МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Сумський державний університет

Факультет електроніки та інформаційних технологій

Кафедра комп'ютерних наук

«До захисту допущено»

В.о. завідувача кафедри

_____ Оксана ШОВКОПЛЯС
(підпис)

04 грудня 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня магістр

зі спеціальності 122 «Комп'ютерні науки»

освітньо-професійної програми «Інформатика»

на тему: Інформаційна технологія моніторингу здоров'я для підтримки прийняття рішень та забезпечення персоналізованого підходу

здобувача групи ІН.м-33 Чупіки Артема Миколайовича

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

_____ Артем ЧУПІКА
(підпис)

Керівник,

асистент кафедри комп'ютерних наук

кандидат фізико-математичних наук

Олександр ВЛАСЕНКО _____
(підпис)

Суми - 2024

Сумський державний університет
Факультет електроніки та інформаційних технологій
Кафедра комп'ютерних наук

«Затверджую»
В.о. завідувача кафедри
_____ Оксана ШОВКОПЛЯС
(підпис)

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістра

зі спеціальності 122 «Комп'ютері науки», освітньо-професійної програми «Інформатика»
здобувача групи ІН.м-33 Чупіки Артема Миколайовича

1. Тема роботи: Інформаційна система моніторингу здоров'я для підтримки прийняття рішень та забезпечення персоналізованого підходу
затверджую наказом по СумДУ від *«03» грудня 2024 року № 1257-VI*
2. Термін здачі здобувачем кваліфікаційної роботи *до 04 грудня 2024 року*
3. Вхідні дані до кваліфікаційної роботи _____
4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)
1) Аналітичний огляд. 2) Постановка задачі 3) Вибір методів розв'язання поставленої задачі. 4) Програмна реалізація. 5) Висновки
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)_____
6. Консультанти до проекту (роботи), із значенням розділів проєкту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання «18» серпня 2024 р.

Завдання прийняв до виконання _____ Керівник _____
(підпис) (підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання	Примітка
1	*Аналітичний огляд*		
2	*Постановка задачі*		
3	*Вибір методів розв'язання поставленої задачі*		
4	*Програмна реалізація*		
5	*Висновки*		

Здобувач вищої освіти _____ Керівник _____
(підпис) (підпис)

АНОТАЦІЯ

Записка: 71 стор., 24 рис., 8 табл., 22 використаних джерел, 2 додатки.

Обґрунтування актуальності теми роботи – тема кваліфікаційної роботи є актуальною, оскільки присвячена розробці технології, яка вирішує проблему цифровізації медичних документів та взаємодії пацієнтів з лікарями.

Об’єкт дослідження – процес розробки інформаційної технології управління, зберігання та обміну медичними даними між пацієнтами та медичними працівниками за допомогою інформаційних технологій.

Мета роботи — проєктування та розробка інформаційної технології моніторингу здоров'я для підтримки прийняття рішень та забезпечення персоналізованого підходу.

Методи дослідження — аналіз та оцінка ефективності інформаційної технології для моніторингу здоров'я для підтримки прийняття рішень та забезпечення персоналізованого підходу.

Результати — розроблено інформаційну технологію, що забезпечує допомогу моніторинг здоров'я для підтримки прийняття рішень та забезпечення персоналізованого підходу. Застосунок відповідає всім вимогам щодо безпеки та функціональності, забезпечує зручну роботу з медичними даними, дозволяє взаємодіяти з лікарями та ділитися медичними записами.

ІНФОРМАЦІЙНА ТЕХНОЛОГІЯ, NODEJS, EXPRESS, GRAPHQL, JAVASCRIPT, TYPESCRIPT, REACT NATIVE.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми роботи. Сучасна медицина стикається з багатьма проблемами, пов'язаними з управлінням і зберіганням даних про пацієнтів. Традиційні паперові медичні карти часто є причиною проблем через їх уразливості до втрат, пошкоджень і незручностей при обміні інформацією між медичними працівниками. Зі швидким розвитком інформаційних технологій виникла необхідність перейти на цифрові рішення, які забезпечують легкий доступ, безпечне зберігання і швидке поширення медичних даних [3]. Тому тема розробки інформаційних технологій підтримки спостереження за станом здоров'я важлива і заслуговує глибокого вивчення.

Актуальність. У умовах глобальних проблем, таких як пандемії та зростання хронічних захворювань, виникає потреба в покращенні способу зберігання та передачі даних про стан пацієнта. Використання мобільних застосунків дозволяє оптимізувати процес взаємодії пацієнтів і лікарів, що позитивно позначається на якості медичних послуг. Розробка інформаційних технологій, які дозволять пацієнтам зберігати, упорядковувати та ділитися медичними записами з лікарями, допоможе покращити процес діагностики та лікування, спростити обробку даних та зменшити ризик помилок [1].

Предмет дослідження. Інформаційна технологія проєктування мобільного застосунку моніторингу медичної інформації пацієнтів, який надає можливості обміну даними між пацієнтами та лікарями.

Новизна. Пропонована технологія відрізняється від існуючих аналогів здатністю інтегрувати функції зберігання даних з можливістю їх передачі медичним працівникам по захищених каналах зв'язку. Технологія заснована на новітніх мобільних технологіях і відповідає вимогам безпеки даних, що робить її унікальною на ринку мобільних рішень для охорони здоров'я [2].

Дослідження складається зі вступу, аналітичного огляду, постановки задачі, вибору методу розв’язання поставленої задачі, опису програмного забезпечення інформаційної технології, висновків, списку використаних джерел та додатків.

1. АНАЛІТИЧНИЙ ОГЛЯД

1.1 Сучасний стан

У сучасному світі інформатизація охоплює все більше і більше аспектів повсякденного життя, включаючи охорону здоров'я. Використання цифрових технологій в охороні здоров'я стало необхідністю для ефективного зберігання та передачі даних про пацієнтів, підвищення якості лікування, моніторингу стану здоров'я та забезпечення взаємодії між пацієнтами та медичними працівниками. Згідно з дослідженням [11], за останні п'ять років впровадження мобільних медичних додатків подвоїлося, і очікується, що до 2027 року ринок цифрових рішень для охорони здоров'я зростатиме на 21% на рік (рис.1.1).

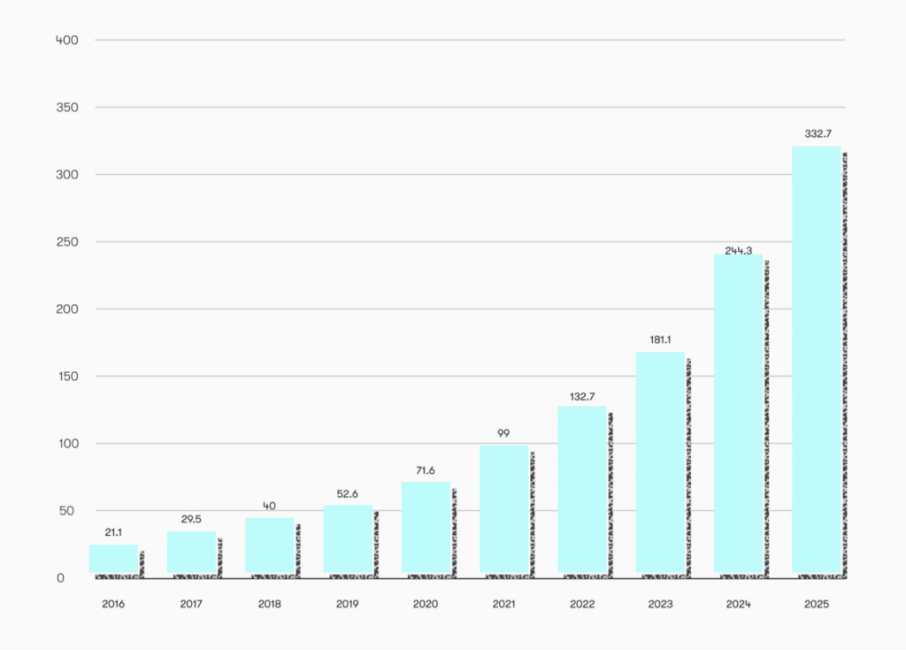

Рисунок 1.1 – Зростання кількості користувачів медичних додатків

Технології мобільного моніторингу здоров'я дозволяють користувачам відстежувати показники свого здоров'я, отримувати поради щодо здорового способу життя та залишатися на зв'язку зі своїми лікарями. Зокрема, за даними Всесвітньої організації охорони здоров'я, понад 58% медичних закладів у розвинених країнах використовують мобільні додатки для віддаленого

моніторингу пацієнтів [22]. Ці додатки пропонують як базовий функціонал, так і розширені можливості з використанням штучного інтелекту для аналізу даних.

