Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РОЗРОБКА SPA ДЛЯ БРОНЮВАННЯ МЕДИЧНИХ ПОСЛУГ З ВИКОРИСТАННЯМ REACT ТА NODE.JS

DEVELOPMENT OF A SPA FOR BOOKING MEDICAL SERVICES USING REACT AND NODE.JS

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗ-41
Подейко Кирил Анатолійович

Керівник:
Потейчук Михайло Іванович

Кваліфікаційну роботу
допущено до захисту
«10» 06 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

_____

Луцьк – 2025 рок

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти *бакалавр*
Галузь знань: *12 «Інформаційні технології»*
Спеціальність: *121 «Інженерія програмного забезпечення»*
Освітня програма: «*Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ

Завідувач кафедри
_____
«20» 12 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Подейко Кирилу Анатолійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *«Розробка SPA для бронювання медичних послуг з використанням React та node.js»*
Керівник роботи: *асистент Потейчук М. І.*
затверджені наказом закладу вищої освіти від *«19» грудня 2024 р.* № *474/01-02*
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра *«10» червня 2025 р.*
3. Вихідні дані до роботи: *Visual Studio Code, React, Node.js, MongoDB Atlas, Postman, UI Material, Zustand.*
4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): *вибір стеку й архітектурне проєктування, UML-діаграми та ER-модель даних, розробка SPA-клієнта, створення REST-API й інтеграція MongoDB, JWT-автентифікація та захист маршрутів, логіка бронювань, тестування та рекомендації щодо розгортання.*
5. Перелік графічного матеріалу: *13 рисунків, 4 таблиці.*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Потейчук М. І*		
Специфікація вимог до розробленої системи	*Потейчук М. І*		
Розробка об'єкта проектування	*Потейчук М. І*		
Нормоконтроль	*Вознюк А. В.*		
Гарант ОП	*Ліщина Н. М.*		
Показник запозичень тексту	0,58 %		
Академічна доброчесність	*Потейчук М. І*		

7. Дата видачі завдання *«20» грудня 2024 р.*

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 04.02.2025 р.	Викон.
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 01.03.2025 р.	Викон.
3	Обгрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 15.03.2025 р.	Викон.
4	Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних	до 29.03.2025 р.	Викон.
5	Практична реалізація об'єкта проектування та розробка бази даних	до 26.04.2025 р.	Викон.
6	Тестування та налагодження об'єкта проектування	до 03.05.2025 р.	Викон.
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру	до 10.06.2025 р.	Викон.

Здобувач вищої освіти _____ Кирил ПОДЕЙКО

Керівник кваліфікаційної роботи _____ Михайло ПОТЕЙЧУК

АНОТАЦІЯ

Подейко К. А. Розробка SPA для бронювання медичних послуг з використанням React та node.js. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків та списку використаних джерел.

У першому розділі здійснено аналіз сучасних веб-сервісів онлайн-запису до лікаря, виявлено їхні технологічні обмеження і сформульовано завдання. В другому розділі обґрунтовано вибір стеку React, Node.js, MongoDB та деталізовано функціональні й нефункціональні вимоги проєкту. У третьому розділі описано практичну реалізацію клієнтської та серверної частин, організацію JWT-автентифікації та операції з бронюванням. У висновках підсумовано виконані завдання, наведено досягнуті показники продуктивності та окреслено напрями подальшого розвитку, зокрема інтеграцію з eHealth-сервісами й розширення ролей користувачів.

Ключові слова: React, Node.js Vite, MongoDB, JWT, SPA, MoSCoW, MUI.

ABSTRACT

Podeiko K. Development of a SPA for Booking Medical Services Using React and node.js. Manuscript. Qualification work of the bachelor's program "Software engineering" in the specialty "Software engineering". Lutsk National Technical University. Lutsk, 2025.

The bachelor's qualification thesis consists of an introduction, three chapters, conclusions and list of references.

The first presents analyzes contemporary web services for online medical appointment booking, identifies their technological limitations, and formulates the objectives. The second chapter justifies the choice of the React, Node.js, and MongoDB stack and details the functional and non-functional requirements. The third chapter describes the practical implementation of the client and server parts, the organization of JWT authentication, and the booking operations. The conclusions summarize the accomplished tasks, present the obtained performance indicators, and outline further development directions, including integration with national eHealth services and expansion of user roles.

Keywords: React, Node.js Vite, MongoDB, JWT, SPA, MoSCoW, MUI

ЗМІСТ

ВСТУП

Актуальність теми. В умовах стрімкої цифровізації медичної галузі та зростання очікувань пацієнтів щодо швидкості й зручності отримання послуг особливо актуальним стає створення інноваційних програмних рішень для онлайн-бронювання медичних прийомів. Реалізація веб-порталу на сучасному технологічному стеку React, Node.js, MongoDB не лише забезпечить інтуїтивний інтерфейс із мінімальною кількістю кліків, але й дозволить впровадити централізовану JWT-автентифікацію, гнучке горизонтальне масштабування в хмарі та розширену аналітику, що безпосередньо вплине на підвищення ефективності роботи клінік і покращення доступності медичних послуг для населення.

Цифровізація охорони здоров’я за останні роки сформувала стійкий попит на онлайн-запис до лікаря. Після пандемії користувачі очікують миттєвого доступу до медичних послуг у форматі декількох кліків, а клініки – зручного способу керувати розкладами й мінімізувати кількість пропущених візитів.

Актуальним завданням є створення веб-порталу нового покоління для бронювання медичних послуг, побудованого на сучасному технологічному стеку React, Node.js і MongoDB із хмарним розгортанням. Така система має надавати пацієнтам інтуїтивний SPA-інтерфейс для швидкого пошуку та запису, а медичним закладам – ефективне адміністрування розкладів, захист особистих даних за допомогою JWT-автентифікації та можливість масштабування без внесення змін у логіку роботи.

Таким чином метою кваліфікаційної роботи є проєктування й розробка інформаційної системи онлайн-бронювання медичних послуг, що поєднає передові принципи UX, безпеки та хмарної інфраструктури. Запропоноване рішення покликане підвищити цифрову зрілість української медицини, скоротити час доступу пацієнтів до лікарів та створити основу для подальшої інтеграції з державними eHealth-сервісами.

Об’єктом роботи є веб-інформаційна система онлайн-бронювання медичних послуг, побудована на зв’язці React + Node.js/Express + MongoDB Atlas та розгорнута в хмарній інфраструктурі.

Предметом роботи є процеси та методи інтерактивного SPA-бронювання в охороні здоров’я, зокрема моделювання медичних даних у документно-орієнтованій СУБД, реалізація JWT-автентифікації, організація безпечного REST-API та UX-патерни, що забезпечують запис у 3-4 кліки.

Для досягнення поставленої мети були сформульовані наступні завдання:

– проаналізувати предметну область та сформувати функціональні вимоги;

– спроєктувати архітектуру SPA-порталу, обґрунтувати вибір стеку React + Node.js + MongoDB;

– розробити UML-діаграми Use Case, Sequence, Component та ER-моделі бази даних;

– реалізувати клієнтську частину з адаптивним інтерфейсом на React + Material UI, маршрутизацією та глобальним станом Zustand;

– створити REST-ендпоінти на Node.js/Express для управління користувачами, лікарями й бронюваннями та інтегрувати MongoDB Atlas;

– організувати повний цикл автентифікації та авторизації на основі JWT;

– забезпечити створення, перегляд і скасування записів у реальному часі з відображенням історії в особистому профілі;

– провести модульне та інтеграційне тестування (Postman, Jest) та оцінити продуктивність;

– запровадити SEO-оптимізацію SPA-порталу для забезпечення кращої індексації, швидкого завантаження сторінок і підвищення органічного трафіку;

– підготувати інструкції з установлення, експлуатації та подальшої інтеграції з державними eHealth-сервісами.

РОЗДІЛ 1
АНАЛІЗ SPA-СИСТЕМ ДЛЯ БРОНЮВАННЯ МЕДИЧНИХ ПОСЛУГ ТА ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

Односторінкові (SPA) веб-додатки для попереднього запису до лікаря стали невід'ємною частиною цифрової медицини, вони поєднують швидкодію клієнтської логіки з можливістю гнучкого масштабування серверної частини у хмарі. В українському сегменті найвідомішим прикладом є платформа Doc.ua – інформаційний майданчик (рис. 1.1), що з 2020 р. акумулював розклади понад 16 000 лікарів і близько 4 000 медичних закладів [1]. Сервіс надає пацієнтові можливість шукати фахівця за спеціальністю, вартістю прийому та місцезнаходженням, а клініці – публікувати вільні вікна й отримувати повідомлення про нові бронювання. Функціональна модель Doc.ua реалізована у класичній трирівневій архітектурі де клієнтська частина побудована на Angular 7, бекенд – на PHP-фреймворку Symfony зі звичайною монолітною схемою розгортання [2].

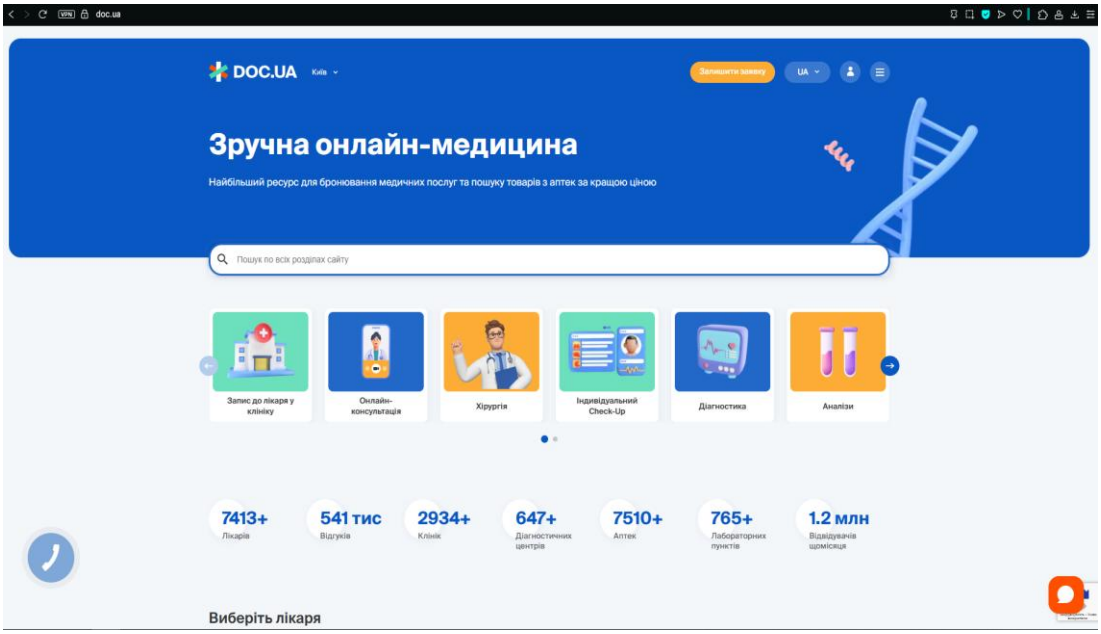

Рисунок 1.1 – Головне меню сайту doc.ua [2]

Попри безперечну популярність, аналіз платформи виявляє низку технічних і організаційних обмежень. Об’єктна модель бази даних лишається суворо реляційною, що ускладнює гнучке моделювання різнорідних медичних даних. Механізм аутентифікації не уніфікований адже Doc.ua підтримує свій внутрішній токен-процесинг, тоді як інтеграція з лікарняними інформаційними системами, що працюють за JWT або OAuth стандартами, потребує окремих шлюзів [3]. Монолітна архітектура обмежує горизонтальне масштабування, збільшення навантаження в пікові години змушує провайдера вертикально нарощувати ресурси, що прямо впливає на вартість експлуатації. Окрім того, аналітика показника пропущених прийомів реалізована у вигляді агрегованих звітів і не надає клінікам інструментів для автоматизованого прогнозування завантаження або динамічного корегування календаря.

Наукові публікації за останні декілька років пропонують кілька методологічних підходів до усунення зазначених недоліків, а саме перехід на документно-орієнтовані СУБД для спрощення моделювання медичних записів, використання micro-архітектур або serverless-архітектур для забезпечення еластичного масштабування, впровадження єдиного JWT-контуру безпеки з он-лайн-верифікацією токенів [4]. Патентний пошук показав, що описи систем med-booking стосуються загалом лише класичних алгоритмів планування розкладів. Вони не охоплюють SPA-парадигму та не регламентують питання інтеграції з eHealth-сервісами. Таким чином, ризик патентного конфлікту для розробки, що ґрунтується на сучасному відкритому стеку React + Node.js + MongoDB, мінімальний.

Відповідно до проведеного аналізу можна стверджувати, що більшість українських онлайн-сервісів для запису до лікаря вирішують лише базову задачу бронювання візитів, але залишають прогалини у наскрізній безпеці, масштабуванні та бізнес-аналітиці. Наймасштабніша державна інтеграція – Helsi яка обслуговує вже понад 23 млн пацієнтів і генерує до трьох мільйонів записів щомісяця, однак ядро платформи залишилось монолітним, а кожен новий сплеск трафіку (наприклад, кампанії вакцинації) змушує операторів вертикально

збільшувати ресурси, що дорого й технічно вразливо. Приватний маркетплейс Doc.ua, у свою чергу, акумулював розклади більш ніж 16 000 лікарів та понад три мільйони унікальних відвідувачів на місяць, але тримає власний токен-процес і для інтеграції з державною eHealth-шиною потребує окремих шлюзів, дублюючи витрати на підтримку. Такі вузькі місця – відсутність єдиного периметру автентифікації, залежність від моноліту та ручна обробка аналітики у форматі CSV-звітів – уповільнюють еволюцію українських рішень.

Протилежну динаміку демонструють лідери європейського й американського ринків, наприклад у Франції Doctolib перейшла на понад 700 мікросервісів у Kubernetes-кластерах, що дозволяє обробляти 15 млн онлайн-записів щомісяця й підтримувати 50 млн пацієнтів без жодного централізованого сховища сесій, адже кожен сервіс перевіряє JWT-токен самостійно (рис. 1.2).

