 10 * 60_000);

    await this.prisma.client.verificationCode.create({
      data: {
        userId,
        phone: newPhone,
        code,
        type: CodeType.PHONE_CHANGE,
        expiresAt,
        payload: { newPhone },
      },
    });

      if (isDev()) console.log(`[Mock SMS] Код зміни номера для ${newPhone}: ${code}`);

    return {
      message: `Код підтвердження відправлено на ${newPhone}`,
      ...(isDev() && { devCode: code }),
    };
  }

  async resendCode(resendCodeDto: ResendCodeDto) {
    const { phone } = resendCodeDto;

    // Look for any pending verification code for this phone
    const existing = await this.prisma.client.verificationCode.findFirst({
      where: {
        phone,
```

```
                type: { in: [CodeType.REGISTRATION, CodeType.PHONE,
CodeType.PHONE_CHANGE] },
              },
              orderBy: { createdAt: 'desc' },
            });

            if (!existing) {
              // Fallback: check if user exists and phone is unverified (legacy PHONE
type)
              const user = await this.prisma.client.user.findUnique({ where: { phone }
});
               if (!user) throw new BadRequestException('Немає активного коду
підтвердження для цього номера');
                if (user.isPhoneVerified) throw new BadRequestException('Номер
телефону вже підтверджено');

              // Re-send a legacy PHONE verification code
              const code = this.generateCode();
              const expiresAt = new Date(Date.now() + 10 * 60_000);
              await this.prisma.client.verificationCode.create({
                data: { userId: user.id, phone, code, type: CodeType.PHONE, expiresAt
},
              });
              if (isDev()) console.log(`[Mock SMS] Новий код підтвердження для
${phone}: ${code}`);
              return {
                message: 'Код підтвердження успішно відправлено',
                ...(isDev() && { devCode: code }),
              };
            }
```

```
        // Re-create the code of the same type, preserving payload
        await this.prisma.client.verificationCode.delete({ where: { id: existing.id
} });
        const code = this.generateCode();
        const expiresAt = new Date(Date.now() + 10 * 60_000);
        await this.prisma.client.verificationCode.create({
          data: {
            userId: existing.userId ?? undefined,
            phone,
            code,
            type: existing.type,
            expiresAt,
            payload: existing.payload ?? undefined,
          },
        });

        if (isDev()) console.log(`[Mock SMS] Новий код підтвердження для
${phone}: ${code}`);

        return {
          message: 'Код підтвердження успішно відправлено',
          ...(isDev() && { devCode: code }),
        };
      }

      // ——— OTP (Passwordless) Auth
——————————————————————————————————————

      /**
```

```
     * Initiates an OTP login for any phone number.
     * Works for: new users (no account), shadow users (admin-created), regular users.
     * Always returns 200 to avoid leaking whether the account exists.
     */
    async otpSend(phone: string) {
     const user = await this.prisma.client.user.findUnique({ where: { phone } });

     // Clean up stale PHONE-type OTP codes for this number
     await this.prisma.client.verificationCode.deleteMany({
      where: { phone, type: CodeType.PHONE },
     });

     const code      = this.generateCode();
     const expiresAt = new Date(Date.now() + 10 * 60_000); // 10 min

     await this.prisma.client.verificationCode.create({
      data: {
       userId:    user?.id,   // null for brand-new users — created on otpVerify
       phone,
       code,
       type:      CodeType.PHONE,
       expiresAt,
      },
     });

     if (isDev()) console.log(`[Mock SMS] OTP для ${phone}: ${code}`);

     return {
```

```
      message: 'OTP надіслано',
      ...(isDev() && { devCode: code }),
    };
  }

  /**
   * Verifies an OTP code and issues JWT cookies.
   * - If userId present in code record — updates isPhoneVerified, logs in.
   * - If no userId (brand-new phone) — auto-creates CLIENT user, then logs in.
   * Returns { user, isNewUser } so the frontend can show a welcome message.
   */
  async otpVerify(dto: OtpVerifyDto, res: Response) {
    const record = await this.prisma.client.verificationCode.findFirst({
      where: { phone: dto.phone, code: dto.code, type: CodeType.PHONE },
      include: { user: true },
    });

    if (!record || record.expiresAt < new Date()) {
      if (record) await this.prisma.client.verificationCode.delete({ where: { id: record.id } });
      throw new BadRequestException('Невірний або прострочений OTP код');
    }

    await this.prisma.client.verificationCode.delete({ where: { id: record.id } });

    let user = record.user;
```

```
let isNewUser = false;

if (!user) {
  // Brand-new phone — auto-register as CLIENT (passwordless)
  user = await this.prisma.client.user.create({
    data: {
      fullName:       dto.phone, // placeholder, updatable in profile
      phone:          dto.phone,
      passwordHash:   '',
      role:           'CLIENT',
      isPhoneVerified: true,
    },
  });
  isNewUser = true;
} else {
  // Existing user (regular, shadow, or re-login) — mark phone verified
  user = await this.prisma.client.user.update({
    where: { id: user.id },
    data:  { isPhoneVerified: true },
  });
}

 const payload: JwtPayload = { sub: user.id, phone: user.phone!, role: user.role };
        const { accessToken, refreshToken } = await this.generateTokenPair(payload);

res.cookie(REFRESH_COOKIE, refreshToken, COOKIE_OPTIONS);
 res.cookie(ACCESS_COOKIE, accessToken, ACCESS_COOKIE_OPTIONS);
```

```
    return {
      user: {
        id:             user.id,
        phone:          user.phone,
        role:           user.role,
        isPhoneVerified: true,
      },
      isNewUser,
    };
  }

  // ─── Helpers ──────────────────────────────────────────────────────

  private generateCode(): string {
    // 6-digit numeric code: 100000–999999
    return String(100000 + (randomBytes(3).readUIntBE(0, 3) % 900000));
  }
}
```

Б.2 Коду файлу schedules.service.ts

```
import { Injectable, BadRequestException, ForbiddenException, ConflictException, Logger } from '@nestjs/common';
import { PrismaService } from '../core/database/prisma.service';
import { CreateScheduleDto, CreateExceptionDto, GetAvailabilityDto } from './dto/schedules.dto';
import { BookingStatus, Role } from '@velvet-razor/database';
import { JwtPayload } from '../common/interfaces/jwt-payload.interface';
```

```
@Injectable()
export class SchedulesService {
  private readonly logger = new Logger(SchedulesService.name);
  constructor(private prisma: PrismaService) {}

  // ─── Ownership helper ──────────────────────────────────────────────────────

  /**
   * For Prisma `@db.Date` fields: parse a YYYY-MM-DD string as UTC midnight.
   * e.g. "2026-05-10" → 2026-05-10T00:00:00.000Z
   * This ensures PostgreSQL DATE stores the CORRECT calendar date regardless of server TZ.
   */
  private parseDateUTC(dateStr: string): Date {
    return new Date(dateStr + 'T00:00:00.000Z');
  }

  /**
   * For Appointment.startTime (`DateTime`) range queries: local day boundaries.
   * Returns [localMidnight, localEndOfDay] so we match appointments booked in local time.
   */
  private getLocalDayRange(dateStr: string): { start: Date; end: Date } {
    const [y, m, d] = dateStr.split('-').map(Number);
    return {
```

```
    start: new Date(y, m - 1, d, 0, 0, 0, 0),
    end:   new Date(y, m - 1, d, 23, 59, 59, 999),
   };
  }

    private async resolveBarberIdForUser(user: JwtPayload, requestedBarberId: string): Promise<string> {
   if (user.role === Role.ADMIN) return requestedBarberId;

   // BARBER may only manage their own schedule
    const barber = await this.prisma.client.barber.findUnique({ where: { userId: user.sub } });
   if (!barber) throw new ForbiddenException('Ваш профіль барбера не знайдено');
   if (barber.id !== requestedBarberId)
    throw new ForbiddenException('Ви можете керувати тільки власним розкладом');
   return barber.id;
  }

  // ── Schedule CRUD ──────────────────────────────────────────────────────

  /**
   * Returns all future PENDING/CONFIRMED appointments for the given barber on `dayOfWeek`.
   * If newStartTime/newEndTime are provided, only returns appointments that fall OUTSIDE
   * (i.e. conflict with) the new working hours.
```

```
        * If no time range is given (e.g. deleting the day entirely), returns ALL future appointments
        * on that day of week.
        */
       async getScheduleConflicts(
        barberId: string,
        dayOfWeek: number,
        newStartTime?: string,
        newEndTime?: string,
        user?: JwtPayload,
       ) {
        if (user) await this.resolveBarberIdForUser(user, barberId);

        const now = new Date();
        const appointments = await this.prisma.client.appointment.findMany({
         where: {
          barberId,
          startTime: { gte: now },
          status: { in: [BookingStatus.PENDING, BookingStatus.CONFIRMED] },
         },
         include: {
          client: { select: { id: true, fullName: true, phone: true } },
          services: { include: { service: { select: { name: true } } } },
         },
         orderBy: { startTime: 'asc' },
        });

        // Filter to only appointments on the target dayOfWeek
```

```
        const onDay = appointments.filter(a => a.startTime.getDay() === dayOfWeek);

        // If no new hours given — all appointments on this day conflict (day is being deleted)
        if (!newStartTime || !newEndTime) return onDay;

        // Otherwise: return only appointments that start/end outside the new shift window
        const newStart = this.timeToMinutes(newStartTime);
        const newEnd   = this.timeToMinutes(newEndTime);

        return onDay.filter(a => {
            const apptStartMins = a.startTime.getHours() * 60 + a.startTime.getMinutes();
            const apptEndMins   = a.endTime.getHours()   * 60 + a.endTime.getMinutes();
            // Conflict: appointment starts before new window starts OR ends after new window ends
            return apptStartMins < newStart || apptEndMins > newEnd;
        });
    }

    async createSchedule(data: CreateScheduleDto, user: JwtPayload) {
        await this.resolveBarberIdForUser(user, data.barberId);

        const existing = await this.prisma.client.workSchedule.findFirst({
            where: { barberId: data.barberId, dayOfWeek: data.dayOfWeek },
        });
```

```
        if (existing && !data.force) {
          // Return a structured error so the frontend can ask "Replace?"
          throw new ConflictException({
            code: 'SCHEDULE_EXISTS',
            message: 'У майстра вже є зміна на цей день. Замінити?',
            existing: { startTime: existing.startTime, endTime: existing.endTime },
          });
        }

        // If cancelConflicts: cancel future appointments that fall outside the new hours
        if (data.force && data.cancelConflicts) {
          const conflicts = await this.getScheduleConflicts(
            data.barberId,
            data.dayOfWeek,
            data.startTime,
            data.endTime,
          );
          for (const appt of conflicts) {
            await this.prisma.client.appointment.update({
              where: { id: appt.id },
                data: { status: BookingStatus.CANCELLED, statusUpdatedAt: new Date() },
            });
            this.logger.log(
                `[SMS STUB] Appointment ${appt.id} cancelled due to schedule change. ` +
              `Client: ${appt.client?.fullName} (${appt.client?.phone})`,
            );
          }
```

```
        this.logger.log(`[SCHEDULE] Cancelled ${conflicts.length} conflict(s) for barber ${data.barberId} day ${data.dayOfWeek}`);
      }

      if (existing) {
        return this.prisma.client.workSchedule.update({
          where: { id: existing.id },
          data: { startTime: data.startTime, endTime: data.endTime },
        });
      }
      return this.prisma.client.workSchedule.create({
        data: { barberId: data.barberId, dayOfWeek: data.dayOfWeek, startTime: data.startTime, endTime: data.endTime },
      });
    }

    async getBarberSchedule(barberId: string) {
      return this.prisma.client.workSchedule.findMany({
        where: { barberId },
        orderBy: { dayOfWeek: 'asc' },
      });
    }

    async deleteScheduleDay(
      barberId: string,
      dayOfWeek: number,
      user: JwtPayload,
      cancelConflicts = false,
    ) {
      await this.resolveBarberIdForUser(user, barberId);
```

```
        // If cancelConflicts: cancel ALL future appointments on this day of week
        if (cancelConflicts) {
          const conflicts = await this.getScheduleConflicts(barberId, dayOfWeek);
          for (const appt of conflicts) {
            await this.prisma.client.appointment.update({
              where: { id: appt.id },
                data: { status: BookingStatus.CANCELLED, statusUpdatedAt: new
Date() },
            });
            this.logger.log(
              `[SMS STUB] Appointment ${appt.id} cancelled due to day removal
from schedule. ` +
              `Client: ${appt.client?.fullName} (${appt.client?.phone})`,
            );
          }
          this.logger.log(`[SCHEDULE] Cancelled ${conflicts.length} conflict(s)
before removing day ${dayOfWeek} for barber ${barberId}`);
        }

        return this.prisma.client.workSchedule.deleteMany({ where: { barberId,
dayOfWeek } });
      }

      // ── Exceptions CRUD
──────────────────────────────────────────────────────────────────

      async createException(data: CreateExceptionDto, user: JwtPayload) {
        await this.resolveBarberIdForUser(user, data.barberId);
```

```
        // Build list of all UTC-midnight dates in the range [date .. dateEnd]
        const dates: Date[] = [];
        const cursor = this.parseDateUTC(data.date);
          const end = data.dateEnd ? this.parseDateUTC(data.dateEnd) : this.parseDateUTC(data.date);

        while (cursor <= end) {
         dates.push(new Date(cursor));
          cursor.setUTCDate(cursor.getUTCDate() + 1); // advance in UTC to avoid DST edge cases
        }

         if (dates.length === 0) throw new BadRequestException('Невірний діапазон дат');

                        // ── Existing-exception guard ──────────────────────────────────────────

        // Check if any of the requested dates already have an exception for this barber.
                        const existingExceptions = await this.prisma.client.scheduleException.findMany({
         where: {
          barberId: data.barberId,
          date: { in: dates },
         },
         orderBy: { date: 'asc' },
        });

        if (existingExceptions.length > 0 && !data.force) {
```

```
            // Build a human-readable description of what's already there
            const descriptions = existingExceptions.map(ex => {
              const dateLabel = new Date(ex.date).toLocaleDateString('uk-UA', {
                day: 'numeric', month: 'long', timeZone: 'UTC',
              });
                  const typeLabel = ex.isDayOff ? 'Вихідний' : `Особл. час ${ex.startTime}–${ex.endTime}`;
              return `${dateLabel}: ${typeLabel}`;
            });

            throw new ConflictException({
              code: 'EXCEPTION_EXISTS',
              message: 'На деякі дати вже є винятки. Замінити?',
              existing: descriptions,
            });
          }

          // If force=true: delete all old exceptions for these exact dates before inserting
          if (existingExceptions.length > 0 && data.force) {
            await this.prisma.client.scheduleException.deleteMany({
              where: {
                barberId: data.barberId,
                date: { in: dates },
              },
            });
            this.logger.log(`[EXCEPTION] Replaced ${existingExceptions.length} existing exception(s) for barber ${data.barberId}`);
          }
```