Основні тенденції сучасного розвитку включають:

- Підвищення уваги до персоналізації послуг;
- Використання алгоритмів ШІ для аналізу медичних даних;
- Інтеграція з різними сенсорами для збору показників здоров'я, таких як серцебиття, рівень цукру в крові тощо [4].

1.2 Аналіз аналогічних проєктів

Аналіз існуючих технологій є важливим етапом для розуміння потреб ринку, визначення сильних та слабких сторін існуючих рішень. Розглянемо кілька популярних медичних платформ:

1. Apple Health. Цей додаток об'єднує дані з різних фітнес-додатків і медичних пристроїв. Сильна сторона – інтеграція з екосистемою Apple та можливість відстеження великого спектра показників здоров'я. Недоліком є обмежена доступність функціоналу для користувачів без iOS-пристроїв [10] (табл.1.1).

Таблиця 1.1 Оцінка функціональності додатку Apple Health

Функціональність	**Опис**	**Оцінка користувачів**
Збір медичних даних	Автоматичний імпорт з підключених пристроїв	4.5/5
Аналіз здоров'я	Пропозиції для поліпшення способу життя	4.0/5

2. Google Fit. Відомий додаток від Google, що зосереджений на фітнес-показниках, таких як фізична активність та серцебиття. Його ключова

перевага – синхронізація з великою кількістю пристроїв Android. Однак, Google Fit має обмежену підтримку специфічних медичних даних [8] (рис.1.2).

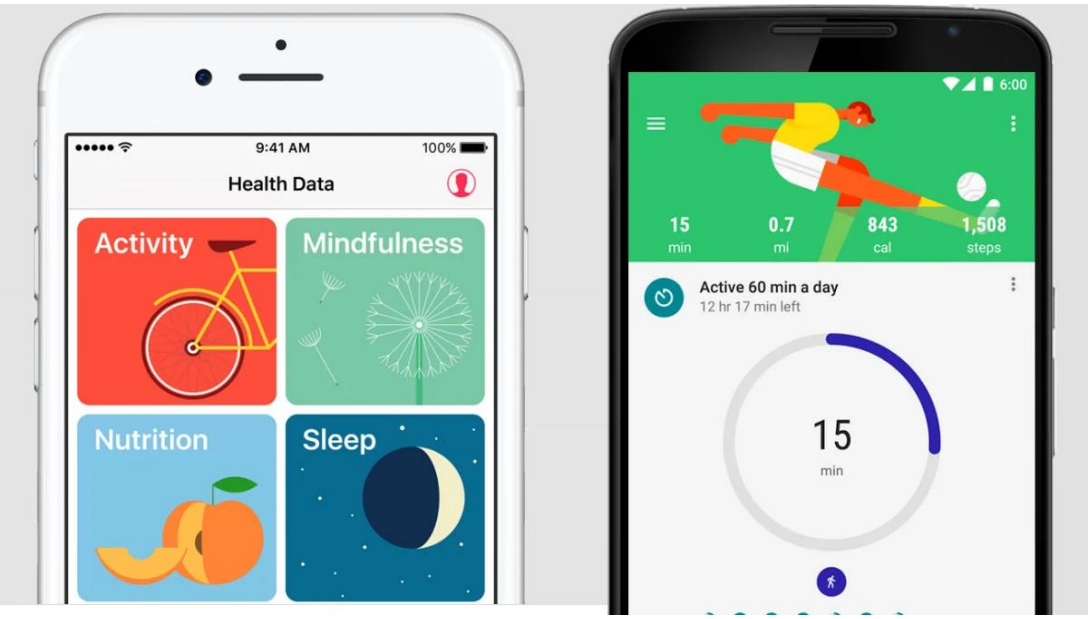

Рисунок 1.2 Інтерфейси Google Fit та Apple Health

3. MyChart. Ця платформа дозволяє пацієнтам безпечно переглядати свої медичні записи, зберігати історію хвороб та зв'язуватися зі своїм лікарем через мобільний застосунок. Відмінна риса – висока безпека та відповідність медичним стандартам, таким як HIPAA. Проте, технологія часто вимагає інтеграції з медичними установами та має складний інтерфейс [17].

Порівняння переваг та недоліків цих трьох додатків можна побачити нижче (табл.1.2).

Таблиця 1.2 Оцінка функціональності додатку Apple Health

Платформа	**Переваги**	**Недоліки**
MyChart	Безпека, доступ до історії хвороб	Складний інтерфейс
Apple Health	Інтеграція, багатий набір даних	Платформа-залежність
Google Fit	Сумісність з Android	Обмеженість медичних даних

Аналіз аналогічних проєктів дозволяє зробити висновок про необхідність створення технології, яка б об'єднувала переваги перерахованих рішень і була максимально доступною, простою у використанні та здатною забезпечити високий рівень безпеки.

1.3 Постановка задачі

Метою даної роботи є розробка інформаційної технології, яка дозволить користувачам зберігати медичні записи, ділитися ними з медичними працівниками та отримувати інформацію про стан свого здоров'я. Основними задачами є:

- Створення архітектури мобільного додатку для зручного та безпечного зберігання медичних карток.
- Розробка функціоналу для обміну медичними записами з лікарем через захищені канали.
- Забезпечення інтеграції з різними сенсорами для збору показників здоров'я.
- Створення зручного інтерфейсу з можливістю пошуку та перегляду записів пацієнта.

Також важливо розуміти основні функції технології (табл.1.3).

Таблиця 1.3 Основні функції технології

Функція	**Опис**
Збереження медичних записів	Зберігання та організація записів пацієнта
Пошук та перегляд	Зручний пошук по медичних категоріях
Обмін інформацією з лікарем	Надсилання медичних даних лікарю

2. ВИБІР МЕТОДІВ РОЗВ'ЯЗАННЯ ПОСТАВЛЕНОЇ ЗАДАЧІ

2.1 Загальні принципи вибору технологій

Вибір відповідних технологій для розробки інформаційної технології має велике значення для досягнення основної мети проєкту. Для створення мобільного додатку, який дозволяє зберігати медичні записи та забезпечує обмін інформацією між пацієнтами та лікарями, необхідно обрати технології, що забезпечують швидкодію, масштабованість та зручність інтеграції.

Сучасні інформаційні технології повинні підтримувати високий рівень інтерактивності та відповідати вимогам мобільності. Зокрема, обрані технології повинні підтримувати ефективну роботу з великими обсягами даних і забезпечувати високу надійність. Інтеграція різних технологій та використання інноваційних підходів дозволяє досягти найкращих результатів при створенні медичних додатків.

2.2 Вибір мов програмування

Одним із ключових аспектів є вибір мови програмування, яка б забезпечувала ефективний розвиток проєкту, легкість у використанні та можливість розширення функціональності. JavaScript зарекомендував себе як одна з найбільш популярних і універсальних мов, що дозволяє створювати як фронтенд, так і бекенд частини додатків. Завдяки своїй підтримці сучасних стандартів мова дозволяє створювати динамічні та інтерактивні користувацькі інтерфейси.

