ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Цифрова система обслуговування медичної лабораторії з web- та мобільним доступом на основі TypeScript»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Кирило НІКОЛІН
(підпис)

Виконав: здобувач(ка) вищої освіти групи ТЦР-42

Кирило НІКОЛІН

Керівник: Вікторія ЖЕБКА
д-р техн. наук, проф.

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Технології цифрового розвитку
Ступінь вищої освіти Бакалавр
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Технології Цифрового розвитку
_____Вікторія ЖЕБКА
« _____» ___2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Ніколніу Кирилу Андрійовичу

1. Тема кваліфікаційної роботи: «Цифрова система обслуговування медичної лабораторії з web- та мобільним доступом на основі TypeScript» керівник кваліфікаційної роботи професор Вікторія ЖЕБКА, затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «___» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: офіційна документація TypeScript, Node.js, Express, React, React Native, PostgreSQL, Prisma ORM; наукові джерела з автоматизації медичних лабораторій та побудови LIMS-систем; матеріали з проєктування медичних інформаційних систем; документація Firebase Cloud Messaging, Docker, MinIO.

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

1. Аналіз предметної галузі автоматизації медичних лабораторій та існуючих LIMS-рішень

2. Аналіз засобів реалізації та архітектурного проєктування веб- та мобільного застосунку.
3. Програмна реалізація серверної частини, веб-кабінету та мобільного додатку для медичної лабораторії.
4. Реалізація інтерфейсу та тестування функціональності системи.

5. Перелік графічного матеріалу: *презентація*
1. Аналіз аналогів(Synevo, ДІЛА, HealthLink).
2. Вимоги до програмного забезпечення.
3. Програмні засоби реалізації(технологічний стек).
4. Діаграма архітектури системи.
5. Діаграма варіантів використання (пацієнт).
6. Діаграма варіантів використання (лаборант та адміністратор).
7. Діаграма структури бази даних.
8. Діаграма переходів статусів замовлення.
9. Екранні форми (веб-кабінет та мобільний додаток).
10. Апробація результатів дослідження.

6. Дата видачі завдання «20» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	20.02-04.03.2026	
2	Аналіз та дослідження існуючих LIMS-аналогів	05.03-13.03.2026	
3	Огляд технологій TypeScript-екосистеми, проєктування архітектури	14.03-21.03.2026	
4	Проєктування бази даних та REST API	22.03-04.04.2026	
5	Програмна реалізація серверної частини (Node.js, Express, Prisma)	05.04-20.04.2026	
6	Програмна реалізація веб-кабінету (React, Ant Design)	21.04-30.04.2026	

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
7	Програмна реалізація мобільного застосунку (React Native)	01.05-10.05.2026	
8	Тестування системи та оформлення роботи	11.05-18.05.2026	
9	Розробка демонстраційних матеріалів та попередній захист	19.05-30.05.2026	

Здобувач вищої освіти _____ Кирило НІКОЛІН
(підпис)

Керівник
кваліфікаційної роботи _____ Вікторія ЖЕБКА
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 108 стор., 12 табл., 40 рис., 27 джерел.

Мета роботи – підвищити зручність обслуговування пацієнтів медичної лабораторії та скоротити час доставки результатів за рахунок автоматизації повного операційного циклу в єдиному застосунку, що об'єднує онлайн-запис, оплату, перегляд результатів та динаміки показників – від запису до отримання верифікованого результату на телефон пацієнта.

Об'єкт дослідження – процес автоматизації операційної діяльності медичної лабораторії.

Предмет дослідження – програмне забезпечення для автоматизації лабораторних процесів із підтримкою веб- та мобільних платформ.

Короткий зміст роботи: у роботі проаналізовано сучасні підходи до автоматизації медичних лабораторій та існуючі системи (Synevo, ДІЛА, HealthLink). Обґрунтовано вибір технологічного стеку на базі TypeScript-екосистеми. Спроєктовано архітектуру монорепозиторію з трьома застосунками: серверна частина (Node.js, Express, Prisma, PostgreSQL 16), веб-кабінет (React 19, Redux Toolkit, Ant Design 6) та мобільний додаток (React Native 0.85). Реалізовано 28 серверних сервісів, 21 контролер, базу даних із 30+ моделями. Система забезпечує JWT-автентифікацію, рольовий контроль доступу (PATIENT/LAB_TECH/ADMIN), генерацію PDF-звітів, push-повідомлення через Firebase Cloud Messaging, модуль графічного відображення динаміки лабораторних показників та аудит-лог критичних операцій. Проведено функціональне тестування (80 тестів API), кросбраузерне та мобільне тестування.

КЛЮЧОВІ СЛОВА: ВЕБ-ЗАСТОСУНОК, МОБІЛЬНИЙ ЗАСТОСУНОК, TYPESCRIPT, REACT, REACT NATIVE, МЕДИЧНА ЛАБОРАТОРІЯ, ЛАБОРАТОРНА ІНФОРМАЦІЙНА СИСТЕМА, JWT, POSTGRESQL.

ЗМІСТ

ВСТУП

Актуальність: сучасна медична діагностика базується на лабораторних дослідженнях, які, за різними оцінками, формують до 70% клінічних рішень лікарів [1]. В Україні функціонують сотні приватних та державних лабораторій, проте значна частина з них досі використовують застарілі методи документообігу – паперові бланки результатів, телефонне інформування пацієнтів, ручне ведення журналів зразків [2]. Такий підхід призводить до затримок у доставці результатів (від кількох годин до доби), ризиків помилок при ручному копіюванні даних та неможливості для пацієнта оперативно отримувати інформацію про стан здоров'я.

Глобальний тренд цифровізації охорони здоров'я демонструє ефективність впровадження лабораторних інформаційних систем (Laboratory Information Management System, LIMS) [3; 4]. За даними досліджень, автоматизація лабораторних процесів скорочує час обробки замовлень на 25-40% та зменшує кількість помилок на 60-80% [1]. В Україні існує кілька комерційних рішень (Synevo, ДІЛА), однак вони мають обмежений мобільний функціонал, не надають графічного відображення динаміки показників та не пропонують self-hosted варіант розгортання [2].

Об'єкт дослідження – процес автоматизації операційної діяльності медичної лабораторії.

Предмет дослідження – програмне забезпечення для автоматизації лабораторних процесів із підтримкою веб- та мобільних платформ.

Мета роботи – підвищити зручність обслуговування пацієнтів медичної лабораторії та скоротити час доставки результатів за рахунок автоматизації повного операційного циклу в єдиному застосунку, що об'єднує онлайн-запис, оплату, перегляд результатів та динаміки показників – від запису до отримання верифікованого результату на телефон пацієнта.

Для досягнення поставленої мети визначено наступні задачі:

1. Провести аналіз існуючих рішень у сфері автоматизації медичних лабораторій.
2. Обґрунтувати вибір технологічного стеку та спроєктувати архітектуру системи.
3. Реалізувати серверну частину з JWT-автентифікацією, RBAC та REST API.
4. Розробити веб-застосунок з персональними кабінетами для трьох ролей.
5. Створити мобільний застосунок із push-повідомленнями та PDF-звітами.
6. Провести тестування та оцінити ефективність розробленого рішення.

Методи дослідження: аналіз наукових джерел у галузі медичних інформаційних систем в[1-5]; порівняльний аналіз LIMS-платформ; методи системного аналізу та проєктування програмного забезпечення; практичні методи розробки і тестування веб- та мобільних застосунків.

Наукова новизна полягає у створенні єдиної TypeScript-екосистеми для всіх рівнів медичної лабораторної системи з інтеграцією модуля динаміки лабораторних показників, що відсутній у конкурентних рішеннях українського ринку.

Практична значущість – self-hosted рішення (Docker Compose) зі скороченням часу доставки результатів на ~30% та повним аудит-логуванням операцій відповідно до законодавства про захист медичних даних [5].

Апробація результатів та публікації. Результати роботи представлено на двох науково-технічних конференціях у ДУІКТ (м. Київ, 2026 р.): VI Міжнародній «Сучасний стан та перспективи розвитку IoT» і VI Всеукраїнській «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». За темою дослідження опубліковано двоє тез доповідей [26, 27]. Запропоновані рішення можуть стати основою для подальших розробок у сфері цифровізації медичних лабораторій.

1. Ніколін К. А., Жебка В. В. Веб- та мобільний застосунок для медичної лабораторії з використанням TypeScript // Матеріали VI Міжнародної науково-технічної конференції «Сучасний стан та перспективи розвитку IoT». Збірник тез. – К.: ДУІКТ, 2026.
2. Ніколін К. А., Жебка В. В. Автоматизація процесів медичної лабораторії засобами TypeScript-екосистеми // Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. – К.: ДУІКТ, 2026.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Характеристика предметної галузі

Лабораторна діагностика є фундаментом сучасної клінічної медицини. За різними оцінками, від 60 до 70 відсотків лікарських рішень прямо чи опосередковано спираються на результати лабораторних досліджень [1]. Ринок приватної лабораторної діагностики в Україні протягом останнього десятиліття зазнав значних змін: якщо до 2014 року більшість пацієнтів користувалися виключно державними лабораторіями при поліклініках, то станом на 2026 рік приватні мережі (Сінево, ДІЛА, Ескулаб та інші) обслуговують мільйони клієнтів щороку [2].

Типовий шлях пацієнта у медичній лабораторії складається з кількох етапів. Спершу людина обирає потрібне дослідження – самостійно або за направленням лікаря. Далі відбувається запис на візит (онлайн через сайт чи за телефоном, або безпосередньо у відділенні). У призначений час пацієнт приходить до відділення, де медична сестра здійснює забір біоматеріалу. Зразок маркується унікальним штрих-кодом та передається до лабораторії, де проходить дослідження на відповідному аналізаторі. Після отримання результату лаборант вносить значення до інформаційної системи, а старший лаборант або лікар-лаборант верифікує дані, підтверджуючи їхню коректність. Нарешті, готовий результат надходить до пацієнта – електронною поштою, через особистий кабінет або у вигляді паперового бланку [3].

Кожен із зазначених етапів потенційно вразливий до помилок та затримок. За даними міжнародних досліджень, до 75 відсотків лабораторних похибок припадають на преаналітичну фазу – тобто на етапи до безпосереднього виконання дослідження: неправильну ідентифікацію зразка, помилки при заповненні направлення, порушення умов транспортування [1]. Автоматизація саме цих процесів – реєстрації пацієнта, маркування зразка, передачі результатів – здатна суттєво знизити ризик похибок та пришвидшити весь цикл обслуговування.

Окремою проблемою є відсутність у більшості існуючих рішень інструментів для моніторингу здоров'я у часі. Пацієнт із хронічним захворюванням (наприклад, цукровим діабетом) регулярно здає одні й ті самі аналізи – глікований гемоглобін, глюкозу натще – і потребує можливості порівняти результати за різні дати, побачити тенденцію на графіку, зрозуміти, чи покращується його стан. Такий функціонал у зарубіжній літературі називають longitudinal tracking або trend analysis, і його наявність визнається однією з ключових переваг сучасних LIMS-систем [4].

Предметна галузь даної кваліфікаційної роботи охоплює проєктування та реалізацію програмного забезпечення, що автоматизує повний операційний цикл медичної лабораторії: від онлайн-запису пацієнта до отримання верифікованих результатів через веб-кабінет та мобільний додаток, із можливістю графічного відображення динаміки показників у часі.

1.2 Аналіз існуючих аналогів

Для об'єктивного порівняння та виявлення напрямів, що потребують удосконалення, у цьому підрозділі розглянуто три платформи: власну розроблювану систему Monolab (як прототип, що є предметом даної кваліфікаційної роботи), а також дві найбільші комерційні лабораторії України – Synevo (Сінево) та ДІЛА. Аналіз проводився на основі практичного дослідження інтерфейсів зазначених платформ автором протягом 2018-2026 рр.

1.2.1 Аналіз платформи Monolab

Monolab – прототип лабораторної інформаційної системи, що розробляється у межах даної кваліфікаційної роботи. Система спроєктована як монорепозиторій із трьох застосунків: серверна частина (Node.js, Express, Prisma, PostgreSQL), веб-кабінет (React 19, Ant Design 6) та мобільний додаток (React Native 0.85). Усі компоненти написані мовою TypeScript [8], що забезпечує перевикористання типів та валідаційних схем між підпроєктами.

Функціональний обсяг Monolab охоплює повний операційний цикл лабораторії. Пацієнт має можливість зареєструватися, обрати дослідження з каталогу, записатися на зручну дату та час, оплатити замовлення, а після готовності – переглянути результати через веб-кабінет або мобільний додаток, завантажити PDF-звіт та відстежити динаміку конкретного показника на лінійному графіку з нанесеними межами нормальних значень [22].

Для лаборанта реалізовано робочу панель із чергою зразків, формами введення та верифікації результатів, сканером штрих-коду. Адміністратор отримує доступ до управління каталогом тестів, профілями користувачів, фінансовою статистикою та аудит-логом, що фіксує кожну критичну дію із зазначенням часу, автора та IP-адреси.

Рольова модель доступу побудована на принципі RBAC (Role-Based Access Control) із трьома ролями: PATIENT, LAB_TECH та ADMIN. Автентифікація реалізована через JWT-токени [18] з парою access (1 година) та refresh (30 днів). Безпека доповнена шифруванням паролів через bcrypt та HTTP-заголовками Helmet.js [27].

Ключовою відмінністю Monolab від комерційних аналогів є модуль динаміки показників, реалізований засобами бібліотеки Recharts [22] у веб-версії та react-native-gifted-charts у мобільному додатку. Цей модуль дозволяє пацієнту обрати конкретне дослідження та побачити графік зміни значень за всі дати, коли він здавав цей аналіз, із горизонтальними лініями верхньої та нижньої меж норми. Такий інструмент забезпечує самостійний моніторинг стану здоров'я без необхідності щоразу консультуватися з лікарем для порівняння результатів [4].

Додатковою перевагою є можливість self-hosted розгортання: вся інфраструктура (PostgreSQL, MinIO S3, серверний API, веб-клієнт) описана у Docker Compose [20] та може бути піднята на серверах будь-якої лабораторії без залежності від зовнішніх SaaS-провайдерів.

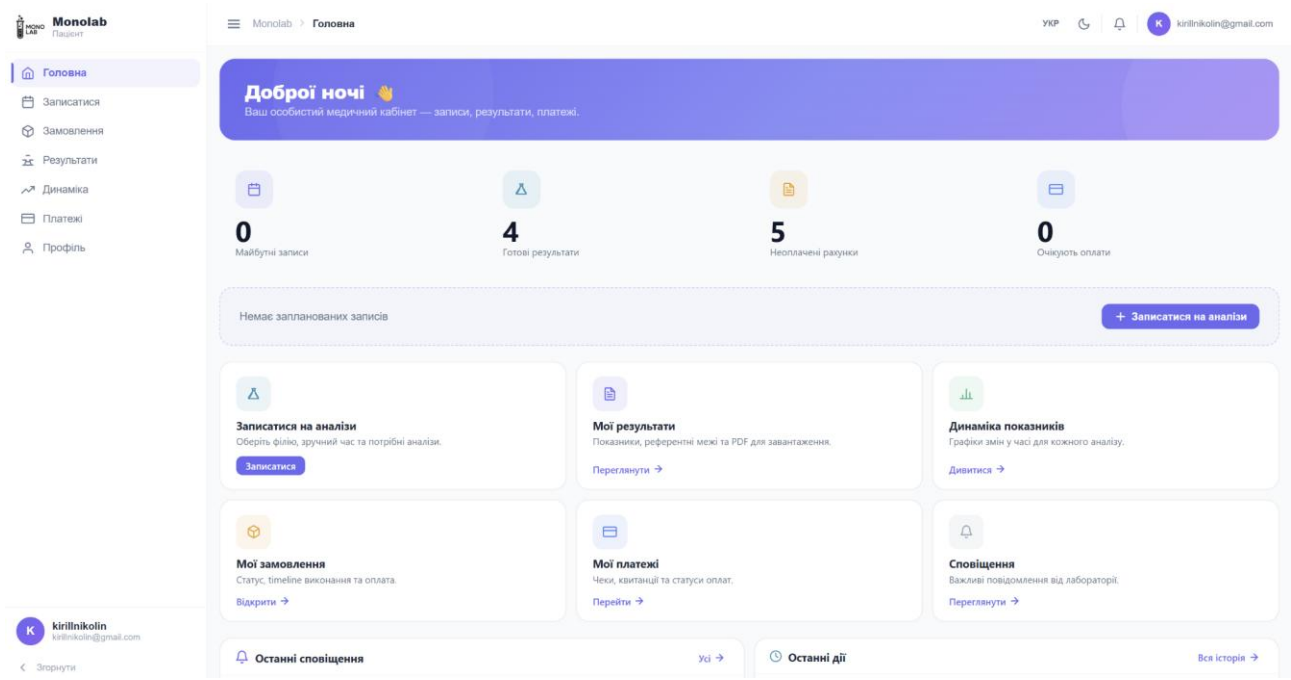


Рис. 1.1 Головна сторінка веб-кабінету Monolab

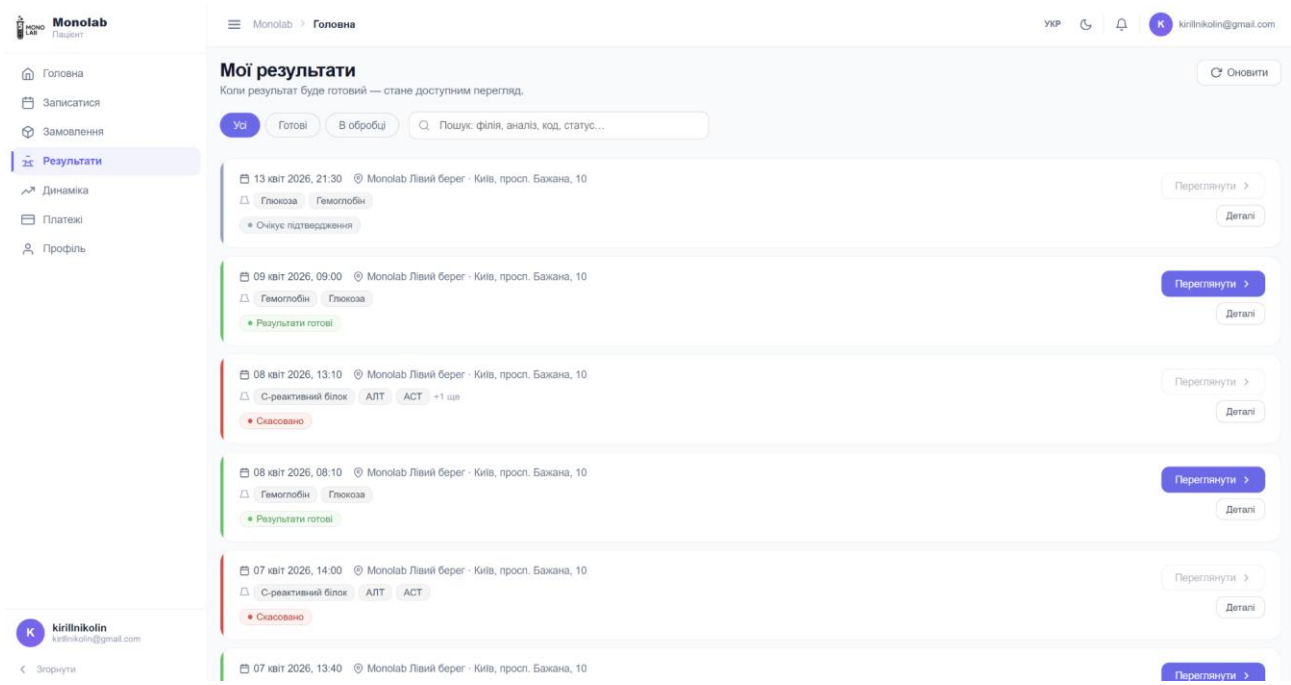


Рис. 1.2 Кабінет пацієнта Monolab – результати аналізів

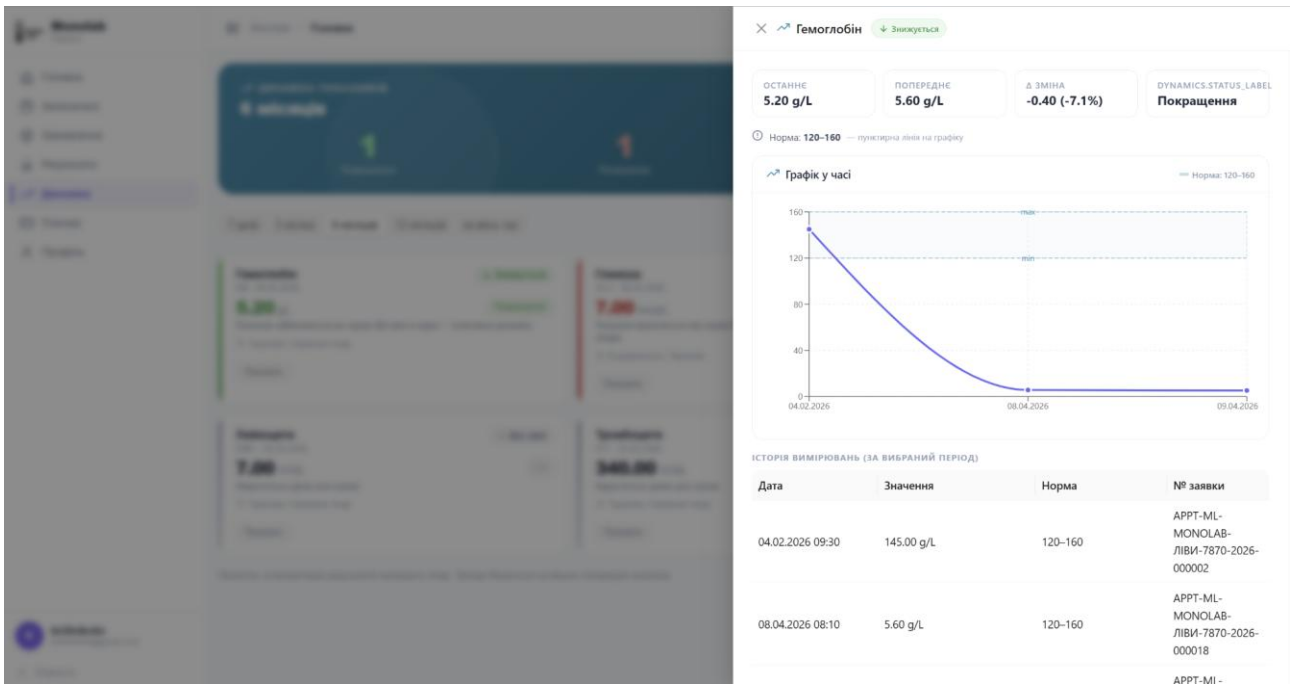


Рис. 1.3 Графік динаміки показника у Monolab

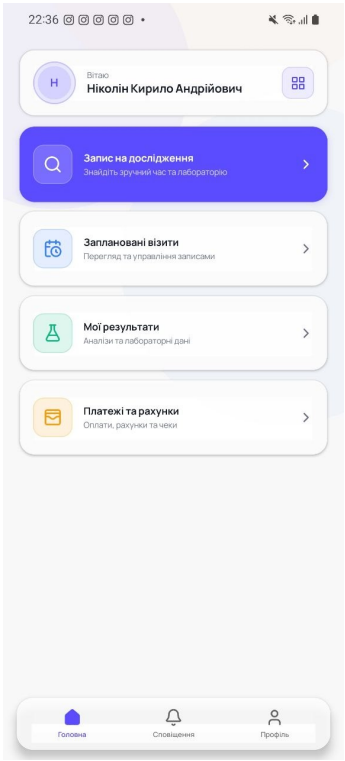


Рис. 1.4 Мобільний додаток Monolab – головний екран

1.2.2 Аналіз платформи Synevo

Synevo (Сінево) – європейська мережа медичних лабораторій, що входить до шведського холдингу Medicover [7]. В Україні Сінево працює з 2007 року і станом

на 2026 рік є найбільшою приватною лабораторною мережею країни: понад 350 відділень та 8 лабораторій у різних регіонах. У 2024 році лабораторія долучилася до Програми медичних гарантій НСЗУ, надавши безоплатно 2,4 млн тестів. Перелік досліджень Сінево охоплює понад 1900 позицій, що є найбільшим показником серед українських лабораторій [2].

Веб-кабінет Сінево (synevo.ua/personal-cabinet) надає доступ до історії замовлень та результатів аналізів. Автор даної роботи мав практичний досвід використання цієї платформи: запис здійснювався безпосередньо на ресепшині відділення, результати надійшли електронною поштою наприкінці того ж дня. В особистому кабінеті зберігається інформація про виконані замовлення, а результати доступні для перегляду онлайн. На сайті присутня вкладка «Динаміка», однак під час практичного дослідження автором вона не відображала жодних графічних даних, хоча у кабінеті зафіксовано результат одного замовлення.

Мобільний додаток MySynevo існує у Google Play та App Store, проте він розрахований виключно на користувачів у Румунії – для активації акаунту необхідно відвідати відділення Synevo саме в Румунії. В описі додатку зазначено такий функціонал: перегляд історії досліджень, графіки еволюції значень у часі, доступ до результатів дитини з одного акаунту, онлайн-запис та push-повідомлення про статус замовлення. Тобто мобільний функціонал, включаючи динаміку показників, технічно реалізований, але недоступний для українського ринку.

Серед переваг Сінево: найбільша мережа відділень в Україні, широкий перелік досліджень (1912 позицій), участь у програмі НСЗУ (безоплатні аналізи за направленням лікаря), міжнародна акредитація ISO 15189, наявність веб-кабінету з історією результатів.

Серед недоліків, виявлених у ході практичного дослідження: мобільний додаток не працює в Україні; вкладка «Динаміка» у веб-кабінеті фактично не функціонує (принаймні за наявності одного результату); відсутні push-повідомлення для українських користувачів; результати надходять лише електронною поштою; відсутня можливість self-hosted розгортання.