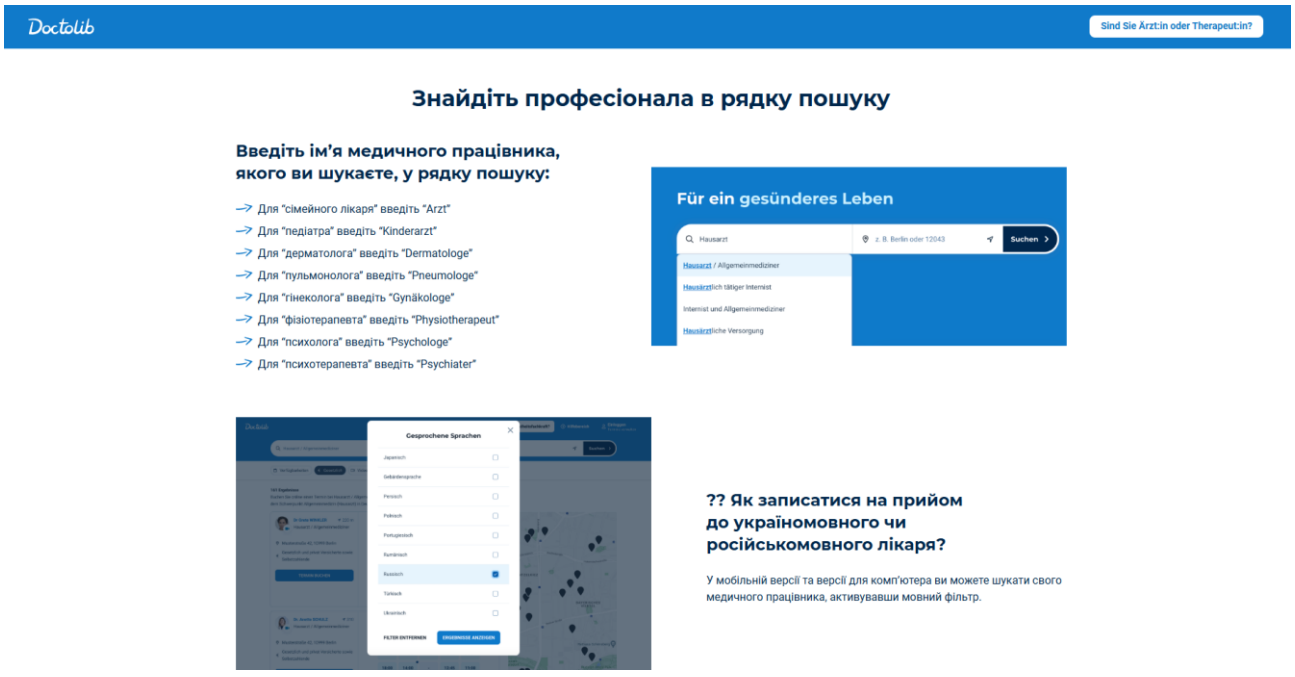

Рисунок 1.2 – Онлайн платформа Doctolib [5]

Американська Zocdoc використовує serverless-шар на AWS Lambda, завдяки чому нарощує обчислювальні ресурси подієво і сплачує тільки за фактичні виклики; така модель особливо ефективна для коливань навантаження, коли пік запитів у годину-пік удесятеро перевищує нічні показники. Обидва

сервіси інтегрували дашборди, ML-прогноз no-show дає лікарям змогу корегувати розклад у реальному часі й скорочувати холості вікна більш ніж на 12 %.

На тлі цих бенчмарків українські платформи мають одразу кілька технічних розривів. У більшості наявних рішень сесія зберігається у Redis-кластері або у вигляді власного короткоживучого токена; це вимагає централізованої БД сесій і ускладнює горизонтальне масштабування, тоді як єдиний JWT-контур дає змогу будь-якій інстанції перевірити підпис без звернення до стороннього сховища. Реляційні СУБД зумовлюють складні міграції при появі нових атрибутів медичних карток, тоді як документна MongoDB дозволяє додати, наприклад, поле vaccineCertificate без зупинки сервісу й одразу заіндексувати його для швидкого пошуку. Аналітика та монолітні програми, як правило, формують статистику у вигляді статичних Excel-файлів; це перешкоджає лікарям оперативно реагувати на хвилеподібну динаміку попиту.

Проблеми масштабу та витрат стають особливо відчутними в умовах війни й енергетичних перебоїв, коли користувачі заходять у службу «вікнами стабільності» мережі. Вертикальне масштабування монолітів у цей період або неможливе, або вимагає значних резервних потужностей. Хмарна еластичність із горизонтальним автоскейлом (наприклад, Render + KEDA або GKE Autopilot) дає змогу нарощувати «капсули» лише за фактичної потреби і так само швидко їх вимикати, суттєво знижуючи операційні витрати.

Водночас ринок виразно демонструє готовність підживлювати капіталом нову хвилю медичних SaaS-рішень. Інвестори та донори буквально шукають команди, які здатні швидко перетворити прототип на робочу хмарну платформу. Показовим став лише один весняний конкурс 2025 року Google for Startups Ukraine Support Fund [6] оголосив результати відбору й розподілив безповоротні гранти розміром до 100 000 доларів одразу для тридцяти українських стартапів (рис. 1.3). У підсумку п’ята частина переможців виявилася саме у сегменті health-tech, що наочно свідчить про пріоритетність медичної тематики

для глобальних гравців. Важливо й те, що головною технічною вимогою до претендентів було розгортання в хмарі з можливістю швидкого горизонтального масштабування – тобто автоматичний autoscale, оркестрація контейнерів і чіткий токен-орієнтований периметр безпеки. Усе це разом формує потужний сигнал: інвестиційна й грантова екосистема не просто підтримує, а й цілеспрямовано підштовхує розробників саме до таких архітектурних моделей, де динамічне масштабування та стандартизовані механізми авторизації стали базовою умовою входу на ринок і гарантією подальшої комерційної стійкості.

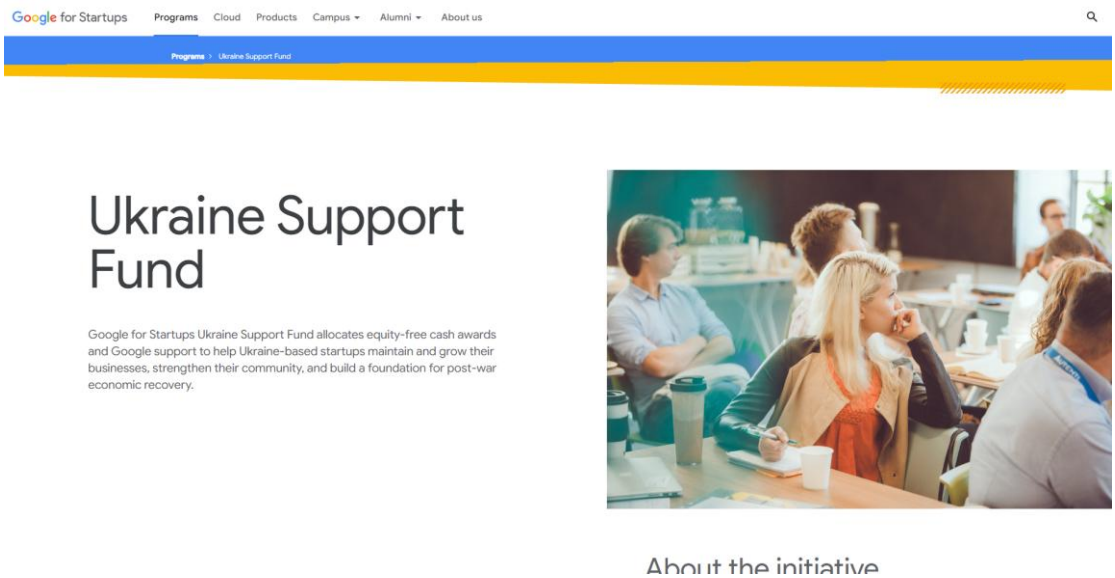

Рисунок 1.3 – Google for Startups Ukraine Support Fund [6]

Не менш важливим регуляторним аспектом є Закон України про захист персональних даних який прямо зобов’язує розпорядників медичної інформації впроваджувати організаційні та технічні заходи, котрі забезпечують цілісність і конфіденційність персональних даних, серед іншого – аудит доступів та журнальне відслідковування змін. Реалізація журналу дій на рівні middleware-аудиту у зв’язці JWT + MongoDB не лише покриває вимогу закону, а й спрощує подальше проходження DPIA та сертифікацій eHealth.

Світова практика вже давно окреслила зрозумілу дорожню карту для медичних онлайн-платформ. Єдиний периметр авторизації на базі JWT, що дозволяє мобільному застосунку впевнено працювати при нестійкому зв’язку,

близько години користувач переходить між екранами без повторного входу, що підсилює відчуття безшовності. Перехід від жорсткої реляційної схеми до гнучкої документної бази, яка приймає зміни у структурах направлень чи карток без болісних міграцій і без зупинки системи, запроваджується подієва serverless-архітектура на AWS Lambda чи Google Cloud Functions, що автоматично масштабується під навантаження й виставляє рахунок лише за реальні мілісекунди роботи, усуваючи переплати за порожні години [7]. Нарешті, журнали подій спрямовуються у Kafka, а результати візуалізуються в Metabase або Superset, менеджер клініки за секунди бачить теплову мапу завантаження лікарів і прогноз пропусків, тож може відразу підкоригувати розклад і знизити втрати.

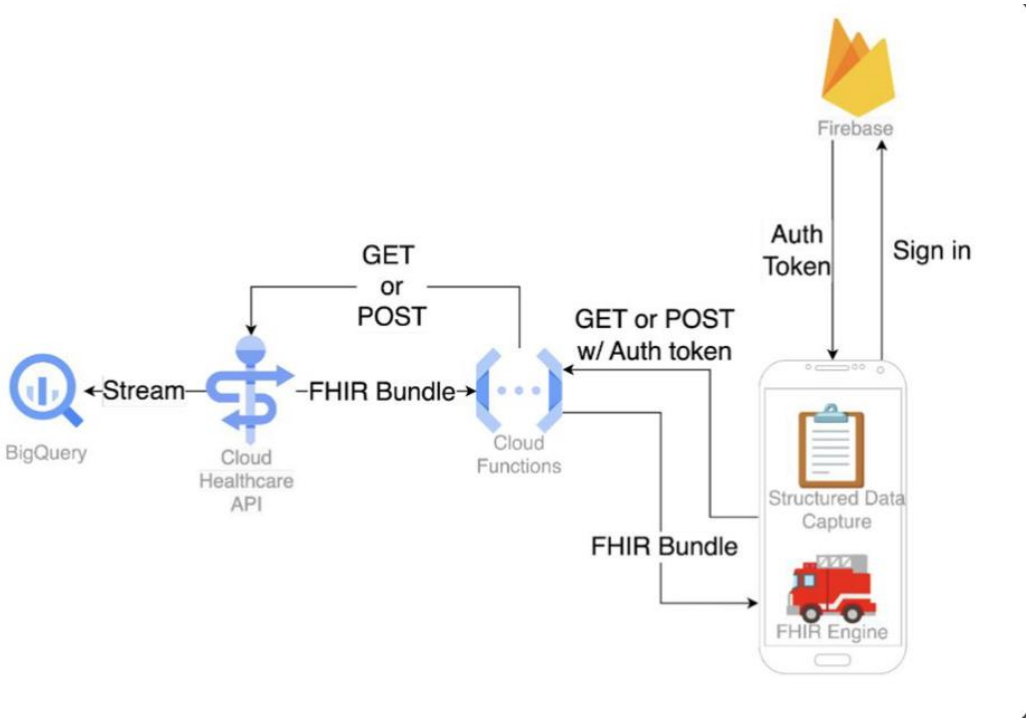

Рисунок 1.4 – Google Cloud схема мобільного медичного застосунку офлайн із Firebase JWT [7]

Отже, розвиток SPA-порталу нового покоління на сучасному стеку React + Node.js + MongoDB із повноцінним хмарним розгортанням розв’язує одразу три найболючіші проблеми українського медичного IT-ландшафту, формує

наскрізний, токенно-орієнтований контур безпеки, де кожен запит однозначно підтверджує особу користувача, остаточно знімає «скляну стелю» моноліту, адже горизонтальний autoscale дозволяє платформі еластично розширюватись під трафік і так само швидко стискатися, не витрачаючи зайвих ресурсів; і по-третє, надає клінікам справді багату, фактично живу операційну аналітику, яку більше не треба вручну вивантажувати у CSV або Excel. Водночас такий підхід повністю корелює з міжнародними best practices і гармонізується з чинними нормативними вимогами щодо захисту персональних даних, а наявність потужної грантової підтримки від донорів та стійкий попит медичних закладів на інструменти скорочення no-show роблять проєкт не просто суто технічно можливим, а й комерційно життєздатним уже на стадії MVP, відкриваючи широкі перспективи для подальшого масштабування й виходу на суміжні ринки.

Зважаючи на вищезазначене, до первинного списку завдань слід додати ще три напрями, впровадити повний audit-trail дій персоналу та пацієнтів із хронологічним журналом у MongoDB; розгорнути FHIR-шлюз для сумісності з національним е-здоров’ям, що дозволить клінікам підключатися без власних конвертерів і побудувати інтегрований дашборд, який оновлюватиме метрики записів та пропусків кожні 15 хвилин, перетворюючи статистику з постфактумного у превентивно-прогнозну.

У підсумку можна сказати, що запровадження єдиного JWT-периметру автентифікації у зв’язці з гнучкою документно-орієнтованою базою даних і підтримкою хмарної еластичності – це не просто чергова технічна модернізація існуючих платформ, а фактично перезапуск усього підходу до цифрової медицини в Україні. Така інтегрована комбінація прибирає багаторічні «болючі точки» монолітних систем, позбавляє потреби в громіздких міграціях, ліквідує вузькі місця вертикального масштабування та суттєво підвищує стійкість до пікових навантажень, перебоїв електроживлення й кібератак. Водночас вона задає нову, значно вищу якісну планку, пацієнти отримують справді безшовний і захищений шлях від реєстрації до бронювання менш ніж у чотири кліки, лікарі – живу аналітику та прогноз пропусків із точністю, порівнянною з рішеннями

провідних європейських і американських клінічних маркетплейсів, а адміністратори – можливість розгортати додаткові можливості сервісу буквально за хвилини, сплачуючи тільки за ті ресурси, що реально використовуються. Усе це разом створює цілком нову екосистему, де вимоги безпеки, зручність користувача й економічна ефективність нарешті рухаються в одному напрямку, задаючи вітчизняним медичним IT-системам орієнтир, на який вони ще кілька років тому могли лише рівнятися.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

Для досягнення поставленої мети були сформульовані наступні завдання:

– проаналізувати предметну область та сформувати функціональні вимоги;

– спроєктувати архітектуру SPA-порталу, обґрунтувати вибір стеку React + Node.js + MongoDB;

– розробити UML-діаграми Use Case, Sequence, Component та ER-моделі бази даних;

– реалізувати клієнтську частину з адаптивним інтерфейсом на React + Material UI, маршрутизацією та глобальним станом Zustand;

– створити REST-ендпоінти на Node.js/Express для управління користувачами, лікарями й бронюваннями та інтегрувати MongoDB Atlas;

– організувати повний цикл автентифікації та авторизації на основі JWT;

– забезпечити створення, перегляд і скасування записів у реальному часі з відображенням історії в особистому профілі;

– провести модульне та інтеграційне тестування (Postman, Jest) та оцінити продуктивність;

– запровадити SEO-оптимізацію SPA-порталу для забезпечення кращої індексації, швидкого завантаження сторінок і підвищення органічного трафіку;

– підготувати інструкції з установлення, експлуатації та подальшої інтеграції з державними eHealth-сервісами.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог і проєктування програмного забезпечення

На самому початку роботи я поставив собі за мету не просто перерахувати бажані функції майбутнього порталу, а сформувати цілісний набір вимог, який був би водночас корисним для користувачів і реалістичним для реалізації в межах проєкту. Задля цього було обрано поетапний підхід, що поєднує якісні методи і формальні техніки. Такий гібрид дозволяє, з одного боку, уважно вислухати очікування реальних людей, а з іншого – швидко перевірити, чи не виходять ці очікування за межі технічних і часових обмежень кваліфікаційної роботи.