```
// ── End guard ──────────────────────────────────────────────────────────

    // If cancelConflicts requested, find and cancel conflicting appointments.
    // For custom (non-dayoff) exceptions, only cancel those that fall outside the new time window.
    if (data.cancelConflicts) {
      const conflicting = await this.getConflicts(
        data.barberId,
        data.date,
        data.dateEnd ?? undefined,
        // Pass custom window only for non-dayoff exceptions
        data.isDayOff ? undefined : (data.startTime ?? undefined),
        data.isDayOff ? undefined : (data.endTime ?? undefined),
      );

      for (const appt of conflicting) {
        await this.prisma.client.appointment.update({
          where: { id: appt.id },
          data: { status: BookingStatus.CANCELLED, statusUpdatedAt: new Date() },
        });
        // [STUB] SMS notification to client
        this.logger.log(
          `[SMS STUB] Appointment ${appt.id} cancelled due to barber exception. ` +
          `Client: ${appt.client?.fullName} (${(appt as any).client?.phone})`,
        );
      }
```

```
            this.logger.log(`[EXCEPTION] Cancelled ${conflicting.length}
appointment(s) for barber ${data.barberId}`);
        }

        // Create one ScheduleException per date in the range
        const created = await this.prisma.client.$transaction(
          dates.map(d =>
            this.prisma.client.scheduleException.create({
              data: {
                barberId: data.barberId,
                date: d,
                isDayOff: data.isDayOff,
                startTime: data.startTime ?? null,
                endTime: data.endTime ?? null,
              },
            })
          )
        );

        return created;
      }

      async getBarberExceptions(barberId: string, from?: string, to?: string) {
        const where: any = { barberId };
        if (from || to) {
          where.date = {};
          if (from) where.date.gte = this.parseDateUTC(from);
          if (to)   where.date.lte = this.parseDateUTC(to);
        }
```

```
        return this.prisma.client.scheduleException.findMany({ where, orderBy: { date: 'asc' } });
      }

      async deleteException(id: string, user: JwtPayload) {
        const exception = await this.prisma.client.scheduleException.findUnique({ where: { id } });
        if (!exception) throw new BadRequestException('Виняток не знайдено');
        await this.resolveBarberIdForUser(user, exception.barberId);
        return this.prisma.client.scheduleException.delete({ where: { id } });
      }

      /**
       * Admin: returns active appointments for a barber in a date range.
       * If customStartTime/customEndTime are provided (custom exception type),
       * only returns appointments that fall OUTSIDE the new time window
       * (i.e. they would no longer be executable by the barber).
       */
      async getConflicts(
        barberId: string,
        dateStart: string,
        dateEnd?: string,
        customStartTime?: string,
        customEndTime?: string,
        user?: JwtPayload,
      ) {
        if (user) await this.resolveBarberIdForUser(user, barberId);

        const { start } = this.getLocalDayRange(dateStart);
```

```
        const { end }   = this.getLocalDayRange(dateEnd ?? dateStart);

        const appointments = await this.prisma.client.appointment.findMany({
          where: {
            barberId,
            startTime: { gte: start, lte: end },
            status: { in: [BookingStatus.PENDING, BookingStatus.CONFIRMED]
},
          },
          include: {
            client: { select: { id: true, fullName: true, phone: true } },
            services: { include: { service: { select: { name: true } } } },
          },
          orderBy: { startTime: 'asc' },
        });

        // For custom (non-dayoff) exceptions: only flag appointments that exceed
the new window
        if (customStartTime && customEndTime) {
          const winStart = this.timeToMinutes(customStartTime);
          const winEnd   = this.timeToMinutes(customEndTime);
          return appointments.filter(a => {
                  const apptStart = a.startTime.getHours() * 60 +
a.startTime.getMinutes();
                const apptEnd   = a.endTime.getHours()   * 60 +
a.endTime.getMinutes();
            // Conflict: appointment starts before window opens OR ends after
window closes
            return apptStart < winStart || apptEnd > winEnd;
          });
```

```
    }

    return appointments;
  }

  /**
   * Admin: returns all work schedules and exceptions for all barbers
    * in a single aggregated object, used for rendering the schedule management table.
   */
  async getAllSchedules() {
    const [schedules, exceptions] = await Promise.all([
      // Only include active (non-archived) barbers
      this.prisma.client.workSchedule.findMany({
        where: { barber: { isArchived: false } },
        orderBy: [{ barberId: 'asc' }, { dayOfWeek: 'asc' }],
      }),
      this.prisma.client.scheduleException.findMany({
        where: { barber: { isArchived: false } },
        orderBy: [{ barberId: 'asc' }, { date: 'asc' }],
      }),
    ]);
    return { schedules, exceptions };
  }

  // ─── Availability engine ──────────────────────────────────────────────────

  private timeToMinutes(t: string): number {
```

```
  const [h, m] = t.split(':').map(Number);
  return h * 60 + m;
 }

 private minutesToTime(m: number): string {
     return `${Math.floor(m / 60).toString().padStart(2, '0')}:${(m % 60).toString().padStart(2, '0')}`;
 }

 async getAvailability(query: GetAvailabilityDto) {
  const { barberId, date, serviceIds } = query;
  const targetDate = this.parseDateUTC(date);
   const dayOfWeek = targetDate.getUTCDay(); // use UTC to match the @db.Date storage

  // FIX: Batch all service lookups into 2 queries instead of N*2
  const [barberServices, allServices] = await Promise.all([
   this.prisma.client.barberService.findMany({
    where: { barberId, serviceId: { in: serviceIds } },
    include: { service: true },
   }),
   this.prisma.client.service.findMany({
    where: { id: { in: serviceIds } },
   }),
  ]);

    const barberServiceMap = new Map(barberServices.map(bs => [bs.serviceId, bs]));
  const globalServiceMap = new Map(allServices.map(s => [s.id, s]));
```

```
        let totalDuration = 0;
        for (const serviceId of serviceIds) {
          const bs = barberServiceMap.get(serviceId);
          if (bs?.customDuration) {
            totalDuration += bs.customDuration;
          } else {
            const s = globalServiceMap.get(serviceId);
              if (!s) throw new BadRequestException(`Service ${serviceId} not found`);
            totalDuration += s.baseDuration + s.bufferTime;
          }
        }

        // Working hours for the day
        let startMins = 0;
        let endMins = 0;

        const exceptionDate = this.parseDateUTC(date);      // @db.Date — must be UTC midnight
                const { start: startOfDay, end: endOfDay } = this.getLocalDayRange(date); // appointment DateTime range

        const [exception, schedule, appointments] = await Promise.all([
          this.prisma.client.scheduleException.findFirst({ where: { barberId, date: exceptionDate } }),
                this.prisma.client.workSchedule.findFirst({ where: { barberId, dayOfWeek } }),
          this.prisma.client.appointment.findMany({
            where: { barberId, startTime: { gte: startOfDay, lte: endOfDay }, status: { not: BookingStatus.CANCELLED } },
```

```
      select: { startTime: true, endTime: true },
    }),
  ]);

  if (exception) {
    if (exception.isDayOff) return [];
    if (exception.startTime && exception.endTime) {
      startMins = this.timeToMinutes(exception.startTime);
      endMins = this.timeToMinutes(exception.endTime);
    }
  } else {
    if (!schedule) return [];
    startMins = this.timeToMinutes(schedule.startTime);
    endMins = this.timeToMinutes(schedule.endTime);
  }

  if (startMins >= endMins) return [];

  const bookedIntervals = appointments.map(a => ({
    start: a.startTime.getHours() * 60 + a.startTime.getMinutes(),
    end: a.endTime.getHours() * 60 + a.endTime.getMinutes(),
  }));

  const availableSlots: string[] = [];
  for (let time = startMins; time <= endMins - totalDuration; time += 15) {
    const slotEnd = time + totalDuration;
    const hasOverlap = bookedIntervals.some(i => time < i.end && slotEnd
> i.start);
    if (!hasOverlap) availableSlots.push(this.minutesToTime(time));
  }
```

```
    return availableSlots;
  }
}
```

Б.3 Код файлу appointments.service.ts

```
import {
  Injectable,
  BadRequestException,
  ForbiddenException,
  NotFoundException,
  ServiceUnavailableException,
  Logger,
} from '@nestjs/common';
import { PrismaService } from '../core/database/prisma.service';
import { SchedulesService } from '../schedules/schedules.service';
import { CreateAppointmentDto, AdminCreateAppointmentDto,
UpdateAppointmentStatusDto } from './dto/appointments.dto';
import { Role, BookingStatus, Prisma } from '@velvet-razor/database';
import { getPrismaSkipTake } from '../common/dto/pagination.dto';

/**
 * Retries an async fn up to `maxAttempts` times on Prisma P2034
 * (PostgreSQL serialization conflict — normal under high concurrency).
 */
async function withSerializableRetry<T>(
  fn: () => Promise<T>,
  maxAttempts = 5,
): Promise<T> {
  let lastErr: unknown;
```

```
    for (let attempt = 1; attempt <= maxAttempts; attempt++) {
     try {
      return await fn();
     } catch (err: any) {
      if (err?.code === 'P2034') {
       lastErr = err;
       // Exponential backoff with jitter: ~0–50ms, ~0–100ms, …
       const delay = Math.random() * 50 * attempt;
       await new Promise(r => setTimeout(r, delay));
       continue;
      }
      throw err; // not a serialization conflict — rethrow immediately
     }
    }
    throw new ServiceUnavailableException(
     'Сервер перевантажений. Будь ласка, спробуйте ще раз.',
    );
   }

   const APPOINTMENT_INCLUDE = {
    barber: { include: { user: { select: { id: true, phone: true, fullName: true,
role: true } }, outlet: true } },
    services: { include: { service: true } },
    client: { select: { id: true, phone: true, fullName: true } },
    outlet: true,
   } as const;

   @Injectable()
   export class AppointmentsService {
    private readonly logger = new Logger(AppointmentsService.name);
```

```
    constructor(
      private prisma: PrismaService,
      private schedulesService: SchedulesService,
    ) {}

    async create(clientId: string, data: CreateAppointmentDto) {
      if (data.startTime < new Date()) {
        throw new BadRequestException('Неможливо створити запис у минулому');
      }

      // ── 0. Prevent barber from booking themselves ──────────────────────
      const barberCheck = await this.prisma.client.barber.findUnique({
        where: { id: data.barberId },
        select: { userId: true },
      });
      if (barberCheck?.userId === clientId) {
        throw new BadRequestException('Ви не можете записатися до самого себе');
      }

      // ── 0.1. Block check + Prepayment gate ─────────────────────────────
      const clientFlags = await this.prisma.client.user.findUnique({
        where: { id: clientId },
        select: { isBlocked: true, requiresPrepayment: true },
      });
```

```
        if (clientFlags?.isBlocked) {
          throw new ForbiddenException(
            'Ваш обліковий запис заблоковано. Онлайн-запис недоступний. Будь ласка, зверніться до адміністратора.',
          );
        }

        if (clientFlags?.requiresPrepayment && !data.isPrepaidStub) {
          throw new BadRequestException(
            'Для створення запису необхідно внести передоплату (50%). Будь ласка, пройдіть процедуру оплати.',
          );
        }

        // ── 1. Resolve services & compute duration/price (outside the transaction) ──
        const [barberServices, allServices] = await Promise.all([
          this.prisma.client.barberService.findMany({
            where: { barberId: data.barberId, serviceId: { in: data.serviceIds } },
            include: { service: true },
          }),
          this.prisma.client.service.findMany({ where: { id: { in: data.serviceIds } } }),
        ]);

        const bsMap = new Map(barberServices.map(bs => [bs.serviceId, bs]));
        const sMap  = new Map(allServices.map(s => [s.id, s]));

        // ── Strict barber-service membership check
```

```
        // Every requested serviceId MUST be in the barber's BarberService list.
        // Without this, a client could book services that were removed from the
        // barber mid-flow, and the server would silently accept them at base price.
        const notOffered = data.serviceIds.filter(id => !bsMap.has(id));
        if (notOffered.length > 0) {
          // Resolve human-readable names for the error message
          const names = notOffered
            .map(id => sMap.get(id)?.name ?? id)
            .join(', ');
          throw new BadRequestException(
            `Майстер більше не надає такі послуги: ${names}. ` +
            `Будь ласка, оновіть вибір послуг.`,
          );
        }

        let totalDuration = 0;
        let totalPrice    = 0;

        for (const serviceId of data.serviceIds) {
          const bs = bsMap.get(serviceId);
          const s  = bs?.service ?? sMap.get(serviceId);
            if (!s) throw new BadRequestException(`Service ${serviceId} not found`);
                totalDuration += bs?.customDuration ?? (s.baseDuration + s.bufferTime);
          totalPrice    += Number(bs?.customPrice ?? s.basePrice);
        }

        const endTime = new Date(data.startTime.getTime() + totalDuration * 60_000);
```

```
        // ── 2. Active appointment limit — max 3 per client ──────────────────────
        const MAX_ACTIVE_APPOINTMENTS = 3;
        const MAX_CANCELLATIONS_PER_DAY = 3;

        const since24h = new Date(Date.now() - 24 * 60 * 60 * 1000);

        const [activeCount, recentCancellations] = await Promise.all([
          this.prisma.client.appointment.count({
            where: {
              clientId,
                                  status:    {    in:    [BookingStatus.PENDING,
BookingStatus.CONFIRMED] },
            },
          }),
          // Count bookings cancelled by the client themselves in the last 24 hours.
          // We use statusUpdatedAt as the cancellation timestamp proxy.
          this.prisma.client.appointment.count({
            where: {
              clientId,
              status: BookingStatus.CANCELLED,
              statusUpdatedAt: { gte: since24h },
            },
          }),
        ]);

        if (activeCount >= MAX_ACTIVE_APPOINTMENTS) {
          throw new BadRequestException(
```

```
                `Ви досягли ліміту активних записів (${MAX_ACTIVE_APPOINTMENTS}). ` +
                  `Для запису на додаткові послуги, будь ласка, зателефонуйте нам.`,
                );
              }

              if (recentCancellations >= MAX_CANCELLATIONS_PER_DAY) {
                throw new BadRequestException(
                  `Ви занадто часто скасовували записи за останні 24 години. ` +
                  `Для нового запису, будь ласка, зателефонуйте нам або зачекайте деякий час.`,
                );
              }

              // ── 3. Validate slot against barber's schedule (outside the transaction) ──
              const d       = data.startTime;
                  const dateStr = `${d.getFullYear()}-${String(d.getMonth() + 1).padStart(2, '0')}-${String(d.getDate()).padStart(2, '0')}`;
              const availableSlots = await this.schedulesService.getAvailability({
                barberId: data.barberId,
                date: dateStr,
                serviceIds: data.serviceIds,
              });

                  const requestedTime = `${d.getHours().toString().padStart(2, '0')}:${d.getMinutes().toString().padStart(2, '0')}`;
              if (!availableSlots.includes(requestedTime)) {
                throw new BadRequestException('Обраний час недоступний. Будь ласка, оберіть інший слот.');
```