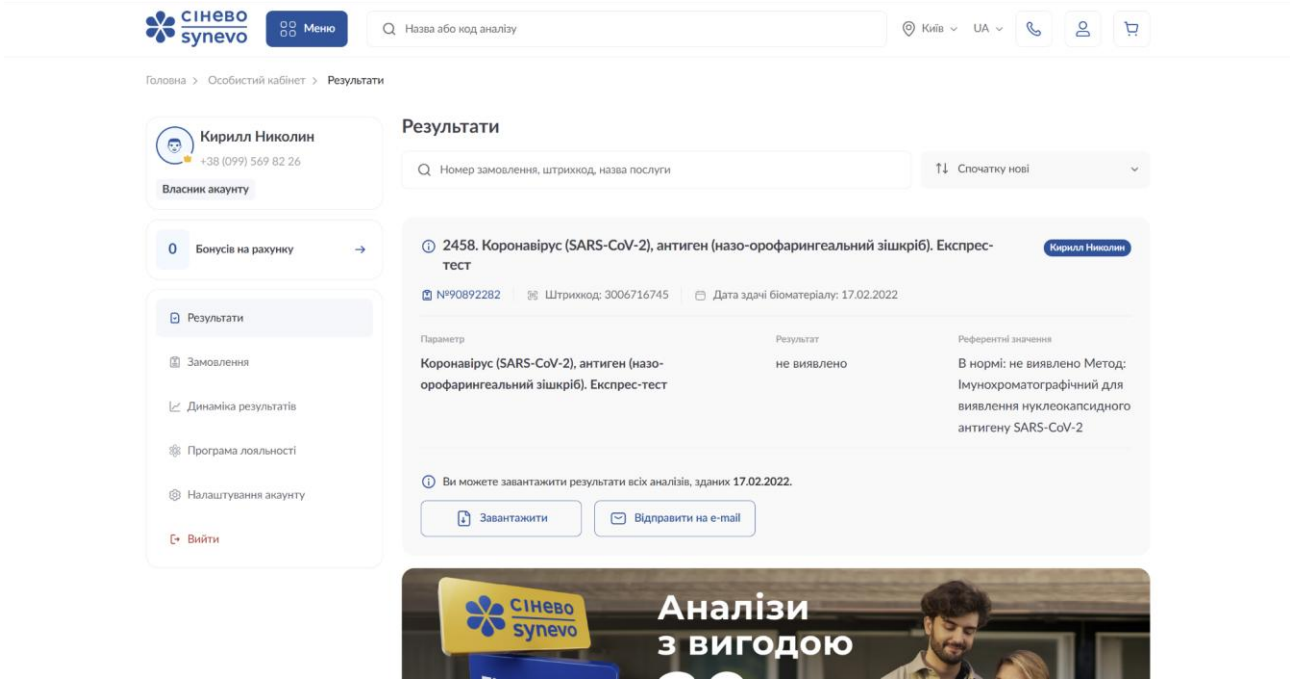


Рис. 1.5 Головна сторінка сайту synevo.ua

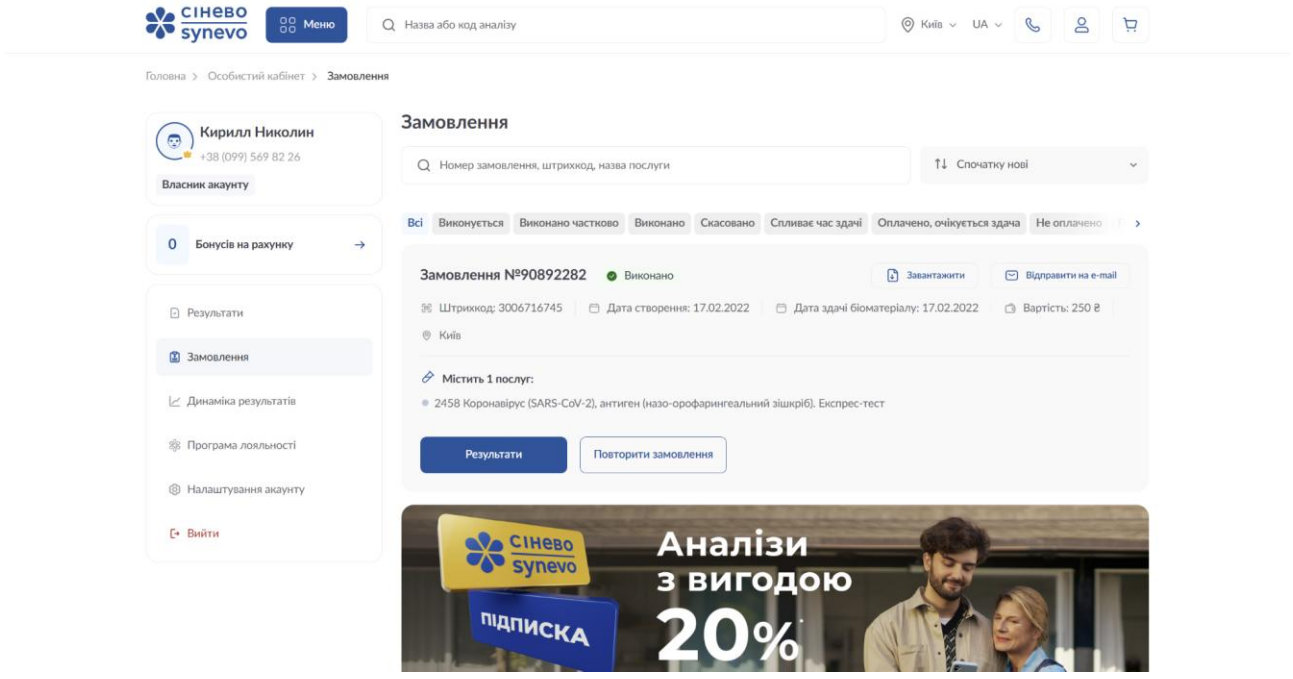


Рис. 1.6 Особистий кабінет Synevo – список замовлень

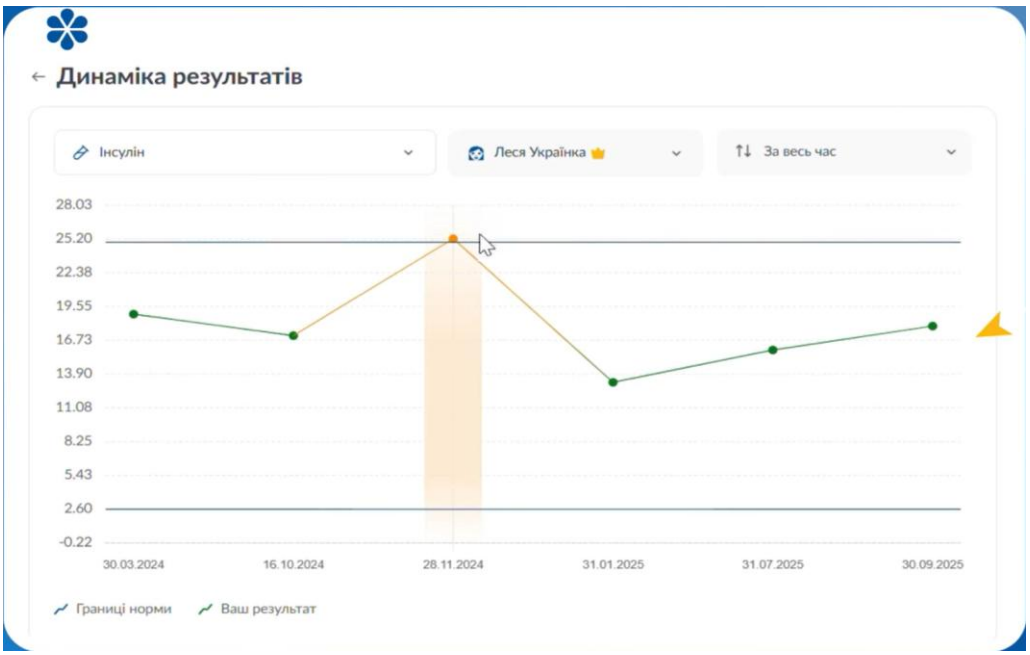


Рис. 1.7 Вкладка «Динаміка» в кабінеті Synevo

1.2.3 Аналіз платформи Діла

ДІЛА – українська мережа медичних лабораторій, заснована у 1998 році в Києві. Станом на початок 2025 року мережа налічує 238 діючих відділень по всій Україні та обслуговує понад 1,3 мільйона клієнтів щороку. У 2024 році виконано близько 9,3 мільйона досліджень, що на 21% більше порівняно з попереднім роком. ДІЛА має міжнародну акредитацію ISO 15189:2022, що підтверджує відповідність найвищим стандартам якості лабораторної діагностики [2].

Автор даної кваліфікаційної роботи користується послугами ДІЛА протягом семи років (з 2018 року), що дозволяє надати детальну оцінку еволюції цифрових сервісів платформи на основі тривалого практичного досвіду.

Веб-кабінет ДІЛА (dila.ua) реалізує кілька ключових функцій. Онлайн-запис на дослідження здійснюється через форму на сайті, де пацієнт обирає місто, відділення, категорію дослідження та зручний час. Після візиту та виконання аналізу результат з'являється в особистому кабінеті, а повідомлення про готовність надходить на вказану електронну пошту. PDF-бланк результату також надсилається на email, зазначений у кабінеті клієнта.

Принципово важливим функціоналом веб-кабінету ДІЛА є вкладка «Динаміка», доступна у десктопній версії сайту. Ця функція дозволяє пацієнту обрати конкретний показник (наприклад, глікований гемоглобін) та побачити графік зміни його значень за всі дати, коли цей аналіз було здано. Автор особисто використовував цю можливість для моніторингу показників протягом кількох років, що підтвердило практичну цінність подібного інструменту для пацієнтів із хронічними захворюваннями.

У березні 2026 року ДІЛА перезапустила мобільний додаток у оновленому форматі. Раніше мобільний додаток існував, але автор ним не користувався. Новий додаток (доступний для Android та iOS) реалізує перегляд результатів, створення замовлень та доступ до результатів родини. Проте, за результатами практичного дослідження, мобільна версія не включає функцію динаміки показників – у додатку доступний лише список результатів із можливістю перегляду деталей окремого замовлення, без графічного порівняння значень за різні дати.

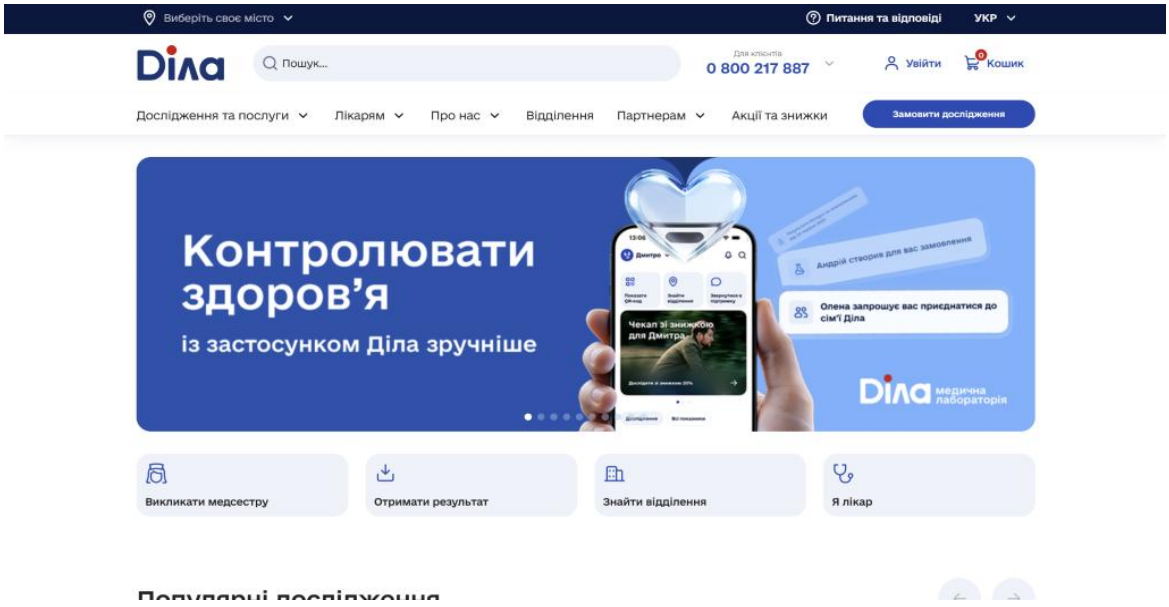


Рис. 1.8 Головна сторінка сайту dila.ua

Серед переваг ДІЛА: широка мережа відділень по Україні (238 на початок 2026), тривала присутність на ринку (з 1998), акредитація ISO 15189:2022, наявність динаміки показників у веб-кабінеті, оновлений мобільний додаток,

повідомлення на email про готовність результатів, можливість управляти результатами родини.

Серед недоліків: відсутність динаміки показників у мобільному додатку; PDF-результат надходить лише на email (не можна завантажити безпосередньо з кабінету чи додатку); відсутність push-повідомлень (повідомлення лише через email); відсутність аудит-логу, доступного для адміністратора; закрита SaaS-архітектура без можливості self-hosted розгортання; обмежений функціонал для лабораторного персоналу (внутрішні процеси ведуться в окремій закритій системі).

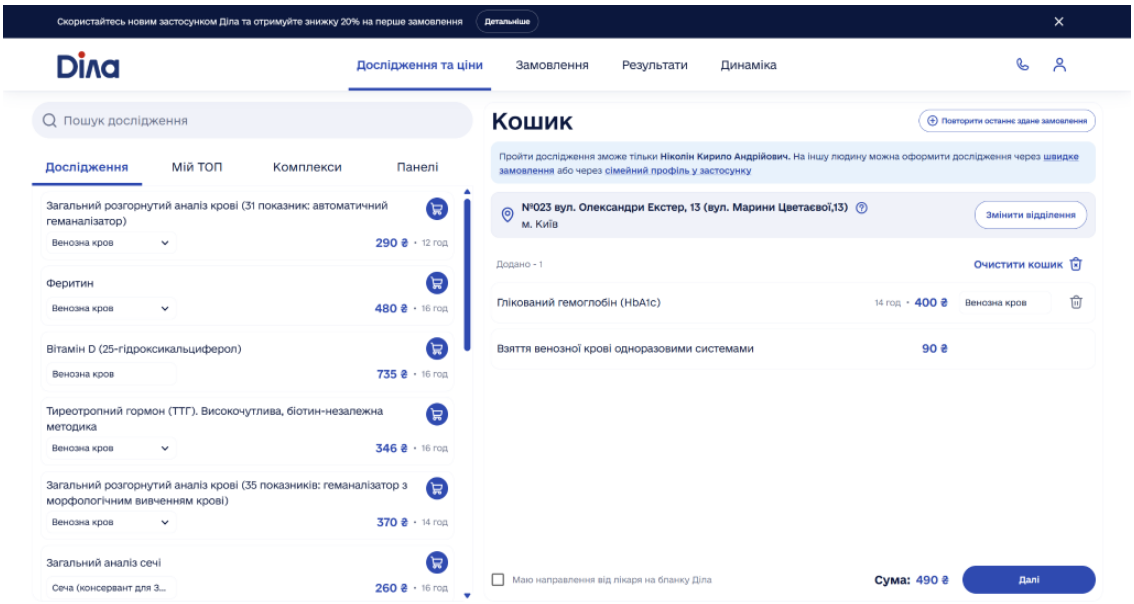


Рис. 1.9 Онлайн-запис на дослідження у ДІЛА

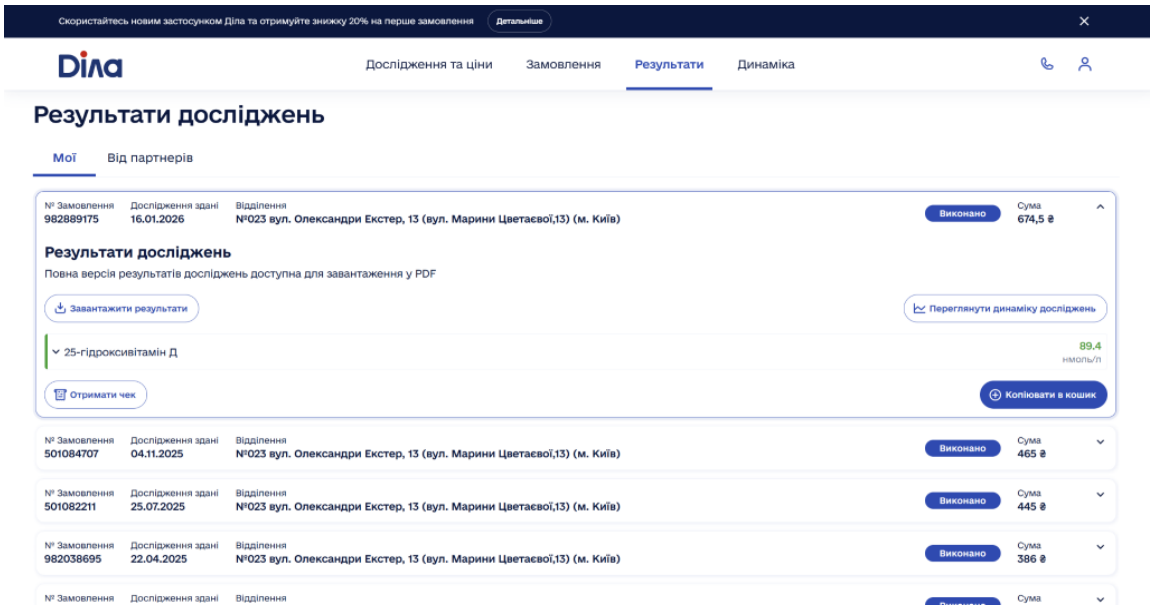


Рис. 1.10 Особистий кабінет ДІЛА – список результатів

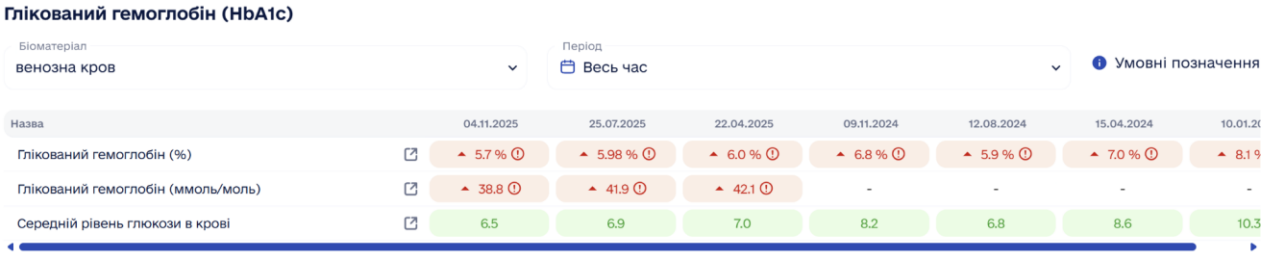


Рис. 1.11 Вкладка «Динаміка» у веб-кабінеті ДІЛА

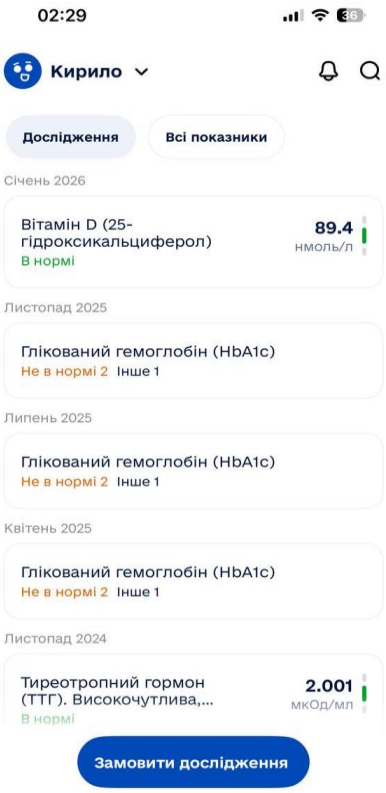


Рис. 1.12 Мобільний додаток ДІЛА (перезапуск березень 2026)

1.3 Визначення проблем предметної області

На підставі проведеного порівняльного аналізу та багаторічного практичного досвіду використання лабораторних сервісів виявлено низку типових проблем, характерних для цифрових платформ медичних лабораторій в Україні.

Першою і найбільш значущою проблемою є фрагментарність реалізації модуля динаміки показників. Серед проаналізованих платформ лише ДІЛА

пропонує графічне відображення зміни значень у часі, і лише у десктопній версії веб-кабінету. У мобільному додатку ДІЛА цей функціонал відсутній. У Сінево вкладка «Динаміка» присутня в інтерфейсі, проте фактично не функціонує. Для пацієнтів із хронічними захворюваннями, які регулярно здають одні й ті самі аналізи, відсутність можливості побачити тренд прямо у телефоні є суттєвим обмеженням [4].

Друга проблема стосується каналів доставки результатів. Обидві проаналізовані комерційні платформи використовують email як основний (а фактично єдиний) канал оповіщення пацієнта про готовність. Push-повідомлення – значно оперативніший і зручніший для мобільних користувачів механізм – у Сінево недоступний для українського ринку, а у ДІЛА реалізований лише у нещодавно оновленому додатку і потребує додаткового дослідження щодо стабільності роботи.

Третя проблема – залежність від зовнішнього провайдера (vendor lock-in). Обидві комерційні платформи функціонують за моделлю SaaS, де дані пацієнтів зберігаються на серверах компанії-провайдера. Для незалежних лабораторій, клінік або медичних центрів, які бажають мати повний контроль над даними своїх пацієнтів (що є вимогою законодавства про захист персональних медичних даних [5]), відсутність self-hosted рішень створює правову та організаційну проблему.

Четверта проблема – обмеженість інструментів для лабораторного персоналу. Жодна з проаналізованих клієнтських платформ не надає публічно доступного інтерфейсу для лаборантів: операційні процеси (прийом зразків, введення результатів, верифікація) ведуться у внутрішніх закритих системах, що не дозволяє оцінити їх з точки зору зручності та ефективності.

П'ята проблема – відсутність прозорого аудит-логування. Медичні інформаційні системи, відповідно до законодавства України та міжнародних стандартів (ISO 15189), зобов'язані забезпечувати прослідковуваність усіх операцій із даними пацієнтів [5; 6]. Жодна з проаналізованих платформ не надає адміністратору інструменту для перегляду та фільтрації аудит-логу через зовнішній інтерфейс.

1.4 Постановка завдання на розробку системи

На основі проведеного аналізу та виявлених проблем сформульовано завдання на розробку системи Monolab – комплексного програмного рішення для медичної лабораторії. Система має складатися з трьох компонентів: серверної частини (REST API), веб-застосунку та мобільного додатку, об'єднаних у монорепозиторій із використанням TypeScript [8] як єдиної мови програмування.

Система має забезпечувати повний операційний цикл лабораторії: онлайн-запис пацієнта, оплата, прийом зразка лаборантом, введення результатів, верифікація старшим лаборантом, автоматична генерація PDF-звіту [21], push-повідомлення пацієнту через Firebase Cloud Messaging [17].

Передбачається реалізація рольової моделі доступу (RBAC) із трьома ролями: PATIENT (пацієнт), LAB_TECH (лаборант), ADMIN (адміністратор). Автентифікація – через JWT [18] з парою access/refresh токенів. Модуль динаміки показників має бути доступний як у веб-версії, так і у мобільному додатку – на відміну від існуючих рішень, де ця функція або відсутня, або обмежена лише десктопною версією.

Систему планується розгортати через Docker Compose [20], що забезпечить можливість self-hosted інсталяції на серверах будь-якої лабораторії. Це дозволить медичним закладам зберігати повний контроль над даними пацієнтів відповідно до вимог законодавства [5].

1.5 Визначення вимог до застосунку

На основі аналізу предметної галузі, виявлених проблем та сформульованої мети проекту визначено вимоги до системи Monolab. Вимоги поділено на функціональні та нефункціональні.

Функціональні вимоги:

1. Самостійна реєстрація пацієнтів через веб-кабінет або мобільний додаток; створення облікових записів лаборантів та адміністраторів виключно через адміністративну панель.
2. Онлайн-запис пацієнта на лабораторне дослідження з вибором категорії тесту, конкретного дослідження, дати та часу.
3. Оплата замовлення через систему.
4. Операційний цикл лаборанта: прийом зразка (зміна статусу), введення результатів через валідовану форму, верифікація старшим лаборантом.
5. Автоматична генерація PDF-звіту після верифікації результатів із збереженням у файловому сховищі.
6. Push-повідомлення пацієнту про готовність результатів.
7. Real-time оновлення статусу замовлення через WebSocket.
8. Графічне відображення динаміки лабораторних показників у часі з межами нормальних значень – як у веб-кабінеті, так і у мобільному додатку.
9. Адміністративна панель для управління каталогом тестів, категоріями, профілями користувачів, замовленнями та ціновою політикою.
10. Фінансова аналітика (дохід за період, кількість досліджень, середній час обробки).
11. Аудит-лог критичних операцій із фіксацією часу, автора та IP-адреси.

Нефункціональні вимоги:

1. Контроль доступу на основі ролей (RBAC) з JWT-автентифікацією [18].
2. Шифрування паролів за допомогою bcrypt.
3. Безпечні HTTP-заголовки через Helmet.js [27].
4. Час відповіді API не більше 500 мс під навантаженням.
5. Кросбраузерна сумісність (Google Chrome, Mozilla Firefox, Microsoft Edge, Safari).
6. Підтримка мобільних платформ Android та iOS.
7. Можливість self-hosted розгортання через Docker Compose [20].
8. Валідація вхідних даних на кожному ендпоінті через Zod-схеми [23].

Таблиця 1.1

Порівняння функціональності платформ

Функціональність/Показник	Monolab	Synevo	Діла
Веб-кабінет пацієнт	+	+	+
Мобільний застосунок	+	-(лише Румунія)	+ (з березня 2026)
Динаміка (веб)	+	+	+
Динаміка (застосунок)		-	-
Повний цикл лаборанта	+	-	-
Push-повідомлення	+	-	+
PDF-звіт із кабінету/застосунку	+	+	+
Онлайн-запис на аналізи	+	+	+
Timeline	+	-	-
Self-hosted розгортання	+	+	+
Фінансова аналітика	+	+	+
Кількість відділень	-(прототип)	350+	238

2 АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ ТА СТРУКТУРИ ПРОГРАМНОЇ СИСТЕМИ

2.1 Середовище розробки та організація моно репозиторію

Як основне середовище розробки для проєкту Monolab обрано WebStorm від JetBrains – професійне IDE, орієнтоване на JavaScript та TypeScript екосистему. WebStorm забезпечує глибоку інтеграцію з TypeScript (автодоповнення на рівні типів, навігація по інтерфейсах, автоматичний рефакторинг), вбудовану підтримку Node.js, React та React Native, зручну роботу з терміналом, Git та Docker. На відміну від VS Code, де аналогічний функціонал досягається через набір окремих розширень, WebStorm надає його одразу «з коробки», що зменшує час на налаштування робочого оточення.

Кодова база Monolab організована як монорепозиторій із трьома підпроєктами у директорії apps/: api (серверна частина), web (веб-кабінет) та mobile (мобільний додаток). Для управління залежностями серверної та веб-частини використовується пакетний менеджер pnpm з механізмом workspaces, що дозволяє: централізовано керувати версіями залежностей, уникати дублювання пакетів за рахунок content-addressable storage, спільно використовувати TypeScript-типи та Zod-схеми валідації між apps/api та apps/web без їх копіювання.

Для мобільного додатку (apps/mobile) використовується окремий менеджер – yarn. Такий вибір обумовлений практичним досвідом: pnpm з його механізмом символічних посилань некоректно працює з iOS-збіркою React Native – проєкт не компілюється через особливості роботи CocoaPods із залежностями. Ця проблема відома у спільноті React Native і на момент розробки не має стабільного вирішення. Використання yarn для мобільної частини забезпечує стабільну збірку під обидві платформи (Android та iOS) без додаткових конфігурацій.

Для локального середовища розробки використовується Docker Compose, який піднімає два контейнери: PostgreSQL 16 (база даних) та MinIO (S3-сумісне

об'єктне сховище для файлів та PDF-звітів). Такий підхід забезпечує ідентичність середовищ розробки та розгортання: розробник отримує ті самі сервіси, що працюватимуть на продакшн-сервері, без необхідності встановлювати PostgreSQL чи MinIO безпосередньо на операційну систему [20].

Контроль версій здійснюється через Git із хостингом на GitHub. Для забезпечення якості коду налаштовано ESLint (статичний аналіз) та Prettier (автоматичне форматування) з єдиною конфігурацією для всіх підпроєктів.

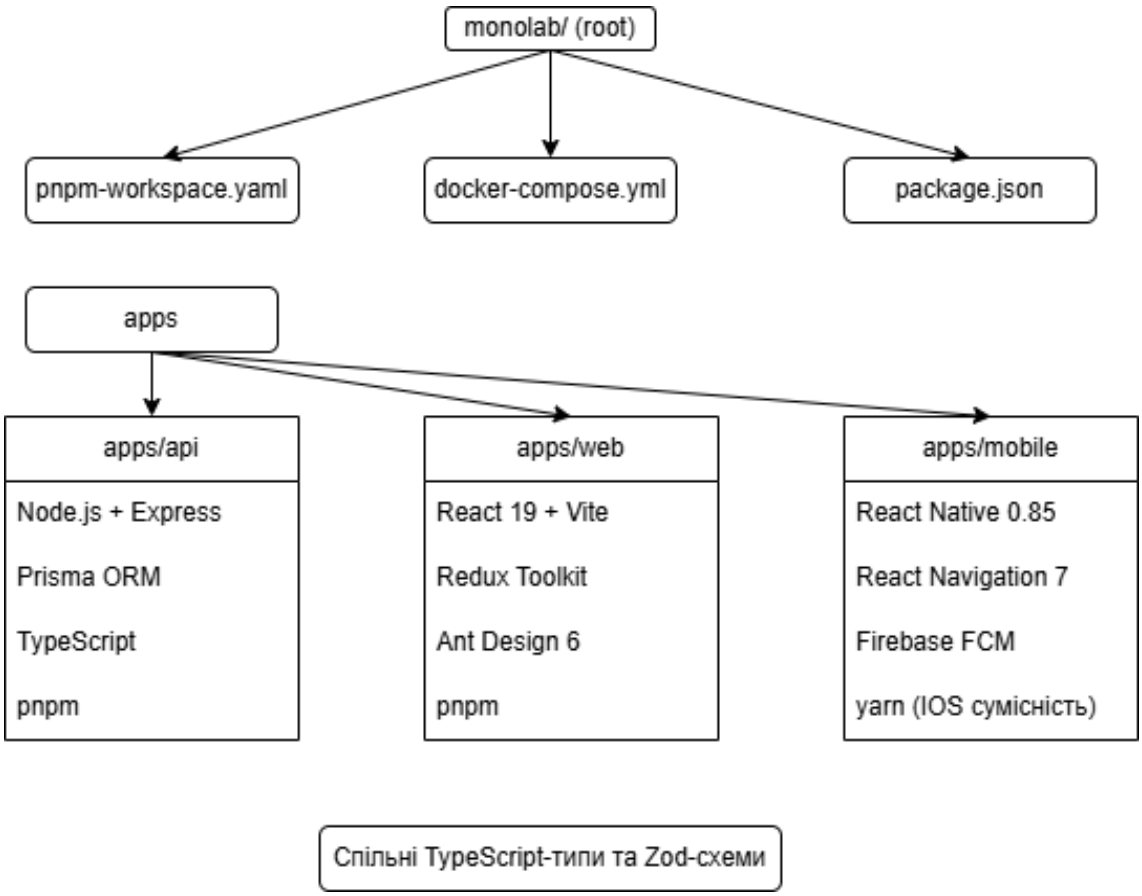


Рис. 2.1 Структура монорепозиторію Monolab

2.2 Вибір мови програмування та серверного фреймворку

Для всіх трьох рівнів системи (сервер, веб, мобільний) обрано TypeScript [8] – статично типізоване надмножинство JavaScript, що компілюється у звичайний JavaScript. Основна причина вибору – практичний досвід роботи автора з цією мовою як у навчальних проєктах університету, так і в комерційній розробці.

TypeScript помітно спрощує підтримку кодової бази середнього та великого розміру завдяки системі типів: компілятор виявляє помилки ще до запуску програми, IDE надає точне автодоповнення, а при рефакторингу зміна інтерфейсу автоматично підсвічує всі місця, де потрібні виправлення.

Використання однієї мови для серверу, веб-клієнта та мобільного додатку є принциповою перевагою обраного підходу. Моделі даних (наприклад, інтерфейс User, enum Role, тип AppointmentStatus) описуються один раз і перевикористовуються у всіх підпроєктах. Zod-схеми валідації, створені для серверних ендпоінтів, можуть застосовуватися і на клієнті для валідації форм, що виключає розбіжності між клієнтською та серверною валідацією [23].

Для серверної частини обрано Node.js [9] з фреймворком Express [10]. Node.js побудований на рушії V8 від Google і використовує однопоточну модель з асинхронним введенням-виведенням (event loop). Такий підхід ефективний для I/O-інтенсивних задач, характерних для API медичної лабораторії: більшість операцій зводяться до запитів до бази даних, читання/запису файлів та відправки HTTP-відповідей. Express, у свою чергу, є мінімалістичним фреймворком, що надає розробнику повний контроль над структурою проєкту та middleware-ланцюжком, не нав'язуючи жорстку архітектуру (на відміну від NestJS або Fastify) [10].

Розглядалися альтернативні варіанти серверного стеку. Python з Django/FastAPI є популярним вибором для медичних систем, проте потребував би окремої мови та набору інструментів, що унеможливило б перевикористання типів із клієнтською частиною. Java зі Spring Boot забезпечує надійну типізацію, але має значно вищий поріг входу та потребує більше часу на конфігурацію. TypeScript з Node.js забезпечує баланс між типобезпекою та швидкістю розробки, що є визначальним фактором для дипломного проєкту з обмеженими часовими ресурсами.