Розширення можливостей JavaScript досягається використанням TypeScript, який є надбудовою над JavaScript і додає статичну типізацію. Це дозволяє знизити кількість помилок на етапі розробки та забезпечити кращу структуру коду. У великих проєктах з великою командою розробників це особливо важливо, оскільки забезпечує зрозумілість і підтримуваність коду.

TypeScript також дозволяє розробникам використовувати сучасні можливості ECMAScript з повною підтримкою у середовищах розробки.

JavaScript і TypeScript обрані як основні мови програмування. JavaScript забезпечує широке використання в розробці веб- та мобільних додатків завдяки своїй універсальності та активній підтримці. TypeScript, як надбудова над JavaScript, дозволяє використовувати типізацію, що забезпечує більшу надійність та зрозумілість коду [13].

Переваги TypeScript:

- Покращене виявлення помилок на етапі компіляції.
- Зрозумілість структури коду для великих команд розробників.
- Підтримка нових стандартів ECMAScript.

Для більшого розуміння можна зробити порівняння JavaScript і TypeScript (табл.2.1).

Таблиця 2.1 Порівняння JavaScript і TypeScript

Параметр	**JavaScript**	**TypeScript**
Типізація	Динамічна	Статична
Зрозумілість коду	Відносно проста	Висока
Підтримка IDE	Стандартна	Покращена

2.3 Вибір технологій для серверної частини

Node.js — це платформа з відкритим вихідним кодом, яка дозволяє запускати JavaScript на серверній стороні, надаючи можливість створювати високопродуктивні та масштабовані додатки. Завдяки подієво-орієнтованій архітектурі та неблокуючій моделі введення-виведення, Node.js здатен ефективно обробляти велику кількість одночасних підключень з мінімальними затримками. Це робить платформу оптимальним вибором для застосунків, де швидкість обробки даних і масштабованість є ключовими вимогами, наприклад, у медичних системах моніторингу здоров'я.

Асинхронна модель роботи Node.js дозволяє виконувати введення та виведення даних без блокування основного потоку виконання. Завдяки цьому сервер може обробляти інші запити, не чекаючи завершення поточних операцій, що значно підвищує ефективність. Такий підхід ідеально підходить для систем, що вимагають швидкої реакції та високої пропускної здатності, особливо у сферах, пов’язаних із динамічною обробкою інформації.

Express.js — це легкий і гнучкий фреймворк для Node.js, який спрощує створення серверної частини. Express.js дозволяє швидко налаштовувати маршрутизацію, обробляти HTTP-запити та легко інтегрувати middleware для покращення функціональності. Його архітектура забезпечує високу гнучкість, що дозволяє розробникам адаптувати серверну частину додатка під різні потреби [16].

Чому Node.js та Express.js:

- Швидка обробка великої кількості запитів.
- Масштабованість і підтримка модулів.
- Простота розробки за рахунок доступності великої кількості бібліотек.

2.4 Вибір архітектури API

Однією з основних складових будь-якої сучасної інформаційної технології є інтерфейс прикладного програмування (API). Обраний підхід до створення API значно впливає на продуктивність, зручність та можливості розширення технології.

GraphQL був обраний як архітектурне рішення для API. Це сучасна альтернатива REST, яка дозволяє клієнтам виконувати запити з більшою гнучкістю та отримувати тільки необхідні дані. GraphQL зменшує проблему надмірного або недостатнього завантаження даних (over-fetching і under-fetching), що є типовими недоліками традиційних REST-запитів. Це забезпечує

ефективніше використання ресурсів серверу та пришвидшує роботу клієнтської частини [9].

GraphQL підтримує модульність і розширюваність завдяки своїй декларативній природі. Розробники можуть легко додавати нові типи даних і розширювати запити, що є важливою перевагою для технологій, які повинні швидко адаптуватися до нових вимог.

Основні переваги GraphQL:

- Одна точка доступу до API.
- Гнучка структура запитів.
- Зменшення over-fetching і under-fetching даних.

Нижче представлена схема роботи GraphQL (рис.2.1).

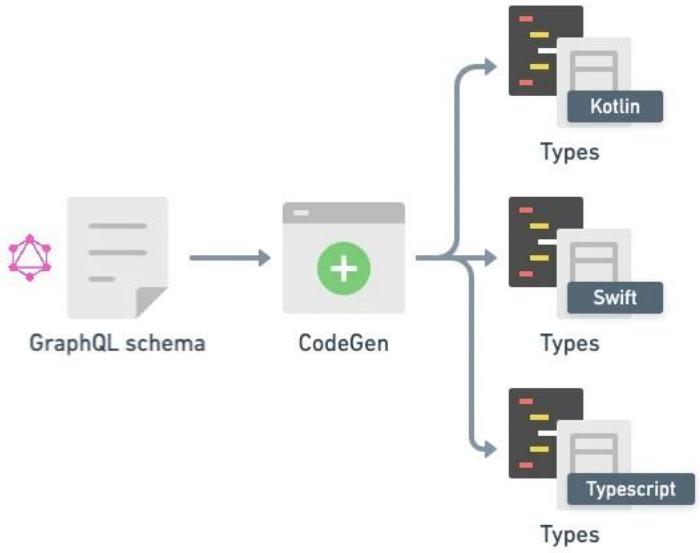

Рисунок 2.1 Схема роботи GraphQL запиту

2.5 Вибір бази даних

Для зберігання медичних даних пацієнтів було обрано MongoDB, яка є нереляційною базою даних. Її документоорієнтована модель добре підходить для зберігання структурованих даних з різними наборами полів, що дозволяє легко зберігати інформацію про пацієнтів та їх медичні записи [15].

MongoDB є однією з найпопулярніших NoSQL баз даних, що широко використовується для побудови сучасних веб- та мобільних додатків. Її документоорієнтована структура базується на зберіганні даних у форматі BSON (бінарна форма JSON), що дозволяє зберігати складні та вкладені об'єкти у вигляді документів. Для інформаційної технології, яка має справу з медичними записами, MongoDB надає значні переваги завдяки своїй гнучкості, масштабованості та продуктивності.

Гнучкість MongoDB особливо важлива в системах, де структура даних може змінюватися з часом. Наприклад, медичні записи різних пацієнтів можуть містити різні поля та вкладені структури залежно від специфіки захворювань або медичних процедур. Завдяки можливості зберігати документи без жорсткої схеми, MongoDB дозволяє адаптувати базу даних до зміни вимог, не потребуючи значної реконструкції. Це робить процес розробки більш динамічним та зручним для інновацій.

Підтримка горизонтального масштабування у MongoDB реалізується за допомогою шардінгу, що дозволяє розподіляти дані по різних серверах. Це особливо корисно для високонавантажених систем, де обсяг інформації може значно зрости. Шардінг дозволяє забезпечити високу доступність і безперебійну роботу системи навіть при великій кількості одночасних запитів. У випадку інформаційної технології для моніторингу здоров'я, де кількість користувачів може бути значною, ця функція забезпечує стабільну роботу та надійність сервісу. MongoDB також має вбудовані функції реплікації, що забезпечують високу доступність даних і захист від втрати інформації у разі апаратних або програмних збоїв. Реплікація дозволяє зберігати копії бази даних на різних серверах, що покращує відмовостійкість системи. Для медичних додатків, де захист даних є критично важливим, це суттєва перевага.