```
        }

        // ── 3. Resolve outlet + income share snapshot from barber's profile ──────────
        const barberForOutlet = await this.prisma.client.barber.findUnique({
          where: { id: data.barberId },
          select: { outletId: true, incomeShare: true },
        });
        const resolvedOutletId = barberForOutlet?.outletId;
        // Snapshot the current income share so historical stats stay accurate
        // even after the admin later changes the barber's rate.
        const barberIncomeShare = barberForOutlet?.incomeShare ?? undefined;

        // ── 4. Short Serializable transaction with automatic retry on P2034 ──────────
        return withSerializableRetry(() =>
          this.prisma.client.$transaction(
          async (tx) => {
            const overlapping = await tx.appointment.findFirst({
              where: {
                barberId: data.barberId,
                status: { not: BookingStatus.CANCELLED },
                AND: [{ startTime: { lt: endTime } }, { endTime: { gt: data.startTime } }],
              },
            });

            if (overlapping) {
              throw new BadRequestException('Цей час вже зайнятий іншим клієнтом. Оберіть інший слот.');
```

```
    }

        // ── 4a. Compute prepayment data if applicable ──────────────────────
    const isPrepaid = data.isPrepaidStub === true;
    const prepaidAmount = isPrepaid
      ? new Prisma.Decimal(totalPrice * 0.5).toDecimalPlaces(2)
      : undefined;

    const appointment = await tx.appointment.create({
      data: {
        clientId,
        barberId: data.barberId,
        outletId: resolvedOutletId,
        startTime: data.startTime,
        endTime,
        totalPrice,
        barberIncomeShare,
        isPrepaid,
        prepaidAmount,
        notes: data.notes,
        services: { create: data.serviceIds.map(id => ({ serviceId: id })) },
      },
      include: { services: { include: { service: true } } },
    });

    if (isPrepaid) {
      this.logger.log(
        `[Mock Payment] Appointment ${appointment.id}: prepayment of ${prepaidAmount} UAH accepted for client ${clientId}.`,
```

```
            );
          }
          this.logger.log(`[STUB] Appointment created: ${appointment.id}`);
          return appointment;
        },
            { isolationLevel: Prisma.TransactionIsolationLevel.Serializable,
timeout: 15_000 },
        )); // end withSerializableRetry
      }

      /**
       * Admin-only "god mode" create: bypasses schedule availability and
overlap checks.
       * Admin provides explicit endTime and totalPrice.
       * outletId is ALWAYS resolved from the barber's profile — cannot be
overridden.
       */
      async adminCreate(data: AdminCreateAppointmentDto) {
        // Always resolve outletId + income share snapshot from barber's profile
        const barber = await this.prisma.client.barber.findUnique({
          where: { id: data.barberId },
          select: { outletId: true, incomeShare: true },
        });
        const resolvedOutletId = barber?.outletId ?? undefined;
        const barberIncomeShare = barber?.incomeShare ?? undefined;

        const appointment = await this.prisma.client.appointment.create({
          data: {
            clientId:        data.clientId,
            barberId:        data.barberId,
```

```
        outletId:        resolvedOutletId,
        startTime:       data.startTime,
        endTime:         data.endTime,
        totalPrice:      data.totalPrice,
        barberIncomeShare,
        notes:           data.notes,
        services:        { create: data.serviceIds.map(id => ({ serviceId: id })) },
      },
      include: APPOINTMENT_INCLUDE,
    });

    this.logger.log(`[ADMIN] Force-created appointment: ${appointment.id}`);
    return appointment;
  }

  async findOne(id: string, userId: string, role: Role) {
    const appointment = await this.prisma.client.appointment.findUnique({
      where: { id },
      include: {
        barber: { include: { user: { select: { id: true, role: true } } } },
        services: { include: { service: true } },
        client: { select: { id: true, phone: true, fullName: true } },
      },
    });

    if (!appointment) throw new NotFoundException('Запис не знайдено');

    // Ownership check
    const isOwner = appointment.clientId === userId;
```

```
    const isBarber = appointment.barber.userId === userId;
    const isAdmin = role === Role.ADMIN;

        if (!isOwner && !isBarber && !isAdmin) throw new ForbiddenException();

    return appointment;
  }

  async getMyClientAppointments(
    clientId: string,
    page: number,
    limit: number,
    filter: 'upcoming' | 'past' | 'cancelled' | 'all' = 'all',
  ) {
    const { skip, take } = getPrismaSkipTake(page, limit);
    const now = new Date();

    /**
     * Server-side filtering ensures that count() and data are always in sync,
     * so pagination never shows "phantom" pages.
     *
     * upcoming  — PENDING/CONFIRMED appointments whose endTime is in the future
     * past      — COMPLETED/NO_SHOW, OR any non-cancelled appointment that has expired
     * cancelled — CANCELLED status only
     * all       — no filter (default)
     */
    let statusWhere: object = {};
```

```
        switch (filter) {
          case 'upcoming':
            statusWhere = {
                                    status:    {    in:    [BookingStatus.PENDING,
BookingStatus.CONFIRMED] },
              endTime: { gt: now },
            };
            break;
          case 'past':
            statusWhere = {
              OR: [
                                  {    status:    {    in:    [BookingStatus.COMPLETED,
BookingStatus.NO_SHOW] } },
                {
                  status: { notIn: [BookingStatus.CANCELLED] },
                  endTime: { lte: now },
                },
              ],
            };
            break;
          case 'cancelled':
            statusWhere = { status: BookingStatus.CANCELLED };
            break;
          // 'all': no extra filter
        }

        const where = { clientId, ...statusWhere };

        const [data, total] = await Promise.all([
          this.prisma.client.appointment.findMany({
```

```
        where,
        include: APPOINTMENT_INCLUDE,
        orderBy: { startTime: filter === 'upcoming' ? 'asc' : 'desc' },
        skip,
        take,
      }),
      this.prisma.client.appointment.count({ where }),
    ]);

    return { data, total };
  }

  async getBarberAppointments(
    barberUserId: string,
    dateStr?: string,
    page = 1,
    limit = 200,
    startDate?: string,
    endDate?: string,
    fromDate?: string,
    status?: string,
  ) {
      const barber = await this.prisma.client.barber.findUnique({ where: { userId: barberUserId } });
    if (!barber) throw new ForbiddenException('Ви не є барбером');

    const { skip, take } = getPrismaSkipTake(page, limit);
    const where: any = { barberId: barber.id };

    if (dateStr) {
```

```
        // Single-day journal view
        const start = new Date(dateStr + 'T00:00:00'); start.setHours(0, 0, 0, 0);
        const end   = new Date(dateStr + 'T00:00:00'); end.setHours(23, 59, 59,
999);
        where.startTime = { gte: start, lte: end };
      } else if (startDate || endDate) {
        // Archive range view
        where.startTime = {};
        if (startDate) {
          const s = new Date(startDate + 'T00:00:00'); s.setHours(0, 0, 0, 0);
          where.startTime.gte = s;
        }
        if (endDate) {
          const e = new Date(endDate + 'T00:00:00'); e.setHours(23, 59, 59, 999);
          where.startTime.lte = e;
        }
      } else if (fromDate) {
        // Journal "All dates": today and beyond
        const f = new Date(fromDate + 'T00:00:00'); f.setHours(0, 0, 0, 0);
        where.startTime = { gte: f };
      }

        // Status scoping: supports single or comma-separated list
("PENDING,CONFIRMED")
      if (status) {
        const parts = status.split(',').map(s => s.trim()).filter(Boolean) as
BookingStatus[];
        if (parts.length === 1)    where.status = parts[0];
        else if (parts.length > 1) where.status = { in: parts };
      }
```

```
    const [data, total] = await Promise.all([
      this.prisma.client.appointment.findMany({
        where,
        include: {
          client: { select: { id: true, phone: true, fullName: true } },
          services: { include: { service: true } },
        },
        orderBy: { startTime: 'asc' },
        skip,
        take: dateStr ? 500 : take,
      }),
      this.prisma.client.appointment.count({ where }),
    ]);
    return { data, total };
  }

  /**
   * Admin findAll:
   * - `date`: returns all appointments for that specific day (grid view)
   * - `startDate`/`endDate`: returns appointments within a date range (archive view)
   * - `fromDate`: lower bound for startTime — used by Journal list to show today+future only
   * - No date params: returns paginated list, ordered by startTime desc
     * - `status`: single status or comma-separated list (e.g. "PENDING,CONFIRMED,COMPLETED")
   */
  async findAll(
    page: number,
```

```
    limit: number,
    status?: string,
    date?: string,
    startDate?: string,
    endDate?: string,
    fromDate?: string,
  ) {
    const { skip, take } = getPrismaSkipTake(page, limit);
    const where: any = {};

    // Multi-status support: "PENDING,CONFIRMED,COMPLETED" → { in: [...] }
    if (status) {
      const parts = status.split(',').map(s => s.trim()).filter(Boolean) as BookingStatus[];
      if (parts.length === 1) {
        where.status = parts[0];
      } else if (parts.length > 1) {
        where.status = { in: parts };
      }
    }

    if (date) {
      // Single-day mode (grid view)
      const start = new Date(date); start.setHours(0, 0, 0, 0);
      const end   = new Date(date); end.setHours(23, 59, 59, 999);
      where.startTime = { gte: start, lte: end };
    } else if (startDate || endDate) {
      // Date range mode (archive/history view)
      where.startTime = {};
```

```
    if (startDate) {
      const s = new Date(startDate); s.setHours(0, 0, 0, 0);
      where.startTime.gte = s;
    }
    if (endDate) {
      const e = new Date(endDate); e.setHours(23, 59, 59, 999);
      where.startTime.lte = e;
    }
  } else if (fromDate) {
    // Journal list mode: only today+future
    const f = new Date(fromDate); f.setHours(0, 0, 0, 0);
    where.startTime = { gte: f };
  }

  const [data, total] = await Promise.all([
    this.prisma.client.appointment.findMany({
      where,
      include: APPOINTMENT_INCLUDE,
      orderBy: { startTime: date ? 'asc' : 'desc' },
      skip,
      take: date ? 500 : take,
    }),
    this.prisma.client.appointment.count({ where }),
  ]);

  return { data, total, page, limit };
 }

  async updateStatus(id: string, userId: string, userRole: Role, dto:
UpdateAppointmentStatusDto) {
```

```
const appointment = await this.prisma.client.appointment.findUnique({
  where: { id },
  include: { barber: true },
});

if (!appointment) throw new NotFoundException('Запис не знайдено');

const oldStatus = appointment.status;
const newStatus = dto.status;

// ── 1. Ownership check ──────────────────────────────────────────────

if (userRole === Role.CLIENT && appointment.clientId !== userId) {
  throw new ForbiddenException('Це не ваш запис');
}
 if (userRole === Role.BARBER && appointment.barber.userId !== userId) {
   throw new ForbiddenException('Ви можете керувати лише своїми записами');
}

// ── 2. FSM: allowed transition matrices per role ────────────────────

//
// Format: ALLOWED_TRANSITIONS[role][oldStatus] = [allowedNewStatuses...]
// If a (oldStatus, newStatus) pair is NOT in the matrix → reject.
//
const S = BookingStatus;
```

```
const ALLOWED_TRANSITIONS: Partial<Record<Role, Partial<Record<BookingStatus, BookingStatus[]>>>> = {
  [Role.CLIENT]: {
    // Client can only cancel future appointments — history is concrete.
    [S.PENDING]:    [S.CANCELLED],
    [S.CONFIRMED]:  [S.CANCELLED],
  },

  [Role.BARBER]: {
    // Barber closes their own appointments (PENDING or CONFIRMED) once time has come.
    // PENDING → COMPLETED/NO_SHOW allowed so barbers don't need to ask admin to confirm first.
    [S.PENDING]:    [S.COMPLETED, S.NO_SHOW],
    [S.CONFIRMED]:  [S.COMPLETED, S.NO_SHOW],
    // Barber may correct their own mis-clicks between terminal states (<24h window).
    [S.NO_SHOW]:    [S.COMPLETED],
    [S.COMPLETED]:  [S.NO_SHOW],
  },

  [Role.ADMIN]: {
    // Standard journal transitions:
    [S.PENDING]:    [S.CONFIRMED, S.CANCELLED, S.COMPLETED],
    [S.CONFIRMED]:  [S.COMPLETED, S.NO_SHOW, S.CANCELLED],
    // Archive correction transitions (<24h window):
```

```
                    [S.CANCELLED]:    [S.CONFIRMED,    S.COMPLETED,
S.NO_SHOW],
                  [S.COMPLETED]:  [S.NO_SHOW, S.CANCELLED],
                  [S.NO_SHOW]:    [S.COMPLETED, S.CANCELLED],
                },
              };

              const allowedForRole = ALLOWED_TRANSITIONS[userRole];
              if (!allowedForRole) {
                throw new ForbiddenException('Невідома роль — зміна статусу
заборонена');
              }

                                  //  ──  Idempotency  guard
──────────────────────────────────────────────────────────────────────
──────

              // If two admins click the same button concurrently, the second request
arrives
              // after the first has already applied the transition. Instead of throwing a
              // confusing 400, we treat "same → same" as a no-op and return success.
              if (oldStatus === newStatus) {
                return this.prisma.client.appointment.findUnique({
                  where: { id },
                  include: APPOINTMENT_INCLUDE,
                });
              }

              const allowedNewStatuses = allowedForRole[oldStatus] ?? [];
              if (!allowedNewStatuses.includes(newStatus)) {
                throw new BadRequestException(
```

```
`Перехід зі статусу "${oldStatus}" до "${newStatus}" заборонений для вашої ролі.`,
        );
      }

      // ── 3. 24-hour correction window ──────────────────────────────────────

      // Applied when the current status is terminal (CANCELLED / COMPLETED / NO_SHOW)
      // and the transition is a reversal / correction.
      const TERMINAL_STATUSES: BookingStatus[] = [S.CANCELLED, S.COMPLETED, S.NO_SHOW];
      if (TERMINAL_STATUSES.includes(oldStatus)) {
        const ageMs = Date.now() - new Date(appointment.statusUpdatedAt).getTime();
        const TWENTY_FOUR_HOURS_MS = 24 * 60 * 60 * 1000;
        if (ageMs > TWENTY_FOUR_HOURS_MS) {
          throw new ForbiddenException(
            'Термін виправлення цього запису (24 год) вичерпано. Зміна статусу заборонена.',
          );
        }
      }

      // ── 4. Transactional update + audit log ───────────────────────────────

      const updated = await this.prisma.client.$transaction(async (tx) => {
        const result = await tx.appointment.update({
          where: { id },
          data: {
```

```
            status: newStatus,
            statusUpdatedAt: new Date(),
          },
        });

        await tx.bookingHistory.create({
          data: {
            appointmentId: id,
            changedById: userId,
            oldStatus,
            newStatus: dto.status,
          },
        });

        return result;
      });

      this.logger.log(
        `[AUDIT] User ${userId} (${userRole}) changed appointment ${id}: ${oldStatus} → ${dto.status}`,
      );