Таблиця 2.1

Порівняння мов програмування для серверної частини

Критерії	TypeScript + Node.js	Python + Django	Java + Spring Boot
Типізація	Статична	Динамічна	Статична
Спільні типи з фронтендом	+	-	-
Швидкість розробки	Висока	Висока	Середня
Екосистема для веб/мобайл	React, RN	Окрема мова	Окрема мова
Асинхронність	Event loop	async/await	Потоки
Власний досвід	Комерційний	Навчальний	Навчальний

2.3 Реалізація безпеки системи

Безпека медичної інформаційної системи є першочерговою вимогою, оскільки система оперує персональними даними пацієнтів, що підпадають під дію законодавства про захист персональних даних у сфері охорони здоров'я [5; 6]. У системі Monolab реалізовано багаторівневий підхід до безпеки, що охоплює автентифікацію, авторизацію, захист від поширених веб-атак та валідацію даних.

Автентифікація побудована на протоколі JWT (JSON Web Token) [18] з парою токенів: access-токен з терміном дії 1 година та refresh-токен з терміном 30 днів. Access-токен передається у заголовку Authorization кожного HTTP-запиту та містить ідентифікатор користувача та його роль. При закінченні терміну дії access-токену клієнт автоматично надсилає refresh-токен для отримання нової пари – цей механізм реалізований через Axios interceptor на веб-клієнті та аналогічний interceptor у мобільному додатку.

Паролі зберігаються у базі даних виключно у хешованому вигляді. Для хешування використовується алгоритм bcrypt із параметром salt rounds = 12, що забезпечує стійкість до brute-force атак та унеможливлює відновлення паролю навіть при повному витоку бази даних (на відміну від MD5 або SHA-256 без salt, де можливі rainbow table атаки) [6].

Авторизація реалізована за моделлю RBAC (Role-Based Access Control) із трьома ролями. PATIENT (пацієнт) має доступ до власних результатів, запису на аналізи та профілю. LAB_TECH (лаборант) має доступ до черги зразків, введення та верифікації результатів. ADMIN (адміністратор) має повний доступ до всіх ресурсів системи, включаючи управління користувачами та аудит-лог. Принципова особливість: самостійна реєстрація доступна лише для ролі PATIENT; облікові записи лаборантів та адміністраторів створюються виключно через адміністративну панель, що унеможливлює несанкціоноване підвищення привілеїв.

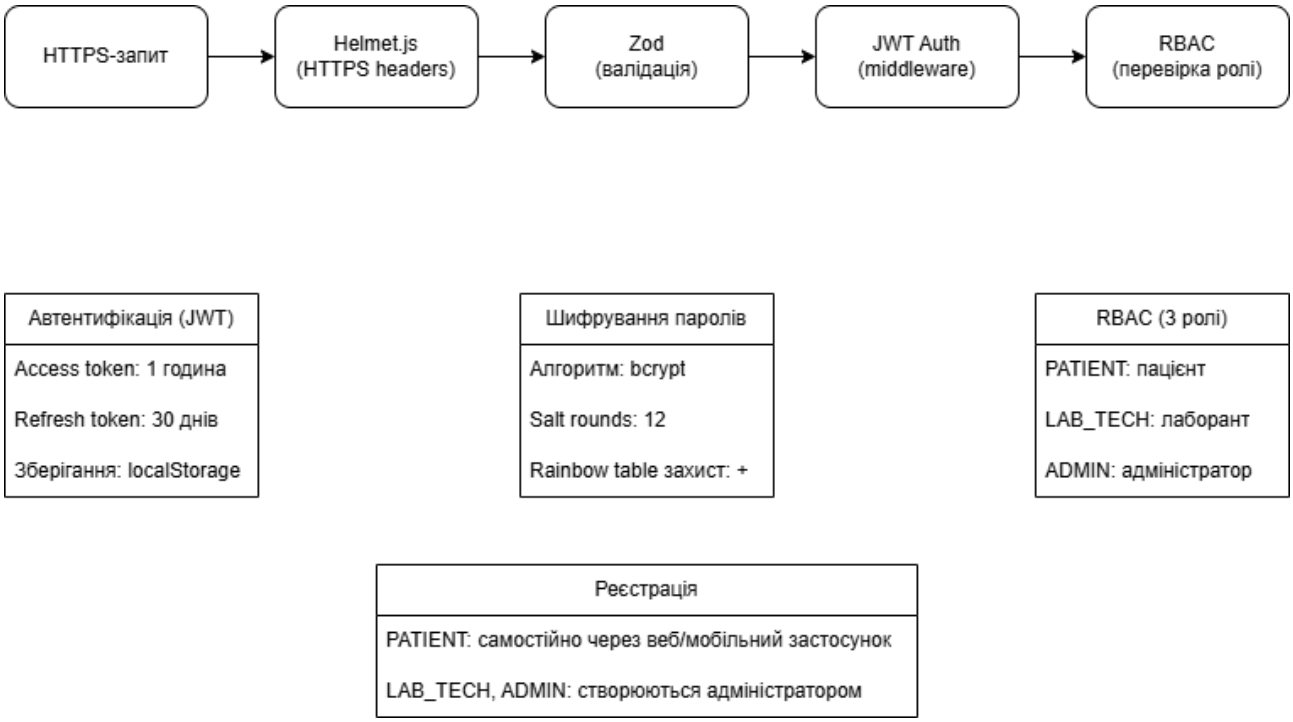


Рис. 2.2 Архітектура безпеки системи Monolab

Додаткові рівні захисту: бібліотека Helmet.js [27] встановлює безпечні HTTP-заголовки (Content-Security-Policy, X-Frame-Options, X-Content-Type-Options, Strict-Transport-Security), що захищають від XSS, clickjacking та MIME-sniffing атак. Валідація вхідних даних на кожному ендпоінті реалізована через Zod-схеми [23] – декларативний валідатор, що перевіряє типи, формати, діапазони та обов'язковість полів ще до потрапляння даних у бізнес-логіку.

2.4 Клієнтська частина веб-застосунку

Веб-кабінет Monolab побудований на бібліотеці React 19 [11] зі збирачем Vite. React обрано з кількох причин: по-перше, це основна бібліотека, з якою автор працює як у навчальних проєктах, так і в комерційній розробці; по-друге, React має найбільшу екосистему серед фронтенд-рішень (бібліотеки компонентів, інструменти стану, маршрутизатори); по-третє, знання React безпосередньо переносяться на React Native для мобільного додатку - компонентна модель, хуки та паттерни управління станом ідентичні.

Для збирання проєкту обрано Vite замість традиційного Create React App (CRA). Vite використовує нативні ES-модулі браузера під час розробки, що забезпечує миттєвий старт dev-сервера та оновлення модулів (HMR) за мілісекунди – на відміну від CRA на базі webpack, де час холодного старту на проєктах середнього розміру сягає 10-30 секунд.

Управління станом застосунку реалізовано через Redux Toolkit [15] – офіційний рекомендований спосіб використання Redux. RTK спрощує створення store, reducers та async thunks, автоматично інтегрується з TypeScript та надає RTK Query для кешування серверних відповідей. У контексті Monolab Redux зберігає: дані авторизованого користувача та JWT-токени, кешовані результати аналізів, стан активних форм (запис на аналіз, введення результатів), повідомлення та статуси замовлень.

Як бібліотеку UI-компонентів обрано Ant Design 6 [16] – зрілу бібліотеку з понад 60 готовими компонентами (таблиці, форми, модальні вікна, кроки, графіки завантаження тощо). Ant Design забезпечує професійний вигляд інтерфейсу, вбудовану адаптивність, підтримку темізації та інтернаціоналізації. Для відображення графіків динаміки лабораторних показників використовується Recharts [22] – React-обгортка над D3.js, що забезпечує декларативний підхід до побудови лінійних діаграм з інтерактивними tooltip та анімаціями.

Комунікація з сервером реалізована через бібліотеку Axios з налаштованим interceptor, що автоматично додає JWT-токен до кожного запиту та оновлює його

при отриманні відповіді 401. Для real-time оновлень (зміна статусу замовлення) використовується WebSocket-з'єднання, ініціалізоване при авторизації.

2.5 Мобільна платформа та інструменти

Мобільний додаток Monolab розроблений на React Native 0.85 [12] – найновішому stable-релізі на момент завершення роботи (квітень 2026). Використовується React Native CLI (bare workflow), а не Expo. Причин дві: автор має комерційний досвід роботи саме з CLI-підходом, і CLI надає повний контроль над нативним кодом обох платформ (Android та iOS), що критично для інтеграції з Firebase Cloud Messaging [17], камерою та файловою системою.

Expo спрощує старт розробки, але має обмеження при роботі з нативними модулями поза стандартним Expo SDK. Для медичного додатку, де потрібна глибока інтеграція з push-повідомленнями (foreground та background), камерою (сканер штрих-кодів) та файловим менеджером, ці обмеження є критичними.

Версія React Native 0.85 працює з увімкненою New Architecture: Fabric (новий рендерер) замість старого UIManager та TurboModules (синхронні нативні модулі) у Bridgeless-режимі. Старий JS – Native bridge більше не використовується на критичних шляхах, що зменшує лаг серіалізації та підвищує продуктивність. Разом із RN 0.85 у проєкті використовується React 19, Reanimated 4.3 з виокремленим пакетом react-native-worklets 0.8 та Gesture Handler 2.31.

У ході розробки було замінено кілька deprecated-бібліотек на актуальні аналоги, сумісні з New Architecture. react-native-camera замінено на react-native-vision-camera v4.7, оскільки перша бібліотека давно не підтримується, а vision-camera є поточним стандартом для роботи з камерою у React Native. Для графіків замість react-native-svg-charts (unmaintained) обрано react-native-gifted-charts v1.4, яка активно розвивається. Модальні вікна (react-native-modalize) замінено на @gorhom/bottom-sheet v5.1, що є Fabric-native. PDF-перегляд реалізовано через WebView (react-native-webview) замість react-native-pdf, який не сумісний з Fabric.

Загалом видалено 14 зайвих залежностей (moment, d3-shape, react-native-fs тощо), замінених на нативні альтернативи.

Таблиця 2.2

Замінені deprecated-бібліотеки при міграції на New Architecture

Було(deprecated)	Стало	Причина зміни
react-native-camera	vision-camera v4.7	camera deprecated, vision-camera – стандарт
react-native-svg-charts	gifted-charts v1.4	svg-charts unmaintained
react-native-modalize	@gorhom/bottom-sheet v5.1	modalize unmaintained, bottom-sheet – Fabric-native
react-native-pdf	WebView (react-native-webview)	PDF-рендерер не сумісний з Fabric
react-native-splash-screen	react-native-bootsplash v6.3	Не підтримує New Architecture
react-native-document-picker	@react-native-documents/picker v10.1	Новий scoped пакет від того ж автора

Навігація реалізована через React Navigation 7 [24] з комбінацією трьох навігаторів: Stack Navigator для послідовних переходів, Bottom Tab Navigator для основних розділів (головна, сповіщення, профіль) та Stack всередині кожної вкладки. Захист маршрутів побудований через кастомний AuthGuard, що перевіряє JWT-токен перед кожним захищеним екраном.

Push-повідомлення інтегровані через Firebase Cloud Messaging (@react-native-firebase/messaging v23.8) [17]. Серверна частина надсилає повідомлення при зміні статусу замовлення, готовності результатів та системних оповіщеннях. Додаток підтримує foreground (in-app банер через Notifee) та background (системне повідомлення). Токен пристрою реєструється при першому запуску та оновлюється при кожній авторизації.

Загалом у ході міграції видалено 14 зайвих залежностей (moment, d3-shape, react-native-fs тощо), які було замінено на нативні альтернативи або видалено як непотрібні. Firebase SDK (@react-native-firebase/app та /messaging) оновлено до версії 23.8, повністю сумісної з New Architecture.

2.6 Робота з базою даних

Як систему управління базами даних обрано PostgreSQL 16 [13]. Цей вибір обумовлений насамперед практичним досвідом автора: усі попередні проєкти (навчальні та комерційні) побудовані на PostgreSQL, що дозволяє ефективно використовувати її можливості без додаткового часу на вивчення. Серед технічних переваг: підтримка складних типів даних (JSON/JSONB для зберігання метаданих), повнотекстовий пошук, надійна система транзакцій (ACID), розвинена індексація (B-tree, GIN, GiST) та зрілий оптимізатор запитів.

Порівняно з MySQL, PostgreSQL пропонує більш гнучку роботу з JSON-даними (що використовується для зберігання метаданих аудит-логу та налаштувань користувачів), кращу підтримку складних запитів із підзапитами та оконними функціями, а також суворіше дотримання стандарту SQL. У порівнянні з MongoDB (документна NoSQL база), PostgreSQL забезпечує цілісність зв'язків між таблицями через зовнішні ключі, що критично для медичної системи, де зв'язок між пацієнтом, замовленням, зразком та результатом має бути гарантовано коректним [13].

Доступ до бази даних реалізований через Prisma ORM [14] – сучасний ORM для TypeScript та Node.js. Prisma генерує типобезпечний клієнт на основі декларативної схеми (файл schema.prisma), де кожна модель описується з полями, типами та зв'язками. Компілятор TypeScript перевіряє коректність запитів до бази ще на етапі написання коду – помилка у назві поля або неправильний тип фільтра виявляється до запуску програми. Prisma-схема Monolab містить понад 30 моделей та 850+ рядків, що охоплюють сутності користувачів, тестів, категорій, замовлень, зразків, результатів, платежів, сповіщень, аудит-записів та файлів.

Для зберігання файлів (PDF-звітів лабораторних досліджень, аватарів користувачів) використовується MinIO [19] – self-hosted об'єктне сховище, повністю сумісне з API Amazon S3. MinIO обрано замість прямого збереження файлів у файловій системі серверу з кількох причин: масштабованість (файлове сховище може бути на окремому сервері), стандартизований API (при міграції на

хмарне S3 зміни мінімальні), контроль доступу на рівні бакетів та об'єктів. Інтеграція здійснюється через офіційний JavaScript SDK (@minio/minio-js).

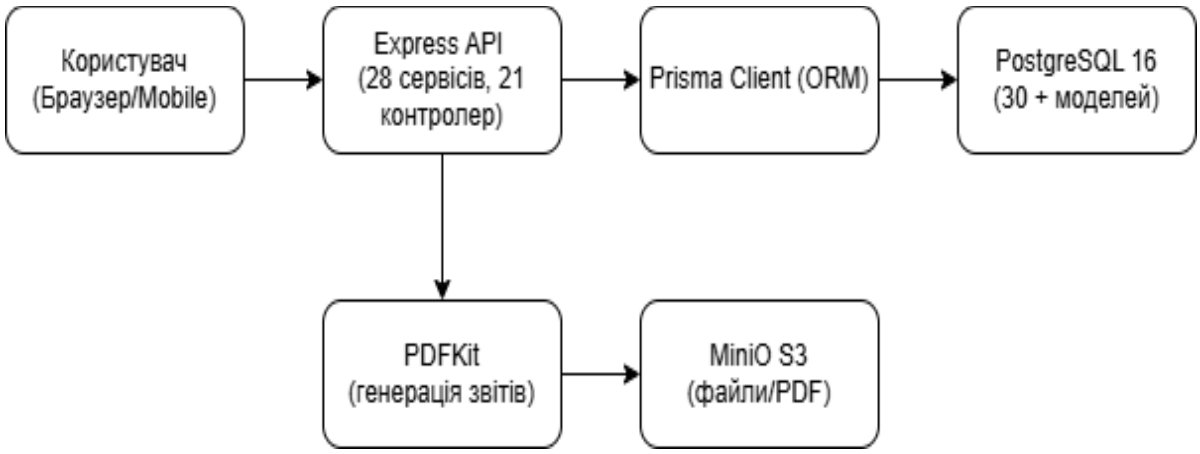


Рис. 2.3 Взаємодія компонентів з базою даних

Таблиця 2.3

Технологічний стек системи Monolab

Рівень	Технологія	Призначення та обгрунтування
Backend	Node,js 20 LTS	Середовище виконання JavaScript на сервері [9]
Backend	Node,js 20 LTS	Мінімалістичний HTTP-фреймворк [10]
Backend	Prisma ORM	Типобезпечний ORM з генерацією клієнта [14]
Backend	PostgreSQL 16	Реляційна СУБД з JSON та повнотекстовим пошуком [13]
Backend	JWT(jsonwebtoken)	Автентифікація з access/refresh токенами [18]
Backend	bcrypt	Хешування паролів (salt rounds: 12)
Backend	Zod	Декларативна валідація вхідних даних [23]
Backend	Helmet.js	Безпечні HTTP-заголовки (CSP, X-Frame-Options) [27]
Backend	PDFKit	Програмна генерація PDF-звітів [21]
Backend	MiniO S3	Self-hosted об'єктне сховище для файлів [19]
Backend	Nodemailer	Відправка email-повідомлень
Backend	Socket.IO	WebSocket для real-time оновлень
Web	React 19	Бібліотека для побудови UI [11]

Рівень	Технологія	Призначення та обгрунтування
Web	Vite	Збирач з миттєвим HMR
Web	Redux Toolkit	Управління станом + RTK Query [15]
Web	Ant Design 6	Бібліотека UI-компонентів (60+ компонентів) [16]
Web	Recharts	Лінійні графіки динаміки показників [22]
Web	Axios	HTTP-клієнт з interceptors
Mobile	React Native 0.85	Кросплатформний фреймворк, New Architecture (Fabric) [12]
Mobile	React Navigation 7	Stack + Tab навігація [24]
Mobile	Firebase FCM v23.8	Push-повідомлення (foreground + background) [17]
Mobile	vision-camera v4.7	Сканер штрих-кодів зразків (замість deprecated camera)
Mobile	Reanimated 4.3	Анімації на UI-потоці (worklets)
Mobile	gifted-charts v1.4	Графіки динаміки (замість svg-charts)
Mobile	@gorhom/bottom-sheet v5.1	Модальні вікна (Fabric-native)
DevOps	Docker Compose	Контейнеризація PostgreSQL + MinIO [20]
DevOps	pnpm + yarn	pnpm для API/web, yarn для mobile (iOS)
DevOps	ESLint + Prettier	Статичний аналіз та форматування коду
DevOps	Git + GitHub	Контроль версій та хостинг репозиторію

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Загальна архітектура програмного забезпечення

Серверна частина Monolab побудована за трирівневою архітектурою, де кожен рівень має чітко визначену зону відповідальності. На вхідному рівні розташовані контролери (21 файл), які приймають HTTP-запити, здійснюють первинну маршрутизацію та формують відповіді. Другий рівень – сервіси (28 файлів), де зосереджена вся бізнес-логіка: обробка замовлень, верифікація результатів, генерація PDF-звітів, відправка повідомлень. Третій рівень – Prisma Client, який забезпечує типобезпечний доступ до PostgreSQL 16 без написання SQL-запитів вручну [14].

Між клієнтом та контролерами розташований ланцюжок middleware, що послідовно обробляє кожен запит. Першим спрацьовує Helmet.js [27], встановлюючи безпечні HTTP-заголовки. Далі Zod-схема [23] валідує тіло запиту, параметри та query-рядок – якщо дані не відповідають схемі, запит відхиляється з кодом 400 ще до потрапляння у контролер. Наступним кроком JWT-middleware перевіряє access-токен [18], а RBAC-middleware звіряє роль користувача з дозволеними для даного маршруту. Замикає ланцюжок централізований обробник помилок (Error Handler), який перехоплює виключення з будь-якого рівня та повертає клієнту структуровану відповідь з відповідним HTTP-кодом.

Такий підхід – Controller, Service, Prisma Client – обрано свідомо замість більш складного чотирирівневого варіанту з окремим Repository-шаром. Prisma Client вже забезпечує абстракцію над SQL та типобезпеку запитів, тому додатковий Repository став би дублюванням без суттєвої переваги. У разі потреби міграції на іншу ORM (наприклад, TypeORM або Drizzle) достатньо буде змінити виклики Prisma у сервісах – контролери залишаться незмінними [14].

Комунікація між клієнтами (веб та мобільний) та сервером відбувається через два канали. Основний – REST API через HTTP/HTTPS для усіх CRUD-операцій

(створення замовлення, введення результатів, отримання даних). Допоміжний – WebSocket-з'єднання через Socket.IO для real-time оновлень: коли лаборант змінює статус замовлення або публікує результат, пацієнт бачить зміну у веб-кабінеті миттєво, без перезавантаження сторінки. Push-повідомлення для мобільного додатку доставляються через Firebase Cloud Messaging [17] – окремий канал, не пов'язаний з WebSocket.

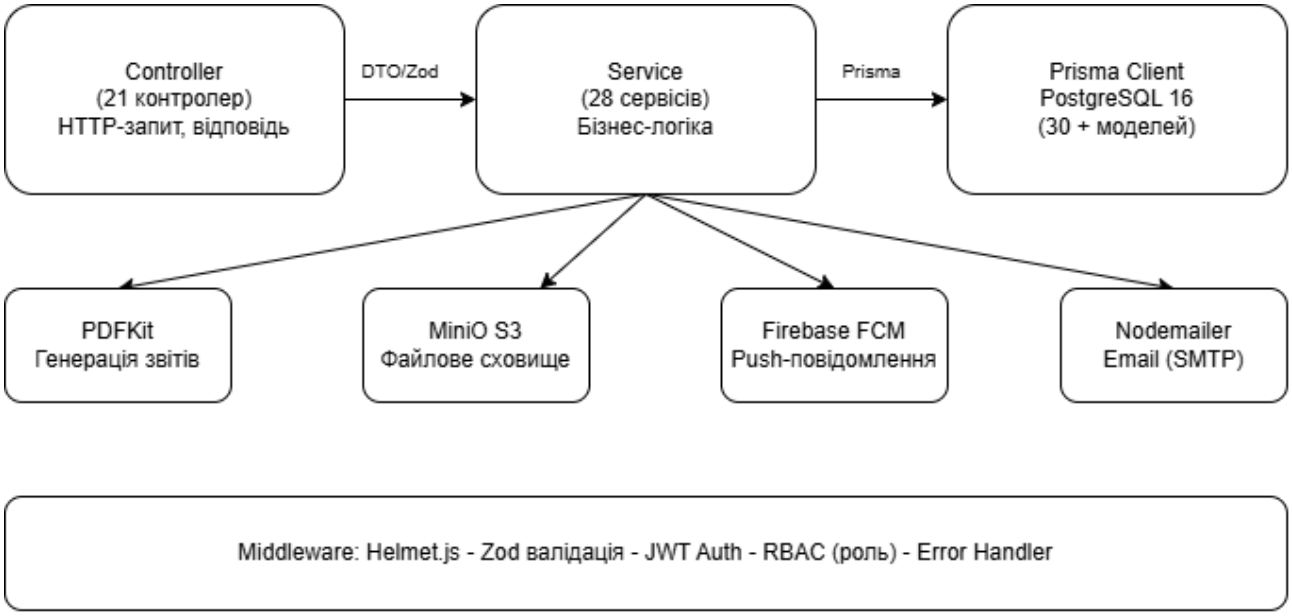


Рис. 3.1 Трирівнева архітектура серверної частини Monolab

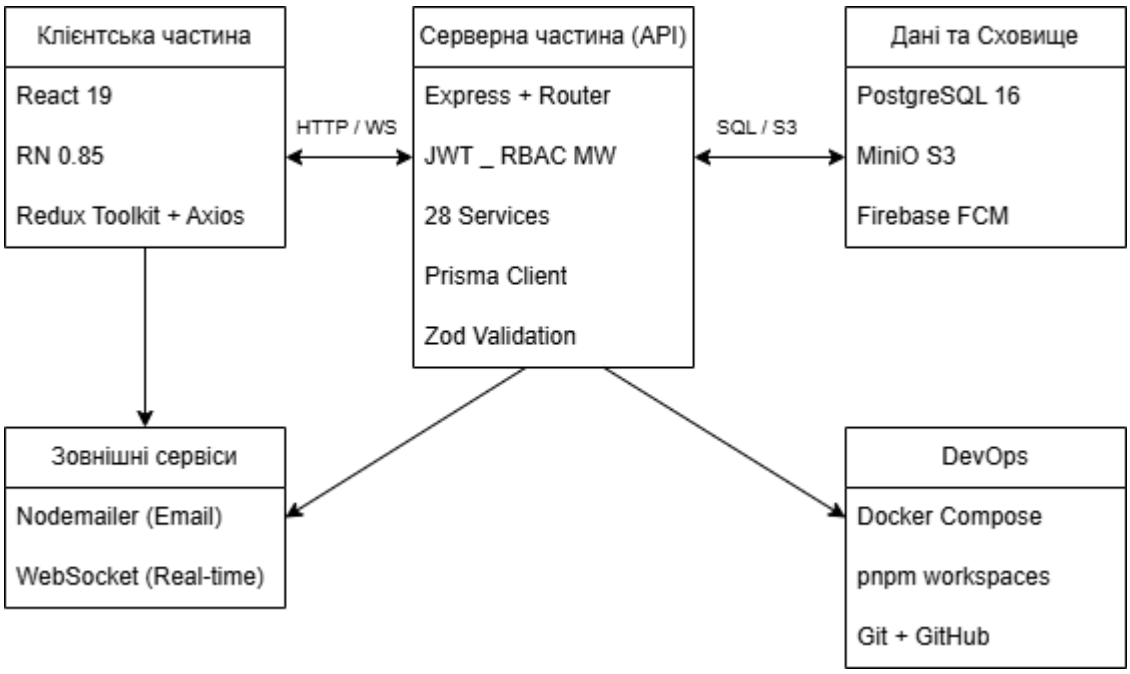


Рис. 3.2 Загальна архітектура системи Monolab

3.2 Сценарії взаємодії пацієнта із системою

Пацієнт є основним споживачем послуг системи і має найширший спектр доступних дій. Реєстрація доступна через веб-кабінет або мобільний додаток: пацієнт вводить ім'я, email, пароль та підтверджує електронну адресу через код, надісланий на пошту (TTL коду – 10 хвилин, максимум 5 спроб). Після авторизації пацієнт потрапляє до персонального кабінету.

Процес створення замовлення побудований за принципом wizard (покрокова форма). На першому кроці пацієнт обирає відділення лабораторії. На другому – обирає конкретні дослідження з каталогу або готові панелі (наприклад, «Загальний аналіз крові», «Гормони щитоподібної залози»). Система автоматично розраховує загальну вартість. На третьому кроці пацієнт бронює зручний слот – система генерує доступні часові проміжки на основі розкладу відділення (BranchScheduleRule + BranchScheduleConfig) з урахуванням вихідних та свят (BranchScheduleException). На четвертому кроці – оплата: система підтримує імітацію платіжного процесу з інтерфейсом вибору способу оплати (Google Pay, Apple Pay, ручне введення картки). Архітектура PaymentService підготовлена для підключення реальних платіжних провайдерів (LiqPay, Easypay) через абстракцію платіжного шлюзу.

Після оплати замовлення отримує статус CONFIRMED, а запис (Appointment) переходить у стан, що очікує на візит. При неоплаті протягом доби спрацьовує планувальник (scheduler.service.ts), який автоматично скасовує прострочені записи та замовлення [9].

Перегляд результатів реалізовано через список із кольоровою індикацією: зелений – значення у межах норми, червоний – відхилення, сірий – результат очікується. Референсні діапазони підбираються автоматично з урахуванням віку та статі пацієнта через refRange.service.ts. Деталізація результату включає: назву дослідження, числове значення, одиницю вимірювання, нижню та верхню межі норми, дату виконання, ПІБ лаборанта.

Модуль динаміки показників (trends.service.ts на сервері, PatientDynamicsPage на вебі, PatientDynamicsScreen у мобільному) дозволяє пацієнту обрати конкретне дослідження та побачити лінійний графік зміни значень за всі дати з горизонтальними лініями верхньої та нижньої меж норми. На вебі використовується Recharts [22], у мобільному додатку – react-native-gifted-charts v1.4 [12].

Додатково пацієнт має доступ до: журналу активності (patient-activity.service.ts – хронологія дій), списку сповіщень із лічильником непрочитаних, управління профілем (персональні дані, дата народження, стать – останні два поля впливають на підбір референсних діапазонів), завантаження PDF-звіту з MinIO та його передачі через системний Share.

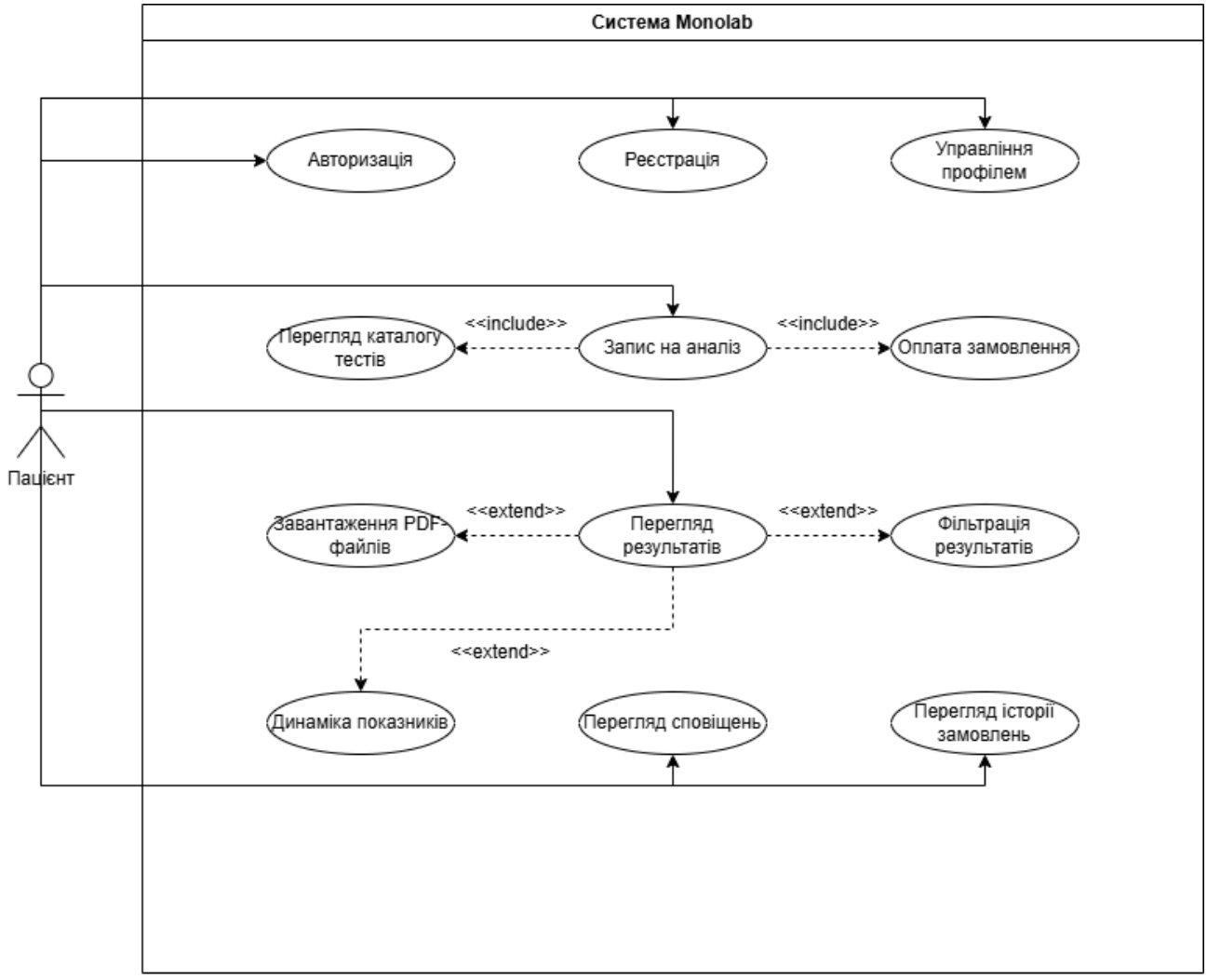


Рис. 3.3 Діаграма варіантів використання актора «Пацієнт»

3.3 Сценарій взаємодії лаборанта із системою

Лаборант (роль LAB_TECH) працює виключно через веб-кабінет і має доступ до операційних процесів лабораторії. Обліковий запис лаборанта створюється адміністратором – самостійна реєстрація для цієї ролі заблокована на рівні AuthService.