Ще однією важливою перевагою є індексація даних. MongoDB дозволяє створювати індекси на різні поля, що прискорює виконання запитів і знижує навантаження на сервери. Це важливо для систем, які обробляють запити на пошук медичних записів, аналіз історії хвороб або вибірки даних за

критеріями. Вбудована підтримка агрегаційних фреймворків дозволяє створювати складні аналітичні запити для обробки даних на сервері безпосередньо. Це допомагає зменшити обсяг даних, що передаються на клієнт, і підвищити продуктивність додатка. Наприклад, у медичному додатку можна швидко отримати статистичні звіти про частоту різних захворювань або проаналізувати динаміку показників пацієнтів.

Переваги MongoDB:

- Висока масштабованість завдяки горизонтальному шардінгу.
- Гнучка структура документів.
- Підтримка великих обсягів даних.

Для наглядності було порівняно MongoDB та SQL бази даних (табл.2.2).

Таблиця 2.2 Порівняння MongoDB та SQL баз даних

Параметр	**MongoDB**	**SQL**
Структура даних	Документоорієнтована	Таблична
Масштабованість	Висока	Помірна
Гнучкість	Висока	Обмежена

2.6 Технології для фронтенду

React Native обраний для створення мобільного додатку, оскільки дозволяє розробляти кросплатформні додатки з одним базовим кодом, що значно скорочує час і витрати на розробку [19].

React Native є одним із найбільш популярних фреймворків для розробки кросплатформених мобільних додатків. Він дозволяє використовувати одну базу коду для створення додатків під iOS та Android, що значно скорочує час розробки і спрощує підтримку проєкту. Основною перевагою React Native є можливість писати додатки мовою JavaScript або TypeScript, що дозволяє

легко інтегрувати мобільний додаток з іншими веб-додатками та серверними службами, які використовують ті ж технології.

React Native базується на React.js, що забезпечує декларативний підхід до створення користувацьких інтерфейсів. Це дозволяє будувати складні компоненти та ефективно управляти станом додатку, зменшуючи кількість помилок і полегшуючи тестування. Вибір React Native для інформаційної технології моніторингу здоров'я дозволяє створювати інтуїтивно зрозумілі інтерфейси, що спрощують взаємодію користувачів з додатком.

Використання React Native забезпечує доступ до рідних компонентів пристрою, таких як камера, геолокація, та доступ до файлової системи, що важливо для додатків, які працюють із зображеннями або сканами медичних документів. Це робить додаток більш функціональним і дозволяє легко інтегрувати необхідні модулі для забезпечення нових можливостей.

Завдяки підтримці бібліотек і модулів, таких як React Navigation та Redux, розробники можуть легко реалізувати навігацію по додатку та керування станом, що підвищує зручність використання та ефективність роботи з додатком.

Однією з ключових переваг є можливість використовувати Expo, платформу для розробки та тестування додатків на базі React Native. Expo надає широкий спектр вбудованих API, що дозволяє швидко впроваджувати нові функції, не витрачаючи час на налаштування нативного коду.

Важливо зазначити, що React Native має активну спільноту розробників і потужну підтримку з боку Facebook, що гарантує регулярні оновлення та стабільність платформи. Велика кількість готових компонентів і модулів значно знижує час розробки та дозволяє сфокусуватися на бізнес-логіці та специфічних вимогах додатка.

React Native також дозволяє використовувати TypeScript, що додає статичну типізацію і знижує кількість потенційних помилок. Це особливо важливо для великих проєктів, які потребують високої підтримуваності та чіткості в структурі коду.

Основні переваги React Native:

- Можливість використання одного коду для Android і iOS.
- Велика спільнота розробників і доступність готових компонентів.
- Висока продуктивність у порівнянні з гібридними фреймворками.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Реалізація бази даних

3.1.1 Архітектура бази даних

Структура бази даних була розроблена таким чином, щоб відповідати вимогам системи щодо зберігання даних користувачів, лікарів, медичних записів та файлів. Основними колекціями є:

- Users – зберігає інформацію про пацієнтів та лікарів і їх дані.
- VerificationCodes – для зберігання даних про завантажені файли.

Для віддаленого створення бази даних був використаний MongoDB Compass. Окрім фунціональності утиліта має зручний інтерфейс (рис.3.1).

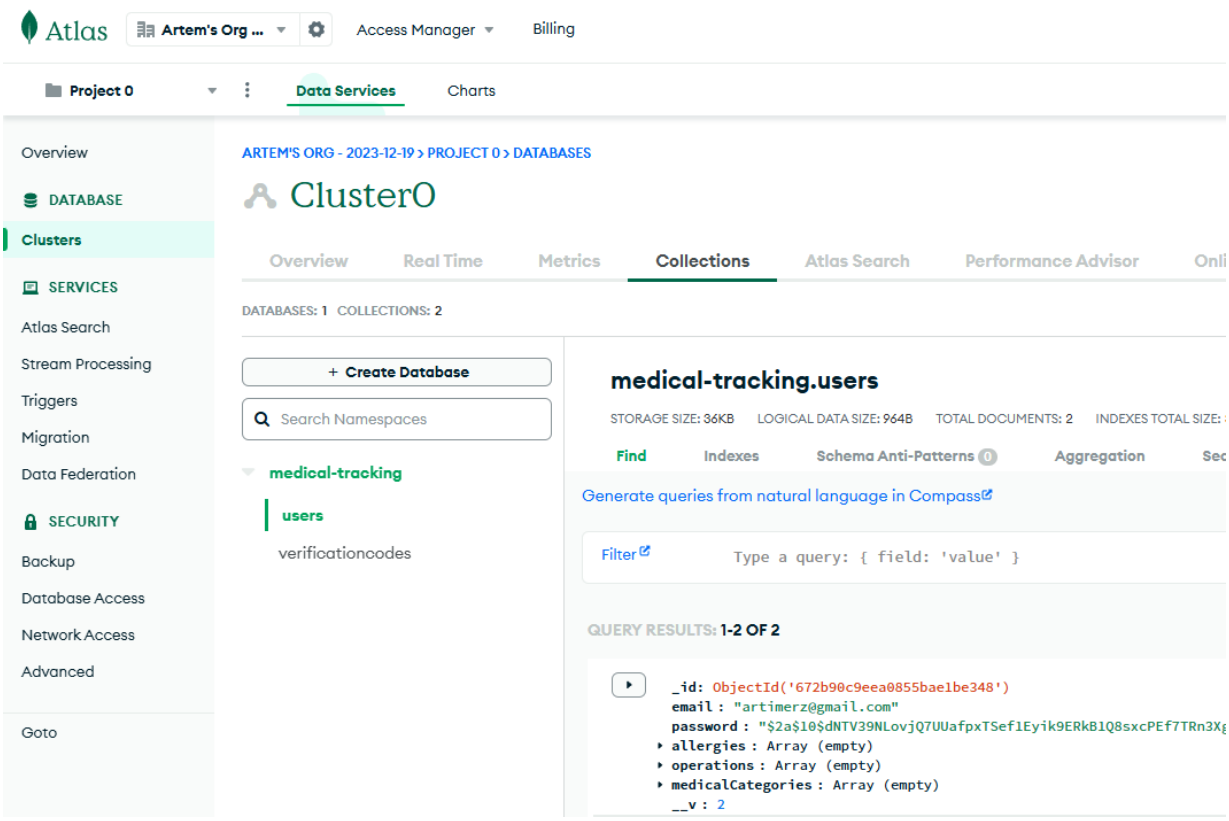

Рисунок 3.1 Інтерфейс MongoDB Compass

3.1.2 Моделювання даних

Для опису структури даних були створені моделі за допомогою Mongoose – бібліотеки для Node.js, що забезпечує легку роботу з MongoDB. Моделі визначають структуру документів у колекціях та валідатори для них.