      // ── 5. [STUB] Status-change notifications ──────────────────────────

      // Fetch enough data to build personalised SMS messages
      const apptFull = await this.prisma.client.appointment.findUnique({
        where: { id },
        include: {
          client: { select: { phone: true, fullName: true } },
```

```
            barber:  { include: { user: { select: { phone: true, fullName: true } } }
},
          },
        });

        if (apptFull) {
            const startStr = apptFull.startTime.toLocaleString('uk-UA', {
timeZone: 'Europe/Kiev' });
          const clientPhone = apptFull.client.phone;
          const clientName  = apptFull.client.fullName;
          const barberPhone = apptFull.barber.user.phone;
          const barberName  = apptFull.barber.user.fullName;

          const STATUS_LABELS: Record<string, string> = {
            PENDING:   'Очікує',
            CONFIRMED: 'Підтверджено',
            CANCELLED: 'Скасовано',
            COMPLETED: 'Завершено',
            NO_SHOW:   'Неявка',
          };
          const newLabel = STATUS_LABELS[newStatus] ?? newStatus;

          if (userRole === Role.CLIENT) {
            // Client cancelled — notify the barber
            this.logger.log(
              `[Mock SMS → BARBER] ${barberPhone} (${barberName}): ` +
              `Клієнт ${clientName} скасував запис на ${startStr}.`,
            );
          } else {
            // Admin or Barber changed status — notify the client
```

```
        const messageMap: Partial<Record<string, string>> = {
          CONFIRMED: `Ваш запис на ${startStr} підтверджено. Чекаємо на вас!`,
          CANCELLED: `Ваш запис на ${startStr} було скасовано. Зверніться для перезапису.`,
          NO_SHOW:  `Ваш запис на ${startStr} позначено як Неявка. Якщо це помилка — зверніться до адміністратора.`,
          COMPLETED: `Дякуємо за візит! Чекаємо вас знову у Velvet Razor.`,
        };
        const message = messageMap[newStatus] ??
          `Статус вашого запису на ${startStr} змінено: «${newLabel}».`;
        this.logger.log(
          `[Mock SMS → CLIENT] ${clientPhone} (${clientName}): ${message}`,
        );
      }

      // ── Refund SMS stub: if a prepaid booking gets cancelled ──────────────
      if (newStatus === BookingStatus.CANCELLED && apptFull.isPrepaid && apptFull.prepaidAmount) {
        this.logger.log(
          `[Mock SMS → CLIENT] ${clientPhone} (${clientName}): ` +
          `Кошти за передоплату (${apptFull.prepaidAmount} грн) успішно повернено на ваш рахунок протягом 3–5 робочих днів.`,
        );
      }
    }
```

```
    return updated;
  }

  async remove(id: string) {
    const appointment = await this.prisma.client.appointment.findUnique({ where: { id } });
    if (!appointment) throw new NotFoundException('Запис не знайдено');
    return this.prisma.client.appointment.delete({ where: { id } });
  }
}
```

Б.4 Код файлу (dashboard)/admin/appointments/page.tsx

```
'use client';

import { useState, useMemo, useEffect } from 'react';
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query';
import { appointmentsApi } from '@/lib/api/appointments.api';
import { barbersApi } from '@/lib/api/barbers.api';
import { usersApi } from '@/lib/api/users.api';
import { servicesApi } from '@/lib/api/services.api';
import { outletsApi } from '@/lib/api/outlets.api';
import { schedulesApi } from '@/lib/api/schedules.api';
import { Modal } from '@/components/ui/Modal';
import { Button } from '@/components/ui/Button';
import { Input } from '@/components/ui/Input';
import { Spinner } from '@/components/ui/Spinner';
import { Pagination } from '@/components/ui/Pagination';
import { toDateString, formatTime, formatPrice } from '@/lib/utils/format';
```

```
import { layoutAppointments, TIME_SLOTS, SLOT_HEIGHT, GRID_HEIGHT } from '@/lib/utils/calendar';
import type { Appointment, Barber, Service } from '@/types/api.types';
import { extractErrorMessage } from '@/lib/utils/error';

// ─── Status config ──────────────────────────────────────────────────────────

const STATUS_COLOR: Record<string, string> = {
  PENDING:   'bg-[#E09B3D]/20 border-[#E09B3D]/50 text-[#E09B3D]',
  CONFIRMED: 'bg-[#4CAF6A]/20 border-[#4CAF6A]/50 text-[#4CAF6A]',
  COMPLETED: 'bg-[#C9A84C]/20 border-[#C9A84C]/50 text-[#C9A84C]',
  CANCELLED: 'bg-white/5 border-white/10 text-[#555]',
  NO_SHOW:   'bg-[#CF4444]/20 border-[#CF4444]/50 text-[#CF4444]',
};
const STATUS_LABELS: Record<string, string> = {
  PENDING: 'Очікує', CONFIRMED: 'Підтверджено', COMPLETED: 'Завершено',
  CANCELLED: 'Скасовано', NO_SHOW: 'Неявка',
};

const H24 = 24 * 60 * 60 * 1000;

/**
 * Returns available status transitions for ADMIN.
 * Enforces the 24h correction window for terminal statuses.
 */
```

```
function getAdminTransitions(appt: import('@/types/api.types').Appointment): string[] {
  const withinCorrection = Date.now() - new Date(appt.statusUpdatedAt ?? appt.startTime).getTime() <= H24;
  switch (appt.status) {
    case 'PENDING':   return ['CONFIRMED', 'COMPLETED', 'CANCELLED'];
    case 'CONFIRMED': return ['COMPLETED', 'NO_SHOW', 'CANCELLED'];
    // Terminal states — only correctable within 24h
    case 'COMPLETED': return withinCorrection ? ['NO_SHOW', 'CANCELLED'] : [];
    case 'CANCELLED': return withinCorrection ? ['CONFIRMED', 'COMPLETED', 'NO_SHOW'] : [];
    case 'NO_SHOW':   return withinCorrection ? ['COMPLETED', 'CANCELLED'] : [];
    default:          return [];
  }
}

// Actions that require a confirmation dialog before executing
const DESTRUCTIVE_STATUSES = new Set(['CANCELLED', 'COMPLETED', 'NO_SHOW']);

// ─── Appointment block overlay in grid cell ──────────────────────────────────

function AppointmentBlock({
  appt,
  onStatusChange,
}: {
```

```
  appt: Appointment;
  onStatusChange: (id: string, status: string) => void;
}) {
  const transitions = getAdminTransitions(appt);
  const [open, setOpen] = useState(false);
  const [pendingStatus, setPendingStatus] = useState<string | null>(null);

  const handleStatusClick = (s: string) => {
    if (DESTRUCTIVE_STATUSES.has(s)) {
      setPendingStatus(s);
    } else {
      onStatusChange(appt.id, s);
      setOpen(false);
    }
  };

  const confirmStatus = () => {
    if (pendingStatus) {
      onStatusChange(appt.id, pendingStatus);
      setPendingStatus(null);
      setOpen(false);
    }
  };

  return (
    <>
      <button
        onClick={() => setOpen(true)}
        className={[
```

```
            'w-full h-full text-left rounded-[8px] border px-2 py-1.5 text-[11px] leading-tight transition-all hover:brightness-110 cursor-pointer overflow-hidden flex flex-col gap-0.5',
            STATUS_COLOR[appt.status] ?? 'bg-white/5 border-white/10',
          ].join(' ')}
        >
          <p className="font-semibold truncate">{appt.client?.fullName ?? appt.client?.phone ?? '—'}</p>
          <p className="opacity-70 truncate">{appt.services?.map(s => s.service.name).join(', ')}</p>
          <p className="opacity-60 shrink-0">{formatTime(appt.startTime)}–{formatTime(appt.endTime)}</p>
          <div className="flex items-center justify-between mt-auto opacity-60">
          <span className="truncate">✂ {appt.barber?.user?.fullName ?? '—'}</span>
                                    <span  className="shrink-0  ml-1">{formatPrice(appt.totalPrice)}</span>
          </div>
          {(appt.outlet ?? appt.barber?.outlet) && (
           <p className="opacity-50 truncate text-[10px]">
            📍 {(appt.outlet ?? appt.barber?.outlet)?.name}
           </p>
          )}
        </button>

          <Modal  open={open}  onClose={() => { setOpen(false); setPendingStatus(null); }} title="Запис" size="sm">
          {pendingStatus ? (
           <div className="space-y-4">
```

```
                <div className="bg-[#CF4444]/10 border border-[#CF4444]/30 rounded-[10px] p-4 text-sm text-[#EBEBEB]">
                    Ви впевнені, що хочете встановити статус <strong className="text-[#CF4444]">«{STATUS_LABELS[pendingStatus]}»</strong>?
                    <p className="text-xs text-[#888] mt-2">Цю дію можна буде скасувати лише протягом 24 годин.</p>
                </div>
                <div className="flex gap-2">
                    <Button size="sm" onClick={confirmStatus}>Підтвердити</Button>
                    <Button size="sm" variant="ghost" onClick={() => setPendingStatus(null)}>Назад</Button>
                </div>
              </div>
            ) : (
              <div className="space-y-3">
                <div className="flex items-start justify-between gap-3">
                  <div>
                    <p className="text-sm font-semibold">{appt.client?.fullName ?? '—'}</p>
                    <p className="text-xs text-[#666]">{appt.client?.phone}</p>
                  </div>
                  <div className="text-right shrink-0">
                    <p className="text-xs text-[#888]">✂ {appt.barber?.user?.fullName ?? '—'}</p>
                    <p className="text-xs text-[#555]">{appt.barber?.outlet?.name ?? ''}</p>
                  </div>
                </div>
```

```
                    <p className="text-xs text-[#888]">{appt.services?.map(s => s.service.name).join(', ')}</p>
                                    <p className="text-sm font-semibold text-[#C9A84C]">{formatPrice(appt.totalPrice)}</p>
                  <p className="text-xs text-[#555]">{formatTime(appt.startTime)} – {formatTime(appt.endTime)}</p>
                   <p className={['inline-flex px-2 py-0.5 rounded-full text-[11px] border', STATUS_COLOR[appt.status]].join(' ')}>
                    {STATUS_LABELS[appt.status]}
                  </p>
                  {transitions.length > 0 && (
                   <div className="flex flex-wrap gap-2 pt-1">
                     {transitions.map(s => (
                      <Button key={s} size="sm" variant="secondary"
                        onClick={() => handleStatusClick(s)}>
                        → {STATUS_LABELS[s]}
                      </Button>
                     ))}
                   </div>
                  )}
                 </div>
                )}
               </Modal>
              </>
             );
            }

            // ——— God-mode create modal ————————————————————————————————————————————————————
```

```
function AdminCreateModal({
  open,
  onClose,
  barbers,
  services,
  defaultDate,
}: {
  open: boolean;
  onClose: () => void;
  barbers: Barber[];
  services: Service[];
  defaultDate: string;
}) {
  const qc = useQueryClient();
  const [clientSearch, setClientSearch] = useState('');
  const [clientId, setClientId] = useState('');
  const [clientSelected, setClientSelected] = useState(false);
  const [clientMode, setClientMode] = useState<'search' | 'new'>('search');
  const [barberId, setBarberId] = useState('');
  const [outletId, setOutletId] = useState('');
  const [serviceIds, setServiceIds] = useState<string[]>([]);
  const [modalDate, setModalDate] = useState(defaultDate); // editable date
inside modal
  const [startTime, setStartTime] = useState('09:00');
  const [endTime, setEndTime] = useState('10:00');
  const [totalPrice, setTotalPrice] = useState('');
  const [notes, setNotes] = useState('');
  // Quick Register panel state
  const [newClientName, setNewClientName] = useState('');
  const [newClientPhone, setNewClientPhone] = useState('');
```

```
  // Sync modalDate when the modal opens (defaultDate may change between opens)
  useEffect(() => { if (open) setModalDate(defaultDate); }, [open, defaultDate]);

  // ── All outlets (for location picker) ──────────────────────────────────
  const { data: outletsData } = useQuery({
    queryKey: ['outlets-all'],
    queryFn: () => outletsApi.findAll().then(r => r.data),
    staleTime: 300_000,
  });
  const outlets = (outletsData ?? []) as import('@/types/api.types').Outlet[];

  // ── Client search ──────────────────────────────────────────────────────
  const { data: usersData } = useQuery({
    queryKey: ['admin-user-search', clientSearch],
    queryFn: () => usersApi.findAll(1, 10, clientSearch).then(r => r.data),
    enabled: clientSearch.length >= 2 && !clientSelected,
    staleTime: 10_000,
  });

  // ── Barber detail: gives us services with customPrice/customDuration ──
  const { data: selectedBarberData } = useQuery({
    queryKey: ['barber-detail-modal', barberId],
    queryFn: () => barbersApi.findOne(barberId).then(r => r.data),
```

```
    enabled: !!barberId,
    staleTime: 60_000,
  });

  // Auto-fill outletId from barber's home outlet when barber changes
   const barberHomeOutletId = (selectedBarberData as any)?.outletId as string | undefined;
  useEffect(() => {
    if (barberHomeOutletId) setOutletId(barberHomeOutletId);
  }, [barberHomeOutletId]);

  // Lookup map: serviceId → { customPrice, customDuration } for selected barber
  const barberServiceMap = useMemo(() => {
    const map = new Map<string, { customPrice?: number; customDuration?: number }>();
      for (const bs of (selectedBarberData?.services ?? []) as import('@/types/api.types').BarberServiceItem[]) {
      map.set(bs.serviceId, {
        customPrice: bs.customPrice ? Number(bs.customPrice) : undefined,
        customDuration: bs.customDuration ?? undefined,
      });
    }
    return map;
  }, [selectedBarberData]);

  // Services available for selected barber (only his services)
  const availableServices = useMemo(() => {
    if (!barberId || barberServiceMap.size === 0) return services.filter(s => s.isActive);
```

```
    return services.filter(s => s.isActive && barberServiceMap.has(s.id));
  }, [barberId, barberServiceMap, services]);

  // ── Auto-computed duration (sum of all selected services) ──────────────

  const computedDurationMin = useMemo(() =>
    serviceIds.reduce((sum, sid) => {
      const bs = barberServiceMap.get(sid);
      if (bs?.customDuration) return sum + bs.customDuration;
      const svc = services.find(s => s.id === sid);
      return sum + (svc ? svc.baseDuration + svc.bufferTime : 0);
    }, 0),
    [serviceIds, barberServiceMap, services],
  );

  // ── Auto-computed price ──────────────────────────────────────────────────

  const computedPrice = useMemo(() =>
    serviceIds.reduce((sum, sid) => {
      const bs = barberServiceMap.get(sid);
      if (bs?.customPrice) return sum + bs.customPrice;
      const svc = services.find(s => s.id === sid);
      return sum + (svc ? Number(svc.basePrice) : 0);
    }, 0),
    [serviceIds, barberServiceMap, services],
  );