Головна сторінка лаборанта (LabHomePage) відображає зведену статистику: кількість записів на поточний день, кількість зразків у роботі, кількість результатів, що очікують верифікації. Список записів (LabAppointmentsPage) відсортований за часом слоту та дозволяє переглядати деталі кожного запису через drawer-панель (LabAppointmentDrawer).

Прийом зразка фіксується зміною статусу замовлення на SAMPLED, при цьому автоматично створюється запис Sample зі статусом CREATED та унікальним штрих-кодом. Лаборант переводить зразок через стадії: CREATED, COLLECTED (зразок прийнятий), IN_LAB (передано до лабораторії), PROCESSED (дослідження виконано). У разі непридатного зразка – статус REJECTED з обов'язковим коментарем причини.

Введення результатів здійснюється через форму, де для кожного тесту у замовленні лаборант вводить числове значення. Форма валідується Zod-схемою [23]: перевіряється тип даних (число), діапазон допустимих значень, обов'язковість полів. Результат зберігається зі статусом DRAFT.

Верифікація – перевірка введених даних перед публікацією. Лаборант із відповідними правами (або інший лаборант, якщо працює модель подвійної перевірки) переглядає значення, порівнює з референсними діапазонами та підтверджує коректність. Статус змінюється на COMPLETED, а потім на VERIFIED. Після верифікації система автоматично запускає ланцюжок дій: PDFService генерує звіт через PDFKit [21], файл завантажується у MinIO S3 [19], пацієнт отримує push-повідомлення через FCM [17] та email через Nodemailer, статус замовлення змінюється на RESULTS_READY.

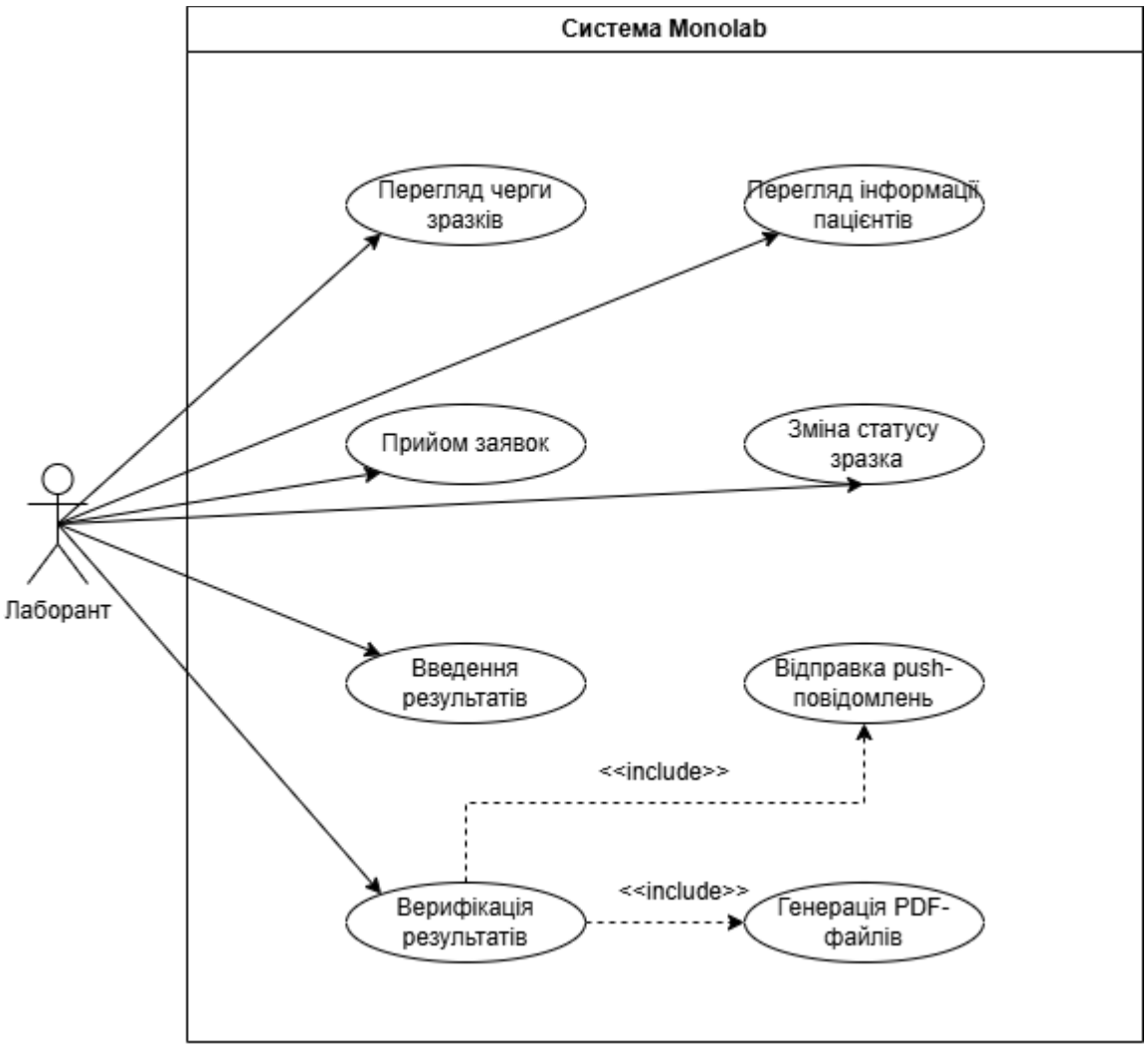


Рис. 3.4 Діаграма варіантів використання актора «Лаборант»

3.4 Сценарій взаємодії адміністратора із системою

Адміністратор (роль ADMIN) має повний доступ до всіх ресурсів системи через веб-кабінет. Обліковий запис адміністратора, як і лаборанта, створюється лише іншим адміністратором через admin.users.service.ts.

Управління каталогом тестів включає створення, редагування та видалення досліджень (tests.service.ts) та панелей тестів (panels.service.ts). Кожен тест містить: код, назву, опис, одиницю вимірювання, ціну, категорію та набір референсних діапазонів (testRefRanges.service.ts), що залежать від віку та статі пацієнта.

Управління відділеннями (branches.service.ts) охоплює створення та редагування локацій з адресою, контактами та графіком роботи. Розклад кожного відділення конфігурується через BranchScheduleRule (правила: робочі дні, час

початку та завершення) та BranchScheduleConfig (крок між слотами, тривалість слоту, буферний час). Для святкових та вихідних днів створюються BranchScheduleException.

Дашборд адміністратора (admin.dashboard.service.ts) відображає ключові метрики: загальна кількість замовлень за обраний період, сума оплат, кількість виконаних досліджень, середній час від створення замовлення до видачі результату, кількість активних пацієнтів.

Аудит-лог (audit.service.ts) фіксує кожну критичну дію в системі: хто (actorId), що зробив (action), над якою сутністю (entity, entityId), коли (timestamp), звідки (IP-адреса, User-Agent, request ID). Адміністратор може фільтрувати лог за типом дії, сутністю, користувачем та часовим діапазоном. Це забезпечує повну прослідковуваність операцій відповідно до вимог ISO 15189 для медичних лабораторій [5; 6].

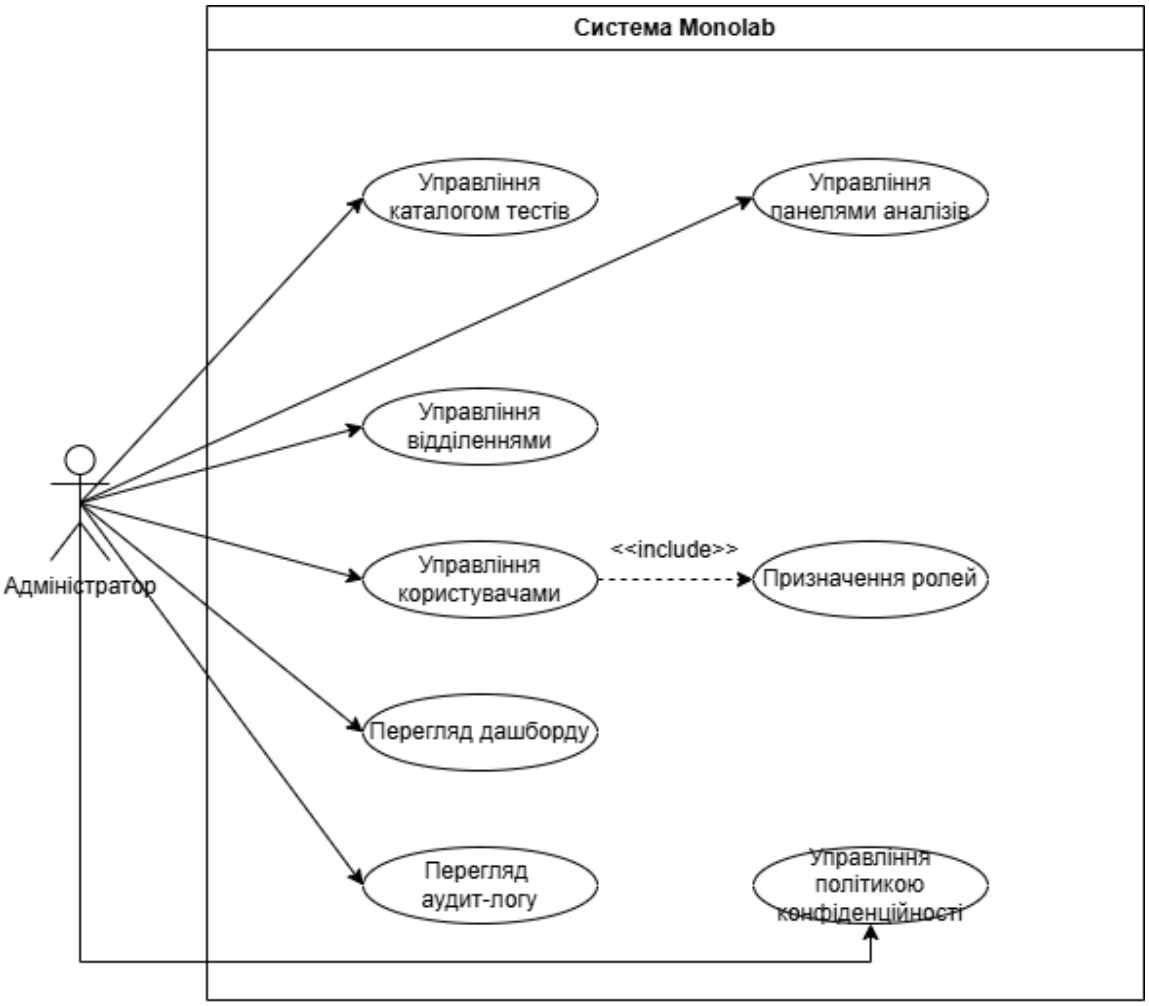


Рис. 3.5 Діаграма варіантів використання актора «Адміністратор»

3.5 Структура серверних сервісів та контролерів

Серверна частина складається з 28 сервісів та 21 контролера, організованих за функціональними доменами. Кожен сервіс інкапсулює бізнес-логіку відповідного модуля та взаємодіє з PostgreSQL через Prisma Client [14]. Контролер відповідає лише за HTTP-рівень: парсинг запиту, виклик сервісу та формування відповіді.

Таблиця 3.1

Повний перелік серверних сервісів

Сервіс	Призначення
auth.service	Реєстрація, логін, верифікація email, скидання паролю, JWT [18]
appointments.service	Бронювання слотів, управління статусами, створення зразків
orders.service	CRUD замовлень, підрахунок вартості, прив'язка тестів/панелей
payments.service	CRUD платежів, імітація платіжного процесу, генерація квитанцій
results.service	Введення результатів лаборантом, верифікація, зміна статусів
results-pdf.service	Генерація PDF результатів через PDFKit [21]
pdf.service	Генерація PDF рахунків та квитанцій
invoices.service	Управління рахунками (invoices)
branches.service	CRUD відділень (локацій)
branchSchedule.service	Розклад роботи відділень + винятки (свята)
tests.service	CRUD каталогу лабораторних досліджень
panels.service	CRUD панелей тестів (групи досліджень)
testRefRanges.service	CRUD референсних діапазонів для тестів
refRange.service	Підбір діапазону за віком та статтю пацієнта
patient.service	Профіль пацієнта, медична історія
patient-activity.service	Журнал активності пацієнта (хронологія дій)
trends.service	Аналіз динаміки показників у часі [22]

Сервіс	Призначення
notifications.service	CRUD сповіщень (in-app, email)
push.service	Firebase FCM push-повідомлення [17]
email.service	Надсилання email через Nodemailer (SMTP)
files.service	Завантаження/отримання файлів через MinIO S3 [19]
admin.users.service	Управління користувачами (створення LAB_TECH/ADMIN)
admin.dashboard.service	Статистика та метрики для дашборду
audit.service	Логування системи, логування критичних дій (аудит-лог) [5]
policies.service	Управління Політикою конфіденційності та consent
profile.service	Управління профілем (дані, аватар, налаштування)
documentNumbers.service	Послідовна нумерація (замовлення/рахунки/результати)
scheduler.service	Щоденні фонові задачі (автоскасування прострочених записів)

Планувальник задач (scheduler.service.ts) реалізовано без зовнішніх cron-бібліотек: використовується setTimeout з timezone-aware розрахунком часу наступного запуску (щоденно о 23:30 UTC). Дві задачі: autoExpireAppointments() автоматично скасовує записи зі статусом CREATED або CONFIRMED, якщо час слоту вже минув, та autoExpireOrders() скасовує неоплачені замовлення, пов'язані зі скасованими записами. Кожне автоскасування фіксується в аудит-лозі.

Таблиця 3.2

Кількісні показники серверної частини

Показник	Значення
Сервіси	28
Контролери	21
Файли маршрутів	21
Моделі Prisma	30+
Рядки Prisma-schema	850+
REST-ендпоінти	88

Показник	Значення
Middleware компонентів	8
Веб-сторінок	31
Мобільних екранів	34
Мобільних компонентів	62
SVG-іконок	100+

3.6 Життєвий цикл сутностей та структура бази даних

Ключовою особливістю архітектури Monolab є чітко визначені життєві цикли для трьох основних сутностей: запису (Appointment), зразка (Sample) та результату (Result). Кожна сутність має набір статусів, між якими можливі лише дозволені переходи – це забезпечується на рівні сервісу через перевірку поточного стану перед зміною.

1. Життєвий цикл запису (Appointment)

Запис проходить п'ять станів. CREATED – пацієнт створив замовлення з записом. CONFIRMED – адміністратор або система підтвердила запис (після оплати). SAMPLED – лаборант зафіксував прийом зразка. RESULTS_READY – результати верифіковані та опубліковані. CANCELLED – запис скасовано (можливо з будь-якого статусу крім RESULTS_READY, а також автоматично планувальником для прострочених записів).

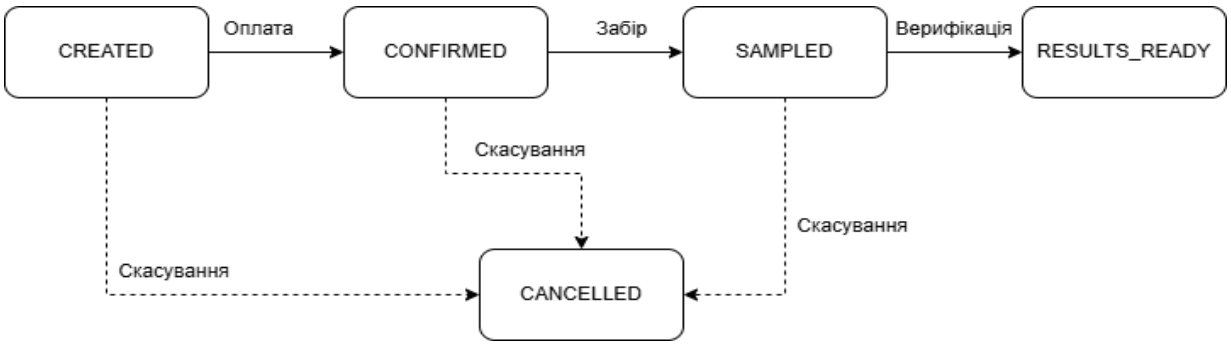


Рис. 3.6 Діаграма переходів статусів запису (Appointment)

2. Життєвий цикл зразка (Sample)

Зразок створюється автоматично при зміні статусу запису на SAMPLED. Послідовність: CREATED, COLLECTED (зразок прийнятий медсестрою), IN_LAB (передано до лабораторії), PROCESSED (дослідження виконано). Якщо зразок непридатний – REJECTED з обов'язковим коментарем.

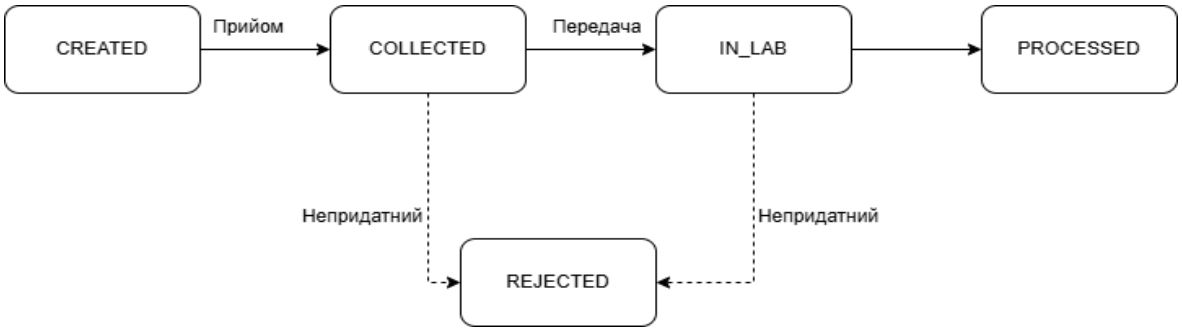


Рис. 3.7 Діаграма переходів статусів зразка (Sample)

3. Життєвий цикл результату (Result)

Результат створюється лаборантом зі статусом DRAFT. Після заповнення всіх значень лаборант переводить його у COMPLETED. Верифікація (іншим лаборантом або старшим фахівцем) фіналізує результат зі статусом VERIFIED. Після верифікації система автоматично генерує PDF, зберігає у MinIO та надсилає push + email пацієнту.

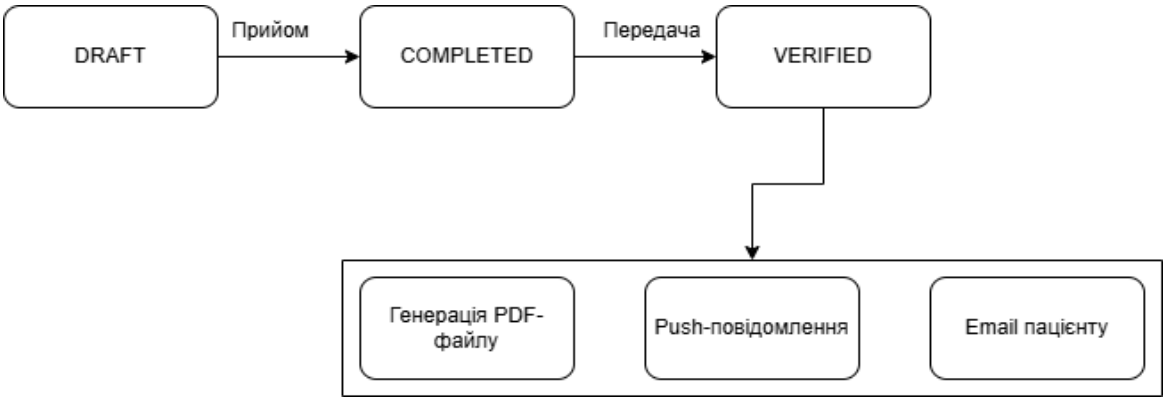


Рис. 3.8 Діаграма переходів статусів результату (Result)

4. Повний операційний цикл

Шість кроків повного циклу об'єднують три ролі: пацієнт (кроки 1-3: замовлення, запис, оплата), лаборант (кроки 4-5: зразок, результат) та система (крок 6: автоматична генерація PDF, push, email). Перехід між кроками контролюється статусами відповідних сутностей – пропуск етапу неможливий.

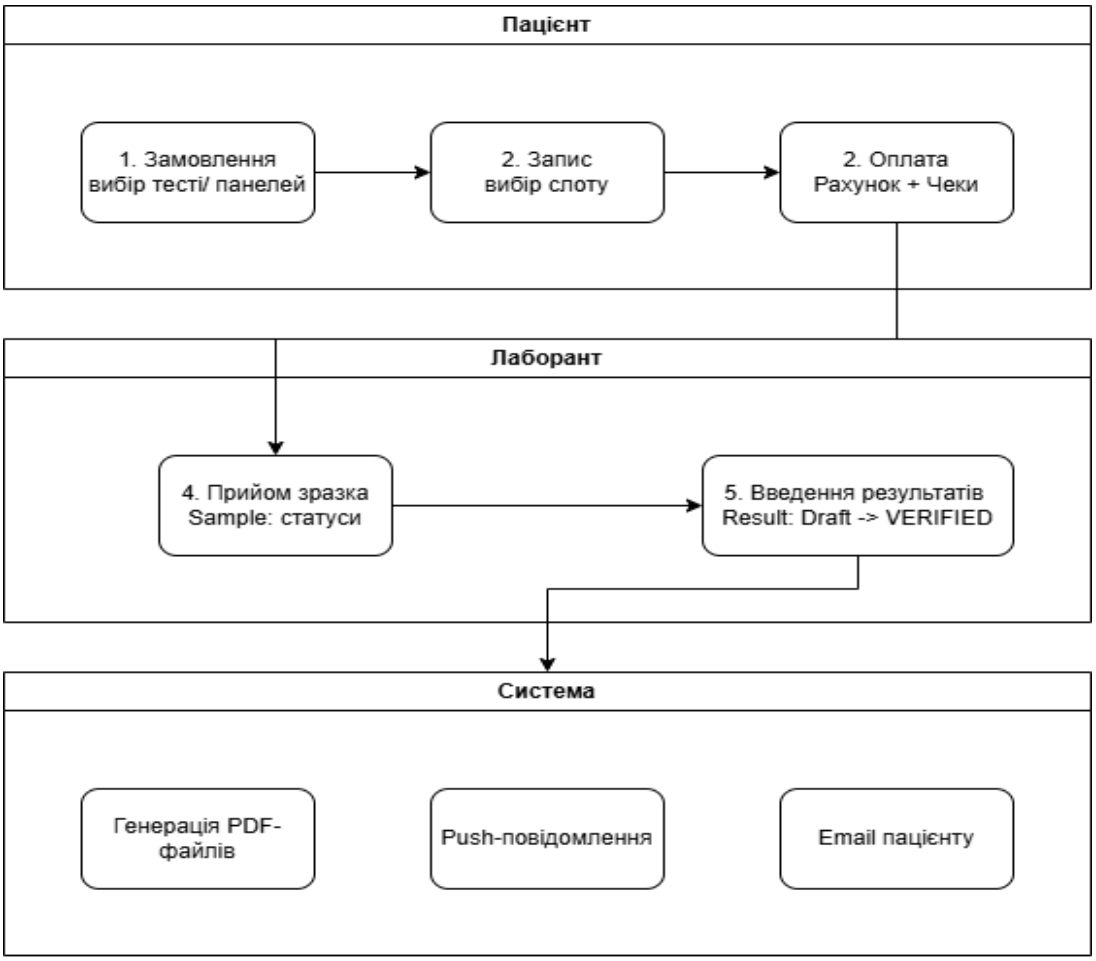


Рис. 3.9 Повний операційний цикл Monolab

5. Структура бази даних

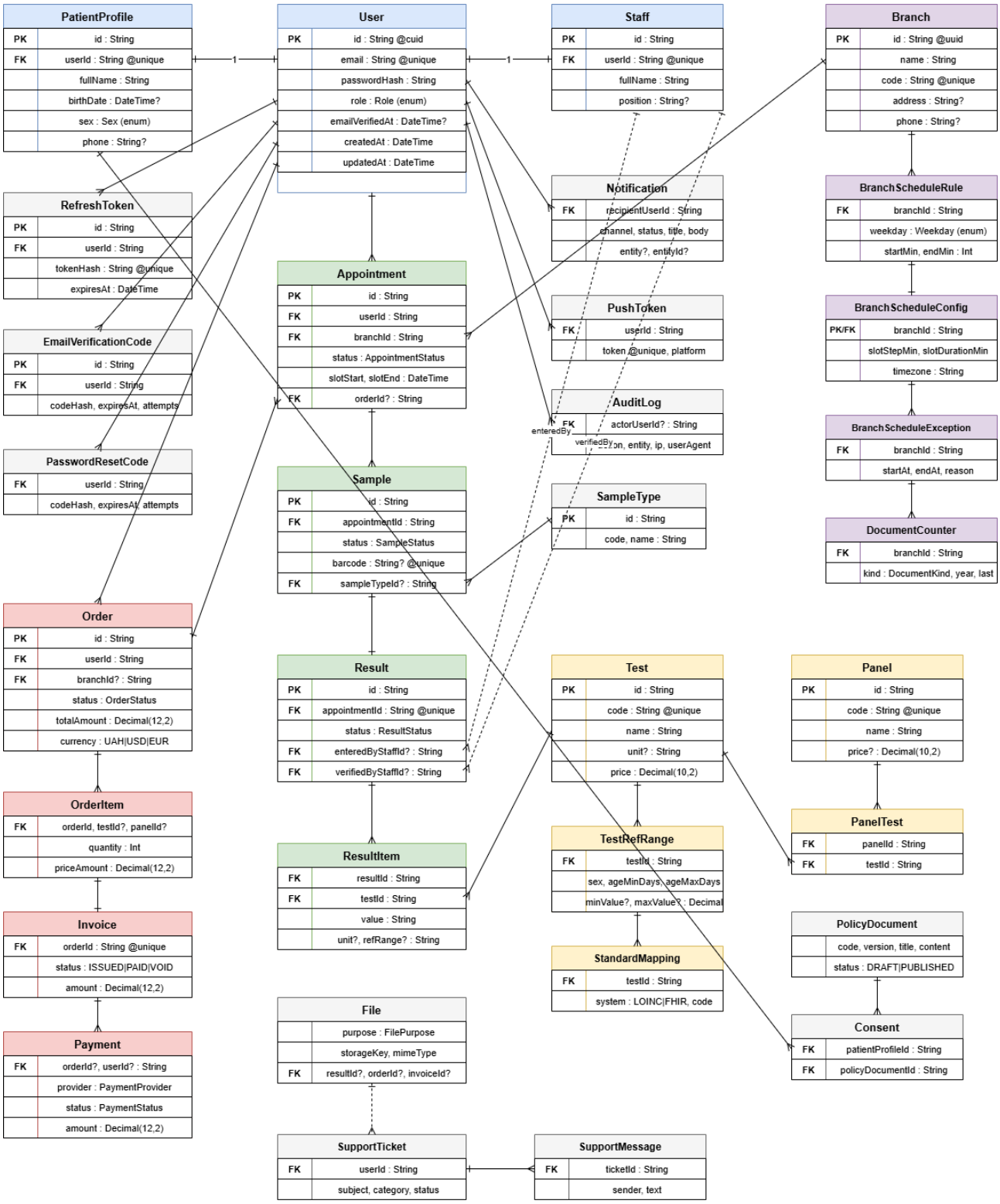


Рис. 3.10 Структура бази даних

Prisma-схема проєкту містить понад 30 моделей та 850+ рядків. Основні сутності та їхні зв'язки наведено на діаграмі (рис. 3.10). Центральними є: User (з полем role: PATIENT/LAB_TECH/ADMIN), Order (замовлення з набором тестів та панелей), Appointment (запис із прив'язкою до слоту відділення), Sample (зразок біоматеріалу), Result та ResultItem (результати по кожному тесту), Payment та Invoice (фінансовий цикл), Notification та PushToken (сповіщення), AuditLog (аудит), DocumentCounter (послідовна нумерація по відділенню/року) [13; 14].

3.7 Модуль генерації PDF-звітів та файлового сховища

Генерація PDF-звітів є завершальним етапом операційного циклу та реалізована через два сервіси: results-pdf.service.ts (звіти результатів) та pdf.service.ts (рахунки та квитанції). Обидва побудовані на бібліотеці PDFKit [21], що дозволяє програмно формувати документи формату A4 з точним контролем над розміткою, шрифтами, таблицями та зображеннями.