Модель користувача

Модель користувача представлена у файлі *src/models/User.ts* і містить важливі дані для забезпечення функціональності додатку, який призначений для моніторингу здоров'я та взаємодії пацієнтів з лікарями. Нижче наведено детальний опис кожного поля цієї моделі у вигляді таблиці (табл.3.1):

Таблиця 3.1 Опис моделі користувача

Поле	**Тип**	**Обов'язковість**	**Опис**
email	string	Так	Електронна адреса користувача. Використовується для ідентифікації користувача в системі.
Password	string	Так	Хешований пароль користувача для забезпечення безпеки доступу.
Role	string	Ні	Роль користувача (може бути або 'User', або 'Doctor'). Допомагає у визначенні прав доступу.
firstName	string	Ні	Ім'я користувача. Може використовуватись для персоналізації інтерфейсу та записів.

Продовження табл. 3.1

lastName	string	Ні	Прізвище користувача.
middleName	string	Ні	По батькові користувача.
bloodGroup	string	Ні	Група крові користувача для медичних записів.
birthDate	Date	Ні	Дата народження користувача, що допомагає визначити вік і можливі ризики для здоров’я.
phone	string	Ні	Номер телефону для зв’язку.
Gender	string	Ні	Стать користувача, що може бути використана в статистичних або медичних цілях.
Allergies	array of strings	Ні	Список алергій користувача.
sharedWith	array of strings	Ні	Масив ID лікарів, з якими поділено медичну картку користувача.
Operations	array	Ні	Список операцій, що користувач мав. Кожен елемент містить дату, опис та фото.
medicalCategories	array	Ні	Містить категорії медичних записів, діагнози та відвідування, що стосуються користувача.
Certificates	array of strings	Ні	Список сертифікатів лікаря (посилання на фото/документи).
Experience	array	Ні	Інформація про досвід роботи (опис, дата початку та кінця).
position	string	Ні	Посада лікаря, що дозволяє зрозуміти його професійні обов'язки.

Пояснення полів зі складними структурами:

1. operations – це поле містить масив об'єктів, де кожен об'єкт описує одну операцію. Кожна операція має дату проведення (date), опис процедури (description), та опціонально масив посилань на фотографії (photos).

2. medicalCategories – це масив, що містить інформацію про медичні категорії. Кожен об'єкт включає:

- category: назва категорії (наприклад, "Кардіологія").
- diagnoses: масив діагнозів, пов'язаних з цією категорією.
- visits: масив відвідувань, що містять дату відвідування (date), діагноз (diagnosis), опис (description), та файли (files).

3. experience – це масив, де кожен об'єкт містить:

- description: опис досвіду роботи.
- startdate: дата початку досвіду.
- enddate: дата завершення досвіду (якщо є).

3.2 Реалізація серверної частини

3.2.1 Структура проекту

Проєкт створений з використанням Node.js та TypeScript, що забезпечує стабільність, надійність і можливість використання типів під час розробки. Проєкт організовано за модульним підходом, що дозволяє легше масштабувати і підтримувати його. Структура проєкту обрана для виконання поставлених задач (рис.3.2).

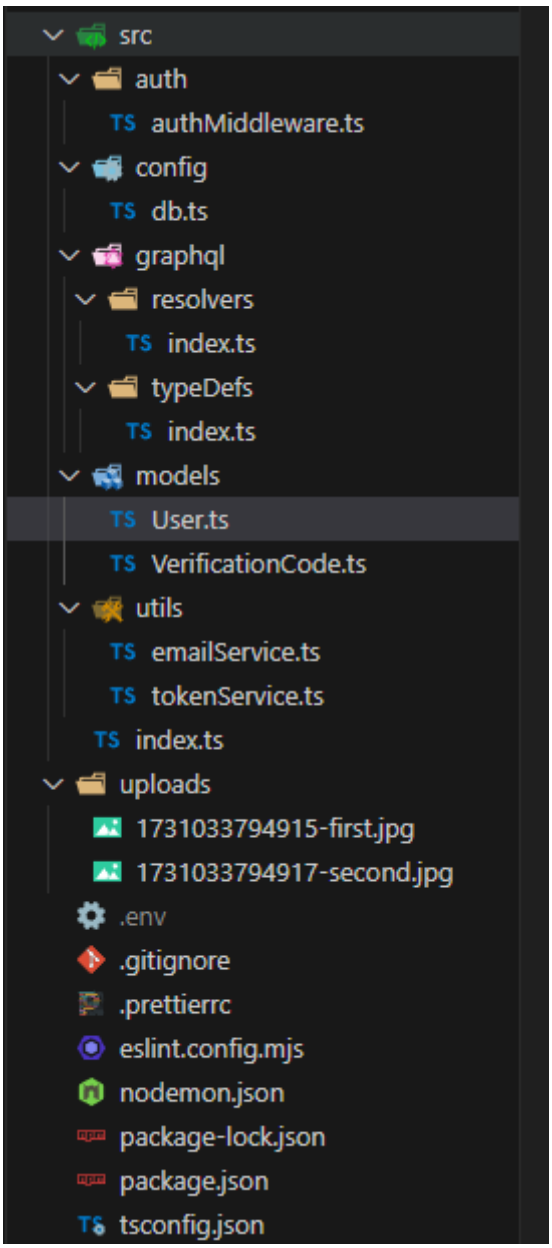

Рисунок 3.2 Структура проєкту

3.2.2 Опис файлів, папок та методів

Детальний опис папок і файлів:

1. /config

Містить конфігураційні файли для підключення до бази даних. Використовується mongoose для роботи з MongoDB. Наприклад, файл db.ts містить функцію для ініціалізації з'єднання з базою даних (рис.3.3).

```
import mongoose from 'mongoose'  871.5k (gzipped: 233.3k)

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGODB_URI!)
    console.log('MongoDB connected')
  } catch (error) {
    console.error('Error connecting to MongoDB', error)
    process.exit(1)
  }
}

export default connectDB
```

Рисунок 3.3 Підключення до бази даних

2. /auth, містить файли для авторизації і перевірки запитів. Наприклад, middleware для аутентифікації запиту перевіряє наявність токена доступу (рис.3.4).

```
1  import jwt from 'jsonwebtoken'  53.9k (gzipped: 16.1k)
2
3  // eslint-disable-next-line @typescript-eslint/no-explicit-any
4  export const authMiddleware = (req: any) => {
5    const authHeader = req.headers.authorization
6
7    if (!authHeader || !authHeader.startsWith('Bearer ')) {
8      throw new Error('No token provided')
9    }
10
11   const token = authHeader.split(' ')[1]
12
13   try {
14     const decoded = jwt.verify(token, process.env.JWT_SECRET!)
15     return decoded
16   } catch {
17     throw new Error('Invalid token')
18   }
19 }
```

Рисунок 3.4 Верифікація токенів

3. /models – містить файли моделей, що описують структуру документів у базі даних. Модель User вже розглянута вище. Використання mongoose дозволяє легко створювати і валідувати структуру даних.

4. /uploads – папка для зберігання файлів та картинок які були відправлені на сервер.

5. /graphql/resolvers – це папка з логікою обробки запитів. Кожен резолвер відповідає за обробку запитів і мутацій, визначених у схемах (табл.3.2).

Таблиця 3.2 Опис методів серверної частини

Метод	**Тип**	**Опис**
getUser	Query	Отримує інформацію про користувача за його ID.
getUsers	Query	Отримує список користувачів з можливістю фільтрації за роллю, позицією та пошуком.
getSharedCards	Query	Отримує список пацієнтів, що поділилися своїми картами з лікарем.
sendCode	Mutation	Відправляє код верифікації на email користувача.
verifyCodeAndRegister	Mutation	Верифікує код і реєструє нового користувача.
login	Mutation	Аутентифікує користувача та генерує токени доступу.
refreshToken	Mutation	Оновлює access token на основі переданого refresh token.
changePassword	Mutation	Змінює пароль користувача після перевірки поточного пароля.
changeEmail	Mutation	Змінює email користувача після верифікації коду.
updateUser	Mutation	Оновлює інформацію про користувача за ID.
shareCard	Mutation	Дає можливість пацієнту поділитися своєю картою з лікарем.
uploadFiles	Mutation	Завантажує файли на сервер і повертає шляхи до них.