  // Auto-update endTime when startTime or duration changes
  useEffect(() => {
```

```
  if (!computedDurationMin) return;
  const [h, m] = startTime.split(':').map(Number);
  const totalMins = h * 60 + m + computedDurationMin;
  // % 24: clamp hours to 0-23 so we never produce invalid "24:30" values
  const endH = (Math.floor(totalMins / 60) % 24).toString().padStart(2, '0');
  const endM = (totalMins % 60).toString().padStart(2, '0');
  setEndTime(`${endH}:${endM}`);
 }, [startTime, computedDurationMin]);

 // Auto-update price when services/barber change
 useEffect(() => {
  if (computedPrice > 0) setTotalPrice(String(computedPrice));
 }, [computedPrice]);

 // ── Barber schedules: to auto-set startTime and show working hours ──────────

 const { data: barberSchedulesPayload } = useQuery({
  queryKey: ['barber-schedules-modal', barberId],
   queryFn: () => barbersApi.getSchedules(barberId).then(r => r.data as any),
  enabled: !!barberId,
  staleTime: 60_000,
 });
   const barberSchedules: any[] = (barberSchedulesPayload as any)?.schedules ?? [];
 // Exceptions are already in the same payload — no extra request needed
   const barberExceptions: any[] = (barberSchedulesPayload as any)?.exceptions ?? [];

 // Find the schedule entry for the modal date's day of week
```

```
  const scheduleForDate = useMemo(() => {
    if (!barberSchedules || !modalDate) return null;
    const dow = new Date(modalDate + 'T00:00:00').getDay();
    return barberSchedules.find((s: any) => s.dayOfWeek === dow) ?? null;
  }, [barberSchedules, modalDate]);

  // Find exception for the selected date — overrides regular schedule
  const exceptionForDate = useMemo(() => {
    if (!modalDate) return null;
    return barberExceptions.find((e: any) => (e.date as string).split('T')[0] === modalDate) ?? null;
  }, [barberExceptions, modalDate]);

  // Whether the appointment crosses midnight (endTime < startTime on same calendar day)
  const crossesMidnight = endTime < startTime;

  // Auto-set startTime to barber's shift start when barber or date changes
  useEffect(() => {
    if (scheduleForDate?.startTime) setStartTime(scheduleForDate.startTime);
  }, [scheduleForDate]);

  // ── Handlers ──────────────────────────────────────────────────────────

  const handleBarberChange = (id: string) => {
    setBarberId(id);
    setServiceIds([]);
    setTotalPrice('');
```

```
    setOutletId(''); // reset outlet — will be refilled by useEffect above
  };

  const toggleService = (id: string) =>
    setServiceIds(prev => prev.includes(id) ? prev.filter(s => s !== id) : [...prev, id]);

  const adminCreate = useMutation({
    mutationFn: () => {
      const startDt = new Date(`${modalDate}T${startTime}:00`);
      const endDt   = new Date(`${modalDate}T${endTime}:00`);
      if (endDt <= startDt) endDt.setDate(endDt.getDate() + 1);
      return appointmentsApi.adminCreate({
        barberId,
        clientId,
        serviceIds,
        startTime: startDt.toISOString(),
        endTime:   endDt.toISOString(),
        totalPrice: +totalPrice,
        notes: notes || undefined,
      });
    },
    onSuccess: () => {
      qc.invalidateQueries({ queryKey: ['admin-appointments'] });
      onClose();
    },
    onError: (e: any) => alert(extractErrorMessage(e, 'Помилка створення запису')),
  });
```

```
        // Quick Register: create Shadow User and auto-select as client
        const createClientMutation = useMutation({
            mutationFn: () => usersApi.adminCreateClient({ fullName: newClientName, phone: newClientPhone }),
          onSuccess: (res) => {
            const u = res.data;
            setClientId(u.id);
            setClientSearch(`${u.fullName} (${u.phone})`);
            setClientSelected(true);
            setClientMode('search');
            setNewClientName('');
            setNewClientPhone('');
            qc.invalidateQueries({ queryKey: ['admin-user-search'] });
          },
          onError: (e: any) => alert(extractErrorMessage(e, 'Помилка створення клієнта')),
        });

        const inputCls = 'w-full bg-[#0A0A0A] border border-[#1E1E1E] rounded-[10px] px-4 py-3 text-sm text-[#EBEBEB] focus:outline-none focus:border-[#C9A84C]/50';

        return (
          <Modal open={open} onClose={onClose} title="Новий запис (God Mode)" size="xl" preventOutsideClose>
            <div className="space-y-4">

              {/* ── Client mode tabs ──────────────────────────────────────── */}
              <div>
```

```
<div className="flex gap-0 mb-3 bg-[#0A0A0A] border border-[#1E1E1E] rounded-[10px] p-1">
<button
onClick={() => { setClientMode('search'); setClientId(''); setClientSelected(false); setClientSearch(''); }}
className={['flex-1 py-1.5 rounded-[7px] text-xs font-medium transition-all cursor-pointer', clientMode === 'search' ? 'bg-[#C9A84C] text-[#080808]' : 'text-[#666] hover:text-[#EBEBEB]'].join(' ')}
>
🔍 Знайти існуючого
</button>
<button
onClick={() => { setClientMode('new'); setClientId(''); setClientSelected(false); setClientSearch(''); }}
className={['flex-1 py-1.5 rounded-[7px] text-xs font-medium transition-all cursor-pointer', clientMode === 'new' ? 'bg-[#C9A84C] text-[#080808]' : 'text-[#666] hover:text-[#EBEBEB]'].join(' ')}
>
👤 Новий клієнт
</button>
</div>

<div className="min-h-[120px]">
{/* Search mode */}
{clientMode === 'search' && (
<>
<Input
label="Пошук (ім'я або телефон)"
value={clientSearch}
```

```
                    onChange={e => { setClientSearch(e.target.value); setClientId('');
setClientSelected(false); }}
                     placeholder="Введіть мінімум 2 символи..."
                    />
                    {usersData?.data && clientSearch.length >= 2 && !clientSelected
&& (
                     <div className="mt-1 border border-[#1E1E1E] rounded-[10px]
bg-[#0A0A0A] overflow-hidden">
                       {usersData.data.filter(u => u.role === 'CLIENT').map(u => (
                        <button
                         key={u.id}
                                                onClick={() => { setClientId(u.id);
setClientSearch(`${u.fullName} (${u.phone})`); setClientSelected(true); }}
                           className={['w-full text-left px-4 py-2 text-sm transition-
colors cursor-pointer', clientId === u.id ? 'bg-[#C9A84C]/10 text-[#C9A84C]' :
'hover:bg-white/5'].join(' ')}
                         >
                          {u.fullName} · {u.phone}
                           {u.isBlocked && <span className="ml-2 text-[#CF4444]
text-xs">[Блок]</span>}
                            {u.requiresPrepayment && <span className="ml-2 text-
[#E09B3D] text-xs">[Передоплата]</span>}
                        </button>
                       ))}
                       {usersData.data.filter(u => u.role === 'CLIENT').length === 0
&& (
                        <p className="px-4 py-2 text-xs text-[#555]">Не знайдено
— спробуйте вкладку «Новий клієнт»</p>
                       )}
                     </div>
```

```
                )}
              </>
            )}

            {/* New client mode */}
            {clientMode === 'new' && (
              <div className="border border-[#C9A84C]/30 rounded-[10px] bg-[#0A0A0A] p-2 space-y-2">
                <div className="grid grid-cols-2 gap-2">
                  <Input label="ПІБ" type="text" value={newClientName} onChange={e => setNewClientName(e.target.value)} placeholder="Іван Іваненко" />
                  <Input label="Номер телефону" type="tel" value={newClientPhone} onChange={e => setNewClientPhone(e.target.value)} placeholder="+380671234567" />
                </div>
                {createClientMutation.isError && <p className="text-[10px] text-[#CF4444] leading-none px-1">Помилка: перевірте формат</p>}
                <Button size="sm" fullWidth loading={createClientMutation.isPending}
                  onClick={() => createClientMutation.mutate()}
                  disabled={newClientName.length < 2 || !newClientPhone.startsWith('+380')}>
                  {clientSelected ? '✓ Клієнта створено та вибрано' : 'Створити (без пароля) та вибрати'}
                </Button>
              </div>
            )}
          </div>
        </div>
```

```
                                {/* ── Date (editable) ──────────────────────────────────────────────────────────── */}
            <div>
                <label className="block text-xs text-[#888] mb-1.5">Дата запису</label>
              <input
                type="date"
                value={modalDate}
                onChange={e => setModalDate(e.target.value)}
                className={inputCls + ' [color-scheme:dark] cursor-pointer'}
              />
            </div>

                                {/* ── Barber ──────────────────────────────────────────────────────────────────── */}
            <div>
                <label className="block text-xs text-[#888] mb-1.5">Майстер</label>
                <select value={barberId} onChange={e => handleBarberChange(e.target.value)} className={inputCls}>
                  <option value="">Оберіть майстра...</option>
                  {barbers.map(b => (
                    <option key={b.id} value={b.id}>{b.user?.fullName ?? b.id}</option>
                  ))}
                </select>
                {barberId && barberSchedulesPayload && (
                  exceptionForDate ? (
```

```
                exceptionForDate.isDayOff ? (
                  // Day off / sick / vacation — strong warning
                  <p className="mt-1.5 text-xs text-[#CF4444]">
                    🚫 Вихідний день ({exceptionForDate.type ?? 'DayOff'}) на {modalDate}. Ви все одно можете створити запис вручну.
                  </p>
                ) : (
                  // Custom hours exception — amber info
                  <p className="mt-1.5 text-xs text-[#E09B3D]">
                    ⏱ Особливий час {exceptionForDate.startTime}–{exceptionForDate.endTime} на {modalDate}.
                  </p>
                )
              ) : scheduleForDate ? (
                <p className="mt-1.5 text-xs text-[#4CAF6A]">✓ Працює {scheduleForDate.startTime} — {scheduleForDate.endTime} ({modalDate})</p>
              ) : (
                <p className="mt-1.5 text-xs text-[#E09B3D]">⚠ Цей майстер не працює {modalDate}. Ви все одно можете створити запис вручну.</p>
              )
            )}
          </div>

          {/* ── Outlet (location) ──────────────────────────────────────── */}
          <div>
            <label className="block text-xs text-[#888] mb-1.5">
              Філія / Локація
```

```
                {barberId && <span className="ml-2 text-xs text-[#666]">— автоматично з профілю майстра</span>}
              </label>
              <select
                value={outletId}
                disabled={true}
                className={inputCls + ' opacity-50 cursor-not-allowed'}
              >
                <option value="">{barberId ? 'Оберіть майстра вище...' : 'Оберіть майстра...'}</option>
                {outlets.map(o => (
                  <option key={o.id} value={o.id}>{o.name} — {o.address}</option>
                ))}
              </select>

            </div>

                  {/* ── Services (filtered by barber) ──────────────────────────────────── */}
            <div>
              <label className="block text-xs text-[#888] mb-1.5">
                Послуги
                {barberId && availableServices.length > 0 && (
                  <span className="ml-2 text-xs text-[#666]">— тільки послуги обраного майстра</span>
                )}
              </label>
              {!barberId ? (
```

```
                    <p className="text-xs text-[#555] py-2">Спочатку оберіть майстра</p>
                ) : availableServices.length === 0 ? (
                    <p className="text-xs text-[#555] py-2">Цей майстер не має призначених послуг</p>
                ) : (
                  <div className="flex flex-wrap gap-2">
                    {availableServices.map(s => (
                      <button
                        key={s.id}
                        onClick={() => toggleService(s.id)}
                        className={['px-3 py-1.5 rounded-full text-xs border transition-colors cursor-pointer', serviceIds.includes(s.id) ? 'border-[#C9A84C] bg-[#C9A84C]/10 text-[#C9A84C]' : 'border-[#1E1E1E] text-[#666] hover:border-[#2E2E2E]'].join(' ')}
                      >
                        {s.name}
                        {barberServiceMap.get(s.id)?.customPrice
                          ? ` · ${barberServiceMap.get(s.id)!.customPrice} ₴`
                          : ` · ${s.basePrice} ₴`}
                      </button>
                    ))}
                  </div>
                )}
              </div>

                                                {/* ── Time ──────────────────────────────────────────────────────────────────────────────────────────────────────────── */}
              <div className="grid grid-cols-2 gap-3">
```

```
            <div>
                <label className="block text-xs text-[#888] mb-1.5">Час початку</label>
                <input type="time" value={startTime} onChange={e => setStartTime(e.target.value)} className={inputCls} />
            </div>
            <div>
              <label className="block text-xs text-[#888] mb-1.5">
                Час закінчення
                {computedDurationMin > 0 && <span className="ml-1 text-[#555]">(авто: {computedDurationMin} хв)</span>}
              </label>
                <input type="time" value={endTime} onChange={e => setEndTime(e.target.value)} className={inputCls} />
            </div>
          </div>

          {/* Midnight-crossing warning */}
          {crossesMidnight && (
            <p className="text-xs text-[#E09B3D] bg-[#E09B3D]/10 border border-[#E09B3D]/20 rounded-[8px] px-3 py-2">
              🌙 Запис перетинає північ — закінчення буде зараховано на наступний день ({modalDate}).
            </p>
          )}

          {/* ── Price + notes ──────────────────────────────────────────────── */}
          <div className="grid grid-cols-2 gap-3">
```

```
                <Input
                  label={computedPrice > 0 ? `Ціна (грн) — авто: ${computedPrice} ₴` : 'Ціна (грн)'}
                  type="number"
                  value={totalPrice}
                  onChange={e => setTotalPrice(e.target.value)}
                />
                      <Input label="Примітка" value={notes} onChange={e => setNotes(e.target.value)} placeholder="Необов'язково" />
              </div>

              <div className="flex gap-2 pt-2">
                <Button
                  loading={adminCreate.isPending}
                  onClick={() => adminCreate.mutate()}
                    disabled={!clientId || !barberId || !serviceIds.length || !totalPrice || !modalDate}
                >
                  Створити запис
                </Button>
                <Button variant="ghost" onClick={onClose}>Скасувати</Button>
              </div>
            </div>
          </Modal>
        );
      }
```

```
//  ——  Main  Page
_____

_____

export default function AdminAppointmentsPage() {
  const today = toDateString(new Date());
  // DST-safe: use calendar arithmetic, not fixed millisecond offsets
  const _yday = new Date(); _yday.setDate(_yday.getDate() - 1);
  const yesterday = toDateString(_yday);
  // Default archive range: last 7 days (yesterday-6 .. yesterday)
  const _astart = new Date(); _astart.setDate(_astart.getDate() - 7);
  const defaultArchiveStart = toDateString(_astart);

  const [mainTab, setMainTab] = useState<'journal' | 'archive'>('journal');
  const [date, setDate] = useState(today);
  const [showCreateModal, setShowCreateModal] = useState(false);
  const [viewMode, setViewMode] = useState<'grid' | 'list'>('grid');
    // Journal: ACTIVE = PENDING+CONFIRMED+COMPLETED composite filter
  const [statusFilter, setStatusFilter] = useState('ACTIVE');
  const [listPage, setListPage] = useState(1);
  // Archive date range — defaults to last 7 days
  const [archiveStart, setArchiveStart] = useState(defaultArchiveStart);
  const [archiveEnd, setArchiveEnd] = useState(yesterday);
  const [archivePage, setArchivePage] = useState(1);
  const [archiveStatus, setArchiveStatus] = useState('ALL');
  const qc = useQueryClient();