PDF-звіт результатів аналізу містить: логотип лабораторії та назву відділення, персональні дані пацієнта (ПІБ, дата народження, стать, номер замовлення), таблицю результатів (для кожного тесту – назва, код, значення, одиниця вимірювання, референсний діапазон, індикація норми або відхилення), дату та час верифікації, ПІБ лаборанта, що верифікував результат, та унікальний номер документа, згенерований через DocumentCounter.

Після генерації PDF-файл завантажується у MinIO S3 [19] через офіційний SDK (@minio/minio-js). Кожен файл зберігається з UUID-ключем, а метадані (оригінальне ім'я, MIME-тип, розмір, зв'язок із результатом) записуються у таблицю File бази даних. Пацієнт отримує PDF двома шляхами: завантаження з MinIO через захищений ендпоінт (files.service.ts перевіряє, що запитувач є власником результату) або як вкладення до email-повідомлення email.service.ts.

3.8 Перспективи інтеграції з зовнішніми системами

Архітектура Monolab спроєктована з урахуванням можливості підключення до зовнішніх систем, що є необхідною умовою для промислової експлуатації медичного програмного забезпечення в Україні. У поточній версії прототипу реальні інтеграції не підключені, проте серверна частина побудована таким чином, що їх додавання не потребує зміни існуючої архітектури.

1. Інтеграція з eHealth (ЕСОЗ)

Електронна система охорони здоров'я (ЕСОЗ, eHealth) – централізована державна платформа МОЗ України, через яку медичні заклади взаємодіють із центральною базою даних: реєструють пацієнтів, фіксують медичні записи, обробляють електронні направлення та е-рецепти. Система eHealth побудована за двокомпонентною архітектурою: заклад використовує медичну інформаційну систему (МІС), яка через REST API обмінюється даними з центральною базою ЕСОЗ.

Для підключення Monolab до eHealth необхідно: зареєструвати систему як МІС на порталі eHealth (portal.ehealth.gov.ua), пройти сертифікацію та тестування на тестовому середовищі ЕСОЗ, реалізувати модуль взаємодії з API eHealth (створення медичних записів, обробка електронних направлень, фіксація результатів лабораторних досліджень). Технічно інтеграція зводиться до створення окремого сервісу (наприклад, ehealth.service.ts), який через HTTP-клієнт (Axios) буде надсилати дані у форматі, визначеному специфікацією API eHealth. Автентифікація з ЕСОЗ здійснюється через КЕП (кваліфікований електронний підпис) медичного закладу.

2. Інтеграція з платіжними провайдерами

Модуль оплати у поточній версії реалізовано з імітацією платіжного процесу (mock payment): пацієнт обирає спосіб оплати, вводить довільні дані та отримує підтвердження. Для підключення реальних провайдерів передбачено абстракцію

PaymentService, де конкретна реалізація (LiqPay, Fondy, Stripe, EasyPay) підключається як стратегія без зміни інтерфейсу. Кожен провайдер потребує укладення договору, отримання API-ключів та налаштування webhook для підтвердження оплати.

LiqPay (ПриватБанк) – найпоширеніший платіжний шлюз в Україні, підтримує Google Pay, Apple Pay та оплату карткою. Інтеграція передбачає формування підписаного запиту з параметрами замовлення (сума, опис, order_id) та перенаправлення на сторінку оплати LiqPay. Після завершення оплати LiqPay надсилає callback на сервер Monolab, де PaymentService оновлює статус платежу та замовлення.

3. Інтеграція з лабораторними аналізаторами

У перспективі розвитку Monolab може бути інтегрований з лабораторним обладнанням через протокол HL7 (Health Level 7) або ASTM (American Society for Testing and Materials) – стандартні протоколи обміну даними між інформаційними системами та медичними приладами. Такая інтеграція дозволить автоматично імпортувати результати з аналізатора у систему, мінімізуючи ручне введення та ризик помилок [3; 4].

4 РЕАЛІЗАЦІЯ ІНТЕРФЕЙСУ ТА ТЕСТУВАННЯ СИСТЕМИ

4.1 Структура навігації веб-застосунку

Навігація веб-кабінету Monolab побудована на React Router v6 із механізмом захищених маршрутів. Точкою входу є AppRouter.tsx, який розподіляє маршрути на три зони: публічну (авторизація, реєстрація, landing page), захищену для авторизованих користувачів та системну (сторінки помилок 403, 404). Захист маршрутів реалізовано через два компоненти: AuthGate.tsx перевіряє наявність та валідність JWT-токену [18], а RequireRole.tsx – відповідність ролі користувача дозволеним для конкретного маршруту.

Кожна роль має ізольовану гілку маршрутів. Пацієнт (/app/patient/*) має доступ до 11 сторінок: дашборд, записи, створення запису (wizard), замовлення, деталі замовлення, результати, деталі результату, платежі, журнал активності, динаміка показників та сповіщення. Лаборант (/app/lab/*) працює з 3 сторінками: головна зі зведеною статистикою, список записів та drawer-панель деталей запису. Адміністратор (/app/admin/*) має 7 сторінок: дашборд, аудит-лог, відділення, користувачі, каталог тестів, панелі та управління політикою конфіденційності. Додатково існує спільна сторінка профілю, доступна для всіх авторизованих ролей.

Загальний layout застосунку (AppLayout.tsx) включає бічне меню з навігацією, верхню панель із пошуком та сповіщеннями, та основну робочу область. Для сторінок авторизації використовується окремий AuthLayout.tsx без бічного меню. Тема інтерфейсу налаштовується через конфігурацію Ant Design 6 (theme/), що забезпечує єдиний візуальний стиль усіх компонентів [16].

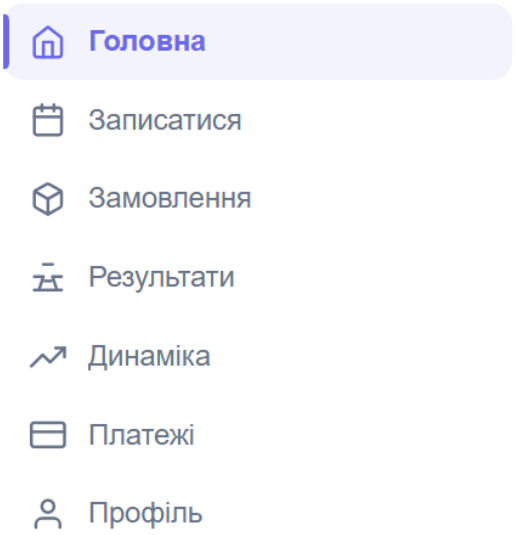


Рис. 4.1 Структура навігації веб-застосунку Monolab

Таблиця 4.1

Перелік сторінок веб-кабінету за ролями

Роль	Сторінка	Призначення
Публіна	LandingPage	Головна сторінка сервісу
Публіна	Login / Register	Авторизація та реєстрація сервісу
Публіна	ForgotPassword / ResetPassword	відновлення паролю через електронну пошту
Публіна	Privacy	Політика конфіденційності
PATIENT	PatientHomePage	Головна сторінка пацієнта із зведеною інформацією
PATIENT	PatientAppointmentsPage	Список замовлень на дослідження
PATIENT	PatientBookAppointmentPage	Запис на проведення дослідження
PATIENT	PatientOrdersPage / OrderPage	Список та деталі замовлення
PATIENT	PatientResultsPage / ResultPage	Список та деталі результатів із кольоровою індикацією
PATIENT	PatientPaymentsPage	Історія оплат
PATIENT	PatientDynamicsPage	Графіки динаміки показників пацієнта
PATIENT	PatientActivityPage	Журнал активності пацієнта

Роль	Сторінка	Призначення
PATIENT	PatientNotificationsPage	Сповіщення
LAB_TECH	LabHomePage	Головна стоірнка лаборанта, Зведена статистика дня
LAB_TECH	LabAppointmentsPage	Черга записів, внесення результатів через модалку
ADMIN	AdminHomePage	Головна сторінка адміністратора із аналітикою системи
ADMIN	AdminBranchesPage	Управлення відділенням та розкладом
ADMIN	AdminTestsPage / AdminPanelsPage	Каталог досліджень та панелей
ADMIN	AdminUsersPage	Управління користувачами та ролями
ADMIN	AdminAuditPage	Аудит-лог системи із продуманою фільтрацією
ADMIN	AdminPrivacyPolicyPage	Управління політикою конфіденційності
Спільна	ProfilePage	Профіль користувача

4.2 Реалізація інтерфейсу веб-кабінету

Веб-кабінет побудований з використанням Ant Design 6 [16], що надає понад 60 готових компонентів: таблиці з сортуванням та пагінацією (Table), покрокові форми (Steps), drawer-панелі (Drawer), модальні вікна (Modal), графіки завантаження (Spin, Skeleton) та картки з KPI-метриками. Redux Toolkit [15] централізує стан авторизації, кешованих результатів, активних форм та сповіщень. RTK Query використовується для кешування серверних відповідей із автоматичною інвалідацією при мутаціях.

Сторінка авторизації складається з двох форм – входу та реєстрації – з валідацією полів через Ant Design Form та серверною перевіркою через Zod [23]. При помилці авторизації відображається повідомлення без розкриття конкретної причини (email чи пароль), що відповідає вимогам безпеки OWASP. Після реєстрації пацієнт отримує код підтвердження на email (TTL 10 хвилин, максимум 5 спроб введення).

Кабінет пацієнта організований навколо дашборду (PatientHomePage), де відображаються KPI-картки: кількість активних записів, останні результати, непрочитані сповіщення. Список результатів (PatientResultsPage) реалізований через Ant Design Table із кольоровою індикацією статусу: зелена позначка – значення у межах референсного діапазону, червона – відхилення від норми, сіра – результат ще не готовий. Натискання на рядок відкриває деталізацію (PatientResultPage) з повною інформацією: назва тесту, числове значення, одиниця, нижня та верхня межі норми, дата, ПІБ лаборанта.

Сторінка динаміки показників (PatientDynamicsPage) побудована на бібліотеці Recharts [22]. Пацієнт обирає конкретне дослідження з випадаючого списку, і система будує лінійний графік (LineChart) із точками даних за кожну дату здачі. На графіку нанесено дві горизонтальні референсні лінії (ReferenceLine) – верхня та нижня межі норми. Це дозволяє візуально оцінити динаміку: чи знаходяться значення у коридорі норми, чи є тренд до покращення або погіршення. Tooltip відображає точне значення та дату при наведенні на точку.

Wizard запису на аналіз (PatientBookAppointmentPage) реалізований через компонент Steps з чотирма кроками. Перший крок – вибір відділення з карткового списку (Card) із адресою та графіком роботи. Другий – вибір досліджень або панелей із каталогу (Transfer або Checkbox group) з автоматичним підрахунком вартості. Третій – вибір дати та часового слоту (DatePicker + сітка доступних слотів, згенерованих на основі BranchScheduleConfig). Четвертий – підтвердження та оплата через модальне вікно з імітацією платіжного процесу.

Кабінет лаборанта (LabHomePage, LabAppointmentsPage) орієнтований на операційну ефективність. Список записів на поточний день відсортований за часом слоту. Натискання на запис відкриває Drawer-панель (LabAppointmentDrawer) із повною інформацією: дані пацієнта, замовлені тести, поточний статус, кнопки зміни статусу. Форма введення результатів генерується динамічно на основі списку тестів у замовленні – для кожного тесту відображається поле вводу з підказкою референсного діапазону.

Адміністративна панель (AdminHomePage) містить KPI-картки (KpiCard) із ключовими метриками: кількість замовлень за обраний період, сума оплат, кількість активних пацієнтів, середній час обробки. Графіки доходу та кількості досліджень побудовані на Recharts (BarChart, LineChart). Аудит-лог (AdminAuditPage) відображається через Table із фільтрацією за типом дії, сутністю, користувачем та часовим діапазоном (DateRangePicker).

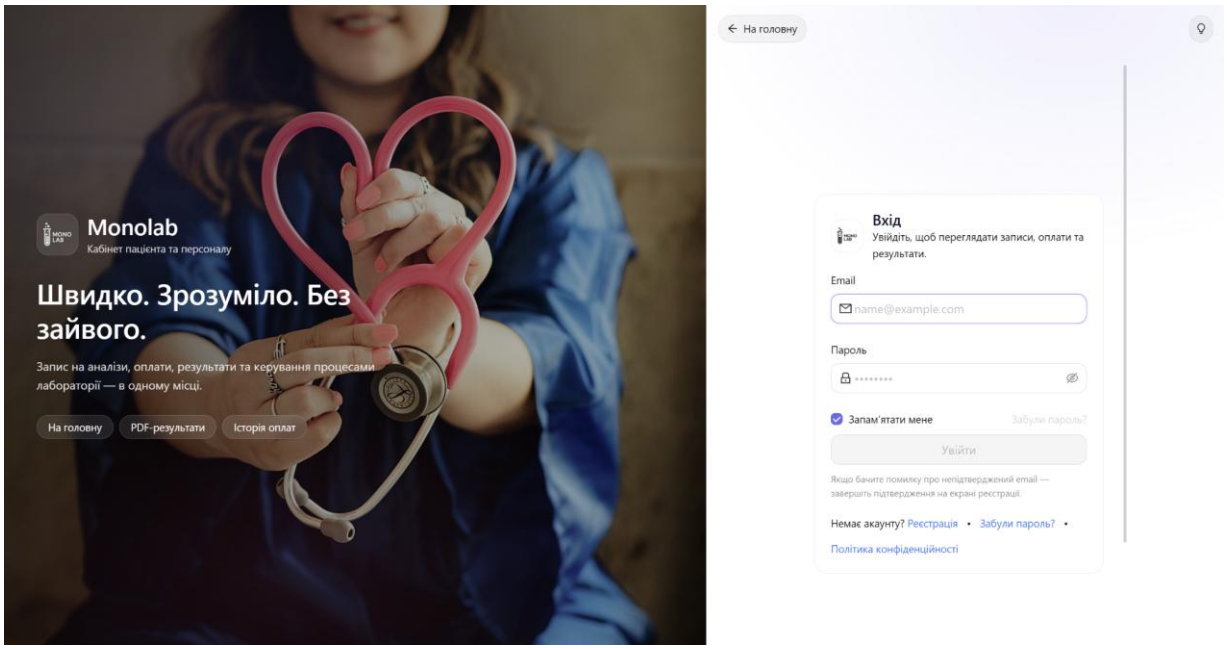


Рис. 4.2 Сторінка авторизації веб-кабінету

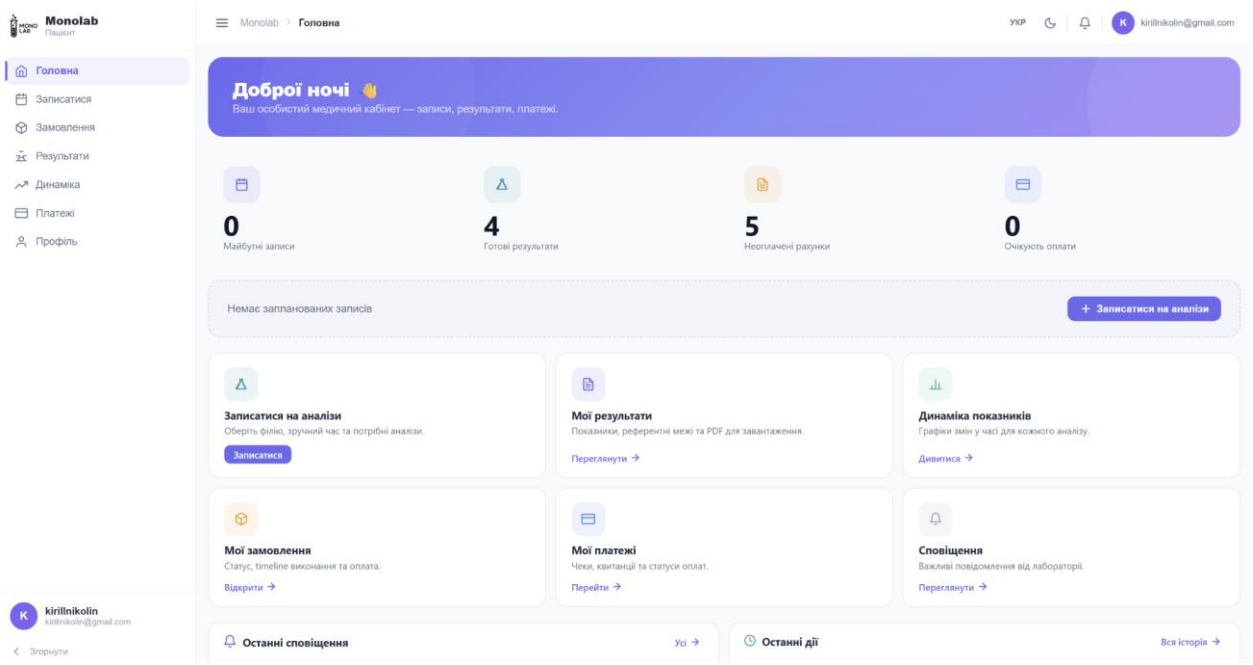


Рис. 4.3 Дашборд пацієнта з KPI-картками

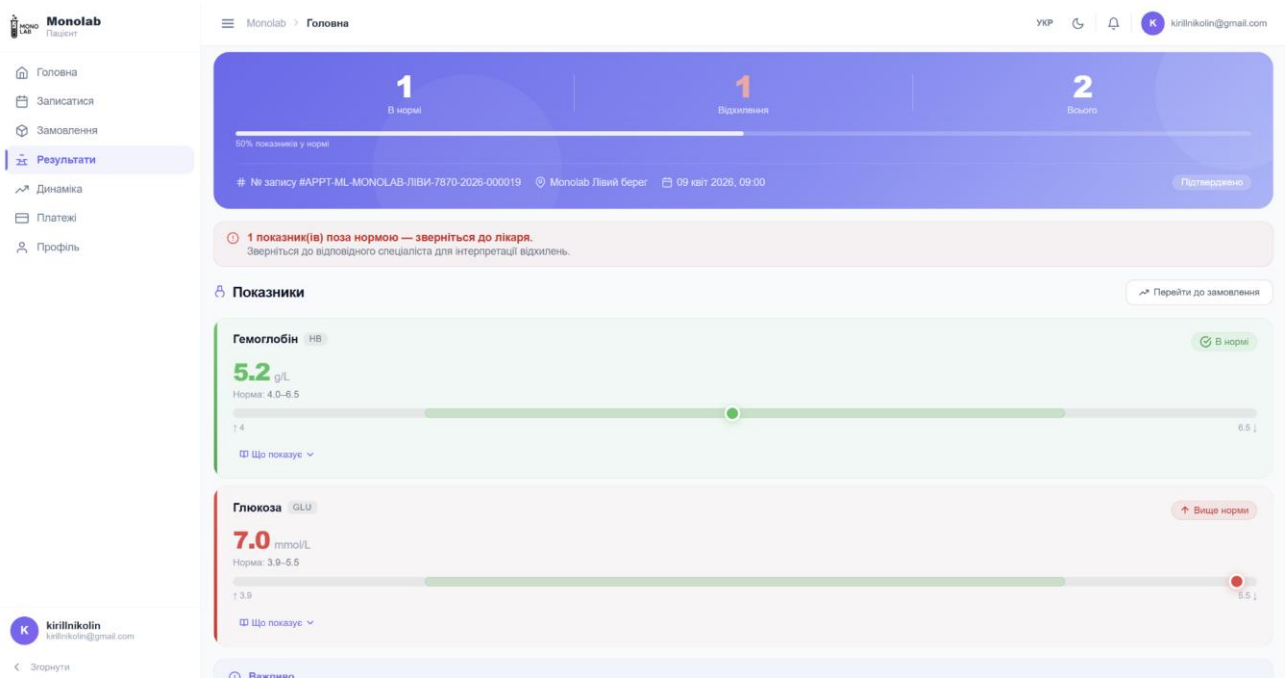


Рис. 4.4 Список результатів із кольоровою індикацією

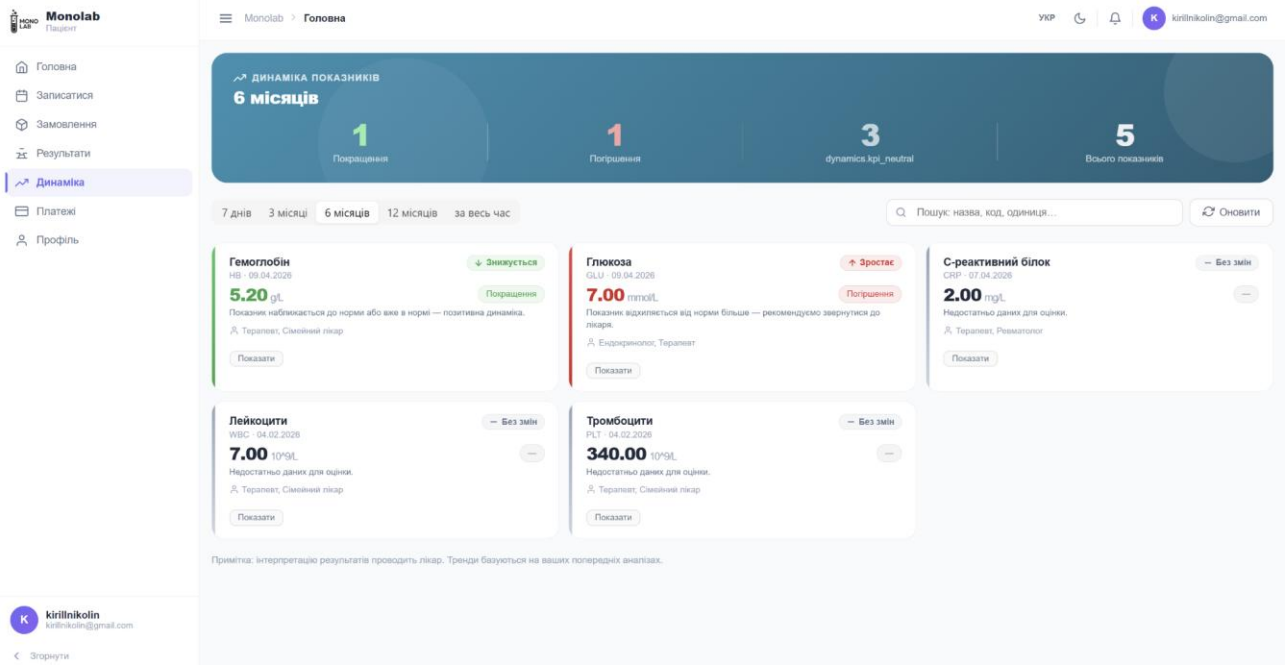


Рис. 4.5 Графік динаміки показника (Recharts)

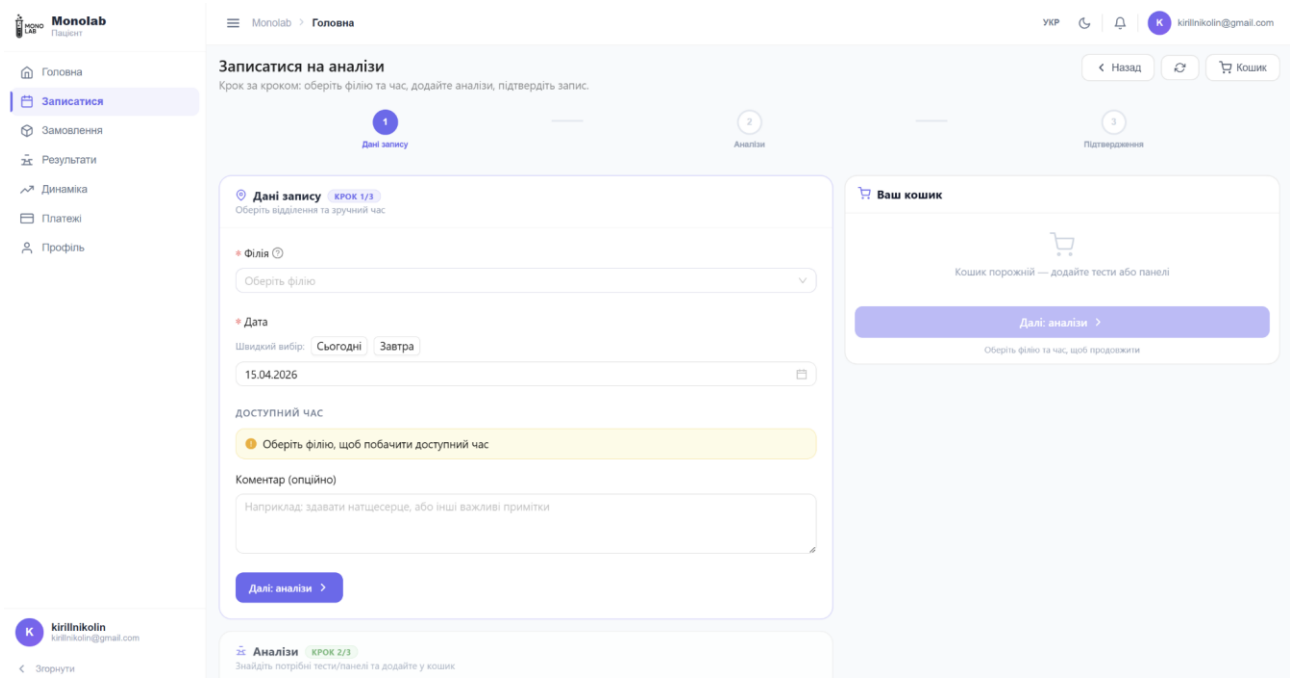


Рис. 4.6 Wizard запису на аналіз (крок вибору слоту)

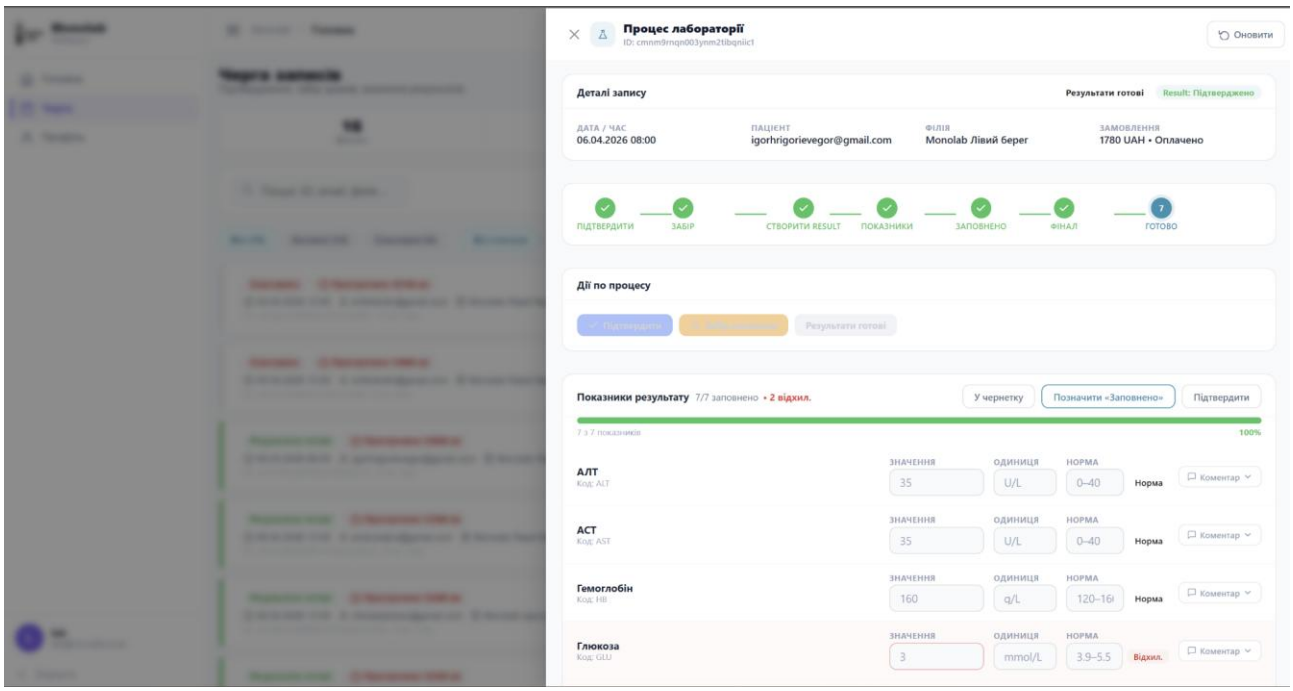


Рис. 4.7 Кабінет лаборанта – drawer деталей запису

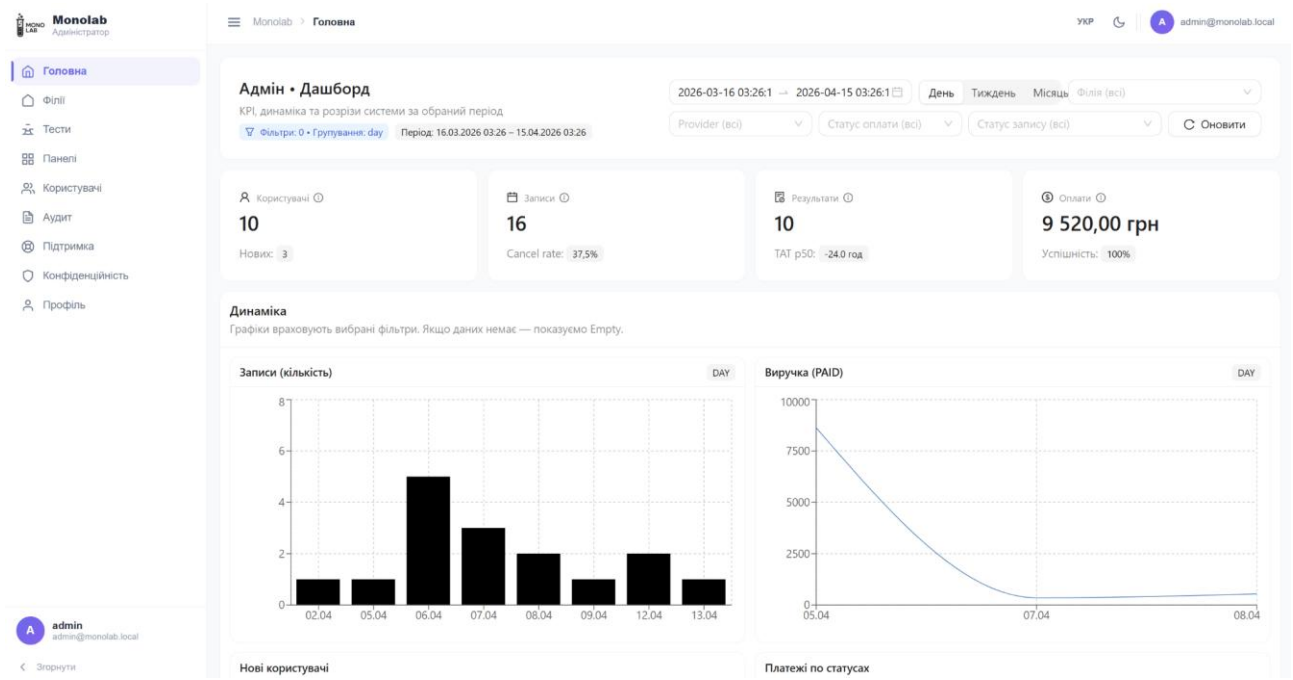


Рис. 4.8 Адміністративна панель – дашборд із аналітикою

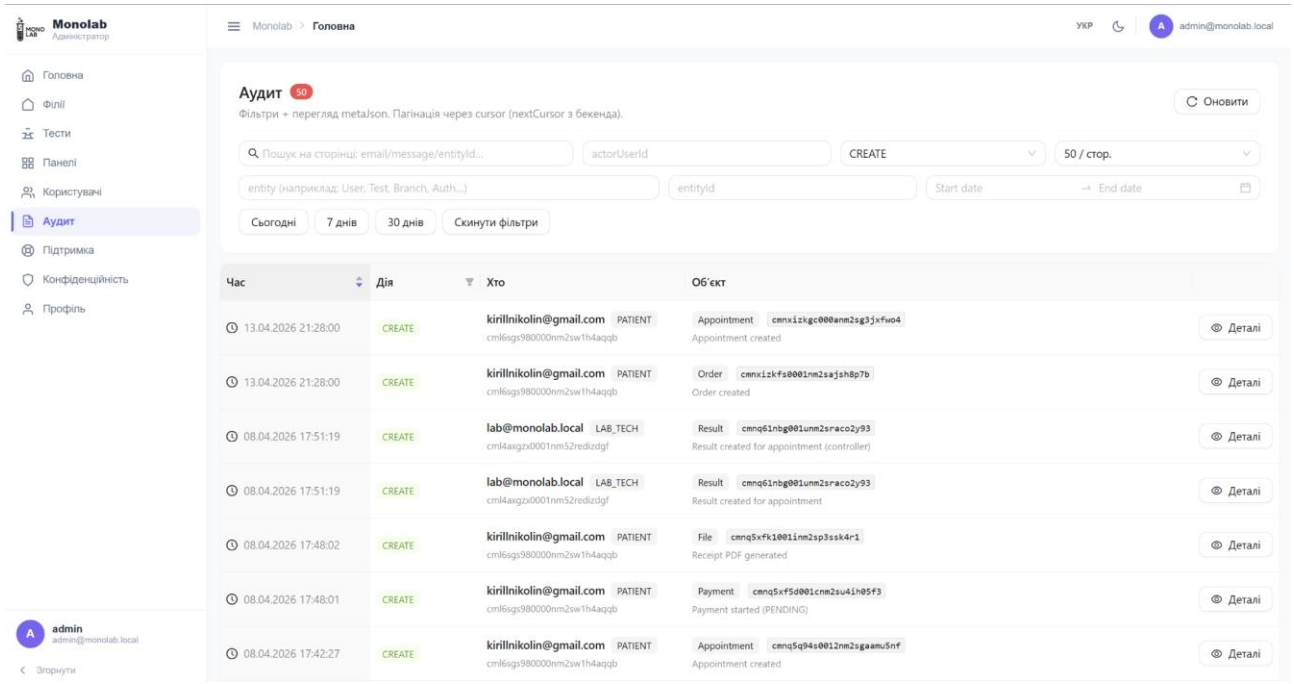


Рис. 4.9 Аудит-лог із фільтрацією

4.3 Реалізація мобільного застосунку

Мобільний додаток побудований на React Native 0.85 [12] з увімкненою New Architecture (Fabric + TurboModules, Bridgeless-режим). Інтерфейс складається з 34 екранів та 62 компонентів, організованих у три навігаційних стеки: авторизація (6