6. /graphql/typeDefs, папка для файлів зі схемами GraphQL. Схема визначає типи запитів, мутацій і об'єкти, доступні через GraphQL API (табл.3.3).

Таблиця 3.3 Опис схем GraphQL

Схема	**Тип**	**Опис**
User	Type	Описує користувача та його атрибути, включаючи особисті дані та токени.
Operation	Type	Містить дані про медичні операції користувача.
Visit	Type	Містить дані про візити користувача до лікаря.
MedicalCategory	Type	Описує категорії медичних даних користувача.
Experience	Type	Описує досвід роботи, актуальний для лікарів.
UpdateUserInput	Input	Використовується для оновлення інформації про користувача.
OperationInput	Input	Вхідні дані для операцій при оновленні користувача.
VisitInput	Input	Вхідні дані для візитів при оновленні користувача.
MedicalCategoryInput	Input	Вхідні дані для медичних категорій при оновленні користувача.
ExperienceInput	Input	Вхідні дані для досвіду роботи при оновленні інформації про користувача.
Upload	Scalar	Скалярний тип для завантаження файлів.
Mutation	Type	Визначає всі мутації, які можуть виконуватися (наприклад, sendCode, updateUser).
Query	Type	Визначає всі запити, які можна виконати (наприклад, getUser, getUsers).

3.3 Інструмент для тестування серверної частини

Для забезпечення належного функціонування серверної частини застосунку необхідно проводити ґрунтовне тестування. Це включає перевірку правильності роботи API-ендпоінтів, перевірку цілісності даних, валідацію заходів безпеки та забезпечення того, щоб система відповідала очікуванням у різних умовах. Одним із найбільш ефективних інструментів для цього є Postman – популярне середовище для розробки та тестування API, що спрощує процес надсилання запитів до сервера та аналізу отриманих відповідей.

Postman – це платформа для розробки та тестування API, яка надає зручний інтерфейс для створення, надсилання та аналізу HTTP-запитів. Вона підтримує різні HTTP-методи, такі як GET, POST, PUT та DELETE, що є основою для CRUD-операцій на сервері. Postman також дозволяє використовувати параметри, заголовки та дані тіла, що робить його ідеальним для тестування складних API [18]. Структура запитів у Postman відповідає методам серверної частини (рис. 3.5).

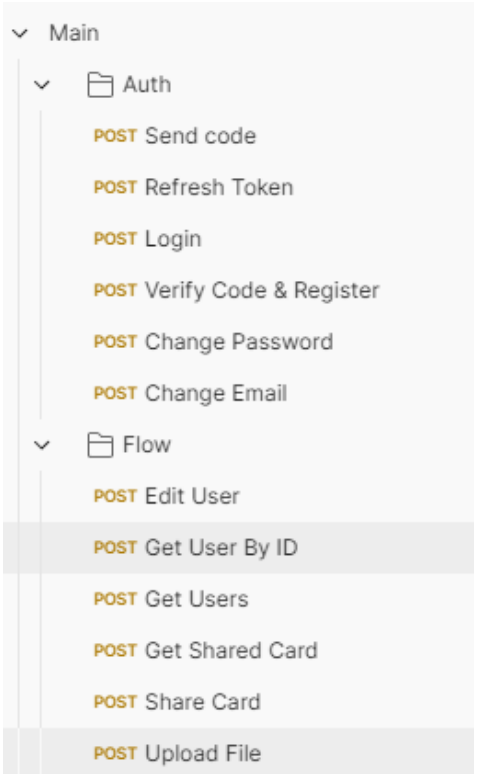

Рисунок 3.5 Запити тестування у Postman

Детальний опис кожного запиту та їхнє застосування в процесі тестування:

1. Надсилання коду на електронну пошту

Цей запит використовується для тестування функції надсилання коду верифікації на вказану електронну пошту. Ця функція є важливою для процесу реєстрації користувачів або зміни електронної адреси. У Postman можна перевірити відповідь сервера, яка повинна підтвердити успішне відправлення коду (рис. 3.6).

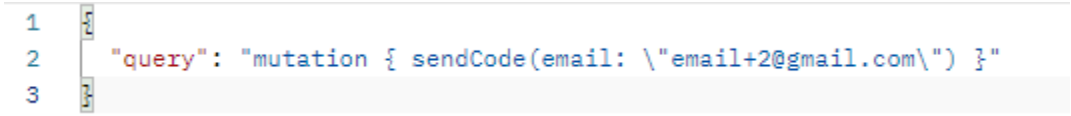

Рисунок 3.6 Запит відправки коду

2. Оновлення токена доступу

Цей запит перевіряє функцію оновлення токену доступу. Користувач подає запит на оновлення, передаючи наявний токен. У відповідь сервер має повернути новий токен доступу для забезпечення безперебійної авторизації в системі (рис. 3.7).

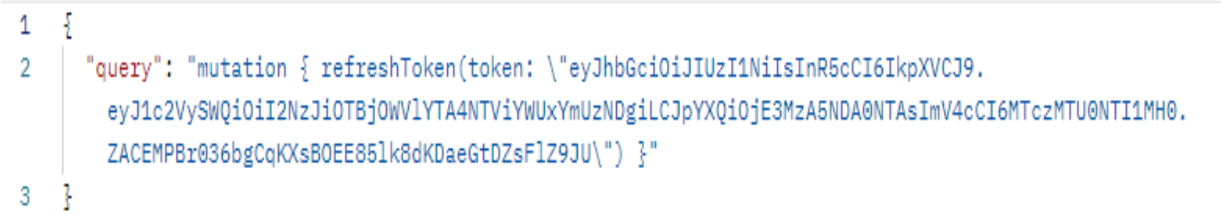

Рисунок 3.7 Запит оновлення токену

3. Авторизація користувача

Цей запит тестує функціонал входу користувача. Після успішної авторизації сервер повинен повернути об'єкт користувача, що містить його ID, електронну адресу, а також токени доступу та оновлення. Postman дозволяє

переконатися в правильності відповіді та тестувати поведінку сервера при різних комбінаціях електронної адреси та пароля (рис. 3.8).

```
{
  "query": "mutation { login(email: \"email@gmail.com\", password: \"123456\") { _id email accessToken
    refreshToken } }"
}
```

Рисунок 3.8 Запит авторизації користувача

4. Підтвердження коду та реєстрація

Даний запит перевіряє процес реєстрації, коли користувач вводить отриманий код верифікації. У відповідь сервер повертає ID та електронну адресу нового користувача. Це тестування важливе для забезпечення коректної роботи системи реєстрації з підтвердженням коду (рис. 3.9).

```
{
  "query": "mutation { verifyCodeAndRegister(email: \"email@gmail.com\", code: \"113778\", password:
    \"123456\") { _id email } }"
}
```

Рисунок 3.9 Запит підтвердження коду та реєстрації

5. Зміна пароля

Цей запит перевіряє функцію зміни пароля користувача. Користувач вводить свою поточну електронну адресу, старий та новий паролі. У Postman можна перевірити, чи правильно сервер обробляє запит і чи оновлює пароль користувача (рис. 3.10).

```
{
  "query": "mutation { changePassword(email: \"email@gmail.com\", currentPassword: \"1234567\",
    newPassword: \"123456\") }"
}
```

Рисунок 3.10 Запит зміни паролю

6. Зміна електронної адреси

Цей запит використовується для тестування функції зміни електронної адреси користувача з підтвердженням коду. Сервер повинен успішно оновити адресу, якщо код верифікації є правильним. Postman дозволяє перевірити відповіді на цей запит, щоб підтвердити, що система працює належним чином (рис. 3.11).

```
{
  "query": "mutation { changeEmail(currentEmail: \"email@gmail.com\", newEmail: \"artimerz+2@gmail.com\",
    code: \"292744\") }"
}
```