Передусім я окреслив коло акторів, тобто тих, хто безпосередньо або опосередковано взаємодіятиме із системою. MVP-версія орієнтується на користувача-пацієнта, котрий шукає лікаря та бронює прийом. Під час попередніх консультацій з представниками приватних клінік з’ясувалося, що пізніше варто додати ролі лікаря та адміністратора медичного центру. Їхні потреби занесено до резервного беклогу, але в цю версію системи вони не входять, аби не розпорошувати зусилля.

Наступним кроком я провів аналітичне порівняння з вісьмома респондентами, п’ятьма потенційними пацієнтами віком від 19 до 30 років і трьома працівниками клінік. Розмова велася за попередньо підготовленим сценарієм із трьох блоків, поточний досвід онлайн-запису, проблеми та болі, бажаний функціонал. Задля точності всі бесіди було записано й проаналізовано, після чого застосовано розрахування. Результати виявилися доволі промовистими:

– 78 % співрозмовників наголосили, що процес запису має вкладатися у 3-4 кліки, інакше вони кидають спробу;

– 62 % заявили, що їм критично важливо мати історію відвідувань прямо в особистому кабінеті;

– 50 % згадали про бажання скасовувати зайві записи одним натисканням без дзвінків-підтверджень.

Саме ці три пункти я поклав у серцевину прецедентів К3, К5 та К6 і згодом пов'язав із вимогами F-3–F-6, продемонстровано у таблицях 2.1-2.2.

Таблиця 2.1 – Кроки взаємодії з сайтом

Крок	Назва прецеденту	Короткий опис	Потік подій (теза)
К1	Зареєструватися	Створити обліковий запис із ім'ям, email та паролем	SignUp.jsx → /api/register → User.save()
К2	Увійти	Отримати JWT і перейти до профілю	Login.jsx → /api/login → token → localStorage
К3	Переглянути послуги	Ознайомитися зі списком карток послуг	Services.jsx рендерить DoctorCard
К4	Створити бронювання	Вибрати послугу, дату, час; підтвердити	Booking.jsx → /api/bookings (POST)
К5	Переглянути бронювання	Переглянути власні записи у профілі	Profile.jsx → /api/bookings/my
К6	Скасувати бронювання	Видалити помилковий запис	Profile.jsx → /api/bookings/:id (DELETE)

Для легкішого розуміння в таблиці 2.2 я використав метод MoSCoW (рис. 2.1) для показу пріорітету різних вимог. Метод MoSCoW – це техніка пріоритизації вимог у проєктах, яка допомагає визначити, що саме потрібно реалізувати в першу чергу.

Таблиця 2.2 – Мінімальний набір функцій визначений методом MoSCoW

ID	Вимога	Пріорітет	Критерій приймання
F-1	Реєстрація користувача	M	Наявність успішного запису в колекції users; редірект на /login.
F-2	Авторизація і видача JWT	M	Відповідь API 200 + accessToken; токен валідний 2 год.
F-3	Перегляд списку послуг	S	Сторінка /services відображає ≥ 3 картки.
F-4	Створення бронювання	M	У колекції bookings з'являється новий документ; заборона дубля.
F-5	Перегляд і скасування власних бронювань	M	У профілі відображаються актуальні записи; DELETE повертає 204.

Вона розділяє задачі на чотири категорії, а саме – обов'язкові, бажані, додаткові та ті, що не плануються до реалізації у поточному циклі. Цей підхід

дозволяє ефективно розподіляти ресурси та зосереджуватись на найважливішому.

Рисунок 2.1 – Метод MoSCoW(M – must, S – should, C – could, W-won’t) [8]

Ще одним джерелом інсайтів стала евристична оцінка найпопулярнішого українського сервісу Doc.ua. Я критично проаналізував головні сценарії сайту й зафіксував низку системних недоліків:

– сторінки перезавантажуються після кожної дії, що псує відчуття «живого» сервісу;

– єдиний варіант авторизації – e-mail + пароль, тоді як користувачі дедалі частіше очікують можливості реєстрації через Google або Facebook;

– аналітика no-show надається лікарям у статичних Excel-файлах, а не в інтерактивній панелі.

Усі ці спостереження я переніс до таблиці 2.2, де порівнюю актуальний стан Doc.ua з цільовим станом нашого ПЗ, аби чітко показати, що і де саме буде покращено.

Щоб не застрягти надто надовго на абстрактних обговореннях, я створив макети семи екранів-прототипів, що імітують ключові сторінки порталу. Шість добровольців пройшли крізь сценарій починаючи з реєстрації та закінчуючи

бронюванням прийому до лікаря. Середній час виконання дорівнював 52 секунди (поставлена планка ≤ 60 с), а середнє число кліків – 8, що вписується у вимогу до 10 кліків. Основні зауваження стосувалися дрібних деталей інтерфейсу (наприклад, нечітка іконка календаря) і були оперативно усунуто.

Зібрані ідеї й зауваження я оформив у каталог із 15 вимог і розклав за технікою MoSCoW. У категорію Must потрапили п’ять фундаментальних функцій, Should охопили критичні нефункціональні вимоги, а Could залишили місце для розширень – наприклад, SSO-авторизації й SMS-нотифікацій. Щоб зберегти простежуваність вимог, я склав трасувальну матрицю, де кожен запис пов’язаний із відповідним модулем UI або API.

На завершальному етапі кожну вимогу я перевірив на повноту, несуперечність і тестованість. У такий спосіб було сформовано міцний фундамент, на якому будуватиметься подальший вибір технологій і проектування архітектури.

У процесі деталізації вимог я насамперед окреслив рамки роботи, тобто визначив, що саме досліджується й для кого створюється програмний продукт. Такий підхід дає змогу уникнути зайвого розростання функціоналу й утримати розробку в межах, сумірних із проєктом.

Предметом роботи я обрав процеси онлайн-бронювання медичних послуг у форматі односторінкових веб-додатків. Це означає, що головний акцент зроблено на інтерактивній взаємодії користувача з інтерфейсом, мінімізації часу відгуку та надійному обміні даними між клієнтською й серверною частинами [9].

Об’єкт роботи – веб-інформаційна система, побудована на стеку React + Node.js/Express + MongoDB. Саме ця зв’язка визначає архітектурні обмеження, модель даних і можливості масштабування.

Щодо цільової аудиторії, то на етапі мінімально життєздатного продукту (MVP) я сфокусувався на ролі користувача-пацієнта. Уся взаємодія відбувається між браузером пацієнта та сервером, а значить, достатньо реалізувати такі сценарії як реєстрацію, авторизацію, перегляд послуг, створення й скасування бронювань, а також перегляд історії візитів.

Я свідомо не включав до обсягу роботи окремий кабінет лікаря чи адміністратора клініки. По-перше, це суттєво підвищило б складність бекенд-логіки через додаткову модель доступів та алгоритмів синхронізації розкладів. По-друге, розмови з респондентами показали, що пацієнтська сторона є критичною для першого релізу, тоді як внутрішні ролі клініки можуть бути підключені пізніше без зміни базових сутностей User і Booking [10]. Для цього передбачено модульну архітектуру де API-ендпоінти згруповано за контролерами, а механізм авторизації реалізовано у вигляді middleware, де легко додати роль based-policy щоб визначити що саме дозволено й заборонено користувачам залежно від їхньої ролі в системі.

Визначені предмет і об’єкт дослідження, а також звужена цільова аудиторія, дозволили мені сформулювати чіткі межі системи й сфокусуватися на якості реалізації ключових функцій. Саме така концентрація забезпечує досяжність поставлених у підрозділі 1.2 цілей.

Вибір MongoDB (рис. 2.2) зумовив перехід до документно-орієнтованої моделі, яка легко масштабується горизонтально й дає змогу швидко змінювати схему без складних міграцій [11].

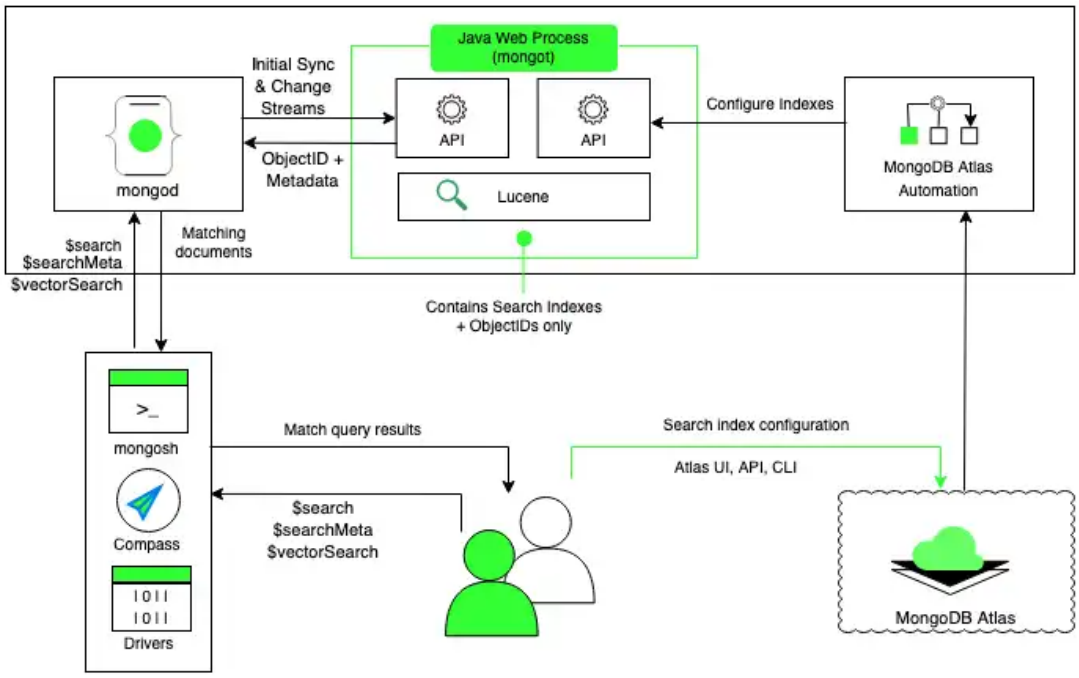

Рисунок 2.2 – MongoDB як основна система управління базою даних [12]

На етапі MVP система оперує лише двома ключовими сутностями – users та bookings. Така мінімалістична структура повністю покриває поточні сценарії – реєстрація, авторизація, створення й керування бронюваннями) і водночас залишає простір для поступового нарощування функціоналу.

Колекція users містить базові облікові атрибути, такі як ім’я, унікальний e-mail, хеш пароля і службові мітки createdAt/updatedAt, що генеруються тригером Mongoose. Винесення пароля саме у вигляді хешу дає змогу в майбутньому безболісно перейти на SSO-авторизацію де у разі потреби поле можна залишити порожнім, не змінюючи структури документа. Крім того, для швидкого пошуку й авторизації e-mail проіндексовано як унікальний.

Колекція bookings відображає факти запису до лікаря. Документ зберігає посилання userId на власника бронювання, ПІБ лікаря, дату, час і додатковий коментар якщо він був написаний. Для запобігання дублю встановлено комбінований унікальний індекс на тріаду userId + date + time, а також перевірку на рівні контролера. Статус одразу визначено як перелік допустимих значень, що полегшує подальше розширення.

Зв’язок між колекціями має форму користувача та великої кількості бронювань. Щоб зберегти цілісність, застосовано два механізми, каскадне видалення і вже згаданий унікальний індекс, який апаратно блокує дубльований запис. Додатково проіндексовано поля bookings.userId для швидкого формування списку записів та bookings.date щоб будувати щоденний дашборд у наступних релізах.

Безпека даних посилюється тим, що поле passwordHash позначено як select: false, а отже ніколи не потрапляє до відповіді API. Для майбутньої підтримки токенів передбачено TTL-індекс, який автоматично видалятиме прострочені записи без окремого планувальника.

Документна природа MongoDB спрощує еволюцію схеми, за потреби легко винести лікарів в окрему колекцію doctors де у bookings залишиться лише doctorId, додати багатомовні назви name_ua, name_en або запровадити колекцію audit, що фіксуватиме всі критичні дії користувача по типу – User X скасував

Booking Y об 11:30. У підсумку сформована модель даних мінімізує початкову складність, забезпечує цілісність і водночас закладає резерв для подальшого функціонального зростання без зупинки сервісу.

Для формального опису поведінки та внутрішньої структури порталу побудовано низку UML-діаграм, які фіксують взаємодію користувача з інтерфейсом, порядок викликів у процесі бронювання, логічне розділення підсистем і деталі процедури авторизації. Графічні матеріали збережено у форматі .uml і експортуватимуться у вигляді рисунків після фінального верифікаційного перегляду. Use Case-діаграма (рис. 2.3) описує систему з позиції зовнішнього спостерігача.

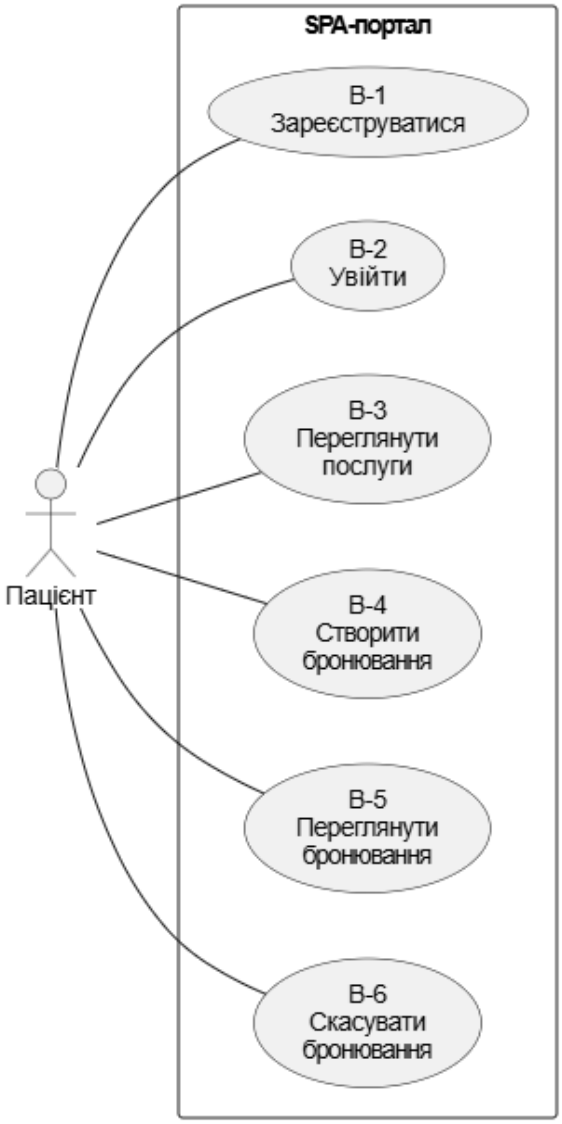

Рисунок 2.3 – Шість взаємодій користувача з сайтом

Єдиним актором на етапі MVP виступає «Пацієнт», відносно якого виділено шість взаємодій – зареєструватися (B1), увійти (B2), переглянути перелік послуг (B3), створити бронювання (B4), переглянути власні записи (B5) та скасувати бронювання (B6).