  /** Maps UI filter key → API status string */
  const mapFilterToApi = (f: string): string | undefined => {
    if (f === 'ACTIVE') return 'PENDING,CONFIRMED,COMPLETED';
```

```
    if (f === 'ALL') return undefined;
    return f || undefined;
  };

  // Grid view — all appointments for selected day, filtered by active statusFilter
  const { data: dayData, isLoading: dayLoading } = useQuery({
    queryKey: ['admin-appointments', 'day', date, statusFilter],
    queryFn: () => appointmentsApi.findAll(1, 500, mapFilterToApi(statusFilter), date).then(r => r.data.data),
    enabled: mainTab === 'journal' && viewMode === 'grid',
    staleTime: 30_000,
    refetchInterval: 15_000,
  });

  // Journal list view — always bounded to today+future (fromDate=today when no specific date)
  const { data: listData, isLoading: listLoading } = useQuery({
    queryKey: ['admin-appointments', 'list', statusFilter, listPage, date],
    queryFn: () => appointmentsApi.findAll(
      listPage, 20,
      mapFilterToApi(statusFilter),
      date || undefined,
      undefined,
      undefined,
      !date ? today : undefined,
    ).then(r => r.data),
    enabled: mainTab === 'journal' && viewMode === 'list',
    staleTime: 30_000,
    refetchInterval: 15_000,
```

```
  });

  // Archive view — past records only (endDate capped at yesterday)
  const safeArchiveEnd = archiveEnd && archiveEnd >= today ? yesterday
: archiveEnd;
  const safeArchiveStart = archiveStart && archiveStart >= today ? yesterday
: archiveStart;
  const { data: archiveData, isLoading: archiveLoading } = useQuery({
    queryKey: ['admin-appointments', 'archive', archiveStatus, archivePage,
safeArchiveStart, safeArchiveEnd],
    queryFn: () => appointmentsApi.findAll(
      archivePage, 20,
      mapFilterToApi(archiveStatus),
      undefined,
      safeArchiveStart || undefined,
      safeArchiveEnd || undefined,
    ).then(r => r.data),
      enabled: mainTab === 'archive' && !!safeArchiveStart &&
!!safeArchiveEnd,
    staleTime: 60_000,
  });

  // Barbers and services for the create modal
  const { data: barbersData } = useQuery({
    queryKey: ['barbers-all'],
    queryFn: () => barbersApi.findAll(1, 100).then(r => r.data as any),
    staleTime: 60_000,
    refetchInterval: 60_000,
  });
  const { data: servicesData } = useQuery({
```

```
  queryKey: ['services-all-admin'],
  queryFn: () => servicesApi.findAll(true).then(r => r.data as Service[]),
  staleTime: 60_000,
  refetchInterval: 60_000,
});

// Schedules + exceptions — for today's status badges in the grid header
const { data: schedulesData } = useQuery({
  queryKey: ['admin-schedules'],
  queryFn: () => schedulesApi.getAllSchedules().then(r => r.data),
  staleTime: 60_000,
  refetchInterval: 60_000,
});

  // ── Work status per barber for the selected date ──────────────────────────

const selectedDateStr = date || toDateString(new Date());
const selectedDow     = new Date(selectedDateStr + 'T00:00:00').getDay();

type TodayStatus = { label: string; sub?: string; color: 'green' | 'amber' | 'red' | 'gray' };
const APPT_STATUS_COLORS: Record<string, string> = {
  green: 'bg-[#4CAF6A]/10 text-[#4CAF6A] border-[#4CAF6A]/30',
  amber: 'bg-[#E09B3D]/10 text-[#E09B3D] border-[#E09B3D]/30',
  red:   'bg-[#CF4444]/10 text-[#CF4444] border-[#CF4444]/30',
  gray:  'bg-[#1A1A1A] text-[#555] border-[#2A2A2A]',
};

const scheduleMap = useMemo(() => {
```

```
    const map: Record<string, Record<number, { startTime: string; endTime: string }>> = {};
    for (const s of schedulesData?.schedules ?? []) {
      if (!map[s.barberId]) map[s.barberId] = {};
      map[s.barberId][s.dayOfWeek] = { startTime: s.startTime, endTime: s.endTime };
    }
    return map;
  }, [schedulesData]);


  const changeStatus = useMutation({
    mutationFn: ({ id, s }: { id: string; s: string }) => appointmentsApi.updateStatus(id, s),
    onSuccess: () => {
      qc.invalidateQueries({ queryKey: ['admin-appointments'] });
    },
    onError: (e: any) => alert(extractErrorMessage(e, 'Помилка зміни статусу')),
  });

  // Group appointments by barberId for grid columns
  const barbers: Barber[] = useMemo(() =>
    (barbersData?.data ?? []) as Barber[], [barbersData]);

  // ── Today's status map (depends on barbers, must be after it) ──────────

  const todayStatusMap = useMemo((): Record<string, TodayStatus> => {
    const result: Record<string, TodayStatus> = {};
    for (const b of barbers) {
```

```
        const exc = (schedulesData?.exceptions ?? []).find(
          // Slice ISO string instead of re-parsing via Date — avoids UTC→local timezone shift
          e => e.barberId === b.id && (e.date as string).split('T')[0] === selectedDateStr,
        );
        if (exc) {
          result[b.id] = exc.isDayOff
            ? { label: 'Вихідний', color: 'red' }
            : { label: 'Особл. час', sub: `${exc.startTime}–${exc.endTime}`, color: 'amber' };
        } else {
          const shift = scheduleMap[b.id]?.[selectedDow];
          result[b.id] = shift
            ? { label: 'Працює', sub: `${shift.startTime}–${shift.endTime}`, color: 'green' }
            : { label: 'Не працює', color: 'gray' };
        }
      }
      return result;
    }, [barbers, schedulesData, scheduleMap, selectedDateStr, selectedDow]);

    const apptsByBarber = useMemo(() => {
      const map: Record<string, Appointment[]> = {};
      for (const b of barbers) map[b.id] = [];
      for (const a of (dayData ?? []) as Appointment[]) {
        if (map[a.barberId]) map[a.barberId].push(a);
      }
      return map;
    }, [dayData, barbers]);
```

```
const goDay = (offset: number) => {
  const base = date || today;
  const d = new Date(base + 'T00:00:00');
  d.setDate(d.getDate() + offset);
  setDate(toDateString(d));
};

const switchToGrid = () => {
  if (!date) setDate(today);
  setViewMode('grid');
};

const switchToAllDates = () => {
  setDate('');
  setViewMode('list');
};

 const STATUS_FILTERS_JOURNAL = ['ACTIVE', 'PENDING', 'CONFIRMED', 'COMPLETED', 'NO_SHOW', 'CANCELLED', 'ALL'];
 const STATUS_FILTERS_ARCHIVE = ['ALL', 'COMPLETED', 'NO_SHOW', 'CANCELLED'];
const SLABEL: Record<string, string> = {
  '': 'Всі', ALL: 'Всі', ACTIVE: 'Активні',
   PENDING: 'Очікує', CONFIRMED: 'Підтверджено', COMPLETED: 'Завершено',
  CANCELLED: 'Скасовано', NO_SHOW: 'Неявка',
};
```

```
/** Reusable appointment list row with inline confirm for destructive actions */
const AppointmentListRow = ({ a, showDate = false }: { a: Appointment; showDate?: boolean }) => {
 const transitions = getAdminTransitions(a);
 const [pendingS, setPendingS] = useState<string | null>(null);
 return (
  <div className="flex items-start justify-between bg-[#111] border border-[#1E1E1E] rounded-[12px] px-5 py-4 gap-4">
   <div className="flex-1 min-w-0">
    <div className="flex items-center gap-2 mb-1 flex-wrap">
     <span className={['text-[11px] px-2 py-0.5 rounded-full border', STATUS_COLOR[a.status]].join(' ')}>
      {STATUS_LABELS[a.status]}
     </span>
     <p className="text-sm font-medium">{formatTime(a.startTime)} – {formatTime(a.endTime)}</p>
     {showDate && (
      <span className="text-xs text-[#555]">
       {new Date(a.startTime).toLocaleDateString('uk-UA', { day: 'numeric', month: 'short' })}
      </span>
     )}
    </div>
    <p className="text-sm">{a.client?.fullName ?? a.client?.phone ?? '—'}</p>
    {a.client?.phone && (
     <a href={`tel:${a.client.phone}`}
      className="text-xs text-[#C9A84C]/70 hover:text-[#C9A84C] transition-colors mt-0.5 block font-mono">
```

```
                {a.client.phone}
              </a>
            )}
              <p className="text-xs text-[#555] mt-0.5">{a.services?.map(s => s.service.name).join(', ')}</p>
            {/* Barber name + outlet */}
            <p className="text-xs text-[#555] mt-1">
              ✂ {a.barber?.user?.fullName ?? '—'}
                {a.barber?.outlet?.name && <span className="text-[#888]"> · {a.barber.outlet.name}</span>}
              </p>
            </div>
            <div className="flex flex-col items-end gap-2 shrink-0">
                          <p  className="text-sm  font-semibold  text-[#C9A84C]">{formatPrice(a.totalPrice)}</p>
              {pendingS ? (
               <div className="flex flex-col items-end gap-1.5">
                          <p  className="text-xs  text-[#CF4444]">Встановити «{STATUS_LABELS[pendingS]}»?</p>
                    <p  className="text-[10px]  text-[#555]">Скасувати можна протягом 24 год.</p>
                 <div className="flex gap-1">
                   <Button size="sm" onClick={() => { changeStatus.mutate({ id: a.id, s: pendingS }); setPendingS(null); }}>Так</Button>
                          <Button  size="sm"  variant="ghost"  onClick={()  =>  setPendingS(null)}>Hi</Button>
                 </div>
               </div>
            ) : transitions.length > 0 ? (
               <div className="flex gap-1 flex-wrap justify-end">
```

```
                {transitions.map(s => (
                  <Button key={s} size="sm" variant="secondary"
                          onClick={() => DESTRUCTIVE_STATUSES.has(s) ?
setPendingS(s) : changeStatus.mutate({ id: a.id, s })}>
                    {STATUS_LABELS[s]}
                  </Button>
                ))}
              </div>
            ) : null}
          </div>
        </div>
      );
    };

    return (
      <div className="animate-fade-up space-y-6">
        {/* Header */}
        <div className="flex items-center justify-between">
          <div>
            <p className="text-xs uppercase tracking-widest text-[#C9A84C]
mb-1">Адміністратор</p>
            <h1 className="font-display text-3xl font-bold">Записи</h1>
          </div>
          {/* Button only shown in Journal tab */}
          {mainTab === 'journal' && (
            <Button onClick={() => setShowCreateModal(true)}>+ Новий
запис</Button>
          )}
        </div>
```

```
                {/* ── Main Tabs: Журнал | Архів ──────────────────────────────────────────── */}
                <div className="flex gap-1 bg-[#111] border border-[#1E1E1E] rounded-[14px] p-1 w-fit">
                <button onClick={() => setMainTab('journal')}
                    className={['px-5 py-2 rounded-[10px] text-sm font-medium transition-all cursor-pointer', mainTab === 'journal' ? 'bg-[#C9A84C] text-[#080808]' : 'text-[#666] hover:text-[#EBEBEB']'].join(' ')}>
                  📋 Журнал
                </button>
                <button onClick={() => setMainTab('archive')}
                    className={['px-5 py-2 rounded-[10px] text-sm font-medium transition-all cursor-pointer', mainTab === 'archive' ? 'bg-[#C9A84C] text-[#080808]' : 'text-[#666] hover:text-[#EBEBEB']'].join(' ')}>
                  🗄 Архів
                </button>
              </div>

              {/* ═══════════ ЖУРНАЛ TAB ═══════════ */}
              {mainTab === 'journal' && (
                <>
                  {/* View toggle and date */}
                  <div className="flex flex-wrap items-center gap-3 justify-between bg-[#111] border border-[#1E1E1E] rounded-[16px] p-2">
                    <div className="flex items-center gap-2 flex-wrap">
                      <div className="flex gap-1 bg-[#0A0A0A] border border-[#1E1E1E] rounded-[10px] p-1">
                        <button onClick={switchToGrid}
```

```
                    className={['px-4 py-1.5 rounded-[7px] text-sm transition-all cursor-pointer', viewMode === 'grid' ? 'bg-[#C9A84C] text-[#080808] font-semibold' : 'text-[#666] hover:text-[#EBEBEB]'].join(' ')}>
                    ⊞ Сітка
                  </button>
                        <button onClick={() => { if (!date) setDate(today); setViewMode('list'); }}
                    className={['px-4 py-1.5 rounded-[7px] text-sm transition-all cursor-pointer', viewMode === 'list' && !!date ? 'bg-[#C9A84C] text-[#080808] font-semibold' : 'text-[#666] hover:text-[#EBEBEB]'].join(' ')}>
                    ≡ Список
                  </button>
                  <button onClick={switchToAllDates}
                    className={['px-4 py-1.5 rounded-[7px] text-sm transition-all cursor-pointer whitespace-nowrap', viewMode === 'list' && !date ? 'bg-[#C9A84C] text-[#080808] font-semibold' : 'text-[#666] hover:text-[#EBEBEB]'].join(' ')}>
                    ∞ Всі дати
                  </button>
                </div>

                {(viewMode === 'grid' || !!date) && (
                  <div className="flex items-center gap-1 bg-[#0A0A0A] border border-[#1E1E1E] rounded-[10px] p-1 pl-2">
                    <button onClick={() => setDate(today)} className="text-sm font-medium text-[#C9A84C] hover:text-[#E09B3D] cursor-pointer mr-1">Сьогодні</button>
                    <Button
                      variant="ghost" size="sm"
                      onClick={() => goDay(-1)}
                      disabled={date === today}
```

```
                    className="h-7 w-7 p-0 flex items-center justify-center rounded-[6px] disabled:opacity-30 disabled:cursor-not-allowed">
                    ←
                  </Button>
                  <input type="date" value={date} min={today}
                    onChange={e => { const v = e.target.value; if (!v) { switchToAllDates(); } else { setDate(v); } }}
                    className="bg-transparent text-sm text-[#EBEBEB] font-medium outline-none cursor-pointer [color-scheme:dark]" />
                  <Button variant="ghost" size="sm" onClick={() => goDay(1)} className="h-7 w-7 p-0 flex items-center justify-center rounded-[6px]">→</Button>
                </div>
              )}
            </div>