Рисунок 3.11 Запит зміни емейлу

7. Оновлення інформації користувача

Цей запит використовується для перевірки функціоналу оновлення даних користувача. Зміни можуть включати ім'я, прізвище, роль, позицію, сертифікати та досвід роботи. Використовуючи Postman, можна переконатися, що сервер коректно обробляє вхідні дані та зберігає зміни (рис. 3.12).

```
{
  "query": "mutation updateUser($_id: ID!, $input: UpdateUserInput!) { updateUser(_id: $_id, input:
    $input) { _id, firstName, lastName, role, position, certificates, experience { description, startDate,
    endDate } } }",
  "variables": {
    "_id": "672b90c9eea0855bae1be348",
    "input": {
      "firstName": "Test",
      "lastName": "Family",
      "role": "User",
      "position": "Dantist",
      "certificates": ["link1", "link2"],
      "experience": [
        {
          "description": "Work at Clinic A",
          "startDate": "2021-01-01",
          "endDate": "2023-01-01"
        }
      ]
    }
  }
}
```

Рисунок 3.12 Запит оновлення даних користувача

8. Отримання інформації про користувача

Цей запит перевіряє функцію отримання даних про користувача за його ID. Використання Postman дозволяє перевірити правильність інформації, що повертається сервером (рис. 3.13).

```
query {
  getUser(_id: "672c0e8bdd7e620c5427c24d") {
    _id
    email
    firstName
    lastName
    role
  }
}
```

Рисунок 3.13 Запит на отримання даних про користувача

9. Отримання списку користувачів

Цей запит тестує отримання списку користувачів з використанням параметрів пагінації. У Postman можна перевірити, чи коректно сервер повертає користувачів відповідно до вказаних параметрів сторінки та обмеження (рис. 3.14).

```
query {
  getUsers(page: 2, limit: 1) {
    _id
    email
    firstName
    lastName
    role
    position
  }
}
```

Рисунок 3.14 Запит на отримання даних користувачів

10. Отримання списку карток пацієнтів

Цей запит дозволяє тестувати функціонал отримання списку пацієнтських карток, якими поділилися з конкретним лікарем. Це корисно для перевірки зв'язків між користувачами та лікарями (рис. 3.15).

```
query {
  getSharedCards(doctorId: "672c0e8bdd7e620c5427c24d") {
    _id
    email
    firstName
    lastName
    role
  }
}
```

Рисунок 3.15 Запит на отримання даних користувачів

11. Поділитися медичною карткою

Запит тестує функціонал, за допомогою якого пацієнт може ділитися своєю медичною карткою з лікарем. Postman допомагає перевірити, чи правильно працює цей процес і чи фіксується він у базі даних (рис. 3.16).

```
{
  "query": "mutation { shareCard(patientId: \"672b90c9eea0855bae1be348\", doctorId:
    \"672c0e8bdd7e620c5427c24d\") }"
}
```

Рисунок 3.16 Запит щоб відправити дані лікарю

12. Завантаження файлів

Цей запит перевіряє завантаження файлів на сервер. За допомогою Postman можна завантажити реальні файли та перевірити їхню обробку сервером (рис. 3.17).

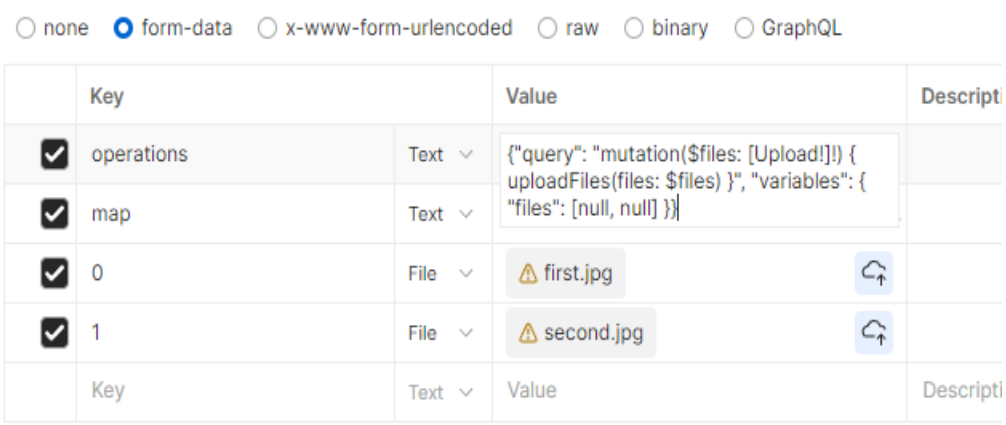

Рисунок 3.17 Запит щоб завантажити файл

3.4 Реалізація клієнтської частини

Для створення клієнтської частини інформаційної технології, спрямованої на моніторинг стану здоров'я, використовувалася технологія React Native. Ця платформа була обрана за її здатність забезпечувати розробку мобільних застосунків з використанням єдиного коду для iOS та Android, що значно скоротило час розробки та спростило підтримку.

React Native забезпечує високий рівень продуктивності застосунків, дозволяючи створювати нативні компоненти та інтегрувати їх у JavaScript-код. Реалізована архітектура додатка має чітку модульну структуру, що полегшує підтримку та розширення функціоналу. Використовувалися найкращі практики React Native для забезпечення зручності користування та стабільності програми.

Для навігації застосовувалася бібліотека React Navigation, яка дозволяє створювати різні типи навігаційних схем: стекову, табуляційну, та інші. Це забезпечило гнучку систему переходів між екранами, що покращує зручність користування та підтримує інтуїтивний інтерфейс.

Однією з ключових функцій застосунку є можливість зйомки фото та відео, що забезпечується завдяки використанню React Native Camera. Ця бібліотека підтримує різні режими роботи камери, що дозволяє користувачам знімати та зберігати вміст високої якості. Крім того, використання камери

оптимізовано для роботи як з фото, так і з відео файлами, а інтерфейс взаємодії є простим та зрозумілим для користувача.

Завантаження файлів на сервер реалізоване за допомогою інтеграції GraphQL з можливістю передачі великих даних. Для цього використовувалася бібліотека apollo-client, що дозволяє ефективно комунікувати з сервером. Завантаження файлів організоване через форму, що підтримує передачу кількох файлів одночасно, та забезпечує зручну взаємодію користувача з системою. Логіка завантаження передбачає обробку файлів різного типу, зокрема документів та медичних записів.

Для комунікації з бекендом використовувалася інтеграція з GraphQL. Це дозволило значно скоротити кількість запитів та підвищити ефективність обміну даними завдяки чіткому визначенню структурованих запитів і мутацій. Клієнтська частина використовує бібліотеку apollo-client, яка є потужним інструментом для управління станом і підтримки запитів у React Native-застосунках.

Для покращення користувацького досвіду додатково використовувалися бібліотеки, що забезпечують стиль та функціональність застосунку. Наприклад, styled-components допомогли структурувати та стилизувати інтерфейс застосунку, забезпечуючи його сучасний вигляд. Компоненти були оптимізовані для різних розмірів екранів та пристроїв, що гарантує адаптивність додатка.

React Native Reanimated додав можливість створювати плавні анімації та переходи, що зробило інтерфейс ще більш привабливим і сучасним. Це дозволяє користувачам взаємодіяти з програмою без затримок та проблем із продуктивністю.

Опис екранів:

Екран авторизації є початковим екраном програми, який надає користувачеві можливість увійти до системи або зареєструватися (рис. 3.18).