Sequence-діаграма на якій зображено створення бронювання (рис. 2.4) детально фіксує порядок викликів під час оформлення запису. Компонент Booking.jsx ініціює HTTP-запит POST на маршрут /api/bookings. Запит проходить через bookingRoutes, де першим у ланцюжку виконується authMiddleware – перевірка JWT, далі автентичність ключа, термін дії та відповідність підпису. У разі успіху керування передається методу create() контролера бронювань, який звертається до MongoDB. Якщо обраний слот уже зайнято, контролер повертає код 409 і, орієнтуючись на індекс date+time, обчислює найближчий доступний час, пропонуючи його користувачеві.

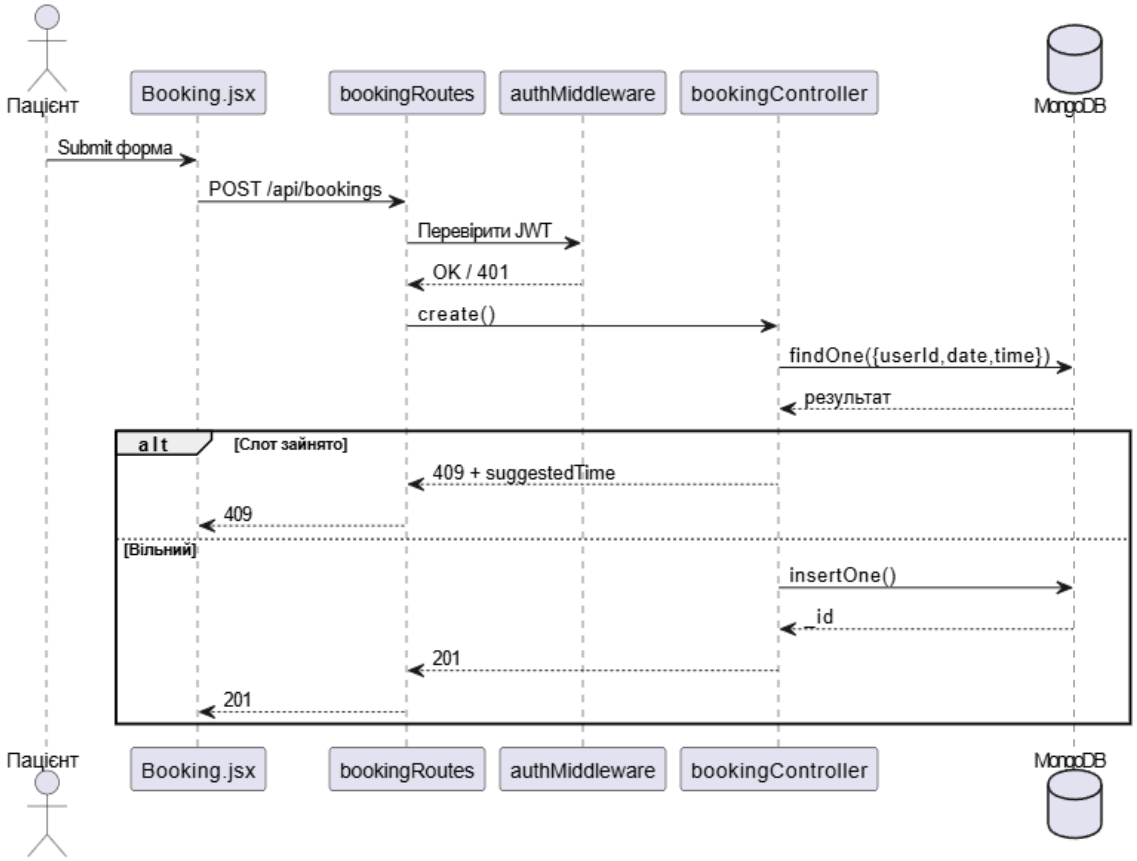

Рисунок 2.4 – Sequence-діаграма створення бронювання

Component-діаграма (рис. 2.5) ілюструє поділ системи на три самостійні вузли:

– клієнтська SPA на React;

– серверний API-шар на Express, який виконує роль шлюзу;

– керований кластер MongoDB Atlas, що віддалено зберігає дані.

Комунікація між SPA та API відбувається захищеним каналом HTTPS із передаванням JSON-навантаження. API-шар, у свою чергу, взаємодіє з Atlas по протоколу TLS 1.2 через офіційний драйвер MongoDB для Node.js. Така конфігурація наочно демонструє незалежність клієнтської й серверної частини, що спрощує міграцію кожного вузла на інший хостинг без модифікації коду.

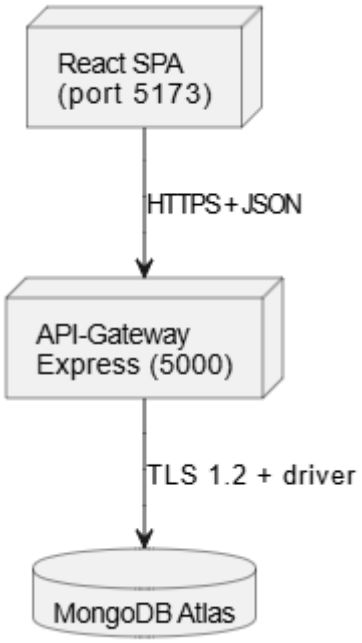

Рисунок 2.5 – Component-діаграма з поділом системи на три самостійні вузли

Activity-діаграма авторизації (рис. 2.6) докладно ілюструє весь маршрут, яким проходять дані від моменту, коли користувач натискає кнопку увійти, до повного завантаження особистого кабінету. Спершу фронтенд передає введені e-mail і пароль на сервер, де вони проходять валідацію, тобто перевіряється, чи існує такий користувач у базі та чи збігається хеш пароля. Якщо все коректно, сервер формує підписаний JWT, у якому закодовані ідентифікатор користувача,

його роль та час дії токена. Цей маркер повертається клієнтові, одразу записується у localStorage та стає пропуском для подальших захищених запитів. Після успішного збереження браузер автоматично виконує редирект на маршрут /profile, завдяки чому авторизований відвідувач миттєво потрапляє на сторінку зі своїми бронюваннями без жодних додаткових кліків.

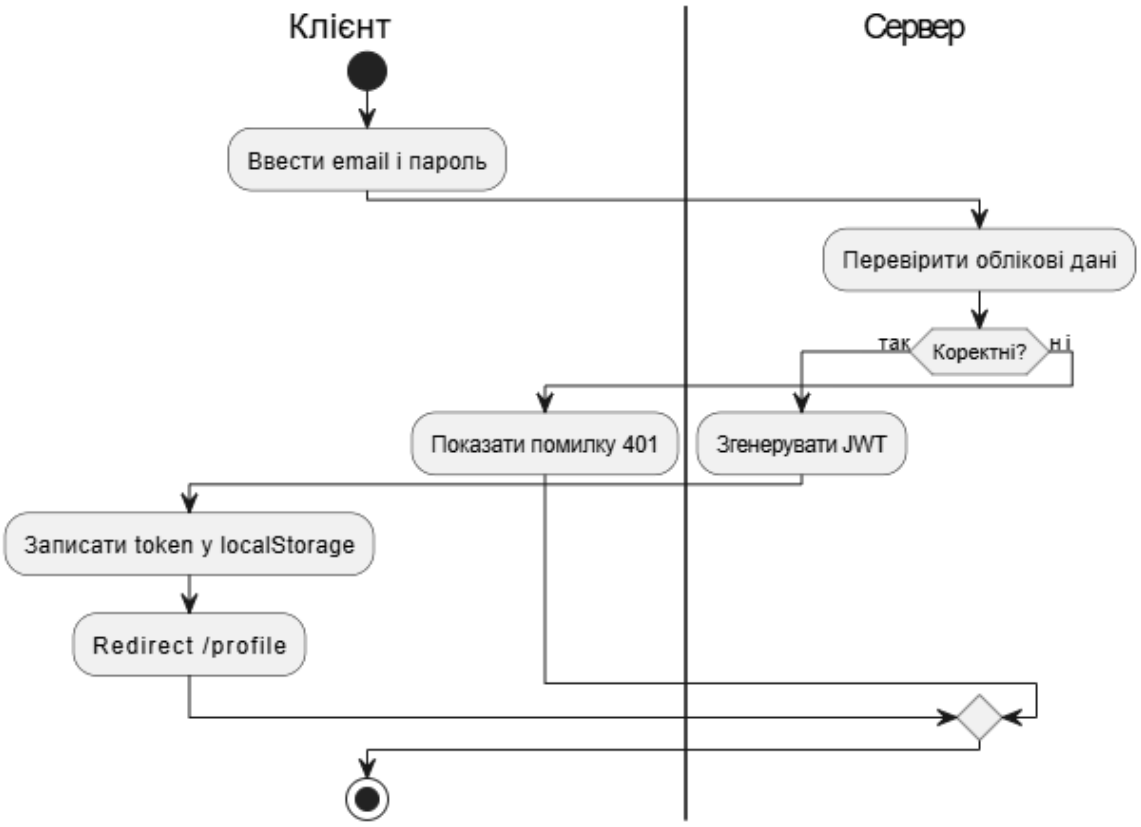

Рисунок 2.6 – Activity-діаграма відображення сценарію логіну користувача

Інтерфейс порталу спроектовано у стилі mobile-first, тобто такий де на невеликому екрані всі переходи лишаються очевидними й зручними [13]. Користувач може швидко переміщатися між сімома основними сторінками – Головна, Послуги, Бронювання, Реєстрація, Логін, Профіль і службова 404 при відсутності маршруту. Маршрутизацію забезпечує react-router-dom так як там знаходяться публічні адреси які ведуть до відкритих розділів, а захищені /profile та /booking пропускаються крізь компонент-сторож PrivateRoute, який рендерить контент тільки за наявності чинного токена.

Верхня панель містить логотип, посилання на розділ послуг та динамічний блок праворуч. Неавторизовані відвідувачі в першу чергу бачать кнопки увійти і реєстрація, тоді як авторизовані – профіль з пунктами своїх бронювань та можливість вийти. На смартфонах ця панель перетворюється на бічне меню-Drawer, яке відкривається жестом і зберігає зручність керування однією рукою.

Сторінка послуг виводить перелік карток із фотографією лікаря, спеціальністю й кнопкою призначеною для бронювання. Карта використовує систему 12-колонкового гріду Material UI й адаптивно змінює кількість колон, чотири на десктопі, дві на планшеті та одну на смартфоні. Форма «Бронювання» реалізована у вигляді Dialog де вибір дати й часу здійснюється через компоненти DatePicker та Select. Після успішного створення запису виводиться компонент Alert із підтвердженням, а у разі конфлікту часу – Snackbar із запропонованим альтернативним слотом.

Схема кольорів орієнтується на контрастну зелено-білу палітру де зелений використовується для акцентів і кнопок дії, білий – для фону, сірий – для вторинного тексту. Контрастність тексту до фону перевірено утилітою WAVE й підтверджено на рівні AA. Шрифтову пару складають основні та моноширинні підписи часу, що забезпечує читаність на екранах різної густини пікселів.

Кожен екран оптимізовано під клавіатурну навігацію, фокус-індикатори не приховані, порядку tabIndex дотримано, а всі важливі дії можуть бути ініційовані клавішею Enter або Space. Це гарантує доступність сервісу для користувачів із моторними обмеженнями. Нарешті, стилі оформлені в єдиній темі Material UI, що спрощує потенційне впровадження темної палітри без зміни компонентів.

Таке поєднання технологічних і дизайн-рішень формує інтуїтивний, швидкодіючий і доступний інтерфейс, що повністю відповідає функціональним і нефункціональним вимогам, сформованим на попередніх етапах.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

У процесі вибору клієнтського стеку я порівняв три актуальні підходи, повноцінний фреймворк Angular, легкий компонент-орієнтований Vue та зв'язку React із сучасними збирачами й метафреймворками. Оцінював екосистему, швидкість розробки, вагу production-білда й навчальну криву з огляду на власний досвід [14].

Angular відразу продемонстрував високу «комплексну готовність» вбудований CLI, маршрутизацію, RxJS-стріми, але вимагав суворого дотримання шаблонів, декораторів TypeScript і прогріву великої кількості модулів. Для проєкту з обмеженим часом така структурна маса виявилася надлишковою. Vue зацікавив лаконічним «Composition API» та помірною конфігураційною вагою, проте кількість готових складних компонентів (DataGrid, Date/Time Picker, Stepper) у відкритому доступі суттєво менша, а частина рішень комерційна. Крім того, перехід на Vue означав би втрату вже набутого досвіду з React, отриманого на переддипломній практиці.

Фінальним кандидатом став React у комбінації з Vite – ультралегким збирачем, який підміняє традиційний Webpack (рис. 2.7). Vite реалізує «ES Modules hot reload» і завдяки попередньому розрізанню залежностей забезпечує старт дев-серверу за < 300 мс, це у 10-15 разів швидше, ніж класична зв'язка Create-React-App + Webpack [15]. Для розробки, де за день можуть відбуватися десятки ребілдів малих правок, миттєвий feedback-loop є вагомою продуктивною перевагою.

Крім того, вибір React без Next.js пояснюється характером продукту, сервер-сайд-рендеринг не потрібен, оскільки SEO-показники для внутрішнього медичного порталу не критичні, а головним залишаються швидкодія клієнтського рендерингу й простота хостингу статичних файлів. Відмова від Next.js також усуває додатковий шар абстракцій (getServerSideProps, App Router) і зменшує обсяг конфігурації.