екранів), головний розділ (12 екранів) та профіль (7 екранів), об'єднаних через Bottom Tab Navigator з трьома вкладками: Головна, Сповіщення, Профіль [24].

Тема додатку підтримує три режими: світлий, темний та системний (автоматичне перемикання відповідно до налаштувань операційної системи). Кольорова палітра визначена через токени (Theme.ts), де кожен колір має гілки light та dark. Усі компоненти використовують виключно токени теми без хардкоднутих кольорів, що перевірено статичним аналізом.

Головний екран (Menu) відображає зведену інформацію: кількість активних записів, останні результати та швидкі дії (новий запис, перегляд результатів). Список результатів (PatientResultsListScreen) реалізований через FlatList з підтримкою pull-to-refresh та фільтрацією через bottom-sheet (@gorhom/bottom-sheet v5.1). Деталі результату (PatientResultDetailsScreen) побудовані з використанням нового шару UI-примітивів: DetailsHeroCard (hero-блок з іконкою та статус-бейджем), DetailsSectionCard (секція з заголовком), DetailsGrid (рядки key/value). Анімації входу реалізовані через Reanimated 4.3: послідовні staggered FadeInDown з наростаючим delay (60 – 120 – 180 мс) для природного відчуття «складання» екрану.

Динаміка показників (PatientDynamicsScreen) побудована на react-native-gifted-charts v1.4, яка замінила deprecated react-native-svg-charts. Графік адаптується до ширини екрану, відображає точки даних із підписами дат та горизонтальні лінії референсних меж. Аномальні показники (значення поза нормою) виділяються через PulseGlow – Reanimated-обгортку, що створює нескінченний м'який пульс opacity для привертання уваги.

Wizard запису (PatientBookAppointmentWizardScreen) оптимізований для сенсорного введення: вибір відділення – горизонтальна прокрутка карток, вибір дати – нативний DateTimePicker, підтвердження – модальне вікно з деталями. Таймлайн замовлення (OrderTimeline) реалізований як горизонтальний stepper із чотирма станами (CREATED – INVOICE – PAID – FULFILLED) з анімованою прогрес-лінією через useSharedValue + withTiming та пульсуючим активним dot.

Для скасованих замовлень використовується ShakeView – Reanimated-обгортка для shake-ефекту на cancel-банері.

Push-повідомлення працюють через Firebase FCM [17] у поєднанні з Notifee для foreground-відображення. Лічильник непрочитаних сповіщень реалізовано через кастомний хук useNotificationBadge, який оновлюється при кожному отриманні push або відкритті екрану сповіщень. Локалізація підтримує українську мову через i18n-словники (ua.json) з fallback-механізмом safeT(t, key, fallback).

Таблиця 4.2

Основні екрани мобільного застосунку

Розділ	Екран	Призначення
Auth(6)	Login, Register, VerifyEmail	Авторизація та підтвердження електроної пошти
Auth	ForgotPasswordEmail, ResetPasswordConfirm	Відновлення пароля
Auth	TermsOfUse	Умови використання сервісу
Main(12)	Menu	Головний екран із швидкими діями
Main	PatientAppointmentsList / BookWizard	Запис на дослідження
Main	PatientOrderDetails	Деталі замовлення з timeline
Main	PatientResultsList / ResultDetails	Результати дослідження з кольоровою індикацією
Main	PatientPaymentsList / PaymentDetails	Список оплат та деталі оплат
Main	PdfViewerScreen	Перегляд pdf-файлів та завантаження через WebView
Main	PatientActivityScreen	Журнал активностей пацієнта

Розділ	Екран	Призначення
Profile(7)	Profile, PersonalData, Settings	Профіль та налаштування
Profile	PatientDynamicsScreen	Графіки динаміки показників пацєнта
Profile	Support, CreateMessage, FAQ	Підтримка
Notification(1)	Notification	Сповіщення

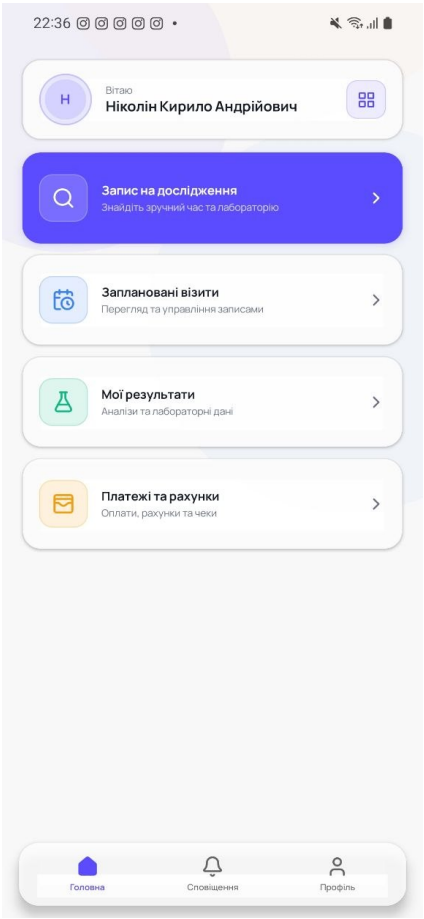


Рис. 4.10 Головний екран мобільного додатку (light theme)

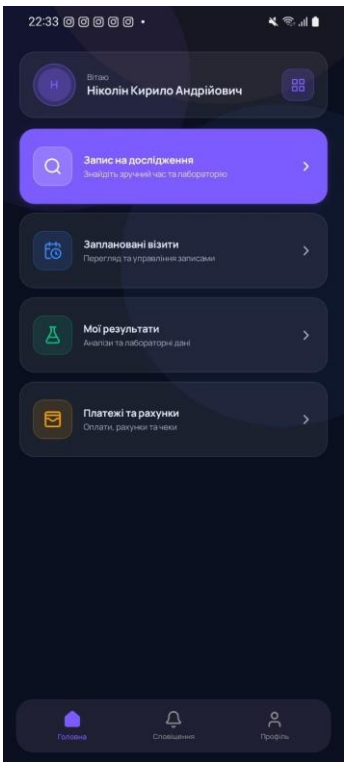


Рис. 4.11 Головний екран мобільного додатку (dark theme)

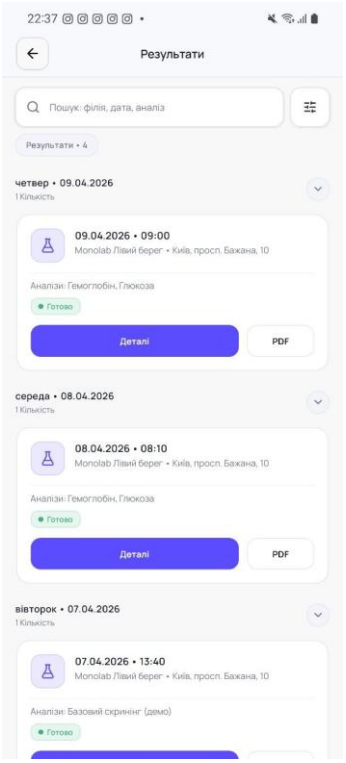


Рис. 4.12 Список результатів у мобільному додатку

Рис. 4.13 Деталі результату з DetailsHeroCard

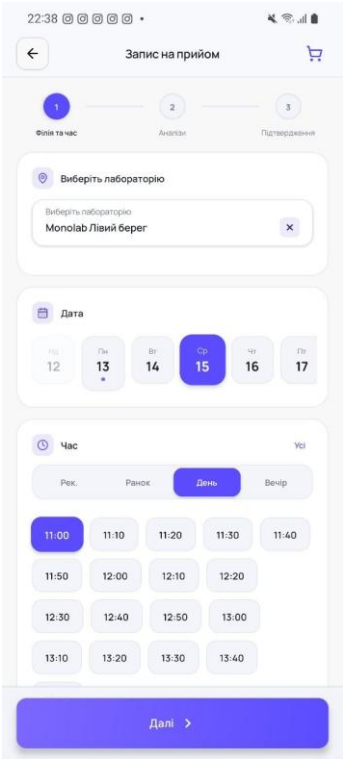


Рис. 4.14 Wizard запису на аналіз (мобільний)

Рис. 4.15 OrderTimeline з анімованою прогрес-лінією

4.4 Тестування системи

4.4.1 Тестування REST API

Тестування серверної частини проводилось методом ручного функціонального тестування із використанням інструменту Postman. Для кожного з 21 контролера перевірялись основні сценарії: успішне виконання (HTTP 200/201), валідаційні помилки (400), відсутність автентифікації (401), відмова в доступі за роллю (403), відсутність ресурсу (404). Автоматизованих unit- або integration-тестів у поточній версії прототипу не реалізовано – акцент зроблено на повноту функціоналу, а не на покриття тестами, що є типовим для дипломних проєктів з обмеженими часовими ресурсами.

Тестування автентифікації охоплювало повний цикл JWT [18]: реєстрація нового пацієнта, підтвердження email через код, вхід з отриманням access + refresh токенів, оновлення access-токену через refresh, вихід з інвалідацією. Перевірялись сценарії помилок: спроба входу з невірним паролем (401), реєстрація з існуючим

email (409), використання простроченого access-токену (401 з подальшим refresh), спроба доступу лаборанта до ендпоінту адміністратора (403).

Тестування бізнес-логіки зосереджувалось на критичному шляху – повному операційному циклі від створення замовлення до видачі результату. Перевірялось дотримання послідовності статусів: Appointment (CREATED – CONFIRMED – SAMPLED – RESULTS_READY), Sample (CREATED – COLLECTED – IN_LAB – PROCESSED), Result (DRAFT – COMPLETED – VERIFIED). Спроба пропуску етапу (наприклад, перехід з CREATED одразу на SAMPLED без CONFIRMED) повертає помилку 422.

Окремо тестувалися модулі генерації PDF (перевірка структури згенерованого документу, коректність даних пацієнта та результатів), завантаження файлів у MinIO (upload, download, delete), відправка email (перевірка через тестовий SMTP-сервер) та аудит-лог (наявність запису після кожної критичної дії).

Таблиця 4.3

Результати функціонального тестування API

Модуль	Перевірені сценарії	Результат
Автентифікація	Реєстрація, вхід, refresh, вихід, помилки	Усі сценарії пройдені
RBAC	Доступ PATIENT/LAB_TECH/ADMIN до чужих ресурсів	403 для недозволених
Замовлення	CRUD, підрахунок вартості, прив'язка тестів	Коректно
Записи	Повний workflow статусів, автоскасування	Послідовність дотримана
Зразки	Створення, зміна статусів, REJECTED	Коректно
Результати	Введення, верифікація, автогенерація PDF	PDF-файли генеруються
Платежі	Mock payment, статуси, квитанції	Імітація працює
PDF-генерація	Структура, дані пацієнта, референси	Документ коректний

Модуль	Перевірені сценарії	Результат
MiniO	Upload, download, delete, перевірка доступу	Захищений доступ
Аудит	Логування дій, фільтрація, актор + IP	Фіксація повна
Нотифікації	FCM push, email, in-app	доставка підтверджена
Планувальник	Автоскасування прострочених записів	Працює кожного дня о 23:30 UTC

4.4.2 Тестування веб-інтерфейсу

Тестування веб-кабінету проводилось ручним методом з використанням інструментів розробника браузерів: Chrome DevTools для аналізу мережевих запитів, продуктивності рендерингу та адаптивності, Firefox Developer Tools для перевірки CSS Grid/Flexbox та доступності. Перевірялась коректність відображення усіх 31 сторінки, робота форм (валідація на клієнті та сервері), фільтрація та сортування таблиць, відображення графіків Recharts, завантаження PDF, обробка помилкових станів (немає даних, мережева помилка, прострочений токен).

Кросбраузерне тестування проводилось у Google Chrome (версія 130+), Mozilla Firefox (версія 132+), Microsoft Edge (версія 130+) та Safari (версія 18+). В усіх браузерах інтерфейс відображався коректно: Ant Design компоненти, Recharts графіки, модальні вікна та drawer-панелі працювали без візуальних артефактів. Адаптивність перевірена через Chrome DevTools Responsive Mode на екранах від 375px (iPhone SE) до 2560px (4K монітор).

Таблиця 4.4

Результати кросбраузерного тестування

Критерії	Chrome	Firefox	Edge	Safari
Верстка та стилі	+	+	+	+
JWT-авторизація	+	+	+	+
Графіки динаміки	+	+	+	+

Критерії	Chrome	Firefox	Edge	Safari
Завантаження PDF-файлів	+	+	+	+
WebSocket	+	+	+	+
Адаптивність	+	+	+	+
Ant-Design компоненти	+	+	+	+

4.4.3 Тестування мобільного застосунку

Мобільний додаток тестувався на реальному пристрої Samsung Galaxy (Android 14) та на емуляторі iOS (Xcode Simulator, iPhone 15 Pro, iOS 17). Додатково debug-збірки розповсюджувались серед групи тестувальників (друзів та знайомих автора) для збору зворотного зв'язку щодо зручності інтерфейсу, зрозумілості навігації та стабільності роботи. На момент написання роботи готується версія 1.0.4, яка буде передана на розширене тестування.

Перевірялись усі 34 екрани додатку, робота нативних модулів (Firebase push у foreground та background, перегляд PDF через WebView, bottom-sheet модальні вікна), коректність анімацій (Reanimated 4.3 staggered входи, OrderTimeline прогрес-лінія, PulseGlow для аномальних показників), перемикання тем (light – dark – system), локалізація (українська мова через i18n).

Продуктивність додатку оцінювалась через Android Studio Profiler та Xcode Instruments. Час холодного старту (cold start) не перевищує 2.5 секунди на пристроях середнього сегменту. Час відповіді API при завантаженні списку результатів складає 150–300 мс (локальний Docker-сервер), що забезпечує суб'єктивне відчуття миттєвої реакції. Споживання оперативної пам'яті стабільно утримується в межах 130 – 190 МБ, що є прийнятним для React Native додатку такого обсягу.

За результатами тестування критичних та блокуючих дефектів не виявлено. Знайдені зауваження (незначне вирівнювання елементів на пристроях з

нестандартним DPI, затримка оновлення лічильника сповіщень при першому вході) було виправлено у поточній ітерації розробки.

Таблиця 4.5

Результати тестування мобільного застосунку

Критерії	Android(Samsung)	IOS(Simulator)
Запуск додатку	2.2с	2с
Авторизація	Працює	Працює
Push-повідомлення	Працює(системні та в застосунку)	Працює(Тестування обмежено симулятор)
Графіки динаміки	Відображення коректне	Відображення коректне
Перегляд PDF-файлів	Працює	Працює
Анімації	Плавні, 60 fps	Плавні, 60 fps
Темна тема	Працює	Працює
Локалізація	Коректна	Коректна
RAM-споживання	130МБ	170МБ
Модалки	Працює	Працює

4.5 Оцінка ефективності розробленого рішення

Для оцінки ефективності системи Monolab проведено порівняння часових витрат на типові операції між традиційним лабораторним процесом (ручна реєстрація, паперові бланки, телефонне/email інформування) та розробленою системою. Оцінка базується на хронометражі операцій під час тестування прототипу та на даних про типові терміни, отриманих з практичного досвіду автора як клієнта лабораторії ДІЛА протягом семи років.

Найбільш суттєве покращення стосується етапу доставки результатів пацієнту. У традиційному процесі після верифікації результат надсилається електронною поштою, що займає від кількох хвилин до кількох годин залежно від внутрішніх процесів лабораторії. У Monolab після натискання кнопки «Верифікувати» система автоматично генерує PDF (PDFKit, приблизно 200 мс),

зберігає його у MinIO (приблизно 100 мс) та надсилає push-повідомлення через FCM (приблизно 300 мс) – загальний час від верифікації до отримання пацієнтом на телефон складає 1 – 3 секунди.

Модуль динаміки показників є унікальною перевагою, що не має прямого аналогу серед конкурентних рішень у повному обсязі. ДІЛА пропонує динаміку лише у десктопній версії веб-кабінету, Сінево має відповідну вкладку, яка фактично не функціонує. У Monolab графіки доступні і на вебі (Recharts), і у мобільному додатку (gifted-charts) – пацієнт може перевірити тренд прямо з телефону у будь-який момент.

Аудит-лог забезпечує повну прослідковуваність операцій, що є обов'язковою вимогою для медичних інформаційних систем відповідно до ISO 15189 та законодавства України про захист персональних медичних даних [5; 6]. Кожна зміна даних фіксується з зазначенням автора, IP-адреси, User-Agent та request ID.

Self-hosted розгортання через Docker Compose [20] дозволяє лабораторії зберігати повний контроль над даними пацієнтів. Вся інфраструктура (PostgreSQL, MinIO, API-сервер, веб-клієнт) розгортається однією командою docker compose up на будь-якому сервері з Docker, без залежності від зовнішніх SaaS-провайдерів.

Таблиця 4.6

Порівняння часових витрат на типові операції

Операція	Традиційний процес	Monolab	Покращення
Запис на дослідження	10-15 хв	2-3 хв	75%
Введення результатів	5-10 хв	2-3 хв	60%
Доставка результатів	30 хв - 24 год	1-3 сек	99%
Пошук старих результатів	10-30 хв	5-10 сек	95%
Порівняння показників	Вручну	1 сек	Новий функціонал
Повний цикл	1-2 доби	2 -6 годин	70%

ВИСНОВКИ

У межах кваліфікаційної роботи бакалавра досягнуто поставленої мети – підвищено зручність обслуговування пацієнтів медичної лабораторії та скорочено час доставки результатів за рахунок автоматизації повного операційного циклу в єдиному застосунку. Спроєктовано та реалізовано комплексне програмне рішення Monolab – веб- та мобільний застосунок, побудований повністю на TypeScript, що об'єднує онлайн-запис, оплату, перегляд результатів та динаміку показників пацієнта. Система охоплює повний операційний цикл лабораторної діагностики – від запису пацієнта до отримання верифікованого результату на його телефон.

Протягом роботи над проєктом було розв'язано шість задач, сформульованих у вступі.

Виконано порівняльний аналіз трьох платформ лабораторної діагностики – власної розробки Monolab, Synevo та ДІЛА. Аналіз проводився на основі практичного семирічного досвіду користування платформою ДІЛА та особистого тестування веб-кабінету Synevo. Встановлено, що жодна з комерційних платформ не забезпечує одночасно: графічне відображення динаміки показників у мобільному додатку, повний інтерфейс для лабораторного персоналу, аудит-лог критичних операцій та можливість self-hosted розгортання. Ці прогалини визначили напрям розробки Monolab.

Обґрунтовано вибір технологічного стеку. TypeScript обрано як єдину мову для усіх рівнів системи (сервер, веб, мобільний), що забезпечило перевикористання типів, інтерфейсів та Zod-схем валідації без дублювання. Серверна частина побудована на Node.js з Express, веб-кабінет – на React 19 з Vite та Ant Design 6, мобільний додаток – на React Native 0.85 з увімкненою New Architecture (Fabric, TurboModules, Bridgeless). Для бази даних обрано PostgreSQL 16 з Prisma ORM, для файлів – MinIO S3. Монорепозиторій організовано через pnpm workspaces для серверної та веб-частини та yarn для мобільної (через несумісність pnpm з iOS-збіркою React Native).

Реалізовано серверну частину обсягом 28 сервісів, 21 контролер та 21 файл маршрутів. Prisma-схема бази даних містить понад 30 моделей (850+ рядків), що

охоплюють повний набір сутностей: користувачі, тести, категорії, панелі, замовлення, записи, зразки, результати, платежі, рахунки, сповіщення, аудит-записи та файлові вкладення. Безпека забезпечена через JWT-автентифікацію з парою access/refresh токенів, рольову модель RBAC із трьома ролями (причому самостійна реєстрація доступна виключно для пацієнтів – акаунти лаборантів та адміністраторів створюються через адмін-панель), шифрування паролів bcrypt, захист HTTP-заголовків через Helmet.js та валідацію вхідних даних через Zod.

Розроблено веб-застосунок із 30 сторінкою, розподіленою між трьома ролями. Пацієнт отримав кабінет із кольоровою індикацією результатів, графіками динаміки показників на базі Recharts, покроковим wizard-записом на дослідження та журналом активності. Лаборант працює з чергою зразків, формами введення результатів з Zod-валідацією та механізмом верифікації. Адміністратор має дашборд із KPI-метриками, управління каталогом тестів, фінансову аналітику та аудит-лог з фільтрацією за типом дії, сутністю, користувачем та часовим діапазоном.

Створено мобільний застосунок із 31 екранами та 111 компонентами, що підтримує Android та iOS. У ході розробки проведено міграцію з React Native 0.76 на 0.85, включаючи перехід на New Architecture та заміну шести deprecated-бібліотек на актуальні аналоги, сумісні з Fabric (react-native-vision-camera замість camera, gifted-charts замість svg-charts, @gorhom/bottom-sheet замість modalize та інші). Реалізовано push-повідомлення через Firebase FCM з підтримкою foreground та background режимів, підтримку трьох тем оформлення (light, dark, system), локалізацію українською мовою та анімації на базі Reanimated 4.3 з виокремленим пакетом worklets.

Проведено ручне функціональне тестування системи. API перевірено через Postman: тестування автентифікації, рольового доступу, повного операційного циклу (послідовність статусів Appointment – Sample – Result), генерації PDF, роботи файлового сховища та аудит-логу. Веб-кабінет протестовано у чотирьох браузерах (Chrome, Firefox, Edge, Safari) із перевіркою адаптивності на екранах від 375px до 2560px. Мобільний додаток тестувався на реальному Samsung Galaxy

(Android 14), емуляторі iOS (iPhone 15 Pro, iOS 17) та розповсюджувався серед групи тестувальників для збору зворотного зв'язку. Критичних дефектів не виявлено.

Практична цінність роботи полягає у створенні self-hosted рішення, що розгортається через Docker Compose без залежності від зовнішніх SaaS-провайдерів. Будь-яка медична лабораторія може розгорнути повну інфраструктуру (PostgreSQL, MinIO, API, веб-клієнт) однією командою на власному сервері, зберігаючи повний контроль над даними пацієнтів. Модуль динаміки лабораторних показників, доступний одночасно у веб-кабінеті та мобільному додатку, є функціоналом, що відсутній у повному обсязі серед проаналізованих конкурентних рішень українського ринку.

Архітектура Monolab спроєктована з урахуванням масштабування, і подальший розвиток системи передбачає кілька напрямів.

Першим та найбільш перспективним напрямом є впровадження інтелектуального асистента на базі нейронної мережі для рекомендації лабораторних досліджень за описаними симптомами. Пацієнт зможе у вільній формі (текстове поле або голосовий ввід) описати свої скарги та симптоми – наприклад, «постійна втома, набір ваги, сухість шкіри» – а система, проаналізувавши введені дані, запропонує перелік рекомендованих досліджень (у цьому випадку: ТТГ, вільний Т4, загальний аналіз крові). Технічна реалізація передбачає використання попередньо навченої мовної моделі, дообученої на медичному датасеті симптомів та відповідних лабораторних досліджень. Модель може бути розгорнута як окремий мікросервіс із REST API, до якого звертається основний сервер Monolab. Це дозволить пацієнтам, які не мають направлення від лікаря, отримати обґрунтовану орієнтовну рекомендацію (з обов'язковим застереженням, що фінальне рішення про призначення досліджень залишається за лікарем). Подібні підходи вже досліджуються у медичній інформатиці під назвою «clinical decision support systems» (CDSS) і визнані перспективними для первинної діагностики.

Другим напрямом є інтеграція з електронною системою охорони здоров'я України (eHealth/ЕСОЗ). Це дозволить системі обробляти електронні направлення від лікарів, фіксувати результати досліджень у центральній базі МОЗ та брати участь у Програмі медичних гарантій НСЗУ. Для цього необхідно: зареєструвати Monolab як медичну інформаційну систему (МІС) на порталі eHealth, пройти сертифікацію на тестовому середовищі та реалізувати модуль взаємодії з API ЕСОЗ через кваліфікований електронний підпис (КЕП) медичного закладу.

Третій напрям – підключення реальних платіжних провайдерів. Поточна архітектура PaymentService побудована через абстракцію, що дозволяє підключити LiqPay (ПриватБанк), Fondy або Stripe без зміни інтерфейсу – достатньо реалізувати конкретну стратегію та налаштувати webhook для підтвердження оплати.

Четвертий напрям – інтеграція з лабораторним обладнанням через протоколи HL7 або ASTM для автоматичного імпорту результатів з аналізаторів у систему, що мінімізує ручне введення даних та знизить ризик помилок.

П'ятий напрям – впровадження автоматизованого тестування: unit-тести для бізнес-логіки сервісів (Jest), integration-тести для API (Supertest) та end-to-end тести для мобільного додатку (Detox). Це підвищить надійність системи та спростить подальший розвиток.

ПЕРЕЛІК ПОСИЛАНЬ

1. Plebani M. Errors in clinical laboratories or errors in laboratory medicine? Clinical Chemistry and Laboratory Medicine. 2006. Vol. 44, No. 6. P. 750–759. DOI: 10.1515/CCLM.2006.123.
2. Медична лабораторія ДІЛА: підсумки роботи за 2024 рік. URL: https://dila.ua/about-us.html (дата звернення: 15.04.2026).
3. Коваленко О. С. Застосування веб-технологій для побудови медичних інформаційних систем. Вісник НТУУ «КПІ». Інформатика, управління та обчислювальна техніка. 2023. № 78. С. 112–119.
4. Бондаренко М. Ф., Шабанов-Кушнаренко Ю. П. Основи проєктування програмного забезпечення медичних інформаційних систем. Харків : ХНУРЕ, 2021. 248 с.
5. Шевченко І. Т. Захист персональних медичних даних в інформаційних системах: правові та технічні аспекти. Правова інформатика. 2023. № 2 (78). С. 33–41.
6. Грищук Р. В., Даник Ю. Г. Основи кібернетичної безпеки : монографія. Житомир : ЖНАЕУ, 2019. 636 с.
7. Synevo – Вікіпедія. URL: https://uk.wikipedia.org/wiki/Synevo (дата звернення: 15.04.2026).
8. TypeScript Documentation. URL: https://www.typescriptlang.org/docs/ (дата звернення: 10.04.2026).
9. Node.js Documentation. URL: https://nodejs.org/en/docs/ (дата звернення: 10.04.2026).
10. Express.js Documentation. URL: https://expressjs.com/ (дата звернення: 10.04.2026).
11. React Documentation. URL: https://react.dev/ (дата звернення: 10.04.2026).
12. React Native Documentation. URL: https://reactnative.dev/docs/getting-started (дата звернення: 10.04.2026).
13. PostgreSQL 16 Documentation. URL: https://www.postgresql.org/docs/16/ (дата звернення: 11.04.2026).