            <div className="flex gap-1.5 flex-wrap">
              {STATUS_FILTERS_JOURNAL.map(s => (
                <button key={s} onClick={() => { setStatusFilter(s); setListPage(1); }}
                  className={['px-3 py-1.5 rounded-[8px] text-xs font-medium transition-colors cursor-pointer', statusFilter === s ? 'bg-[#C9A84C]/20 text-[#C9A84C]' : 'bg-[#1E1E1E]/50 text-[#888] hover:bg-[#1E1E1E] hover:text-[#EBEBEB]'].join(' ')}>
                  {SLABEL[s]}
                </button>
              ))}
            </div>
          </div>
```

```
                {/* Grid view — absolute-positioned appointments */}
                {viewMode === 'grid' && (
                 dayLoading ? (
                    <div className="flex justify-center py-20"><Spinner size="lg" /></div>
                 ) : (
                   <div className="overflow-x-auto rounded-[16px] border border-[#1E1E1E] bg-[#0A0A0A]">
                    <div style={{ minWidth: `${Math.max(barbers.length * 180 + 72, 600)}px` }}>

                     {/* Sticky header */}
                       <div className="grid sticky top-0 z-10 bg-[#111] border-b border-[#1E1E1E]"
                     style={{ gridTemplateColumns: `72px repeat(${barbers.length}, 1fr)` }}>
                      <div className="border-r border-[#1E1E1E]" />
                      {barbers.map(b => {
                       const st = todayStatusMap[b.id];
                       return (
                         <div key={b.id} className="px-3 py-3 text-center border-r border-[#1E1E1E] last:border-0">
                          <p className="text-sm font-semibold">{b.user?.fullName ?? 'Майстер'}</p>
                          {(b as any).outlet?.name && (
                            <p className="text-[11px] text-[#888] mt-0.5">📍 {(b as any).outlet.name}</p>
                          )}
                          {st && (
```

```
                        <span className={`inline-flex items-center gap-1 mt-1 text-[10px] px-1.5 py-0.5 rounded-full border ${APPT_STATUS_COLORS[st.color]}`}>
                          {st.label}{st.sub && <span className="opacity-80">{st.sub}</span>}
                        </span>
                      )}
                    </div>
                  );
                })}
              </div>

              {/* Grid body */}
              <div className="relative" style={{ height: `${GRID_HEIGHT}px` }}>

                {/* Background: time labels + horizontal lines */}
                {TIME_SLOTS.map((slot, si) => (
                  <div key={slot}
                    className="absolute w-full flex pointer-events-none select-none"
                    style={{ top: `${si * SLOT_HEIGHT}px`, height: `${SLOT_HEIGHT}px` }}>
                    <div className={['w-[72px] shrink-0 border-r border-[#1E1E1E] flex items-start pt-1.5 px-3', si % 2 !== 0 ? 'bg-white/[0.01]' : ''].join(' ')}>
                      <span className="text-[11px] text-[#444]">{slot}</span>
                    </div>
                    <div className={['flex-1 border-b border-[#0F0F0F]', si % 2 !== 0 ? 'bg-white/[0.01]' : ''].join(' ')} />
```

```
                </div>
              ))}

              {/* Barber columns */}
              <div className="absolute top-0 right-0 bottom-0"
                   style={{ left: '72px', display: 'grid', gridTemplateColumns:
`repeat(${barbers.length}, 1fr)` }}>
                {barbers.map(b => {
                  const positioned = layoutAppointments(apptsByBarber[b.id]
?? []);
                  return (
                    <div key={b.id} className="relative border-r border-
[#1E1E1E] last:border-0">
                      {positioned.map(({ appt, top, height, leftPercent,
widthPercent }) => (
                        <div key={appt.id}
                          className="absolute p-0.5 box-border"
                          style={{ top: `${top}px`, height: `${height}px`, left:
`${leftPercent}%`, width: `${widthPercent}%` }}>
                          <AppointmentBlock appt={appt}
                            onStatusChange={(id, s) => changeStatus.mutate({ id,
s })} />
                        </div>
                      ))}
                    </div>
                  );
                })}
              </div>
            </div>
```

```
                </div>
              </div>
            )
          )}

          {/* List view */}
          {viewMode === 'list' && (
            listLoading ? (
              <div className="flex justify-center py-20"><Spinner size="lg" /></div>
            ) : (
              <div className="space-y-2">
                {!date && (
                  <div className="flex items-center gap-2 text-xs text-[#555] px-1 pb-1 border-b border-[#1E1E1E]">
                    <span className="text-[#C9A84C] text-base">∞</span>
                    Режим «Всі дати» — показано поточні та майбутні записи
                  </div>
                )}
                {((listData as any)?.data ?? []).map((a: Appointment) => (
                  <AppointmentListRow key={a.id} a={a} showDate={!date} />
                ))}
                {!(listData as any)?.data?.length && (
                  <p className="text-[#444] text-center py-10">Записів немає</p>
                )}
              </div>
            )
          )}
          {viewMode === 'list' && (
```

```
                <Pagination page={listPage} total={(listData as any)?.total ?? 0} limit={20} onPageChange={setListPage} />
              )}
            </>
          )}

          {/* ═══════════ АРХІВ TAB ═══════════ */}
          {mainTab === 'archive' && (
            <>
              <div className="bg-[#111] border border-[#1E1E1E] rounded-[16px] p-4 space-y-3">
                <p className="text-xs uppercase tracking-widest text-[#555]">Фільтр по діапазону дат</p>
                <div className="flex flex-wrap items-center gap-3">
                  <div className="flex items-center gap-2">
                    <label className="text-xs text-[#888]">З</label>
                    <input type="date" value={archiveStart} max={yesterday}
                      onChange={e => { setArchiveStart(e.target.value); setArchivePage(1); }}
                      className="bg-[#0A0A0A] border border-[#1E1E1E] rounded-[8px] px-3 py-1.5 text-sm text-[#EBEBEB] outline-none [color-scheme:dark] cursor-pointer" />
                  </div>
                  <div className="flex items-center gap-2">
                    <label className="text-xs text-[#888]">По</label>
                    <input type="date" value={archiveEnd} max={yesterday}
                      onChange={e => { setArchiveEnd(e.target.value); setArchivePage(1); }}
                      className="bg-[#0A0A0A] border border-[#1E1E1E] rounded-[8px] px-3 py-1.5 text-sm text-[#EBEBEB] outline-none [color-scheme:dark] cursor-pointer" />
                  </div>
```

```
                    <button onClick={() => { setArchiveStart(defaultArchiveStart); setArchiveEnd(yesterday); setArchivePage(1); }}
                      className="text-xs text-[#555] hover:text-[#EBEBEB] cursor-pointer transition-colors">
                      Скинути
                    </button>
                  </div>
                  <div className="flex gap-1.5 flex-wrap">
                    {STATUS_FILTERS_ARCHIVE.map(s => (
                      <button key={s} onClick={() => { setArchiveStatus(s); setArchivePage(1); }}
                        className={['px-3 py-1.5 rounded-[8px] text-xs font-medium transition-colors cursor-pointer', archiveStatus === s ? 'bg-[#C9A84C]/20 text-[#C9A84C]' : 'bg-[#1E1E1E]/50 text-[#888] hover:bg-[#1E1E1E] hover:text-[#EBEBEB]'].join(' ')}>
                        {SLABEL[s]}
                      </button>
                    ))}
                  </div>
                </div>

                {(!archiveStart || !archiveEnd) ? (
                  <p className="text-[#444] text-center py-10">
                    Оберіть початкову та кінцеву дати для пошуку в архіві
                  </p>
                ) : archiveLoading ? (
                  <div className="flex justify-center py-20"><Spinner size="lg" /></div>
                ) : (
                  <div className="space-y-2">
```

```
                {((archiveData as any)?.data ?? []).map((a: Appointment) => (
                  <AppointmentListRow key={a.id} a={a} showDate />
                ))}
                {!(archiveData as any)?.data?.length && (
                  <p className="text-[#444] text-center py-10">
                    За обраний період записів немає
                  </p>
                )}
              </div>
            )}
              <Pagination page={archivePage} total={(archiveData as any)?.total ?? 0} limit={20} onPageChange={setArchivePage} />
            </>
          )}