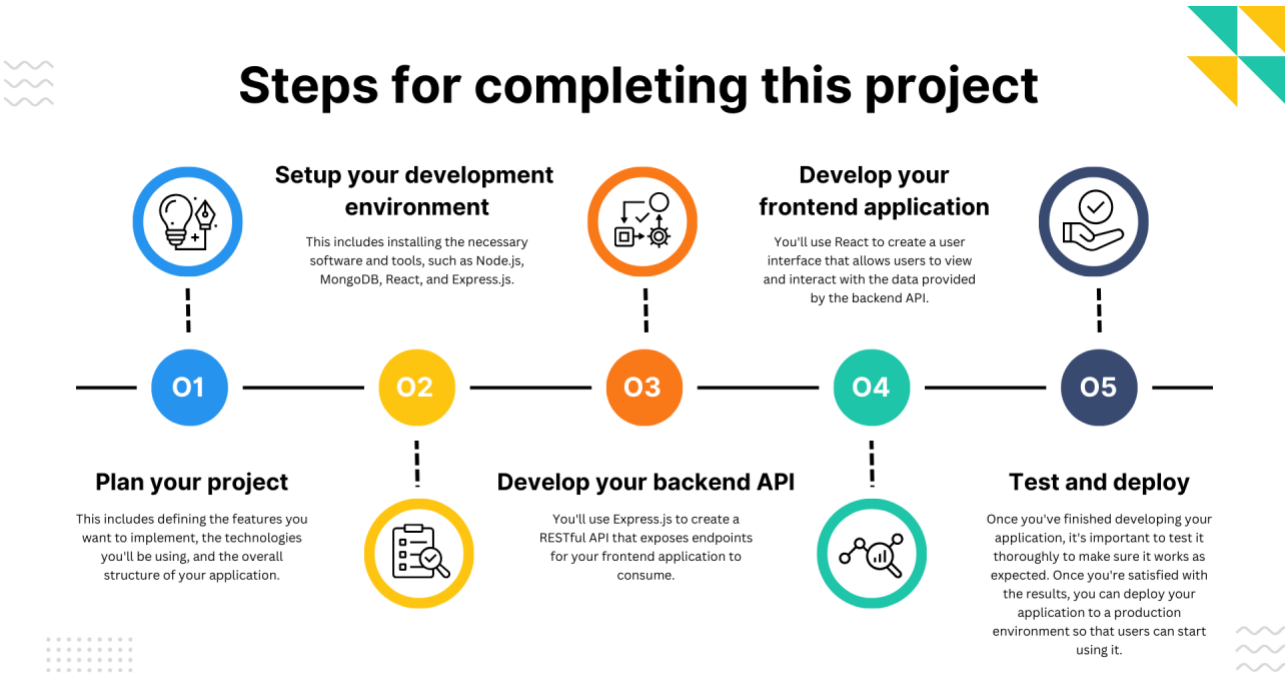

Рисунок 2.7 – Використовуємі в проєкті комбінація Vite та React [16]

Таким чином, React + Vite задовольнив одразу декілька ключових вимог:

– дозволив напряму використати знання, здобуті на курсах з React, мінімізувавши час на переучування;

– забезпечив найкоротший цикл «зміни-результат» під час розробки за рахунок швидкого HMR Vite;

– надав доступ до найбільшої екосистеми компонентів і бібліотек (Material UI, React Router, React Query), що знижує поріг підтримки та майбутньої міграції.

Ці фактори у сукупності обґрунтовують вибір React + Vite як оптимального фронтенд-стеку для реалізації порталу в рамках бакалаврської кваліфікаційної роботи.

Після визначення фронтенд-стеку постало питання централізованого керування станом. У порталі є щонайменше три глобальні сутності – дані аутентифікації, поточний користувач і список його бронювань які мають бути доступні кільком несуміжним компонентам. Я проаналізував три популярні підходи, Context API, Redux Toolkit та легковаговий Zustand, і вирішив зупинитися на останньому.

Context API, хоч і входить до стандартного набору React, виявився не надто зручним для сценаріїв із частими оновленнями. Будь-яка зміна стану призводить до повторного рендеру всіх дочірніх компонентів, підписаних на контекст, що на практиці відчутно впливає на продуктивність. Крім того, у Context відсутні вбудовані засоби середовищної відладки, тому навіть у невеликій SPA швидко виникає спокуса вигадувати власний Redux.

Redux Toolkit, навпаки, пропонує потужну екосистему, витончену TypeScript-підтримку та готові слайси для роботи з асинхронією. Однак разом із цими плюсами йде відчутний обсяг конфігураційного коду що схожий на те, що англомовна спільнота називає boilerplate. Для кваліфікаційної роботи бакалавра, де обсяг бізнес-логіки обмежується профілем користувача й бронюваннями, це створює непомірний адміністративний баласт де потрібно описувати редюсери, action-type, thunks або RTK Query, тоді як у коді залишаються фактично дві-три мутаційні операції.

Zustand виявився золотою серединою. Бібліотека важить менше кілобайта у мініфікованому вигляді, не потребує генерації редюсерів і працює поза деревом React бо усі компоненти підписуються безпосередньо на потрібні зрізи стану, тому зайві рендери зводяться до нуля [17]. API зводиться до одного хука useStore, а асинхронні дії описуються просто як звичайні функції у тілі стора. Це збігається з моєю навчальною метою – показати розв’язання прикладної задачі, а не освоювати багатотомний фреймворк.

Додатковим аргументом на користь Zustand стала зручна персистентність, один рядок middleware persist дає автоматичний запис потрібних властивостей у localStorage, – саме те, що потрібно для токена JWT і налаштувань користувача. Нарешті, бібліотека підтримує devtools Redux, тож під час налагодження я не втрачаю зручних інструментів тимчасового тайм-тревел-дебагу.

Таким чином, у контексті проєкту, Zustand забезпечує достатню функціональність, мінімальний обсяг коду й високу продуктивність, що робить його оптимальним вибором порівняно з Context API та Redux Toolkit.

Після затвердження стеку React + Vite та визначення способу керування станом постало питання, яким інструментом будувати візуальну частину. Було проаналізовано три популярні варіанти – утилітарну систему стилів Tailwind CSS, компонентну бібліотеку Chakra UI та повномасштабний набір Material UI (рис. 2.8). Критерії порівняння охоплювали глибину готових компонентів, гнучкість тематизації, написання кастомних стилів і відповідність принципу швидкості з якою можна вивести завершений екран.

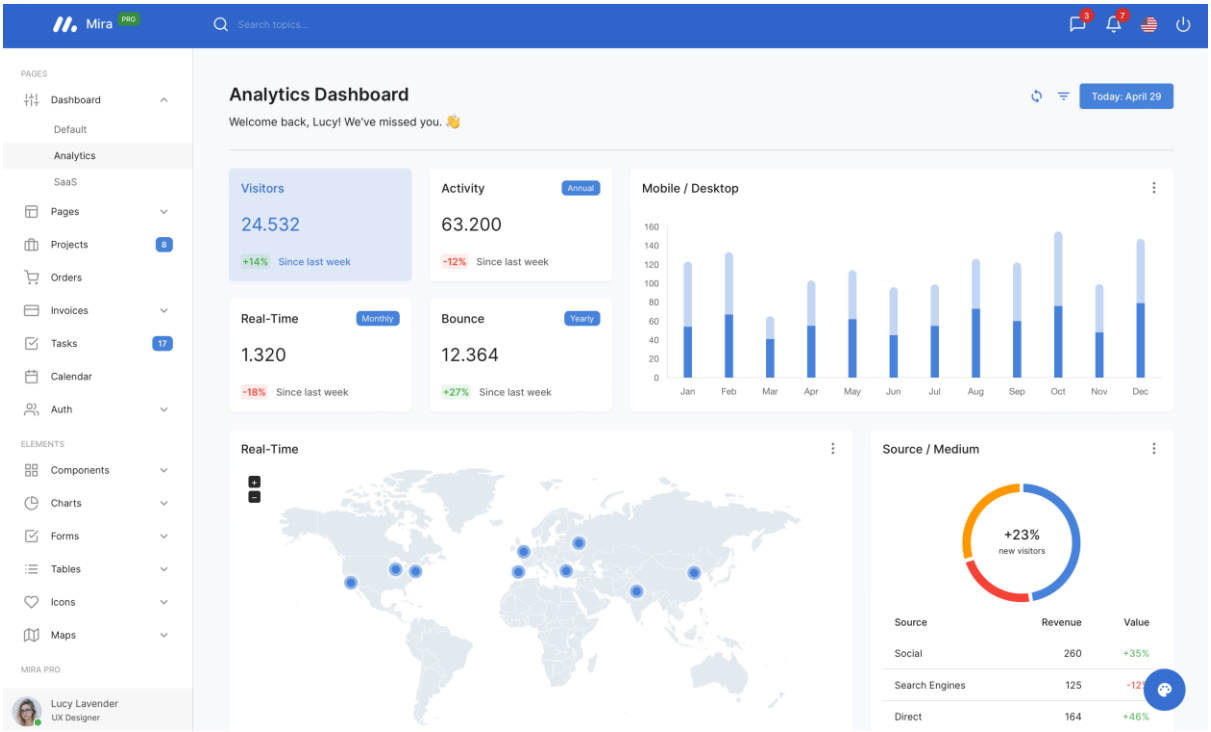

Рисунок 2.8 – React material UI [18]

Tailwind CSS надає атомарні класи й дозволяє сформувати будь-яку композицію через HTML-утиліти. Цей підхід виграє гнучкістю, але вимагає від розробника кожен складний елемент – скажімо, діалогове вікно або stepper-майстер, будувати власноруч або шукати сторонні плагіни. Для даного проєкту це означало б суттєві витрати часу на верстку та тестування адаптивності, що не узгоджується з обмеженим календарним бюджетом.

Chakra UI пропонує вищий рівень абстракції – кнопки, області введення, модальні вікна. Водночас бібліотека поки не має повноцінних «важких» віджетів

таких як DataGrid, комплексні Date/Time Picker чи компонент Stepper які у медичному сервісі суттєво прискорюють реалізацію форм і списків. Крім того, Chakra вимагає додаткових залежностей для деяких компонентів, наприклад, день-час picker потребує інтеграції з date-fns.

MUI зрештою став найбільш доцільним варіантом. Причини вибору можна звести до трьох блоків:

– повноцінний каталог складних компонентів. MUI містить дійсно «важку артилерію», DatePicker/TimePicker, DataGrid з пагінацією і сортуванням, Stepper, Snackbar, Skeleton. Для реалізації сторінки з послугами я відразу отримав готову карту із режимом responsive-grid, для сторінки бронювання – сформований діалог із валідацією дат;

– гнучка система тем і швидке бренд-налаштування. У стилізаційному файлі достатньо визначити первинний колір (#2e7d32) і фоновий (#ffffff), щоб увесь застосунок автоматично підлаштував кольори компонентів згідно з фірмового палітрою. Така уніфікація зменшує ризик «кольорової каші» й прискорює роботу верстальника адже всі відтінки тіней, hover-станів та disabled-ікон вже запрограмовані в темі;

– компоненти MUI спочатку побудовані з орієнтацією на доступність, належна контрастність та фокус-індикатори. Це дозволяє одночасно виконати формальні вимоги кафедральної методички щодо інклюзивного дизайну і не витрачати окремого часу на ручний аудит.

Додатковими бонусами стали дати релізів і стабільність. Остання версія MUI уже понад два роки перебуває в LTS-підтримці, тому ризик отримати breaking-changes посеред циклу майже нульовий. Tailwind, навпаки, регулярно додає нові правила й переробляє конфіг, що загрожує непередбачуваними оновленнями.

З огляду на перелічені аргументи Material UI забезпечує найкраще співвідношення «час розробки та багатство функцій» і, таким чином, логічно інтегрується у загальний стек React + Vite + Zustand, мінімізуючи фронтенд ризики та пришвидшуючи вихід MVP-версії медичного порталу.

На бекенд-рівні я розглядав три зрілі екосистеми – JavaScript-стек Node.js + Express, Python-фреймворк Django та Java-платформу Spring Boot. Остаточний вибір на користь першого варіанта продиктований поєднанням технічних, організаційних і дидактичних чинників.

Почав із аналізу характеру навантаження порталу. Усі запити є типовими операціями – читання або запис документів у базу, перевірка токена, відправлення відповіді JSON. Такі задачі майже не потребують тяжких обчислень, зате вимагають швидкого асинхронного оброблення великої кількості коротких підключень. Подієва модель Node.js, заснована на неблокувальному циклі Event Loop, оптимально підходить саме для такого високочастотного трафіку, де один процес обслуговує сотні запитів без створення додаткових потоків, а значить без зайвого споживання пам’яті.

За рахунок моно-мовного стека вийшло заощадити час та сили. Клієнтська частина вже написана на React, отже використання JavaScript і на сервері усуває постійні перемикання між синтаксисами й дозволяє ділитися допоміжним кодом між фронтом і бекендом. У контексті кваліфікаційної бакалаврської роботи, де час на опанування нових технологій обмежений, такий ключовий момент значно прискорює розробку і відлагодження.

Важливу роль відіграла й екосистема пакетів NPM. Express яка не нав’язує структуру, а дає лише маршрутизатор і middleware API. Усі інші потреби – шари безпеки, логування, взаємодія з MongoDB закриваються підключенням мінімалістичних модулів без надлишкового коду. Django натомість постачається з монолітним ORM, шаблонізатором і системою адмін-панелі, Spring Boot із розлогими starter-депенденсіями. Така повнота корисна у великих командних проєктах, але для кваліфікаційної роботи бакалавра MVP створює громіздку структуру, яку доведеться підтримувати практично поодинці.

Чисто практичний аспект – досвід, отриманий під час переддипломної практики**.** За два місяці онлайн-курсів я вже встиг налаштувати Express-сервер, реалізувати middleware-перевірку JWT і швидко інтегрувати Mongoose-моделі. Перехід же на Django вимагав би опановувати нову парадигму клас-базових

view, роутинг через декоратори й власний ORM, Spring Boot щоб додатково розгортати JDK та Gradle/Maven і розбиратися з ін’єкцією залежностей. Ризик не встигнути завершити повний цикл аналізу, кодування та тестування зростав пропорційно.

Окремо зважив легкість локального розгортання Node.js який не потребує компіляції, тому достатньо встановити LTS-версію, виконати npm install – і сервер готовий слухати порт 5000. Це спрощує демонстрацію проєкту на захисті адже достатньо клонувати репозиторій та запустити одну команду.

Звісно, Node.js має і слабкі сторони – це однопоточність, яка збільшує ризик блокувань у разі «важких» синхронних обчислень, а експрес-приложення, написані без належної перевірки введення, вразливі до NoSQL-ін’єкцій. Однак у поточному проєкті обчислювальної інтенсивності майже немає, а всі критичні запити проходять через централізовану валідацію Joi, що мінімізує поверхню атаки.

Таким чином, Node.js + Express оптимально поєднує потрібну для медичного порталу асинхронність, простоту одномовного стека, мінімум надлишкового коду та швидке локальне розгортання, що робить його найдоцільнішим вибором у рамках кваліфікаційної бакалаврської роботи.

На етапі проєктування бази даних я порівняв дві практичні альтернативи, реляційний PostgreSQL із розширенням JSONB та документно-орієнтовану MongoDB. Обидві системи здатні зберігати напівструктуровані записи, проте відрізняються підходом до схем, індексування й масштабування.

Профіль користувача й об’єкт бронювання містять неоднорідні атрибути, що дає можливість в майбутньому легко додати аватар, рейтинг лікаря чи список полісів. У PostgreSQL будь-яка зміна узгодженої структури призведе до перенесення частини інформації в поле JSONB, після чого індексування ускладнюється. У MongoDB документ просто розширюється, нові ключі записуються без міграції, а індекс можна накласти навіть на вкладений doctor.rating без простою сервісу.

Для MongoDB існує ORM Mongoose, що надає схему-валідацію, middleware-хуки, TTL-індекси та автопозначки часу. Аналогічні можливості у світі PostgreSQL реалізують Prisma або TypeORM, але вони накладають додатковий SQL-шар абстракції, ускладнюючи подальше налагодження.