14. Prisma ORM Documentation. URL: https://www.prisma.io/docs (дата звернення: 11.04.2026).
15. Redux Toolkit Documentation. URL: https://redux-toolkit.js.org/ (дата звернення: 11.04.2026).
16. Ant Design Documentation. URL: https://ant.design/docs/react/introduce (дата звернення: 12.04.2026).
17. Firebase Cloud Messaging Documentation. URL: https://firebase.google.com/docs/cloud-messaging (дата звернення: 12.04.2026).
18. JSON Web Token Introduction. URL: https://jwt.io/introduction (дата звернення: 12.04.2026).
19. MinIO Object Storage Documentation. URL: https://min.io/docs/minio/linux/index.html (дата звернення: 13.04.2026).
20. Docker Documentation. URL: https://docs.docker.com/ (дата звернення: 13.04.2026).
21. PDFKit Documentation. URL: https://pdfkit.org/ (дата звернення: 13.04.2026).
22. Recharts Documentation. URL: https://recharts.org/en-US/api (дата звернення: 14.04.2026).
23. Zod Documentation. URL: https://zod.dev/ (дата звернення: 14.04.2026).
24. React Navigation Documentation. URL: https://reactnavigation.org/ (дата звернення: 14.04.2026).
25. Ніколін К. А., Жебка В. В. Веб- та мобільний застосунок для медичної лабораторії з використанням TypeScript // Матеріали VI Міжнародної науково-технічної конференції «Сучасний стан та перспективи розвитку IoT». Збірник тез. – К.: ДУІКТ, 2026.
26. Ніколін К. А., Жебка В. В. Автоматизація процесів медичної лабораторії засобами TypeScript-екосистеми // Матеріали VI Всеукраїнської науково-технічної конференції «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. – К.: ДУІКТ, 2026.
27. Helmet.js – HTTP Security Headers Middleware. URL: https://helmetjs.github.io/ (дата звернення: 14.04.2026).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ТЕХНОЛОГІЇ ЦИФРОВОГО РОЗВИТКУ

Цифрова система обслуговування медичної лабораторії з web- та мобільним доступом на основі TypeScript

Виконав студент 4 курсу

Групи ТЦР-42

Ніколін Кирило Андрійович

Керівник роботи

Д.т.н, професор, завідувач кафедри ТЦР Жебка Вікторія Вікторівна

Київ – 2026

1

Титульний слайд

Мета, об'єкт та предмет дослідження

МЕТА РОБОТИ

Підвищити зручність обслуговування пацієнтів медичної лабораторії та скоротити час доставки результатів за рахунок автоматизації повного операційного циклу в єдиному застосунку, що об'єднує онлайн-запис, оплату, перегляд результатів та динаміки показників – від запису до отримання верифікованого результату на телефон пацієнта.

ОБ'ЄКТ ДОСЛІДЖЕННЯ

Процес автоматизації операційної діяльності медичної лабораторії.

ПРЕДМЕТ ДОСЛІДЖЕННЯ

Програмне забезпечення для автоматизації лабораторних процесів із підтримкою веб- та мобільних платформ.

Мета, об’єкт та предмет дослідження

Постановка задачі

Для досягнення поставленої мети визначено наступні задачі:

1. Провести аналіз існуючих рішень у сфері автоматизації медичних лабораторій (Synevo, ДІЛА).
2. Обґрунтувати вибір технологічного стеку та спроєктувати архітектуру системи.
3. Реалізувати серверну частину з JWT-автентифікацією, RBAC та REST API.
4. Розробити веб-застосунок з персональними кабінетами для трьох ролей.
5. Створити мобільний застосунок із push-повідомленнями та PDF-звітами.
6. Провести тестування та оцінити ефективність розробленого рішення.

Постановка задачі

Аналіз аналогів

Функціональність	Synevo	ДІЛА	Monolab
Мобільний застосунок (Україна)	Ні*	**Так (з 2026)**	**Так**
Динаміка показників у мобільному	Ні	Ні	**Так ✓**
Повний цикл лаборанта (відкрито)	Закрита система	Закрита система	**Так ✓**
Push-повідомлення	Ні (UA)	**Так**	**Так (FCM)**
Аудит-лог для адміністратора	Закрита система	Закрита система	**Так ✓**
Self-hosted розгортання	Закрита система	Закрита система	**Так ✓**

** Додаток MySynevo доступний лише в Румунії (перевірено у 2026 р. автором роботи)*

Аналіз аналогів

Вимоги до програмного забезпечення

ФУНКЦІОНАЛЬНІ ВИМОГИ

Облікові записи та доступ

ФВ-1 Самостійна реєстрація пацієнтів (PATIENT)

ФВ-2 Створення облікових записів LAB_TECH та ADMIN – лише через адмін-панель

Робота пацієнта

ФВ-3 Онлайн-запис: вибір відділення, тестів, дати та слоту

ФВ-4 Оплата замовлення; перегляд історії та статусів

ФВ-5 Динаміка показників: графік значень у часі з межами норми

Операційний цикл лаборанта

ФВ-6 Прийом зразка зі зміною статусу; введення значень

ФВ-7 Верифікація результатів старшим лаборантом

ФВ-8 Автоматична генерація PDF-звіту після верифікації

Сповіщення та адміністрування

ФВ-9 Push-повідомлення (FCM) та email пацієнту

ФВ-10 Аудит-лог критичних операцій (автор, IP, час)

НЕФУНКЦІОНАЛЬНІ ВИМОГИ

Безпека

НВ-1 JWT-автентифікація: access (1 год) + refresh (30 днів)

НВ-2 RBAC: PATIENT / LAB_TECH / ADMIN; перевірка на кожному маршруті

НВ-3 Хешування паролів bcrypt (salt rounds = 12)

НВ-4 Helmet.js: CSP, X-Frame-Options, HSTS

Надійність та продуктивність

НВ-5 Час відповіді API ≤ 500 мс під типовим навантаженням

НВ-6 Валідація вхідних даних через Zod на кожному ендпоінті

Сумісність

НВ-7 Веб: Chrome, Firefox, Edge, Safari (актуальні версії)

НВ-8 Мобільний: Android 10+ та iOS 15+

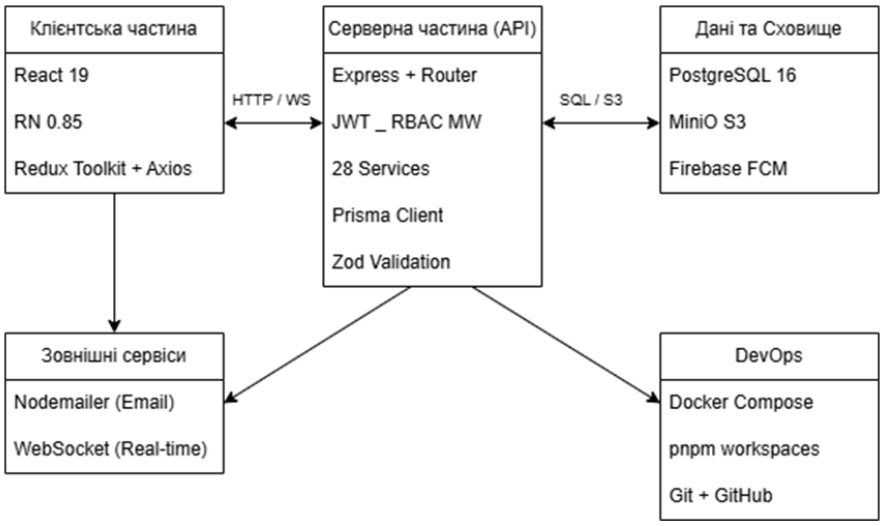
Розгортання

НВ-9 Self-hosted через Docker Compose (PostgreSQL + MinIO + API)

НВ-10 Відповідність вимогам захисту персональних медичних даних

Вимоги до програмного забезпечення

Архітектура системи



Трирівнева архітектура: контролери – сервіси (28) – Prisma Client. Спільні TypeScript-типи між клієнтами та сервером.

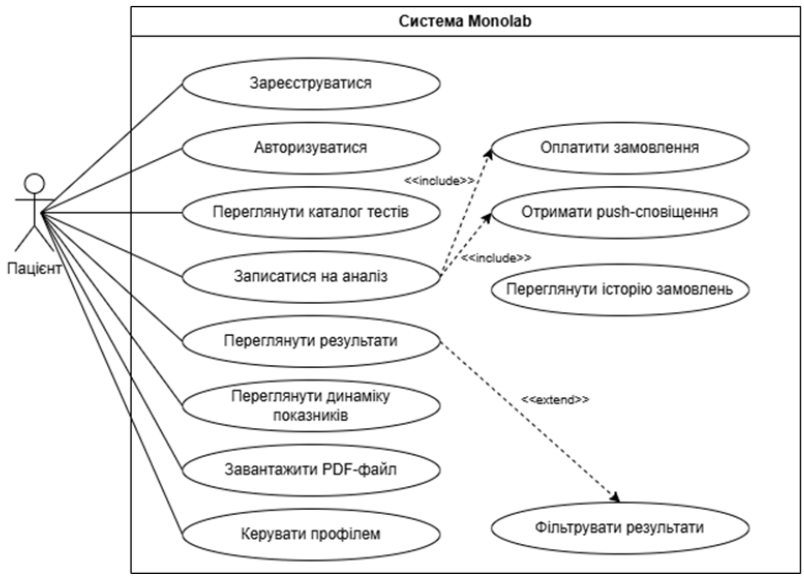
Архітектура системи

Програмні засоби реалізації

Рівень	Технологія	Обґрунтування вибору
Мова	**TypeScript**	Єдина екосистема для API/web/mobile; спільні типи
Backend	**Node.js + Express**	Асинхронний I/O, мінімалістичний фреймворк
База даних	**PostgreSQL 16 + Prisma ORM**	Реляційна СУБД, типобезпечні запити
Веб	**React 19 + Ant Design 6 + Vite**	60+ готових компонентів, миттєвий HMR
Мобільний	**React Native 0.85 (New Architecture)**	Fabric + TurboModules, Android + iOS
Безпека	**JWT + bcrypt + Helmet.js + Zod**	Access/refresh токени, RBAC, валідація
Файли	**MinIO (S3-сумісне)**	Self-hosted, API-сумісність з Amazon S3
Сповіщення	**Firebase Cloud Messaging**	Push на iOS та Android, foreground + background
DevOps	**Docker Compose, Git, ESLint**	Self-hosted розгортання однією командою

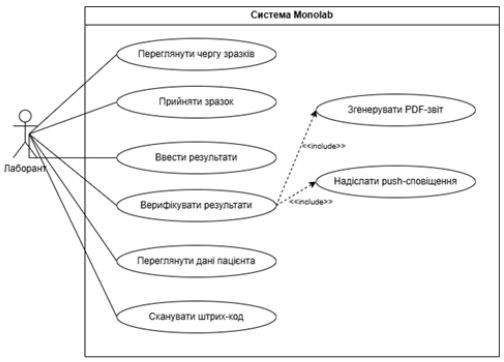
Стек технологій

Діаграма варіантів використання: Пацієнт

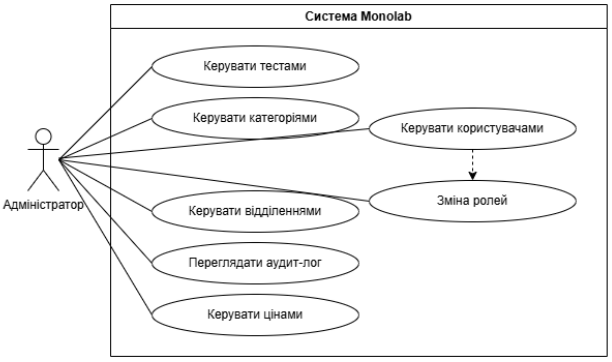


Діаграма варіантів використання (пацієнт)

Діаграми варіантів використання: Лаборант та Адміністратор



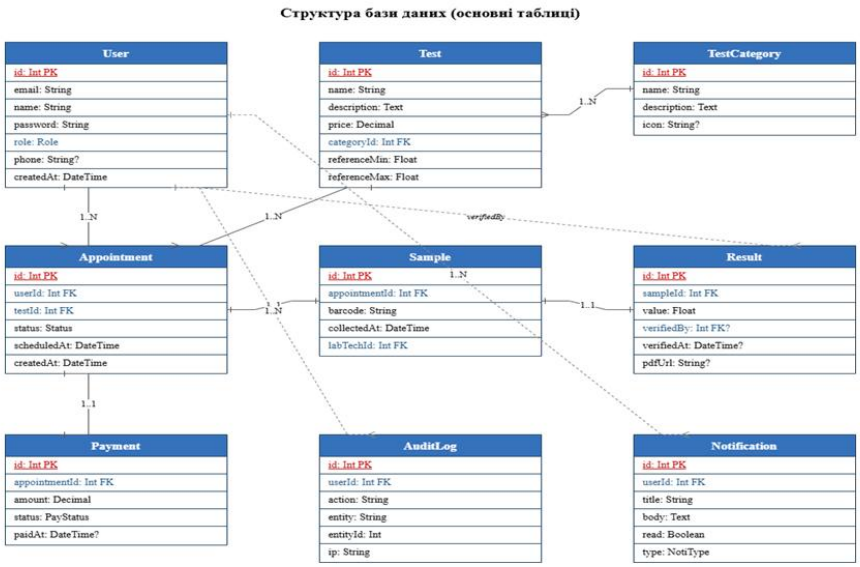
Лаборант: операційний цикл – прийом зразків, введення та верифікація результатів



Адміністратор: керування каталогом, користувачами, ролями; перегляд аналітики та аудит-логу

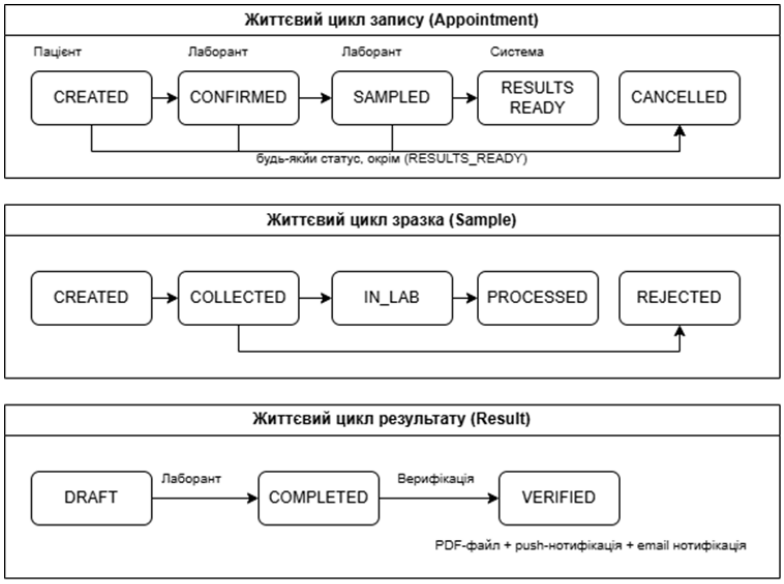
Діаграма варіантів використання (лаборант та адміністратор)

Структура бази даних



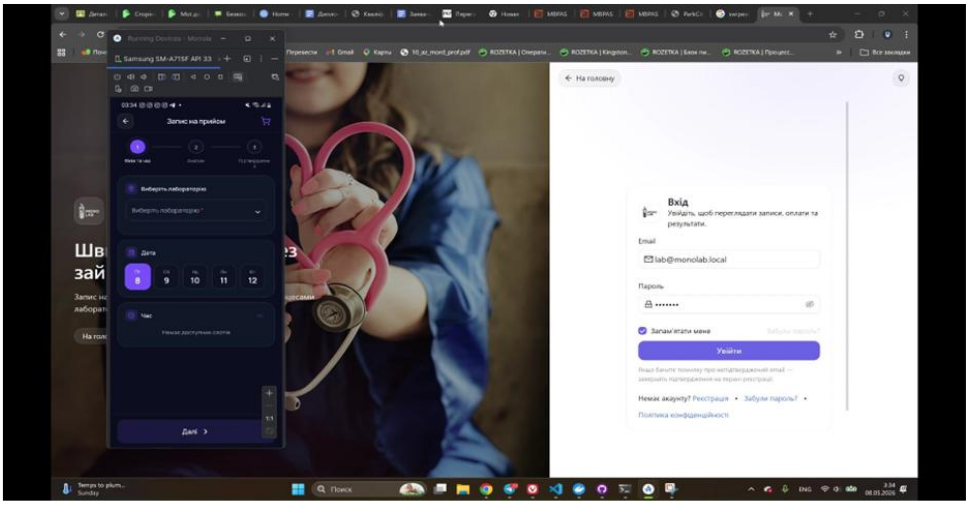
Структура бази даних

Діаграма переходів статусів замовлення



Діаграми переходів статусів замовлення

Демонстрація роботи системи

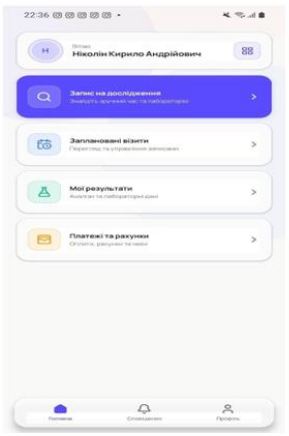


Відеодемонстрація основних сценаріїв: запис на дослідження, перегляд результатів, динаміка показників

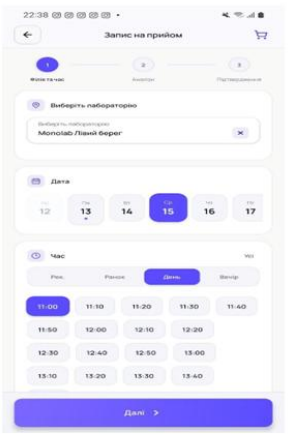
Демонстрація роботи системи

Мобільний застосунок – екранні форми

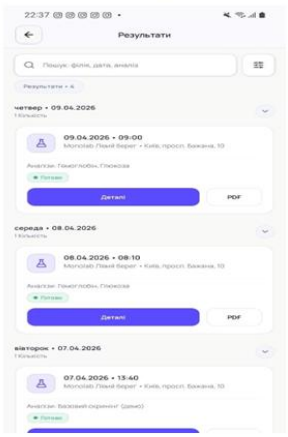
React Native 0.85 · New Architecture (Fabric + TurboModules) · 34 екрани · 62 компоненти



Головний екран



Запис на прийом



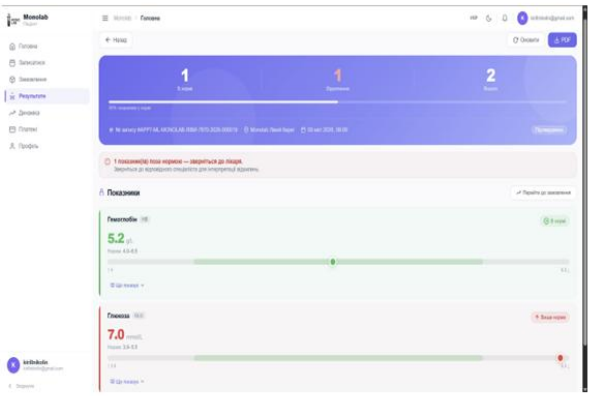
Результати аналізів

Динаміка показників

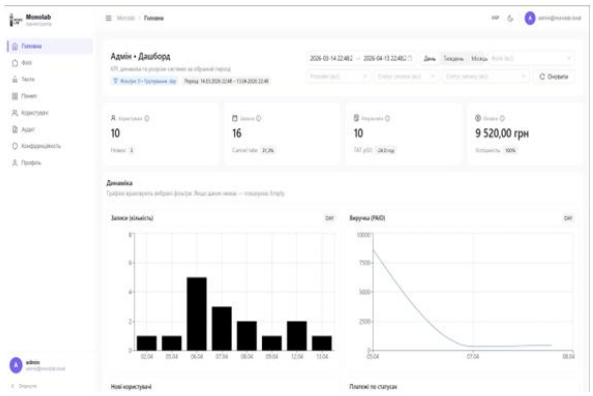
Мобільний застовунок - екранні форми

Веб-кабінет – екранні форми

React 19 + Ant Design 6 · 31 сторінка · Redux Toolkit · Recharts



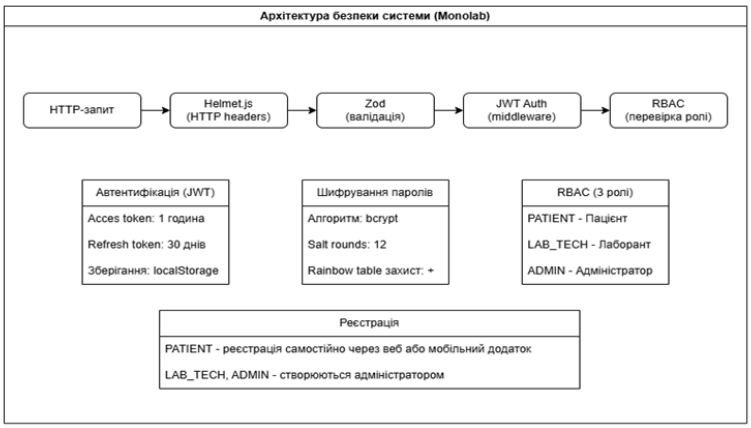
Кабінет пацієнта: результати, динаміка показників



Адмін-панель: KPI-метрики, дашборд, аудит-лог

Веб-панель персоналу - екранні форми

Безпека та real-time комунікація



Ключові механізми

JWT-автентифікація

Access (1 год) + Refresh (30 днів)

RBAC

3 ролі: PATIENT, LAB_TECH, ADMIN

Хешування паролів

bcrypt, salt rounds = 12

HTTP-заголовки

Helmet.js (CSP, X-Frame-Options)

Валідація даних

Zod-схеми на кожному ендпоінті

Аудит-лог

IP, User-Agent, Request ID – кожна дія

WebSocket (Socket.IO)

Real-time оновлення статусів

Безпека та real-time комунікація

Мобільний застосунок – темна тема

Підтримка трьох режимів оформлення: світлий, темний, системний (автоматичне перемикання)

Головний екран

Деталі результату

OrderTimeline

Мобільний застосунок - темна тема

Тестування системи

21	80	4	60
контролер	**API-сценаріїв**	**браузери**	**fps**
120 REST-ендпоінтів	*усі пройдені у Postman*	*Chrome · FF · Edge · Safari*	*Android 14 + iOS Sim*

Рівень	Що тестувалося	Результат
API + бізнес-логіка	Postman: реєстрація, JWT-цикл, повний workflow статусів (Appointment – Sample – Result), генерація PDF, MinIO, аудит-лог	Усі 80 сценаріїв пройдені
Безпека (RBAC)	Спроби доступу PATIENT/LAB_TECH/ADMIN до чужих ресурсів; перевірка 401/403; протестовано через Postman	Несанкціонований доступ блокується
Веб-інтерфейс	Chrome 130+, Firefox 132+, Edge 130+, Safari 18+; адаптивність 375px – 2560px через Responsive Mode	Без артефактів верстки
Мобільний застосунок	Samsung Galaxy (Android 14) + iOS Simulator (iPhone 15 Pro); cold start ≤ 2.5 с; API 150–300 мс; RAM 130–190 МБ	Стабільна робота, 60 fps
User testing	Debug-збірки (v1.0.4) серед групи тестувальників; зворотний зв'язок щодо зручності та стабільності	Критичних дефектів не виявлено

Головний результат: критичних дефектів не виявлено; UX-показники під час тестування підтверджують зручність нового процесу для пацієнта

Результати тестування

Результати та порівняння

Порівняння функціональних можливостей Monolab з провідними комерційними платформами України

Функціональна можливість	Synevo	ДІЛА	Monolab
Онлайн-запис на дослідження	+	+	+
Особистий кабінет із результатами (веб)	+	+	+
PDF-звіт із результатами	*тільки email*	**у кабінеті + push**	**у кабінеті + push**
Мобільний застосунок в Україні	лише Румунія	**+ (з 03.2026)**	+
Push-сповіщення про готовність	–	+	**+ (FCM)**
Динаміка показників у мобільному застосунку	–	–	+
Динаміка показників у веб-кабінеті	вкладка є, не працює	+	+
Інтерфейс для повного циклу лаборанта	закрита система	закрита система	+
Аудит-лог для адміністратора через UI	–	–	+
Self-hosted розгортання (Docker Compose)	закрита система	закрита система	+

УНІКАЛЬНІ МОЖЛИВОСТІ MONOLAB

Динаміка показників у мобільному застосунку · Відкритий інтерфейс для лаборанта · Аудит-лог через UI · Self-hosted розгортання

Результати та порівняння

Апробація результатів та перспективи розвитку

АПРОБАЦІЯ

Результати кваліфікаційної роботи апробовані та прийняті до публікації у вигляді тез на двох науково-технічних конференціях ДУІКТ у 2026 році:

Ніколін К. А., Жебка В. В.
«Веб- та мобільний застосунок для медичної лабораторії з використанням TypeScript»
VI Міжнародна науково-технічна конференція «Сучасний стан та перспективи розвитку IoT». Збірник тез. – К.: ДУІКТ, 2026.

Ніколін К. А., Жебка В. В.
«Автоматизація процесів медичної лабораторії засобами TypeScript-екосистеми»
VI Всеукраїнська науково-технічна конференція «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях». Збірник тез. – К.: ДУІКТ, 2026.

ПЕРСПЕКТИВИ РОЗВИТКУ

AI-асистент
Рекомендація переліку аналізів за описаними симптомами (clinical decision support)

Інтеграція з eHealth
Обмін даними з державною ЕСОЗ через КЕП медзакладу; робота з е-направленнями

Реальні платежі
Підключення LiqPay / Easypay через готову абстракцію PaymentService

Інтеграція з аналізаторами
Автоматичний імпорт результатів за протоколами HL7 / ASTM

Автоматизоване тестування
Unit + integration (Jest, Supertest) і E2E для мобільного (Detox)

Апробація результатів та перспективи розвитку

Висновки

Мету роботи досягнуто: підвищено зручність обслуговування пацієнтів та скорочено час доставки результатів за рахунок автоматизації повного операційного циклу. У межах кваліфікаційної роботи спроєктовано та реалізовано програмне рішення Monolab, що автоматизує повний операційний цикл медичної лабораторії – від онлайн-запису пацієнта до отримання верифікованого результату на телефон. Поставлені у вступі задачі виконано в повному обсязі.

ЧИМ СТАЛО ЗРУЧНІШЕ ДЛЯ КОРИСТУВАЧІВ

Для пацієнта

Запис, оплата, результати і динаміка показників – в одному застосунку на телефоні. Push-сповіщення про готовність – за лічені секунди після верифікації.

Для лаборанта

Єдиний веб-кабінет із чергою зразків, валідованими формами введення значень та автоматичною генерацією PDF після верифікації.

Для адміністратора

Керування каталогом, користувачами та цінами; фінансова аналітика та аудит-лог із фільтрацією – без переходу між системами.

Для лабораторії як бізнесу

Self-hosted розгортання через Docker Compose – повний контроль над даними пацієнтів, без залежності від зовнішнього SaaS.

Підтверджений ефект: інтуїтивніший процес запису, миттєва доставка результатів через push, графічна динаміка показників прямо в застосунку – функціонал, відсутній у наявних рішеннях українського ринку.