          {/* Create modal */}
          <AdminCreateModal
            open={showCreateModal}
            onClose={() => setShowCreateModal(false)}
            barbers={barbers}
            services={servicesData ?? []}
            defaultDate={date}
          />
        </div>
      );
    }
```

ДОДАТОК В

Слайди презентації

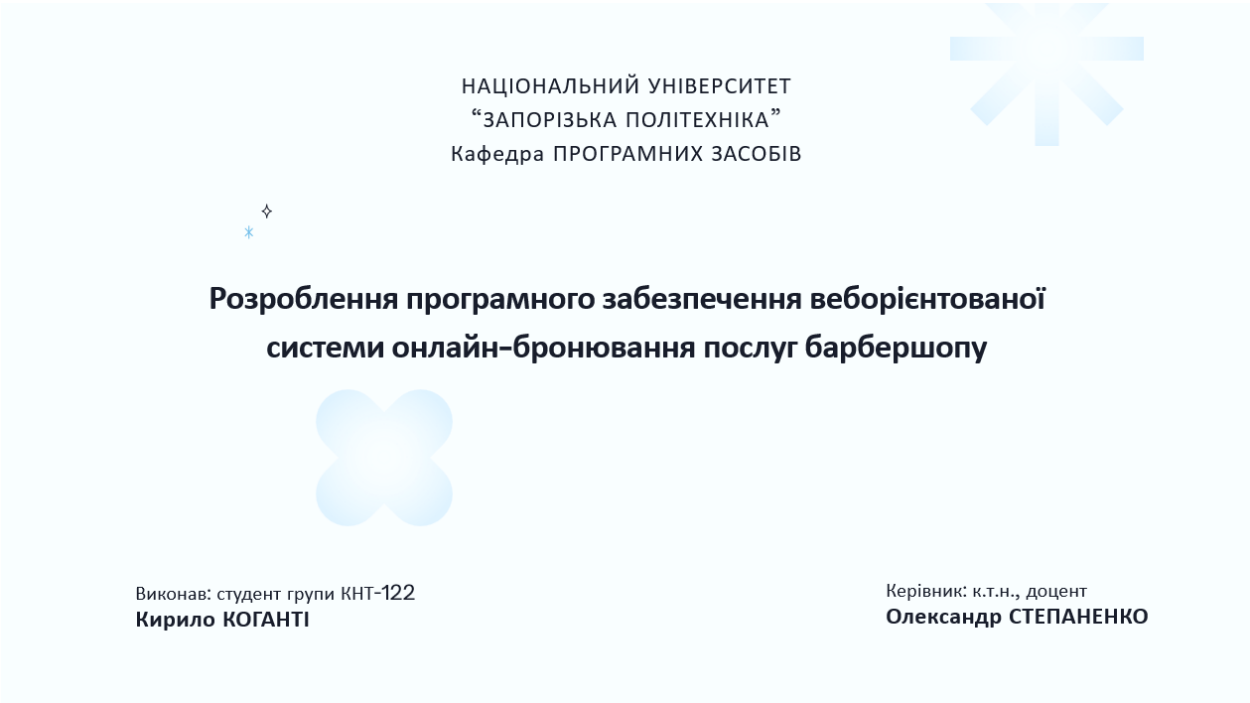

Рисунок В.1 – Слайд №1

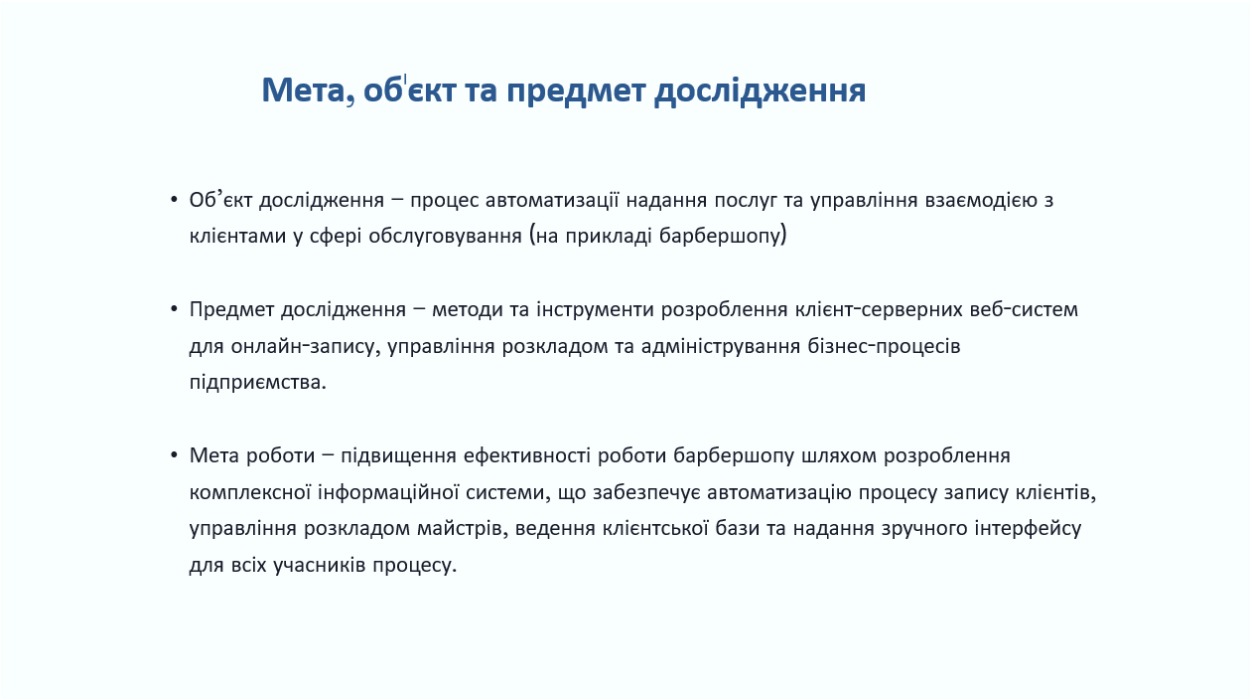

Рисунок В.2 – Слайд №2

Аналіз існуючих рішень

Критерій	Altegio	Booksy	Fresha	Velvet Razor
Веб-інтерфейс та мобільна адаптивність	+	+	+	+
Відсутність щомісячної абонентської плати	-	-	+	+
Відсутність комісій за залучених клієнтів	+	-	-	+
Гнучке управління кастомними перервами	+	-	-	+
Відсутність надлишкового корпоративного функціоналу	-	-	+	+
Повний контроль над клієнтською базою	+	-	-	+
Кастомізація інтерфейсу під бренд закладу	-	-	-	+

Рисунок В.3 – Слайд №3

Архітектура застосунку

- Клієнтський рівень
- Серверний рівень
- Рівень спільних даних
- Рівень даних

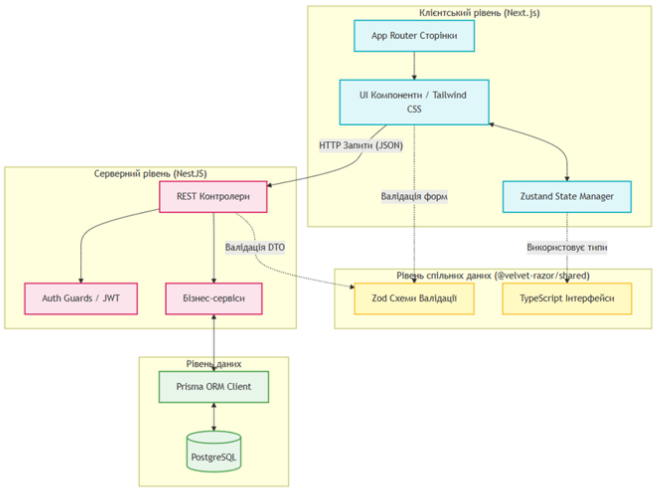

Рисунок В.4 – Слайд №4

Рисунок В.5 – Слайд №5

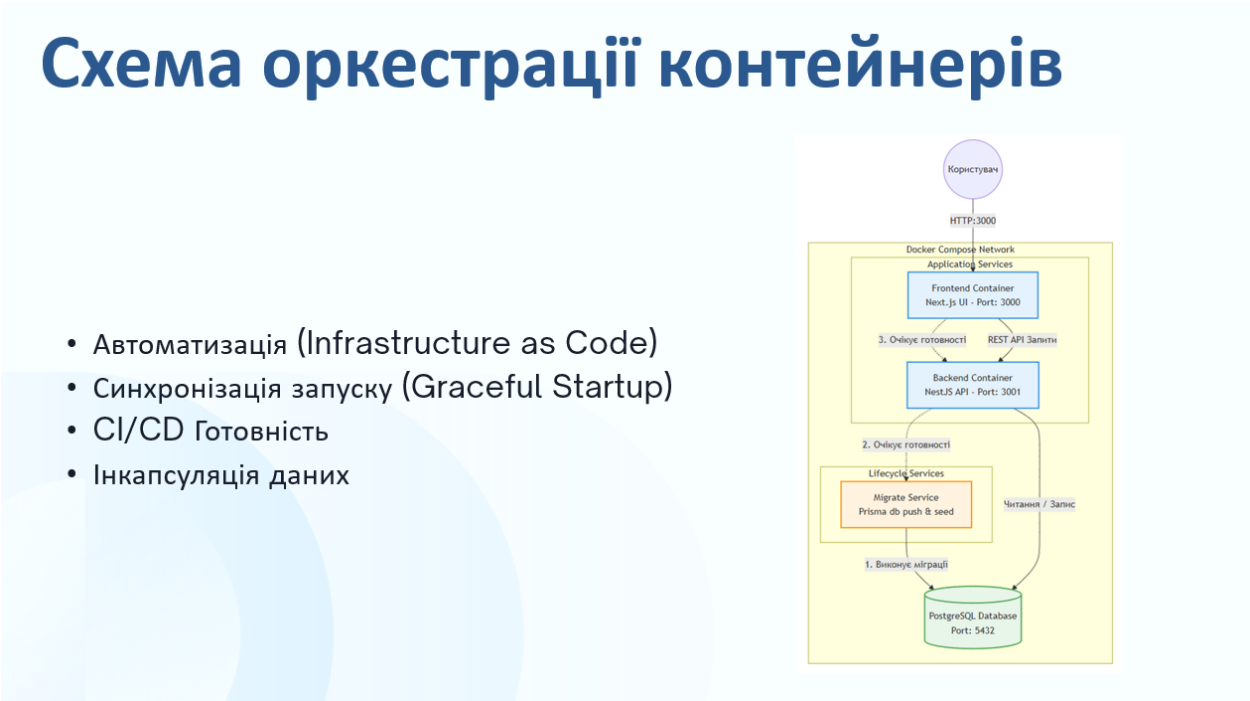

Рисунок В.6 – Слайд №6

Функціональна схема

- Єдина точка входу
- Ізольоване математичне ядро
- Транзакційний рівень доступу
- Модульність адміністративних функцій

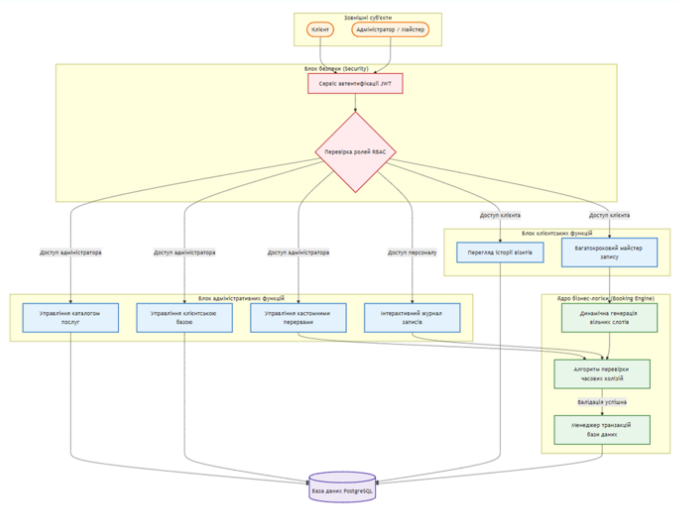

Рисунок В.7 – Слайд №7

Синхронізація даних

- Багаторівнева валідація
- Захищений транспорт
- Транзакційна цілісність
- Реактивне оновлення

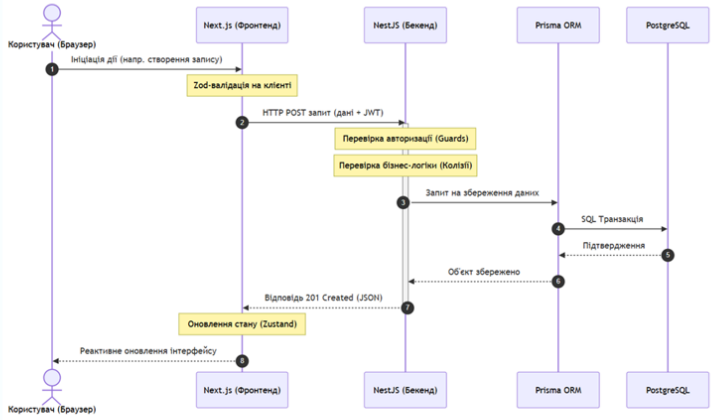

Рисунок В.8 – Слайд №8

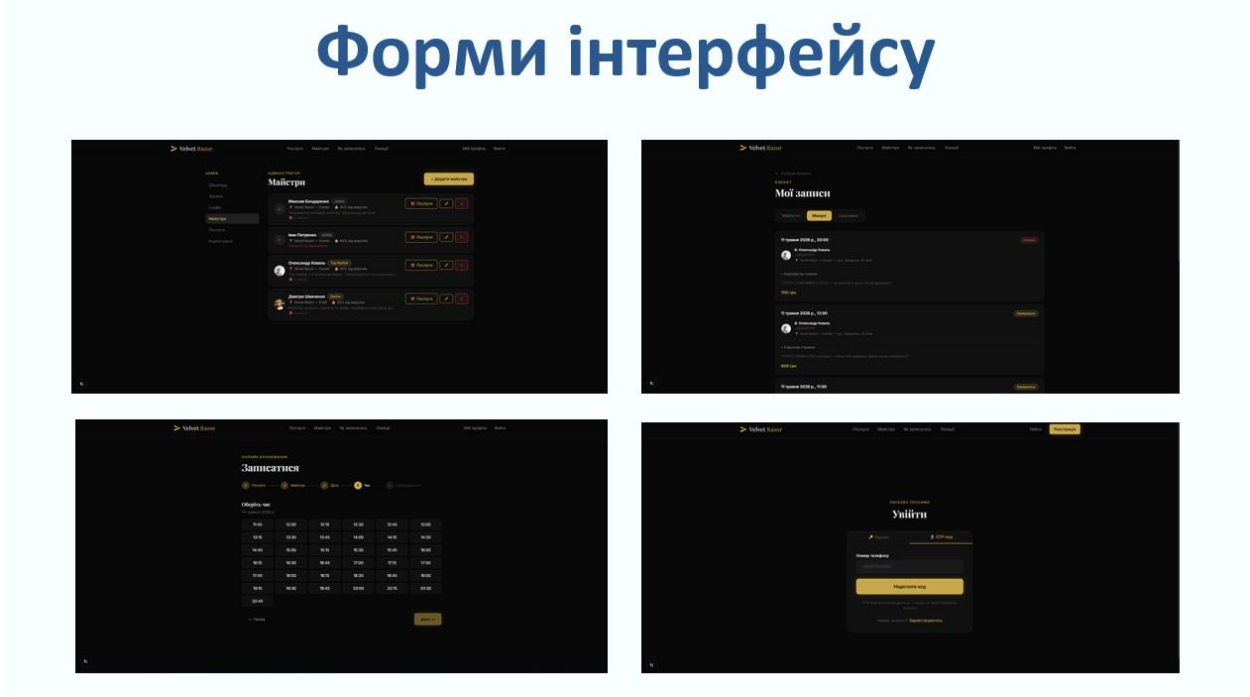

Рисунок В.9 – Слайд №9

Рисунок В.10 – Слайд №10

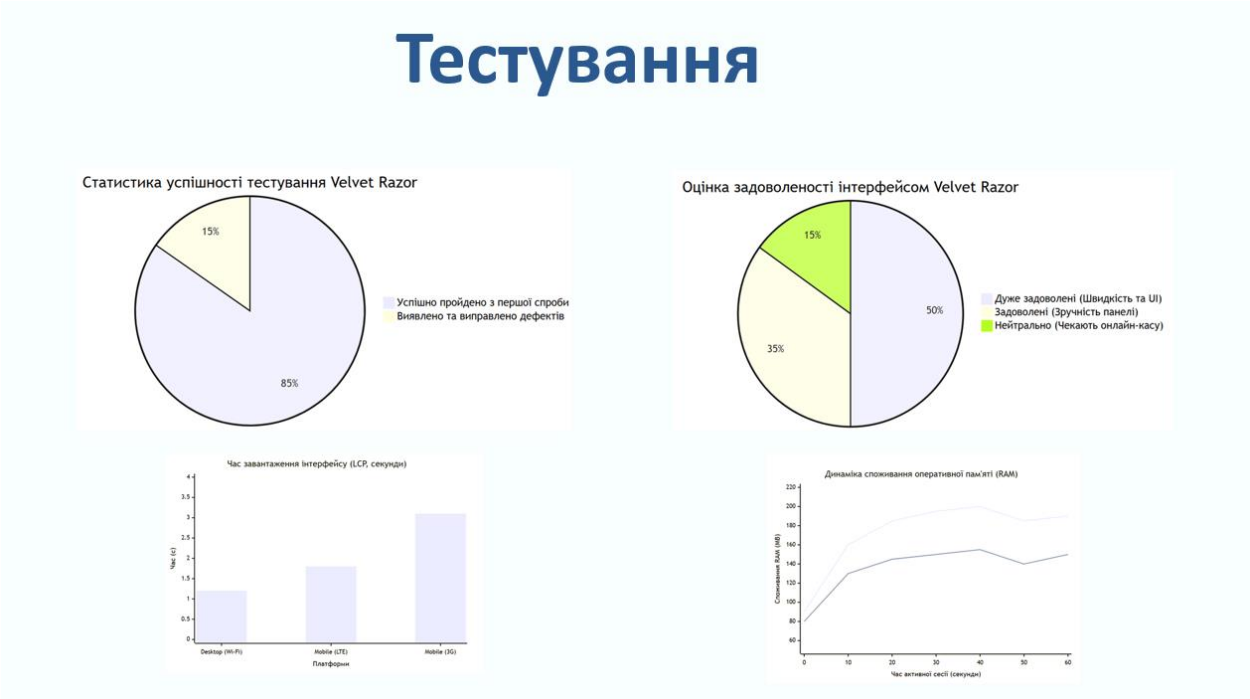

Рисунок В.11 – Слайд №11

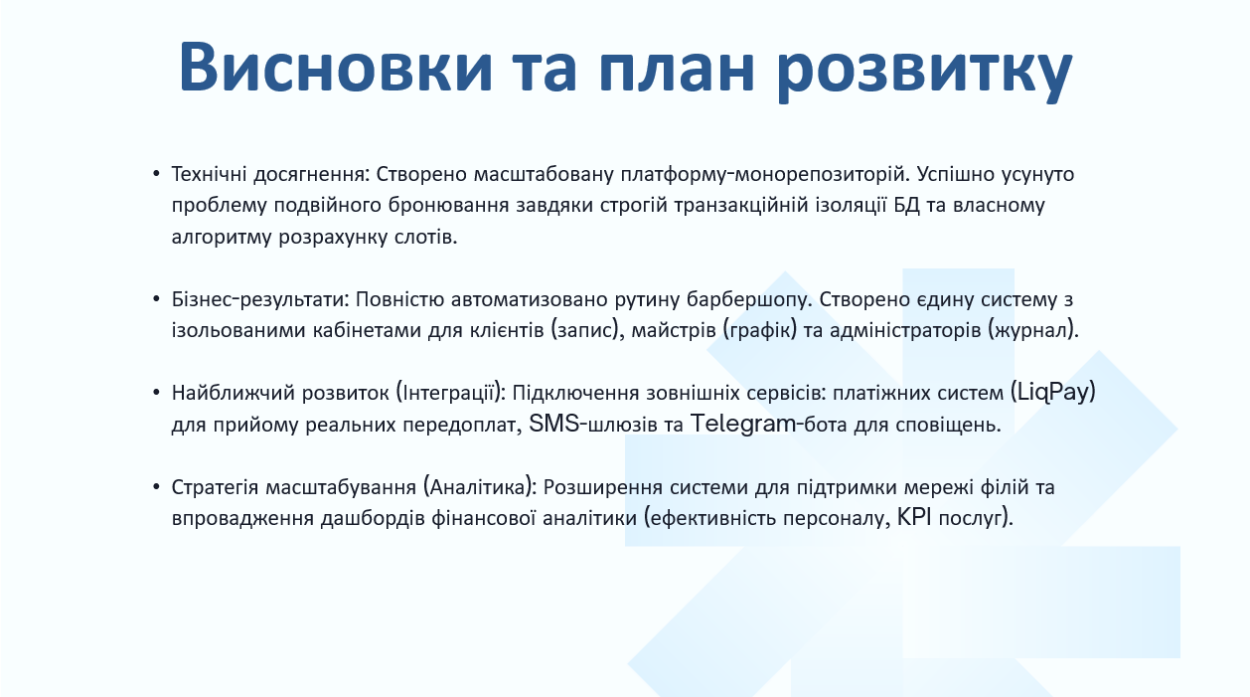

Рисунок В.12 – Слайд №12