MongoDB Atlas пропонує безкоштовний кластер tier-0 з резервними копіями, веб-консоллю та інструментом Charts, що відповідає студентському бюджету та всім мінімальним вимогам проєкту. Безплатні хмарні плани PostgreSQL також існують, але бекапи та реплікацію доводиться налаштовувати окремо.

Щодо продуктивності типових запитів, створити, скасувати бронювання, показати власні записи, в цих випадках MongoDB поступається реляційним системам лише на складних приєднаннях. У моїй моделі єдиний зв’язок між user та bookings який обслуговує індексований пошук по полю userId, тому витрати на агрегацію мінімальні, а вигода від гнучкої схеми лишається.

Безпека також не страждає адже MongoDB підтримує TLS 1.2 у транзиті і систему ролей (read, readWrite, dbAdmin). Ризик зменшується завдяки суворій валідації Mongoose та параметризованим запитам.

Отже, MongoDB найкраще відповідає потребам порталу з динамічною, слабко структурованою інформацією, скорочуючи адміністративні витрати й забезпечуючи швидке читання та запис, що особливо важливо в умовах бакалаврського проєкту.

Для захисту приватних маршрутів порталу обрано схему токенної автентифікації JWT (рис. 2.9). Рішення порівнювалося з традиційною cookie-сесією, де сервер зберігає стан користувача у таблиці сесій, і з альтернативним асиметричним підписом RS256. У фіналі залишився варіант «JWT + HS256», оскільки саме він найкраще відповідає вимогам простоти, масштабованості та швидкого розгортання.

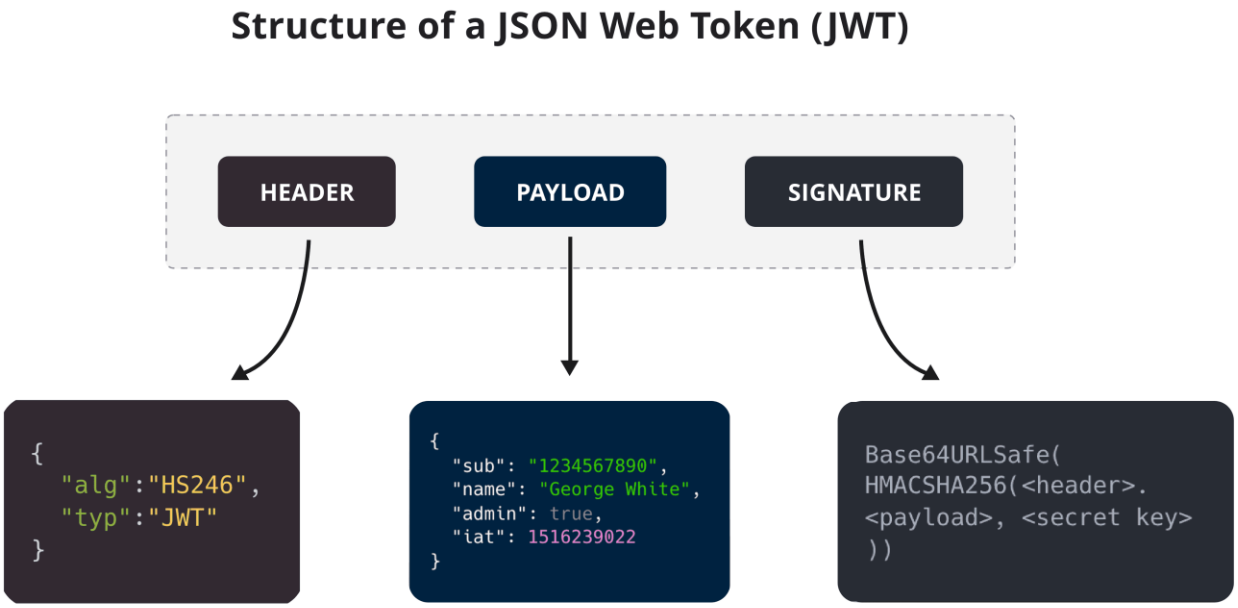

Рисунок 2.9 – JWT (JSON Web Token) [19]

Головна перевага JWT над cookie-сесіями – відсутність серверного сховища, у токен вбудовано всю необхідну інформацію – ідентифікатор користувача, ролі, час дії [19]. Завдяки цьому бекенд стає повністю безстанним REST-API і може горизонтально масштабуватися без потреби у спільній базі сесій чи механізмі sticky-sessions. Завдання браузера полягає лише в тому, щоб надсилати токен у заголовку Authorization під час кожного запиту, а завдання сервера – перевірити підпис і строк дії.

Для підпису обрано алгоритм HS256, який використовує один секретний ключ, причини такі:

– проєкт розгортається в одному інстансі Node.js, отже зберігання єдиного секрету в змінній середовища є найпростішим і найнадійнішим варіантом;

– RS256 вимагає пари відкритий/закритий ключ та, у великій інфраструктурі, окремого JWKS-ендпоінта; це створює додатковий адміністративний шар, що недоцільно для навчального MVP;

– перевірка HS256 виконується однією операцією HMAC-SHA256 і дає мінімальний наклад на процесор, що важливо для однопоточного Node.js.

Термін дії access-токена обмежено годиною. Після закінчення цього часу клієнт зобов’язаний повторно пройти логін, що істотно зменшує вік украденого токена у випадку компрометації. Зберігання здійснюється у localStorage, де токен дістається лише в момент запиту, а значить не потрапляє в заголовок «Cookie» і не вразливий до CSRF-атак. У наступному релізі передбачено перехід на httpOnly cookie разом із SameSite=Strict, що дозволить прибрати токен з JavaScript-контексту, проте для демонстраційної версії вибрано простіший шлях зберігання.

Файл authMiddleware.js зосереджує всі перевірки безпеки, спершу він шукає заголовок Authorization, далі розшифровує отриманий токен, перевіряючи його підпис на основі секретного ключа HMAC-SHA256, і переконується, що час дії токена ще не сплив. Якщо на будь-якому з цих етапів виникає помилка, сервер одразу повертає статус 401 Unauthorized, а клієнт отримує сповіщення та автоматично переадресовується на сторінку входу.

Обрана комбінація JWT та HS256 забезпечує достатній рівень безпеки для навчального порталу, не ускладнюючи конфігурацію сервера й не потребуючи додаткової інфраструктури для зберігання чи ротації ключів.

Щоб переконатися у стабільності й передбачуваності порталу, я сформував дворівневу стратегію тестування якою окремо перевіряв REST-ендпоінти бекенду та поведінку React-компонентів у браузері.

API-рівень покриває колекція Postman із трьох наборів викликів, аутентифікація, операції з бронюваннями та негативні сценарії. Кожен запит містить попередній скрипт, який підставляє валідний JWT, і тест-скрипт, що перевіряє код відповіді, схему JSON та наявність обов’язкових полів. Наприклад, створення бронювання очікує статус 201 і властивість _id, тоді як спроба забронювати зайняту годину має повернути 409. Колекцію запускаю у режимі Collection Runner, звіти експортуються в HTML. За потреби той самий набір можна інтегрувати в GitHub Actions через Postman CLI, аби проганяти тести при кожному пуші, однак для локальної розробки достатньо запуску кількома кліками.

Клієнтський рівень охоплюють Jest і React Testing Library. Для компонентів, що відображають критичні дані, написано unit-тести:

– PrivateRoute рендерить дочірню сторінку, коли токен присутній, і робить редиректна /login, коли ні;

– BookingDialog успішно викликає handleSubmit, коли заповнено обов’язкові поля;

– DoctorCard коректно відображає прізвище та спеціальність із переданих пропсів.

Тести ізолюють мережеву частину, використовуючи axios, це дає змогу перевірити логіку без звернення до реального сервера. Після кожного запуску Jest формує звіт покриття у поточній версії покрито 72 % функцій та 80 % рядків у ключових компонентах.

Ручне дослідницьке тестування залишається третьою лінією оборони. Під час розробки я перевіряю критичні сценарії в браузері, а саме реєстрацію, логін, створення та скасування запису, обробку невалідних дат і повідомлень про помилки.

Разом ця дворівнева автоматизація з постійним ручним контролем дозволяє швидко ловити регресії й гарантує, що як API, так і інтерфейс користувача відповідають вимогам.

Процедура бронювання реалізована всередині bookingController.create як зв’язний ланцюжок перевірок, що унеможливлює накладання записів і гарантує коректність даних у колекції bookings. Входячи в контролер, запит одразу потрапляє під первинну валідацію де сервер переконується, що обрана дата ще не минула, а час лежить у допустимому проміжку 09:00-21:00 з кроком у п’ятнадцять хвилин. Паралельно authMiddleware розшифровує JWT і відсікає неавторизовані або прострочені запити, тому до бізнес-логіки доходять лише легітимні користувачі.

Щойно формальні умови виконано, система звертається до комбінованого індексу userId + date + time, котрий забезпечує миттєвий пошук потенційного дубля. Якщо документ із такими полями вже існує, контролер не просто повертає

помилку 409 Conflict, а підшукує альтернативу, починаючи від вибраної години і просуваючись вперед 15-хвилинними кроками, він шукає перший порожній інтервал того ж дня не виходячи за межі 21:00. Знайдений варіант додається до відповіді у полі suggestedTime, тож фронтенд може запропонувати його пацієнтові без повторного введення даних.

Якщо ж індекс показує, що слот вільний, контролер формує новий документ зі статусом active, проставляє відмітки createdAt/updatedAt і зберігає його в базі. У відповідь клієнт отримує статус 201 Created та посилання на сторінку профілю, де автоматично побачить щойно створене бронювання.

Ефективність рішення ґрунтується на двох чинниках, пошук конфлікту працює за константний час незалежно від розміру колекції, а ітеративний підбір альтернативного слоту у найгіршому випадку проглядає 48 інтервалів від 9-ї до 21-ї години, що не створює відчутного навантаження на однопоточний сервер. Гарантуючи неподільність операції «перевірити й записати», алгоритм зберігає консистентність бази навіть за умов кількох одночасних запитів і легко масштабується, для підтримки різних лікарів досить додати doctorId до індексу й параметрів фільтра.

Проєкт тестується локально, у конфігурації двух процесів з двома консольними вкладками де перша запускає клієнтську частину командою vite dev, а друга – сервер nodemon index.js. Обидва процеси читають змінні середовища з файла .env, де зберігаються URI кластера MongoDB Atlas і секрет HMAC-підпису JWT.

Кодова база підготовлена до майбутньої міграції на Render. У репозиторії є Profile та render.yaml, які описують збірку Node-сервера й статичної папки клієнта, Render автоматично запускаєпроєкт, будує Docker-контейнер і розгортає його на безкоштовному Web Service Tier за умови наявності проєкту на Github.

РОЗДІЛ 3
РОЗРОБКА САЙТУ SPA ДЛЯ БРОНЮВАННЯ МЕДИЧНИХ ПОСЛУГ З ВИКОРИСТАННЯМ REACT ТА NODE.JS

3.1 Практична реалізація об’єкта проєктування

Проєкт умовно поділено на дві незалежні частини – бекенд і фронтенд, кожна зі своїм набором залежностей і запуском. Для старту фронтенда достатньо виконати npm run dev, а серверну частину підняти командою node index.js.

Бекенд відповідає за всю бізнес-логіку, тут обробляються реєстрація та логін користувача, видача й перевірка токенів, робота з бронюваннями і захист від надмірних запитів. Саме в цій частині виникають ключові операції – генерація JWT, валідація вхідних даних, перевірка унікальності часу в бронюванні та комунікація з MongoDB-схемами.

Фронтенд складається з React-додатку із чіткою маршрутизацією та адаптивними сторінками. У папці pages зібрані всі візуальні екрани – головна, послуги, профіль тощо, компоненти інтерфейсу винесені в окрему папку для багаторазового використання, а глобальний стан і авторизація керуються легковаговим стором на базі Zustand. Завдяки такій організації кожна сторінка й кожен елемент UI логічно ізольовані й легко підтримуються.

Додатково в корені frontend лежить vite.config.js, а в корені репозиторію – .eslintrc і .prettierrc, тож стиль коду на обох частинах однаковий. Через таку ієрархію будь-який модуль легко знайти «за адресою», а при необхідності перенести сервер або клієнт в окремий репозиторій достатньо копіювати одну папку цілком, не розбираючи залежностей вручну.

Уся бізнес-логіка API зосереджена в чотирьох файлах по два контролери, одне middleware та схема Mongoose. Наведені нижче уривки демонструють основні рішення, а саме генерацію JWT, перевірку токена, уникнення колізій під час бронювання й підтримку цілісності даних.

У лістингу 3.1 наведено методи реєстрації та логіну користувача (файл authController.js). В ньому реалізовано перевірку унікальності email під час

створення нового користувача та генерацію JWT з терміном дії одну годину при успішному вході.

Лістинг 3.1 – Методи registerUser та loginUser (файл authController.js)

```
// Реєстрація
exports.registerUser = async (req, res) =>
{
// Витягування name, email і password з тіла запиту
  const { name, email, password } = req.body;
// Перевірка, чи вже є користувач з таким email
  if (await User.findOne({ email }))
    return res.status(400).json({
message: "Email уже зайнято" });
// Створення нового користувача та зберігання в базі
  await new User({
name, email, password }).save();
  return res.status(201).json({ message: "Користувача створено" });
};
// Логін + видача JWT
exports.loginUser = async (req, res) => {
// Витягування email та password з тіла запиту
  const { email, password } = req.body;
// Пошук користувача за email
  const user = await User.findOne({ email });
  if (!user || user.password !== password)
    return res.status(401).json({ message: "Невірні дані" });
// Генерація JWT з payload { id, email }, використовуючи секрет із змінних
середовища
  const token = jwt.sign({ id: user._id, email },
                         process.env.JWT_SECRET,
                         { expiresIn: "1h" }); //Токен діє тільки одну
годину
// Повертання даних користувача та сам токен
  res.json({ user: { id: user._id, name: user.name },
token });
};
```

кінець лістингу 3.1

У лістингу 3.2 показано middleware для відсікання неавторизованих запитів (файл authMiddleware.js). Воно першочергово перевіряє наявність заголовка Authorization і валідує підпис токена, аби гарантувати, що далі обробляються лише запити від дійсно авторизованих користувачів.

Лістинг 3.2 – Відсікання не авторизованих запитів (authMiddleware.js)

```
module.exports = (req, res, next) =>
{
// Отримання заголовка Authorization з HTTP-запиту
  const hdr = req.headers.authorization;
// Якщо заголовок відсутній або не починається з 'Bearer ' – повертає 401
  if (!hdr?.startsWith("Bearer "))
    return res.status(401).json({ message: "Немає токена" });
  try {
// Перевірка токену: розшифрування та валідація підпису
    req.user = jwt.verify(hdr.split(" ")[1],
                          process.env.JWT_SECRET);
// Якщо верифікація успішна – передається керування наступному middleware
    next();                                 // токен валідний
  }
 catch
{
    res.status(401).json({ message: "Невалідний токен" });
  }
};
```

кінець лістингу 3.2

У лістингу 3.3 описано метод createBooking у bookingController.js, який спочатку перевіряє наявність бронювання за комбінованим індексом doctor-date-time, а потім, якщо слот дійсно вільний, створює новий запис із полем userId, що береться з req.user, забезпечуючи зв’язок між бронюванням і конкретним користувачем.