Підсумок та перспективи розвитку

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

1. Prisma-схема бази даних (фрагмент)

```
generator client {
 provider = "prisma-client-js"
}
datasource db {
 provider = "postgresql"
 url      = env("DATABASE_URL")
}
enum Role {
 PATIENT
 LAB_TECH
 ADMIN
}
enum AppointmentStatus {
 CREATED
 CONFIRMED
 SAMPLED
 RESULTS_READY
 CANCELLED
}
enum ResultStatus {
 DRAFT
 COMPLETED
 VERIFIED
}
enum SampleStatus {
 CREATED
 COLLECTED
 IN_LAB
 PROCESSED
 REJECTED
}
enum FilePurpose {
 OTHER
 RESULT_PDF
 INVOICE_PDF
 RECEIPT_PDF
}
enum NotificationChannel {
 IN_APP
 EMAIL
 SMS
}
enum NotificationStatus {
 QUEUED
 SENT
 FAILED
 READ
}
enum ConsentStatus {
 GIVEN
 REVOKED
```

```
}
enum AuditAction {
 CREATE
 UPDATE
 DELETE
 LOGIN
 LOGOUT
 REFRESH
 REGISTER
 EXPORT
 AUTH_REGISTER
 AUTH_LOGIN
 AUTH_REFRESH
 AUTH_LOGOUT
}
enum Currency {
 UAH
 USD
 EUR
}
enum PaymentProvider {
 MANUAL
 LIQPAY
 FONDY
 STRIPE
}
enum PaymentStatus {
 PENDING
 PAID
 FAILED
 REFUNDED
}
enum OrderStatus {
 CREATED
 CONFIRMED
 CANCELLED
 PAID
 FULFILLED
}
enum InvoiceStatus {
 ISSUED
 PAID
 VOID
}
enum StandardSystem {
 LOINC
 FHIR
}
enum Sex {
 UNKNOWN
 MALE
 FEMALE
}
enum RefRangeType {
 NUMERIC
```

```
 LESS_THAN
 GREATER_THAN
}
enum Weekday {
 MON
 TUE
 WED
 THU
 FRI
 SAT
 SUN
}
enum DocumentKind {
 ORDER
 INVOICE
 RECEIPT
 APPOINTMENT
 RESULT
}
enum PolicyStatus {
 DRAFT
 PUBLISHED
}
enum TicketStatus {
 OPEN
 WAITING_REPLY
 ANSWERED
 CLOSED
}
enum TicketCategory {
 GENERAL
 BILLING
 RESULTS
 TECHNICAL
 OTHER
}
model User {
 id                 String                  @id @default(cuid())
 email              String                  @unique
 passwordHash       String
 role               Role                    @default(PATIENT)
 emailVerifiedAt    DateTime?
 createdAt          DateTime                @default(now())
 updatedAt          DateTime                @updatedAt
 patientProfile     PatientProfile?
 staff              Staff?
 appointments       Appointment[]
 notifications      Notification[]          @relation("NotificationRecipient")
 orders             Order[]
 payments           Payment[]               @relation("PaymentUser")
 invoices           Invoice[]               @relation("InvoiceUser")
 files              File[]
 refreshTokens      RefreshToken[]
 auditLogs          AuditLog[]              @relation("AuditActor")
 pushTokens         PushToken[]
```

```
 emailVerifications EmailVerificationCode[]
 passwordResets     PasswordResetCode[]
 supportTickets     SupportTicket[]
}
model PatientProfile {
 id        String    @id @default(cuid())
 userId    String    @unique
 fullName  String
 phone     String?
 birthDate DateTime?
 sex       Sex       @default(UNKNOWN)
 createdAt DateTime  @default(now())
 updatedAt DateTime  @updatedAt
 user      User      @relation(fields: [userId], references: [id], onDelete:
Cascade)
 consents  Consent[]
}
model Staff {
 id              String   @id @default(cuid())
 userId          String   @unique
 fullName        String
 position        String?
 createdAt       DateTime @default(now())
 updatedAt       DateTime @updatedAt
 user            User     @relation(fields: [userId], references: [id], onDelete:
Cascade)
 resultsEntered  Result[] @relation("ResultEnteredBy")
 resultsVerified Result[] @relation("ResultVerifiedBy")
}
model Appointment {
 id        String            @id @default(cuid())
 number    String?           @unique
 userId    String
 branchId  String
 status    AppointmentStatus @default(CREATED)
 slotStart DateTime
 slotEnd   DateTime?
 notes     String?
 createdAt DateTime          @default(now())
 updatedAt DateTime          @updatedAt
 user      User              @relation(fields: [userId], references: [id],
onDelete: Restrict)
 branch    Branch            @relation(fields: [branchId], references: [id],
onDelete: Restrict)
 samples   Sample[]
 result    Result?
 orderId   String?
 order     Order?            @relation(fields: [orderId], references: [id],
onDelete: SetNull)
 panelId   String?
 panel     Panel?            @relation(fields: [panelId], references: [id],
onDelete: SetNull)


 @@index([number])
```

```
 @@index([panelId])
 @@index([userId])
 @@index([branchId])
 @@index([slotStart])
 @@index([branchId, slotStart])
 @@index([slotEnd])
 @@index([branchId, slotEnd])
}
model Sample {
 id            String       @id @default(cuid())
 appointmentId String
 sampleTypeId  String?
 status        SampleStatus @default(CREATED)
 barcode       String?      @unique
 qrCode        String?      @unique
 collectedAt   DateTime?
 createdAt     DateTime     @default(now())
 updatedAt     DateTime     @updatedAt
 appointment   Appointment  @relation(fields: [appointmentId], references: [id],
onDelete: Cascade)
 sampleType    SampleType?  @relation(fields: [sampleTypeId], references: [id],
onDelete: SetNull)


 @@index([appointmentId])
 @@index([status])
}
model Result {
 id                String       @id @default(cuid())
 number            String?      @unique
 appointmentId     String       @unique
 status            ResultStatus @default(DRAFT)
 enteredByStaffId  String?
 verifiedByStaffId String?
 createdAt         DateTime     @default(now())
 updatedAt         DateTime     @updatedAt
 appointment       Appointment  @relation(fields: [appointmentId], references:
[id], onDelete: Cascade)
 enteredBy         Staff?       @relation("ResultEnteredBy", fields:
[enteredByStaffId], references: [id], onDelete: SetNull)
 verifiedBy        Staff?       @relation("ResultVerifiedBy", fields:
[verifiedByStaffId], references: [id], onDelete: SetNull)
 items             ResultItem[]
 files             File[]       @relation("ResultFiles")


 @@index([number])
}
model ResultItem {
 id        String   @id @default(cuid())
 resultId  String
 testId    String
 value     String
 unit      String?
 refRange  String?
```

```
 createdAt DateTime @default(now())
 updatedAt DateTime @updatedAt
 result    Result   @relation(fields: [resultId], references: [id], onDelete:
Cascade)
 test      Test     @relation(fields: [testId], references: [id], onDelete:
Restrict)


 @@unique([resultId, testId])
 @@index([resultId])
 @@index([testId])
}
model AuditLog {
 id          String      @id @default(cuid())
 createdAt   DateTime    @default(now())
 requestId   String?
 ip          String?
 userAgent   String?
 actorUserId String?
 actor       User?       @relation("AuditActor", fields: [actorUserId],
references: [id], onDelete: SetNull)
 action      AuditAction
 entity      String?
 entityId    String?
 message     String?
 metaJson    Json?


 @@index([createdAt])
 @@index([actorUserId])
 @@index([requestId])
 @@index([entity, entityId])
 @@index([action])
}
model Order {
 id           String        @id @default(cuid())
 number       String?       @unique
 branchId     String?
 branch       Branch?       @relation(fields: [branchId], references: [id],
onDelete: SetNull)
 userId       String
 status       OrderStatus   @default(CREATED)
 currency     Currency      @default(UAH)
 totalAmount  Decimal       @db.Decimal(12, 2)
 notes        String?
 createdAt    DateTime      @default(now())
 updatedAt    DateTime      @updatedAt
 user         User          @relation(fields: [userId], references: [id],
onDelete: Restrict)
 items        OrderItem[]
 payments     Payment[]
 invoice      Invoice?
 appointments Appointment[]
 files        File[]        @relation("OrderFiles")
```

```
 @@index([userId])
 @@index([branchId])
 @@index([status])
 @@index([createdAt])
}

model Payment {
 id                String          @id @default(cuid())
 receiptNumber     String?         @unique
 branchId          String?
 branch            Branch?         @relation(fields: [branchId], references: [id],
onDelete: SetNull)
 userId            String?
 orderId           String?
 provider          PaymentProvider @default(MANUAL)
 status            PaymentStatus   @default(PENDING)
 currency          Currency        @default(UAH)
 amount            Decimal         @db.Decimal(12, 2)
 providerPaymentId String?
 metaJson          Json?
 createdAt         DateTime        @default(now())
 updatedAt         DateTime        @updatedAt
 paidAt            DateTime?
 user              User?           @relation("PaymentUser", fields: [userId],
references: [id], onDelete: SetNull)
 order             Order?          @relation(fields: [orderId], references: [id],
onDelete: SetNull)
 files             File[]          @relation("PaymentFiles")


 @@index([branchId])
 @@index([status])
 @@index([provider])
 @@index([createdAt])
}
```

2. Middleware автентифікації

```
import { Request, Response, NextFunction } from "express";
import { verifyAccessToken } from "../auth/tokens";


export function authMiddleware(req: Request, res: Response, next: NextFunction) {
 const headerRaw = req.headers.authorization;
 const header = typeof headerRaw === "string" ? headerRaw.trim() : "";


 const [scheme, token] = header.split(/\s+/);


 if (scheme !== "Bearer" || !token) {
   return res.status(401).json({ error: "UNAUTHORIZED" });
 }
```

```
 try {
   const payload = verifyAccessToken(token);
   req.user = { id: payload.sub, role: payload.role };
   return next();
 } catch (e: any) {


   return res.status(401).json({ error: "UNAUTHORIZED" });
 }
}
```

3. Сервіс результатів

```
const allowedResultTransitions: Record<ResultStatus, ResultStatus[]> = {
 DRAFT: [ResultStatus.COMPLETED],
 COMPLETED: [ResultStatus.VERIFIED, ResultStatus.DRAFT],
 VERIFIED: [],
};
export async function setResultStatus(params: SetResultStatusParams) {
 const existing = await prisma.result.findUnique({
   where: { id: params.resultId },
   select: { id: true, status: true },
 });
 if (!existing) {
   await auditSafe({
     actorUserId: params.actorUserId,
     action: AuditAction.UPDATE,
     entity: "Result",
     entityId: params.resultId,
     message: "Result status change failed: RESULT_NOT_FOUND",
     metaJson: { requested: params.status },
     ctx: params.ctx,
   });
   return { error: "RESULT_NOT_FOUND" as const };
 }
 if (existing.status === ResultStatus.VERIFIED) {
   await auditSafe({
     actorUserId: params.actorUserId,
     action: AuditAction.UPDATE,
     entity: "Result",
     entityId: existing.id,
     message: "Result status change blocked: RESULT_LOCKED",
     metaJson: { current: existing.status, requested: params.status },
     ctx: params.ctx,
   });
   return { error: "RESULT_LOCKED" as const };
 }
 const allowed = allowedResultTransitions[existing.status] ?? [];
 if (!allowed.includes(params.status)) {
   await auditSafe({
     actorUserId: params.actorUserId,
     action: AuditAction.UPDATE,
     entity: "Result",
     entityId: existing.id,
```

```
      message: "Result status change failed: INVALID_STATUS_TRANSITION",
      metaJson: { current: existing.status, requested: params.status, allowed },
      ctx: params.ctx,
    });
    return {
      error: "INVALID_STATUS_TRANSITION" as const,
      current: existing.status,
      next: params.status,
    };
  }
  const updated = await prisma.result.update({
    where: { id: params.resultId },
    data: { status: params.status },
    select: { id: true, status: true, appointmentId: true, number: true },
  });
  if (updated.status === ResultStatus.VERIFIED) {
    try {
      const row = await prisma.result.findUnique({
        where: { id: updated.id },
        select: {
          id: true,
          number: true,
          appointment: {
            select: {
              id: true,
              number: true,
              userId: true,
              user: {
                select: {
                  email: true,
                  patientProfile: { select: { fullName: true } },
                },
              },
            },
          },
        },
      });
      const patientUserId = row?.appointment?.userId ?? null;
      const appointmentNumber = row?.appointment?.number ?? null;
      const resultNumber = row?.number ?? null;
      if (patientUserId) {
        await Notifications.notifyResultsReady({
          patientUserId,
          appointmentNumber,
          resultId: row!.id,
        });
      }
      const patientEmail = row?.appointment?.user?.email ?? null;
      if (patientEmail) {
        const fullName = row?.appointment?.user?.patientProfile?.fullName ?? null;
        const pdfBuffer = await
ResultsPdf.buildAppointmentResultPdfBuffer(row!.appointment.id);
        await Email.sendResultsReadyEmail({
          to: patientEmail,
          fullName,
```

```
          appointmentNumber,
          pdfBuffer,
          pdfFileName: `${resultNumber ?? "result"}.pdf`,
        });
      } else {
        console.warn("[results] VERIFIED but patient email is null", { resultId:
updated.id });
      }
    } catch (e) {
      console.error("[results] notify/email on VERIFIED failed", e);
    }
  }
  await auditSafe({
    actorUserId: params.actorUserId,
    action: AuditAction.UPDATE,
    entity: "Result",
    entityId: updated.id,
    message: "Result status changed",
    metaJson: { from: existing.status, to: updated.status },
    ctx: params.ctx,
  });
  return prisma.result.findUnique({
    where: { id: updated.id },
    include: {
      appointment: { include: { user: { select: { id: true, email: true, role:
true, patientProfile: true } }, branch: true, panel: true } },
      items: { include: { test: true }, orderBy: { createdAt: "asc" as const } },
    },
  });
}
```

4. Сервіс генерації PDF

```
export async function renderResultsPdf(params: { appointmentId: string }) {
  const appointmentId = params.appointmentId;
  const result = await prisma.result.findUnique({
    where: { appointmentId },
    include: {
      appointment: {
        include: {
          user: { include: { patientProfile: true } },
          branch: true,
          panel: true,
        },
      },
      items: { include: { test: true }, orderBy: { createdAt: "asc" } },
    },
  });
  if (!result) return { error: "RESULT_NOT_FOUND" as const };
  const appt = result.appointment;
  const patientProfile = appt?.user?.patientProfile;
  const appointmentNumber = appt.number ?? "-";
  const resultNumber = result.number ?? "-";
  const doc = new PDFDocument({
    size: "A4",
```

```
    margin: 48,
    info: {
      Title: `Monolab - Результат ${resultNumber} (Візит ${appointmentNumber})`,
      Author: getBrandConfig().companyName ?? "Monolab",
    },
  });
  applyBrandFonts(doc);
  const pager = createPager(doc, {
    drawHeader: () =>
      drawResultsHeader(doc, {
        guaranteeLine: "Гарантія точності та достовірності результатів досліджень",
        accreditationLine:
          "Навчальний (тестовий) режим: результати сформовано в межах дипломного
проєкту. Акредитація та атестат не застосовуються.",
        addressLine: "Київ, Україна",
        website: "monolab.local",
        email: "support@monolab.local",
        infoRight: "Інформаційно-сервісна служба: +380 XX XXX XX XX",
      }),
    drawFooter: (n) => {
      drawDisclaimerLine(doc);
      addFooter(doc, n);
    },
  });
  drawResultsHeader(doc, {
    guaranteeLine: "Гарантія точності та достовірності результатів досліджень",
    accreditationLine:
      "Навчальний (тестовий) режим: результати сформовано в межах дипломного
проєкту. Акредитація та атестат не застосовуються.",
    addressLine: "Київ, Україна",
    website: "monolab.local",
    email: "support@monolab.local",
    infoRight: "Інформаційно-сервісна служба: +380 XX XXX XX XX",
    reportTitleBelow: `Результат ${resultNumber}`,
  });
  drawSectionBar(doc, "ДАНІ ПАЦІЄНТА ТА ВІЗИТУ", pager.pageBreak);
  drawPatientCard(
    doc,
    {
      patientName: safeText(patientProfile?.fullName ?? appt.user?.email ?? "-"),
      birthDate: patientProfile?.birthDate ?? null,
      sex: (patientProfile as any)?.sex ?? null,
      email: appt.user?.email ?? null,


      branchName: appt.branch?.name ?? null,
      branchAddress: appt.branch?.address ?? null,


      appointmentDate: appt.slotStart ?? null,
      appointmentNumber: appt.number ?? null,
      resultNumber: result.number ?? null,


      panelName: appt.panel?.name ?? null,
```

```
    },
    pager.pageBreak
  );
  drawSectionBar(doc, "РЕЗУЛЬТАТИ ДОСЛІДЖЕНЬ", pager.pageBreak);
  const atDate = appt.slotStart ?? new Date();
  const rrList = await Promise.all(
    result.items.map((it) =>
      resolveRefRange({
        testId: it.testId,
        fallbackRefRange: it.refRange ?? null,
        testRefRange: it.test?.refRange ?? null,
        patientSex: (patientProfile as any)?.sex ?? null,
        patientBirthDate: patientProfile?.birthDate ?? null,
        at: atDate,
      })
    )
  );
  const rows: ResultsRow[] = result.items.map((it, idx) => {
    const rr = rrList[idx];
    const raw = String(it.value ?? "").trim();
    const value = raw;
    const interpretation = interpretValue(value, rr);
    const label = interpretationLabelUA(interpretation);
    const sourceLabel =
      rr.source === "DB"
        ? "Референс: база"
        : rr.source === "RESULT_ITEM"
          ? "Референс: вручну"
          : rr.source === "TEST"
            ? "Референс: тест"
            : "-";
    const testName = it.test?.name ?? it.test?.code ?? "-";
    return {
      name: safeText(testName),
      value: value.length ? value : "-",
      unit: safeText(it.unit ?? it.test?.unit ?? "-"),
      ref: rr.display ?? "-",
      interpretation: label,
      isOutOfRange: interpretation === "OUT_OF_RANGE",
      comment: sourceLabel,
    };
  });
  drawResultsTable(doc, rows, pager.pageBreak);
  drawSectionBar(doc, "ПІДПИСАННЯ РЕЗУЛЬТАТІВ", pager.pageBreak);
  drawSignoff(doc, pager.pageBreak);
  pager.finalize();
  return await bufferFromDoc(doc);
}
export async function buildAppointmentResultPdfBuffer(appointmentId: string):
Promise<Buffer> {
  const out = await renderResultsPdf({ appointmentId });
  if (Buffer.isBuffer(out)) return out;
  throw new Error(out?.error ?? "RESULT_PDF_BUILD_FAILED");
}
```

5. Сервіс динаміки показників

```
export type TrendDirection = "UP" | "DOWN" | "FLAT";
export type TrendStatus = "BETTER" | "WORSE" | "NEUTRAL" | "UNKNOWN";
export type PatientTestTrendPointDto = {
 date: string;
 value: number | null;
 rawValue?: string | null;
 unit?: string | null;
 refRange?: string | null;
 appointmentId?: string | null;
 appointmentNumber?: string | null;
};
export type PatientTestTrendDto = {
 testId: string;
 testCode?: string | null;
 testName: string;
 unit?: string | null;
 lastValue: number | null;
 prevValue: number | null;
 deltaAbs: number | null;
 deltaPct: number | null;
 direction: TrendDirection;
 status: TrendStatus;
 points: PatientTestTrendPointDto[];
};
export type PatientDynamicsListRowDto = {
 testId: string;
 testCode?: string | null;
 testName: string;
 unit?: string | null;
 lastDate: string | null;
 lastValue: number | null;
 direction: TrendDirection;
 status: TrendStatus;
};
type NumericParse = | { kind: "NUM"; value: number; raw: string } | { kind: "LT";
value: number; raw: string } | { kind: "GT"; value: number; raw: string } | {
kind: "NA"; raw: string };
function parseNumericValue(raw: unknown): NumericParse {
 const s = String(raw ?? "").trim();
 if (!s) return { kind: "NA", raw: "" };
 const ss = s.replace(/\s+/g, "").replace(",", ".");
 const mLT = ss.match(/^<\s*([0-9]+(?:\.[0-9]+)?)$/);
 if (mLT) return { kind: "LT", value: Number(mLT[1]), raw: s };
 const mGT = ss.match(/^>\s*([0-9]+(?:\.[0-9]+)?)$/);
 if (mGT) return { kind: "GT", value: Number(mGT[1]), raw: s };
 const mNum = ss.match(/^([+-]?[0-9]+(?:\.[0-9]+)?)$/);
 if (mNum) return { kind: "NUM", value: Number(mNum[1]), raw: s };
 const mAny = ss.match(/([+-]?[0-9]+(?:\.[0-9]+)?)/);
 if (mAny) return { kind: "NUM", value: Number(mAny[1]), raw: s };
 return { kind: "NA", raw: s };
}
```

```
type RangeParse = { min: number | null; max: number | null; raw: string | null };
function parseRefRange(raw: unknown): RangeParse {
 if (raw == null) return { min: null, max: null, raw: null };
 const s0 = String(raw).trim();
 if (!s0) return { min: null, max: null, raw: null };
 const s = s0.replace(",", ".").replace(/\s+/g, "");
 const m = s.match(/^([+-]?[0-9]+(?:\.[0-9]+)?)[--]([+-]?[0-9]+(?:\.[0-9]+)?)$/);
 if (m) {
   const a = Number(m[1]);
   const b = Number(m[2]);
   if (Number.isFinite(a) && Number.isFinite(b)) return { min: Math.min(a, b),
max: Math.max(a, b), raw: s0 };
 }
 const mLess = s.match(/^(?:<|≤)([+-]?[0-9]+(?:\.[0-9]+)?)$/);
 if (mLess) {
   const x = Number(mLess[1]);
   if (Number.isFinite(x)) return { min: null, max: x, raw: s0 };
 }
 const mGreater = s.match(/^(?:>|≥)([+-]?[0-9]+(?:\.[0-9]+)?)$/);
 if (mGreater) {
   const x = Number(mGreater[1]);
   if (Number.isFinite(x)) return { min: x, max: null, raw: s0 };
 }
 return { min: null, max: null, raw: s0 };
}
export async function getTestTrendForUser(params: {
 testId: string;
 actorUserId: string;
 actorRole: Role;
 range?: string | null;
 requestContext?: RequestContext;
}): Promise<PatientTestTrendDto | Err> {
 const ctx = params.requestContext;
 if (!isNonEmptyString(params.actorUserId)) return err(401, "UNAUTHORIZED",
"UNAUTHORIZED", "errors.unauthorized");
 if (!isNonEmptyString(params.testId)) return err(400, "BAD_REQUEST",
"TEST_ID_REQUIRED", "errors.testIdRequired");
 if (params.actorRole !== Role.PATIENT && params.actorRole !== Role.ADMIN &&
params.actorRole !== Role.LAB_TECH) {
   return err(403, "FORBIDDEN", "FORBIDDEN", "errors.forbidden");
 }
 const rangeParsed = parseRangeParam(params.range);
 if ("error" in (rangeParsed as any)) return rangeParsed as Err;
 const test = await prisma.test.findUnique({
   where: { id: params.testId },
   select: { id: true, code: true, name: true, unit: true, refRange: true },
 });
 if (!test) {
   await auditSafe({
     actorUserId: params.actorUserId,
     action: AuditAction.EXPORT,
     entity: "Trend",
     entityId: params.testId,
     message: "Trend failed: test not found",
     metaJson: { testId: params.testId },
```

```
      ctx,
    });
    return err(404, "NOT_FOUND", "TEST_NOT_FOUND", "errors.testNotFound", { testId:
params.testId });
  }
  const baseWhere = buildResultItemWhereForUser({ actorUserId: params.actorUserId,
startAt: rangeParsed.startAt });
  const where: Prisma.ResultItemWhereInput = {
    ...baseWhere,
    testId: params.testId,
  };
  const items = await prisma.resultItem.findMany({
    where,
    orderBy: { result: { appointment: { slotStart: "asc" } } },
    select: {
      id: true,
      value: true,
      unit: true,
      refRange: true,
      createdAt: true,
      result: {
        select: {
          id: true,
          status: true,
          appointment: {
            select: {
              id: true,
              number: true,
              slotStart: true,
              branchId: true,
            },
          },
        },
      },
    },
  });
  const points: PatientTestTrendPointDto[] = items.map((it) => {
    const measuredAt = it.result?.appointment?.slotStart ?? it.createdAt;
    const parsed = parseNumericValue(it.value);
    const valueNum =
      parsed.kind === "NUM" ? parsed.value : parsed.kind === "LT" ? parsed.value :
parsed.kind === "GT" ? parsed.value : null;
    return {
      date: measuredAt.toISOString(),
      value: valueNum,
      rawValue: valueNum == null ? String(it.value ?? "") : null,
      unit: (it.unit ?? test.unit ?? null) as string | null,
      refRange: (it.refRange ?? test.refRange ?? null) as string | null,
      appointmentId: it.result?.appointment?.id ?? null,
      appointmentNumber: it.result?.appointment?.number ?? null,
    };
  });
```

```
 const numericPoints = points.filter((p) => typeof p.value === "number" &&
Number.isFinite(p.value as number));
 const last = numericPoints.length ? numericPoints[numericPoints.length - 1] :
null;
 const prev = numericPoints.length >= 2 ? numericPoints[numericPoints.length - 2]
: null;
 const lastValue = last?.value ?? null;
 const prevValue = prev?.value ?? null;
 let deltaAbs: number | null = null;
 let deltaPct: number | null = null;
 if (lastValue != null && prevValue != null) {
   deltaAbs = lastValue - prevValue;
   deltaPct = safePct(deltaAbs, prevValue);
 }
 let status: TrendStatus = "UNKNOWN";
 if (lastValue != null && prevValue != null) {
   const lastRange = parseRefRange(last?.refRange ?? test.refRange);
   const prevRange = parseRefRange(prev?.refRange ?? test.refRange);
   const internal = deriveTrendFromTwoPoints({
     prevValue,
     prevRange,
     curValue: lastValue,
     curRange: lastRange,
   });
   status = toStatusFromTrend(internal.trend);
 }
 const dto: PatientTestTrendDto = {
   testId: test.id,
   testCode: test.code ?? null,
   testName: test.name,
   unit: test.unit ?? null,
   lastValue,
   prevValue,
   deltaAbs,
   deltaPct,
   direction: toDirection(deltaAbs),
   status,
   points,
 };
 await auditSafe({
   actorUserId: params.actorUserId,
   action: AuditAction.EXPORT,
   entity: "Trend",
   entityId: test.id,
   message: "Trend viewed",
   metaJson: {
     testId: test.id,
     range: rangeParsed.label,
     points: points.length,
   },
   ctx,
 });
 return dto;
}
```

6. Компонент динаміки (веб)

```
export type TrendRange = "7d" | "3m" | "6m" | "12m" | "all";
export type TrendDirection = "UP" | "DOWN" | "FLAT";
export type TrendStatus = "BETTER" | "WORSE" | "NEUTRAL" | "UNKNOWN";
export type PatientTestTrendPointDto = {
 date: string;
 value: number | null;
 rawValue?: string | null;
 unit?: string | null;
 refRange?: string | null;
 appointmentId?: string | null;
 appointmentNumber?: string | null;
};
export type PatientTestTrendDto = {
 testId: string;
 testCode?: string | null;
 testName: string;
 unit?: string | null;
 lastValue: number | null;
 prevValue: number | null;
 deltaAbs: number | null;
 deltaPct: number | null;
 direction: TrendDirection;
 status: TrendStatus;
 points: PatientTestTrendPointDto[];
};
export type PatientDynamicsListRowDto = {
 testId: string;
 testCode?: string | null;
 testName: string;
 unit?: string | null;
 lastDate: string | null;
 lastValue: number | null;
 direction: TrendDirection;
 status: TrendStatus;
};
function parseRefRangeForLines(raw?: string | null): { min: number | null; max:
number | null } {
 if (!raw) return { min: null, max: null };
 const s0 = String(raw).trim();
 if (!s0) return { min: null, max: null };
 const s = s0.replace(",", ".").replace(/\s+/g, "");
 const m = s.match(/^([+-]?[0-9]+(?:\.[0-9]+)?)[--]([+-]?[0-9]+(?:\.[0-9]+)?)$/);
 if (m) {
   const a = Number(m[1]);
   const b = Number(m[2]);
   if (Number.isFinite(a) && Number.isFinite(b)) return { min: Math.min(a, b),
max: Math.max(a, b) };
 }
 const mLess = s.match(/^(?:<|≤)([+-]?[0-9]+(?:₩.[0-9]+)?)$/);
 if (mLess) {
   const x = Number(mLess[1]);
   if (Number.isFinite(x)) return { min: null, max: x };
 }
```

```
 const mGreater = s.match(/^(?:>|≥)([+−]?[0−9]+(?:₩.[0−9]+)?)$/);
 if (mGreater) {
   const x = Number(mGreater[1]);
   if (Number.isFinite(x)) return { min: x, max: null };
 }
 return { min: null, max: null };
}
type SparkPoint = { x: string; y: number | null };
function buildSpark(points: PatientTestTrendPointDto[]): SparkPoint[] {
 return points.map((p) => ({
   x: fmtDate(p.date),
   y: typeof p.value === "number" && Number.isFinite(p.value) ? p.value : null,
 }));
}
```

7. Docker Compose конфігурація

```
version: "3.9"
services:
 postgres:
   image: postgres:16-alpine
   container_name: monolab-postgres
   restart: unless-stopped
   environment:
     POSTGRES_DB: monolab
     POSTGRES_USER: monolab
     POSTGRES_PASSWORD: monolab_password
   ports:
     - "5433:5432"
   volumes:
     - monolab_pg_data:/var/lib/postgresql/data
   healthcheck:
     test: ["CMD-SHELL", "pg_isready -U monolab -d monolab"]
     interval: 5s
     timeout: 3s
     retries: 20
 adminer:
   image: adminer:4
   container_name: monolab-adminer
   restart: unless-stopped
   depends_on:
     postgres:
       condition: service_healthy
   ports:
     - "8081:8080"
 minio:
   image: minio/minio:latest
   container_name: monolab-minio
   command: server /data --console-address ":9001"
   restart: unless-stopped
   environment:
     MINIO_ROOT_USER: minioadmin
     MINIO_ROOT_PASSWORD: minioadmin
   ports:
     - "9000:9000"
```

```
    - "9001:9001"
  volumes:
    - monolab_minio_data:/data
minio-mc:
  image: minio/mc
  container_name: monolab-minio-mc
  depends_on:
    - minio
  entrypoint: >
    /bin/sh -c "
    sleep 5;
    mc alias set monolab http://minio:9000 minioadmin minioadmin;
    mc mb -p monolab/monolab-files || true;
    mc anonymous set download monolab/monolab-files || true;
    exit 0;
    "
api:
  build: ../apps/api
  container_name: monolab-api
  restart: unless-stopped
  depends_on:
    postgres:
      condition: service_healthy
    minio:
      condition: service_started
  ports:
    - "3001:3001"
  environment:
    NODE_ENV: development
    PORT: 3001
    DATABASE_URL: postgresql://monolab:monolab_password@postgres:5432/monolab
    S3_ENDPOINT: http://minio:9000
    S3_REGION: us-east-1
    S3_ACCESS_KEY_ID: minioadmin
    S3_SECRET_ACCESS_KEY: minioadmin
    S3_BUCKET: monolab-files
    S3_FORCE_PATH_STYLE: "true"
    APP_PUBLIC_BASE_URL: http://localhost:3001
    GOOGLE_APPLICATION_CREDENTIALS: /run/secrets/firebase.json
    CI: "true"
    PNPM_STORE_DIR: /pnpm-store
  volumes:
    - ../apps/api:/app
    - monolab_api_node_modules:/app/node_modules
    - monolab_pnpm_store:/pnpm-store
    - ./secrets/monolab-65568-ea38bd2c3351.json:/run/secrets/firebase.json:ro
  command: sh -lc "corepack enable && pnpm i --frozen-lockfile && pnpm prisma
generate && pnpm dev"
volumes:
 monolab_pg_data:
 monolab_minio_data:
 monolab_api_node_modules:
 monolab_pnpm_store:
```