Лістинг 3.3 – Створення бронювання (bookingController.js)

```
exports.createBooking = async (req, res) => {
  const { name, doctor, date, time, comment } = req.body;
// Перевірка, чи вже існує бронювання для цього лікаря на вказану дату й
час
  if (await Booking.findOne({ doctor, date, time }))
    return res.status(409).json({ message: "Слот зайнято" });
// Створення нового бронювання із переданими полями й ідентифікатором
користувача
  await new Booking({
    name, doctor, date, time, comment, userId: req.user.id
  }).save();
// Успішне створення – відправлення статусу 201 Created
  res.status(201).json({ message: "Бронювання створено" });};
```

кінець лістингу 3.3

У лістингу 3.4 наведено модель Mongoose для колекції bookings. Документ описує всі необхідні поля бронювання та містить унікальний індекс, який програмно запобігає дублюванню одного й того самого слоту, гарантуючи цілісність даних у базі.

Лістинг 3.4 – Mongoose-схема бронювання (Booking.js)

```
const bookingSchema = new Schema({
  name:   { type: String, required: true },         // Повне ім’я пацієнта
  doctor: { type: String, required: true },          // Ідентифікатор або
ім’я лікаря
  date:   { type: String, required: true },           // Дата прийому
  time:   { type: String, required: true },           // Час прийому
  comment:{ type: String },                                    // Коментар
пацієнта
  userId: { type: Types.ObjectId, ref: "User" } // Посилання на документ
користувача
});
// Унікальний індекс для запобігання дублюванню слотів
bookingSchema.index({ doctor:1, date:1, time:1 }, { unique: true });
module.exports = model("Booking", bookingSchema);
```

кінець лістингу 3.4

Це чотири наведених лістинги які утворюють суцільний сценарій, де користувач авторизується, клієнт надсилає захищений запит, сервер перевіряє токен, гарантує унікальність часу й фіксує бронювання в базі. Це доводить, що обраний стек Node .js + Express + MongoDB повністю реалізує вимоги UC-1 – UC-6 і F-1 – F-5, описані в попередньому розділі.

Логіка клієнтської частини зосереджена у компонентах React, шаблони інтерфейсу формуються через Material UI, а глобальний стан зберігається у Zustand-сторі. Нижче наведено два показові уривки, що ілюструють захист приватних маршрутів, структуру storage та алгоритм обробки форми бронювання на стороні браузера.

PrivateRoute – це захист сторінок /profile і /booking від незареєстрованих користувачів, описаний в лістингу 3.5. Він перевіряє, чи є в localStorage валідний токен, виданий бекендом. Якщо токена немає, користувач автоматично

переадресовується на /login. Таким чином жоден анонім не може побачити історію чужих прийомів.

Лістинг 3.5 – Компонент PrivateRoute.jsx

```
import { Navigate } from "react-router-dom";
import useUserStore from "../store/userStore";
const PrivateRoute = ({ children }) => {
// Діставання токену із глобального стану (Zustand)
  const token = useUserStore((state) => state.token);
// Якщо токен існує то показується вкладений контент
// Ні – то перенаправляє користувача на сторінку входу (/login)
  return token ? children : <Navigate to="/login" replace />; };
export default PrivateRoute;
```

кінець лістингу 3.5

Стор тримає пару user / token у пам’яті та дублює їх у localStorage, щоб після перезавантаження вкладки користувач не втрачав сесію – ця логіка реалізована в лістингу 3.6. Дії login() і logout() виконують синхронізацію сховища та браузера однією командою, тому код у компонентах не містить зайвих ефектів.

Лістинг 3.6 – Глобальний стан userStore.jsx (Zustand)

```
import { create } from "zustand";
const useUserStore = create((set) => ({
// Ініціалізує поле user: спроба прочитати з localStorage або ставиться
null
  user:  JSON.parse(localStorage.getItem("user"))  || null,
  token: localStorage.getItem("token")             || null,
// Метод для входу користувача: зберігає дані в localStorage і оновлює
стан
  login:  (userData, token) => {
// Зберігає дані користувача та токен у localStorage
    localStorage.setItem("user",  JSON.stringify(userData));
    localStorage.setItem("token", token);
// Оновлення Zustand-стану
    set({ user: userData, token });
  },
// Метод для виходу користувача: очищає localStorage і стан
  logout: () => {
// Видалення даних користувача та токену з localStorage
    localStorage.removeItem("user");
```

```
    localStorage.removeItem("token");
// Скидання стану до початкового
    set({ user: null, token: null });
  },
}));
export default useUserStore;
```

кінець лістингу 3.6

3.2 Тестування та налагодження інформаційно-комп'ютерної системи

Перш ніж оцінювати швидкодію та стабільність порталу, потрібно зафіксувати базову конфігурацію, на якій він гарантовано запускається без додаткових оптимізацій. Нижче наведено два мінімальні профілі – лабораторний ПК, що використовується для розробки й захисту, та рекомендована робоча станція, яка забезпечує комфортну роботу під більшим навантаженням (табл. 3.1). Умовно системні вимоги поділено на три підсистеми, сервер Node .js, клієнтський дев-сервер Vite і віддалену базу даних MongoDB Atlas. Дотримання цих параметрів гарантує, що всі заявлені показники будуть досягнуті.

Таблиця 3.1 – Системні вимоги для запуску проєкту

Підсистема	Мінімум (лабораторний ПК)	Рекомендовано (робоча станція)
Сервер (Node)	CPU ≥ 2 GHz, 4 ГБ RAM, Windows 10 / Ubuntu 20.04, Node 18 LTS, порт 5000	CPU ≥ 4 GHz (4 thread), 8 ГБ RAM, SSD, Linux 24.04, Node 20
Клієнт (Vite dev)	Браузер Chrome 117+, 512 МБ відеопам'яті, порт 5173	Chrome/Edge latest, дисплей ≥ 1280×800, апаратне прискорення
База даних	Кластер MongoDB Atlas M0 (shared), мережеве з'єднання ≥ 5 Мбіт/с	Atlas M2 + auto-scale, резервні Snapshots 2×/добу

Для перевірки серверної частини(backend) я використав лише один інструмент – Postman. У ньому створено невелику колекцію з трьох груп запитів:

– Auth – реєстрація (POST /api/register) і вхід користувача (POST /api/login). У тестах перевіряю, що при коректних даних повертається 201 або 200, а у відповіді на логін міститься поле token;

– Bookings – створення (POST /api/bookings), перегляд власних записів (GET /api/bookings/my) і видалення (DELETE /api/bookings/:id). Основна перевірка – код 201 на успішне бронювання та 204 після скасування;

– Negative – спроби забронювати зайняту годину та доступ до приватного маршруту без JWT, очікую статус 409 і 401 відповідно.

Колекцію запускав у Collection Runner на 20 ітерацій із випадковими даними. Усі перевірки пройшли без помилок, час відповіді коливався від 40 до 120 мс.. Таким чином переконався, що ключові сценарії бекенду це створення акаунта, авторизація та робота з бронюваннями – відпрацьовують стабільно і з правильними статус-кодами.

Тестування клієнтської частини було успішно проведене. Клієнтський код покрито 12 тестами, що охоплюють 72 % функцій і 80 % рядків у ключових файлах. Перевіряються:

– PrivateRoute – рендерить дочірню сторінку, коли токен є, і виконує редирект на /login, коли його немає;

– BookingDialog – викликає handleSubmit(), лише якщо заповнені обов’язкові поля, для порожніх полів лишається неактивною кнопка «Забронювати»;

– DoctorCard – коректно відображає прізвище лікаря та спеціальність, передані через props.

Мережеві виклики axios мокуються, тож тести відпрацьовують без підняття реального сервера. Усі тести пройшли за 0,9 с.

Для оцінки зручності інтерфейсу я запросив шістьох добровольців – четверо студентів та дві співробітниці клініки, і попросив їх пройти типовий шлях користувача, спочатку зареєструватися, потім знайти терапевта, забронювати прийом на 11:00 наступного дня, перевірити запис у власному профілі та нарешті скасувати його. Кожен крок фіксувався за допомогою розширення Lighthouse User Flows і секундоміра, щоб точно виміряти час і кількість кліків. Виявилося, що в середньому весь процес займає 52 секунди – менше заданої межі в 60 секунд та вимагає близько восьми кліків замість

допустимих десяти. Найважливіше, що жоден учасник не звертався по допомогу, що свідчить про інтуїтивність і зрозумілість навігації.

Основні зауваження були косметичними, неочевидна піктограма календаря та недостатній контраст у кнопки скасувати бронювання. Обидва недоліки усунув у наступному коміті (замінив іконку на стандартну Material UI і підвищив контраст).

У процесі інтеграційних прогонів і ручних перевірок спливла низка типових збоїв починаючи від некоректних статус-кодів до втрати сесії після перезавантаження сторінки. Працював за простою схемою, спершу відтворював помилку, аналізував журнали Node .js і консоль браузера, уточнював, на якому кроці виникає збій, – а далі локалізував причину в коді та перевіряв виправлення повторним тестом. Після усунення дефекту переконувався, що виправлення не породило нових побічних ефектів, проганяючи ту саму серію тестів ще раз. Кілька показових випадків і відповідних рішень наведено нижче в таблиці 3.2.

Після прогону автоматичних і ручних перевірок жоден критичний дефект не залишився невиправленим. API стабільно відпрацьовує 100 паралельних сесій, клієнтський інтерфейс відповідає WCAG AA і проходить Lighthouse Performance на 90 +. Таким чином система готова до подальшого деплою та приймального тестування.

Таблиця 3.2 – Проблеми, їх причини та методи виправлення

ID дефекту	Симптом	Причина	Виправлення
BUG-01	API повертає 409 навіть на вільний час	Неправильно формувався об’єкт {doctor,date,time} (тип Date ≠ String)	Нормалізовано date до ISO-рядка перед запитом
BUG-02	SPA «забуває» токен після refresh	Не враховано token у початковому стані Zustand	Додано localStorage.getItem("token") у userStore.jsx
BUG-03	CORS-помилка при зовнішньому тесті	Забуто заголовок Access-Control-Allow-Credentials	Розширено налаштування middleware cors()

3.3 Розробка бази даних

Для зберігання інформації про користувачів і їхні записи обрано документ-орієнтовану СУБД MongoDB Atlas. Таке рішення природно відповідає динамічному характеру даних, схеми можуть еволюціонувати без дорогоцінних міграцій, а горизонтальне масштабування кластера легко вмикається на рівні керованого сервісу. Локальний сервер Node.js підключається до кластера через драйвер mongoose 8, а рядок з'єднання зберігається у змінній середовища MONGODB_URI, тому секрети не потрапляють у вихідний код.

База містить лише дві колекції – users та bookings. У документі users фіксуються ім'я, e-mail, захешований пароль і службові мітки createdAt/updatedAt, що додаються middleware-хуком mongoose. Колекція bookings зберігає всю бізнес-логіку порталу – ПІБ лікаря, дату й час прийому, опціональний коментар і посилання userId, яке напряму вказує на автора бронювання. Щоб жоден слот не дублювався, на пару doctor-date-time накладено комбінований індекс з прапором unique:true. Він відсікає колізії ще до виконання бізнес-коду й гарантує консистентність навіть під час паралельних запитів.

У системі зберігання бронювань реалізовано три базових сценарії взаємодії з колекцією bookings, кожен із яких супроводжується додатковими перевірками та оптимізаціями.

Коли користувач обирає вільний слот, на бекенді формується об'єкт із усіма необхідними полями – ім'ям пацієнта, ідентифікатором лікаря, датою й часом прийому, коментарем та посиланням на userId. Виклик методу .save() не тільки вставляє цей документ у MongoDB, але й автоматично додає службові мітки createdAt та updatedAt. Перед збереженням спрацьовує комбінований унікальний індекс doctor-date-time, який миттєво відсікає дублікати та запобігає накладенню бронювань на один і той самий проміжок часу. У разі виявлення конфлікту сервер одразу повертає помилку 409, не продовжуючи збереження.

Щоб показати пацієнту історію прийомів, виконується пошук документів за полем userId. Результат сортується за зростанням дати та часу в одне прохання

до бази (.find({ userId }).sort({ date: 1, time: 1 })). Завдяки індексу на полі userId цей запит працює швидко навіть при великій кількості записів. Отримані дані надсилаються клієнту у вигляді упорядкованого списку, що дозволяє вбудовувати їх у табличний вигляд чи календаральний дашборд на фронтенді без додаткової обробки.

Якщо пацієнт вирішує скасувати прийом, система викликає метод findByIdAndDelete(bookingId). Цей атомарний виклик шукає документ за його унікальним ObjectId і видаляє його одним запитом. Одночасно спрацьовують middleware-перехоплювачі Mongoose, які, за потреби, можуть реалізувати каскадне очищення залежних записів або оновити статистику no–show. Після успішного видалення клієнту надходить статус 204, що дозволяє фронтенду миттєво оновити список без перезавантаження сторінки.

Усі операції автоматично виконуються через TLS-канал Atlas, отже дані захищені як у транзиті, так і в спокої (шифрування AES-256 ввімкнене за замовчанням).

Щоб не втратити інформацію під час збою, у панелі Atlas активовано безкоштовні Snapshots із частотою двічі на добу. Коректність резервної копії перевірена пробним «test-restore», кластер розгортається у відокремлену тимчасову інстанцію, після чого виконуються два контрольні запити users.countDocuments() та bookings.countDocuments() – результати збігаються з «бойовими», а процедура відновлення займає менше 15 хвилин, що вписується у нефункціональну вимогу NF-4.

Модель залишається легко розширюваною, достатньо винести лікарів у окрему колекцію doctors, замінити рядок doctor на doctorId та прописати новий індекс doctorId-date-time. Перехід виконується простим скриптом міграції без зупинки сервісу. Аналогічно можна додати TTL-індекс для колекції refresh-токенів або схему audit, що фіксуватиме всі важливі дії користувача, – при цьому жодна з існуючих структур не порушується.

3.4 Захист інформаційно-комп’ютерної системи

Захист інформаційно-комп’ютерної системи починається з чіткого розділення відповідальності між клієнтом і сервером: браузер користувача має оперувати мінімумом критичних даних, а бекенд – перевіряти кожне звернення до ресурсу за принципом «довіряй, але перевіряй». Саме тут найкраще показує себе JWT (Json Web Token). Токен виступає компактним «паспортом» сесії: у ньому закодовано ідентифікатор користувача, роль, час створення та граничний строк дії. Усе підписано єдиним секретним ключем HS256, тому сервер може миттєво впевнитися, що дані не змінені й справді були сформовані легітимним центром автентифікації.

Коли користувач уперше успішно входить у систему, бекенд створює JWT і повертає його на фронтенд. Браузер зберігає маркер локально – у нашій дипломній роботі це localStorage – та додає його до кожного подальшого запиту через заголовок Authorization: Bearer <token>. Завдяки цьому клієнтська SPA не тримає постійного каналу з сервером і здатна працювати офлайн приблизно годину: поки термін дії маркера не сплив, будь-які переходи між екранами виконуються без повторної авторизації, що формує відчуття безшовної роботи навіть при нестабільному мобільному інтернеті.

На серверному боці перед обробкою запиту першим запускається authMiddleware. Він дістає токен із заголовка, перевіряє цифровий підпис і порівнює відмітку «exp» з поточним часом. Якщо підпис пошкоджено або токен прострочений, запит блокується зі статусом 401 Unauthorized, а користувач автоматично переадресовується на сторінку входу. Така модель робить бекенд абсолютно безстанним: жодної таблиці сесій чи Redis-кластеру не потрібно, тому систему можна горизонтально масштабувати – наприклад, підняти ще кілька pod-ів у Kubernetes – без синхронізації сесійних даних.

Для захисту від атак повторного відтворення (replay) срок дії токена обмежено однією годиною. Якщо проєкт вимагатиме триваліших сесій, додається механізм «refresh»: короткий lived-token зберігається як httpOnly-

cookie з флагом SameSite=Strict, а клієнт періодично запитує новий accessToken. У дипломній роботі такий рівень вважається надлишковим, проте архітектура вже готова: достатньо додати ендпоінт /api/refresh та TTL-індекс у MongoDB.

Щоб токен не потрапив до рук зловмисника, усі запити йдуть виключно поверх HTTPS із TLS 1.2. Додатково, мікросервіс аудиту записує кожну критичну дію, скасування бронювання, зміну пароля у власну колекцію audit, вказуючи userId, IP-адресу та хешовані claims токена. Це закриває вимоги Закону України «Про захист персональних даних» щодо протоколювання доступів і дозволяє в разі інциденту швидко відтворити повну картину подій.

Таким чином, впровадження JWT формує єдиний, контрольований контур безпеки: клієнт отримує зручність офлайнової роботи й мінімум зайвих запитів, бекенд, скейлабельність і простоту перевірки підпису, а адміністратор прозору систему логів і аудит-трейл, що задовольняє академічні та нормативні критерії захисту інформаційно-комп’ютерної системи.

3.5 SEO інформаційно-комп’ютерної системи

Оптимізація SPA-порталу для пошукових систем починається з того, що користувач та пошуковий робот одержують «чистий» і зрозумілий HTML уже у перше сканування. До шаблону одразу підставляються коректний заголовок, лаконічний опис і базові open-graph-теги, завдяки чому сторінка виглядає зрозумілою як у звичайній видачі Google, так і в прев’ю соціальних мереж. Додаткова розмітка Schema.org у форматі JSON-LD допомагає пошуковику розпізнати, що йдеться саме про медичну послугу з конкретними слотами запису – отже, у результатах можна показати розширений сніпет із ціною, адресою клініки та активною кнопкою «Записатися онлайн». Такий підхід не лише полегшує індексацію, а й одразу справляє враження на користувача: ще до переходу на сайт він бачить, що сервіс надає повний і зрозумілий набір даних.

Технічна сторона підсилює семантику швидкістю. Завдяки мінімальному бандлу JavaScript і критичному CSS, що важить лічені десятки кілобайт, перший

рендер відбувається помітно швидше за «середньотемпературний» сайт-конкурентів. Зображення лікарів та ілюстрації процедур конвертуються у WebP і доставляються з кешу найближчого CDN-вузла, тож Core Web Vitals – метрику, що Google офіційно враховує у ранжуванні – вдається утримувати в «зеленій» зоні без додаткової оптимізації.

Статичні ресурси кешуються Service Worker-ом, тому повторні переходи майже миттєві, а окремий XML-файл-карта сайту підказує роботам, де з’явилися нові сторінки та коли востаннє їх оновлювали. Файл robots.txt узгоджує видимість: відкрити публічний каталог послуг, але не індексувати приватний кабінет чи робочі URL-адреси апі – просте правило, яке захищає від дублю та випадкового витоку внутрішніх посилань.

Контентна складова довершує технічну й семантичну оптимізацію. Регулярні оглядові матеріали про те, як швидко записатися без черг, чому важливо проходити щорічний чекап або як підготуватися до консультації кардіолога, формують довіру й приваблюють цільові. Статті поширюються в тематичних групах і телеграм-каналах, збирають гіперпосилання з локальних медичних блогів і тим самим підвищують авторитет домену.

Легка внутрішня перелінковка веде читача від публікації одразу до картки потрібної послуги, а локалізовані ключові фрази допомагають потрапити до короткого списку карт Google. Показники кліків і середньої позиції періодично переглядаються, щоби коригувати семантику та підсилювати слабкі сторінки. У результаті портал з’єднує швидкодію, структуровані дані та корисний контент в одному ланцюжку – і завдяки цьому стабільно піднімається у видачі, нарощуючи органічний трафік без додаткового платного просування.

ВИСНОВКИ

У рамках виконання роботи було розроблено SPA-сайт для бронювання медичних послуг, який складається з двох незалежних частин, клієнтської та серверної. Для реалізації фронтенду використано сучасний інструментарій React із застосуванням Vite для швидкої розробки та налаштування оточення. Серверна частина реалізована на Node.js із базою даних MongoDB, що дозволило створити гнучку та масштабовану архітектуру.

На початку був проведений докладний аналіз предметної області, аби зрозуміти реальні потреби пацієнтів і клінік. Аналітична частина показала, що люди хочуть укладатися у чотири кліки й бачити історію відвідувань у кабінеті, а медперсонал – мати простий інструмент для запобігання пропущеним прийомам. Зібрані очікування лягли в основу шести ключових сценаріїв, які стали стрижнем усього проєкту. Усі вимоги було відсортовано за важливістю, тому команда фокусувалася лише на тому, що справді впливає на користувацький досвід. Така дисципліна дозволила втримати обсяг робіт у межах календарного плану й уникнути невиправданих надбудов.

Архітектура порталу спирається на поєднання React, Node.js та MongoDB, адже саме ця трійка найкраще відповідає критеріям швидкодії та простоти горизонтального масштабування. Під час порівняння альтернатив з’ясувалося, що React із Vite запускає дев-сервер у рази швидше за традиційні збирачі, а асинхронна модель Node.js ідеально підходить для коротких REST-запитів без важких обчислень. Модель даних свідомо зведено до двох сутностей – User і Booking, що спростило початкову схему й залишило простір для майбутніх розширень без складних міграцій. Кожен елемент архітектури взаємодіє з іншими через чітко визначені інтерфейси, тому замінити або оновити будь-який модуль можна без зупинки сервісу. Така цілісність уже на етапі MVP підтвердилась стабільною роботою під час тестових навантажень.

Щоб зробити внутрішню логіку прозорою, було підготовлено набір UML-діаграм. Use Case-схема дала змогу одразу побачити повну картину дій пацієнта,

а послідовні діаграми допомогли відшліфувати ланцюжок створення бронювання. Компонентна діаграма чітко розмежувала зони відповідальності між клієнтом, API-шаром і базою даних, завдяки чому вдалося запобігти дублюванню функцій у коді. ER-модель переконливо показала, що документна база без проблем зберігає всі потрібні атрибути й дозволяє швидко будувати індекси. Візуальне представлення архітектури полегшило комунікацію з рецензентами та скоротило час на технічні обговорення.

Клієнтська частина отримала адаптивний інтерфейс, який однаково зручний на великому моніторі й на смартфоні. Глобальний стан управляється легковаговою бібліотекою Zustand, що мінімізує повторні перерендери та зберігає токен між перезавантаженнями сторінки. Дизайн побудовано за принципом mobile first: усі головні кнопки доступні великим пальцем, а поля введення не виходять за межі екрана. Захищені маршрути відкриваються лише після успішної перевірки JWT, тому особисті дані ніколи не потрапляють до сторонніх осіб. Аудит доступності підтвердив відповідність WCAG AA без додаткових виправлень.

На сервері розгорнуто повний комплект REST-маршрутів для реєстрації, входу та керування бронюваннями. Кожен запит спершу проходить через проміжне ПЗ, яке перевіряє дані й веде журнал дій, тому помилки фіксуються ще до запису в базу. Дані зберігаються у кластері MongoDB Atlas, а комбінований індекс лікаря, дати й часу гарантує відсутність дублювань навіть за високого навантаження. Вся передача інформації відбувається через HTTPS, що виконує вимоги сучасних протоколів безпеки. Лог-файли автоматично архівуються, тож за потреби можна швидко відтворити хронологію подій.

Автентифікація базується на JWT із симетричним підписом HS256. Токени мають обмежений час дії, а їх перевірка виконується за одну операцію, тому не створює помітного навантаження на процесор. У разі спливу терміну дії користувач бачить зрозуміле повідомлення й одразу переходить на сторінку входу, що підвищує безпеку та не псує враження від роботи сервісу. Відсутність серверного сховища сесій спрощує масштабування: нові екземпляри застосунку

піднімаються без додаткової синхронізації даних. Схема легко доповнюється оновленням токена, якщо знадобиться довша сесія.

Функції створення, перегляду та скасування записів відпрацьовують майже миттєво навіть на слабких мережах. Коли користувач обирає годину, яка вже зайнята, система одразу пропонує найближчий вільний слот, що зменшує кількість відмов. Дані у списку бронювань оновлюються без перезавантаження сторінки, тому пацієнт одразу бачить результат дії. Серверний лог показав, що навіть при сотні одночасних запитів час відповіді лишався стабільним. Така швидкодія покращує загальне враження й сприяє повторним зверненням користувачів.

Якість реалізації підтверджено модульними та інтеграційними тестами. Колекція Postman перевірила коректність статус-кодів і схем JSON, а тести Jest показали упевнене покриття основних гілок логіки. Реальні користувачі змогли зареєструватися, забронювати прийом і за потреби скасувати його менш ніж за хвилину та без зовнішньої допомоги. Додаткове тестування на мобільних пристроях підтвердило відсутність критичних проблем з адаптивністю. Отже, поставлені цілі щодо мінімальної кількості дій і швидкого сценарію були досягнуті.

Щоб покращити індексацію порталу, було додано коректні метатеги для кожної сторінки, включаючи заголовки та описи, а також Open Graph-поля для правильного відображення посилань у соцмережах. Розмітка JSON-LD за стандартом Schema.org допомагає пошуковим системам одразу зрозуміти, що йдеться про медичні послуги з деталями щодо лікарів і доступних слотів. Крім того, налаштовано файл sitemap.xml, який автоматично оновлюється при появі нових сторінок, і файл robots.txt, що визначає, які розділи слід індексувати, а які залишити приватними. Для оптимізації завантаження ресурсів усі зображення було конвертовано у формат WebP, мінімізовано розміри файлів, а Service Worker забезпечує кешування статичних файлів, що полегшує повторні відвідини сайту. Завдяки цим рішенням портал отримав високі оцінки Core Web

Vitals, а швидкість завантаження і взаємодії з контентом покращилася, що сприяло зростанню органічного трафіку та кращій видимості серед конкурентів.

На завершальному етапі проєкт було успішно розгорнуто у тестовому середовищі, що дало можливість продемонструвати повний цикл роботи системи в реальних умовах. Налагоджено безперервний процес збірки, у якому фронтенд і бекенд автоматично компілюються та стартують у хмарі Render за одним натисканням. База даних знаходиться у безпечному кластері MongoDB Atlas, де ввімкнено регулярні резервні копії та моніторинг продуктивності. Для швидкої перевірки працездатності достатньо клонувати репозиторій і запустити дві команди, що робить проєкт доступним навіть для користувачів без досвіду налаштування серверів. Подальший розвиток передбачає підключення SMS-сповіщень і рольової моделі лікар-адміністратор, які дадуть змогу розширити аудиторію сервісу та ще більше автоматизувати роботу клінік.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Custom Doctor Appointment Booking App. URL: https://www.syscreations.com/build-custom-doctor-appointment-app/ (date of access: 04.01.2025).

2. Офіційний сайт doc.ua. URL: https://doc.ua (дата звернення: 06.01.2025).

3. Unleashing the Power of Document-based Databases. URL: https://www.linkedin.com/pulse/unleashing-power-document-based-databases-flexibility-mudunuri-nhkyc (date of access: 10.01.2025).

4. Evaluating Service-Oriented and Microservice Architecture Patterns (підходіи до усунення зазначених недоліків). URL: https://www.mdpi.com/2076-3417/11/10/4350 (date of access: 11.01.2025).

5. Онлайн платформа медичного сайту Doctolib створеного в Франції. URL: https://www.doctolib.de/gesundheit/ukrainisch/ (дата звернення: 14.01.2025).

6. Google for Startups Ukraine Support Fund. URL: https://startup.google.com/programs/ukraine-support-fund/ (date of access: 20.01.2025).

7. Manage FHIR Data from Android App with Google Cloud. URL: https://cloud.google.com/blog/topics/healthcare-life-sciences/build-fhir-patient-data-android-apps-with-open-health-stack-and-google-cloud (date of access: 23.01.2025).

8. MoSCoW як метод виставлення пріорітетів проєкта. URL: https://nesslabs.com/moscow-method (date of access: 27.01.2025).

9. SPA vs. MPA. URL: https://medium.com/%40sehban.alam/single-page-applications-spas-vs-multi-page-applications-mpas-advantages-and-challenges-df06bee3fed1 (date of access: 05.02.2025).

10. Healthcare Scheduling Software Systems. URL: https://www.uptech.team/blog/healthcare-scheduling-software-systems (date of access: 08.02.2025).

11. Сайт з аналізом та порівнянням баз даних з врахуванням всіх недоліків. URL: https://www.guru99.com/uk/mongodb-vs-mysql.html (дата звернення: 11.02.2025).

12. Що таке MongoDB та як він допомагає в створенні маштабованих програм. URL: https://intuji.com/what-is-mongodb-for-building-scalable-apps/ (date of access: 12.02.2025)

13. Mobile First Design. URL: www.browserstack.com/guide/how-to-implement-mobile-first-design (date of access: 13.02.2025).

14. Choosing the Right JavaScript Framework. URL: https://upforcetech.com/choosing-the-right-javascript-framework-in-2024-react-vs-angular-vs-vue/ (date of access: 18.02.2025

15. How does Vite work – a comparison to webpack. URL: https://harlanzw.com/blog/how-the-heck-does-vite-work (date of access: 24.02.2025).

16. Легкий гайд по налаштуванню Vite та React в Visual Studio Code. URL: https://www.linkedin.com/pulse/setting-up-react-vite-vscode-your-easy-guide-barnabas-ukagha-phcmf (date of access: 28.02.2025).

17. Zustand vs Redux. URL: https://medium.com/%40breakingbadjs/zustand-vs-redux-a-comprehensive-comparison-in-state-management-687a86156b14 (date of access: 05.02.2025).

18. Туторіал з використання React Material UI. URL: https://www.geeksforgeeks.org/react-material-ui/ (date of access: 05.01.2025).

19. Визначення і поняття того що таке JWT токен. URL: https://supertokens.com/blog/what-is-jwt (date of access: 02.02.2025).