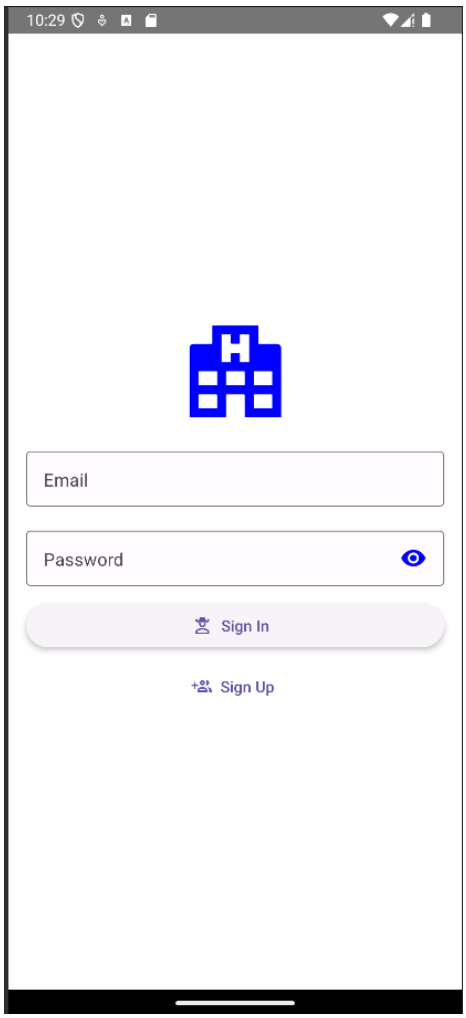

Рисунок 3.18 Екран авторизації

Інтерфейс містить два поля для введення:

- Поле для введення електронної пошти.
- Поле для введення пароля.

Користувач вводить свої облікові дані і натискає кнопку Sign In для виконання авторизації. Якщо авторизація успішна, система перенаправляє користувача на головний екран.

Крім того, на екрані розміщена кнопка Sign Up, яка дозволяє перейти до екрану реєстрації, якщо користувач ще не має облікового запису.

Екран авторизації також включає механізми валідації введених даних:

- Перевірка формату електронної пошти (наприклад, наявність символу "@").
- Перевірка довжини пароля (мінімум 8 символів).

- Відображення повідомлення про помилку у разі порожніх полів або некоректного введення даних.
- Забезпечення безпеки шляхом приховування введеного пароля.
- Такий підхід дозволяє мінімізувати помилки при введенні даних і забезпечує більш комфортний досвід для користувача.

Екран вибору ролі доступний після успішної реєстрації користувача (рис. 3.19).

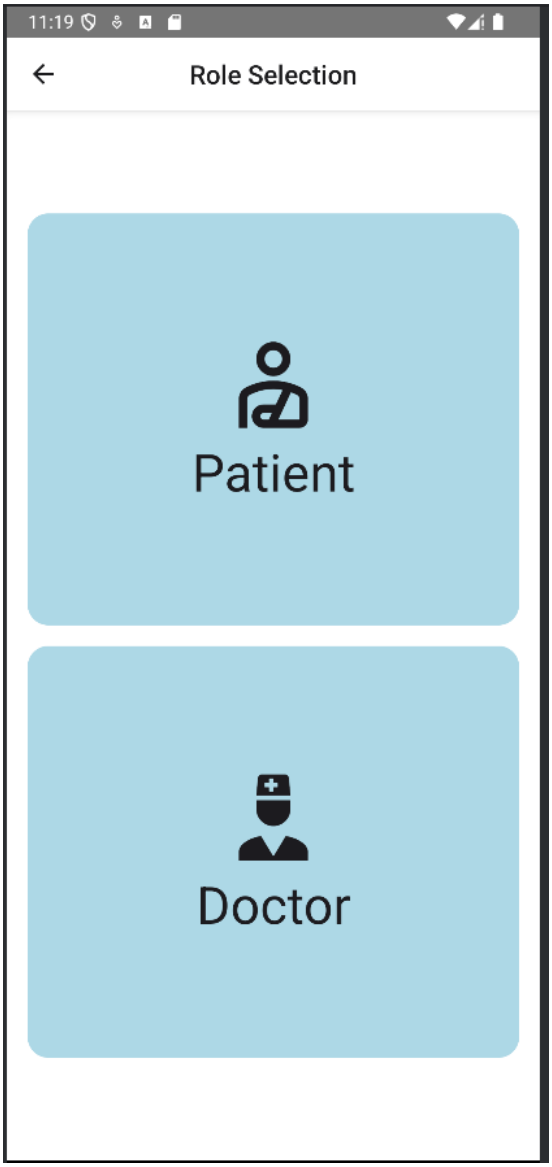

Рисунок 3.19 Екран вибору ролі

На цьому екрані користувач має вибрати одну з двох ролей, які визначають його функціональні можливості в додатку:

- Пацієнт
- Лікар

Інтерфейс складається з двох великих кнопок із відповідними іконками та текстом. Після натискання на одну з кнопок програма зберігає обрану роль користувача та автоматично перенаправляє його на головний екран. Цей етап важливий для персоналізації функціоналу програми, оскільки кожна роль має свій набір доступних функцій.

Для зручності користувачів кнопки займають майже весь екран, що робить інтерфейс інтуїтивно зрозумілим і простим у використанні навіть на мобільних пристроях із невеликим екраном. Екран налаштувань забезпечує можливість керування обліковим записом користувача (рис. 3.20).

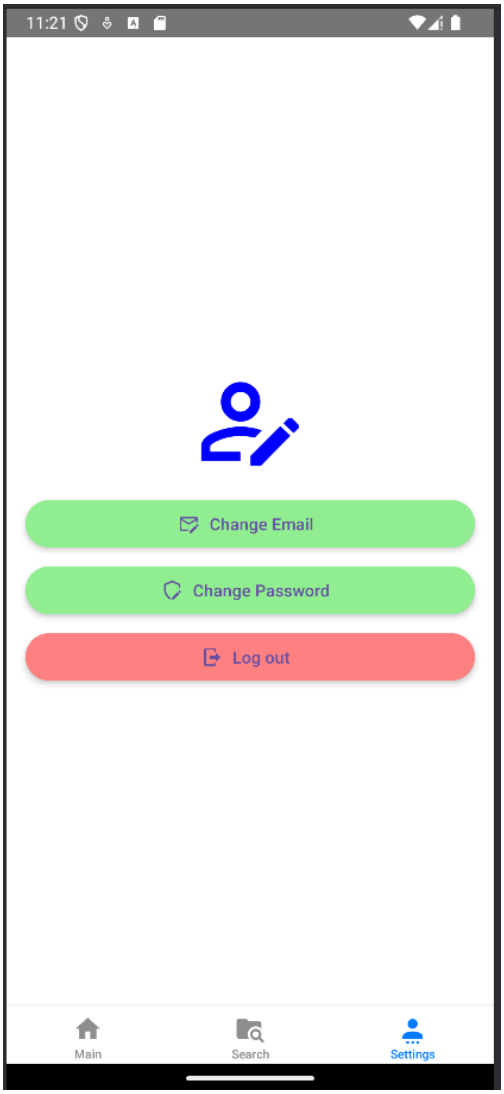

Рисунок 3.20 Екран налаштувань

Він містить три основні кнопки:

Зміна електронної пошти — дозволяє змінити email, який використовується для входу в систему. Користувач вводить нову адресу електронної пошти, після чого система перевіряє її коректність і оновлює дані в профілі.

Зміна паролю — забезпечує функцію зміни пароля. Користувач повинен ввести старий пароль для підтвердження, а також двічі новий пароль для валідації.

Вихід із акаунту — кнопка, яка дозволяє користувачеві завершити сесію. Після натискання користувач автоматично перенаправляється на екран авторизації.

Інтерфейс екрана налаштувань є мінімалістичним і зручним, щоб користувач легко міг виконати потрібні операції. Головний екран є ключовою частиною додатку (рис. 3.21).

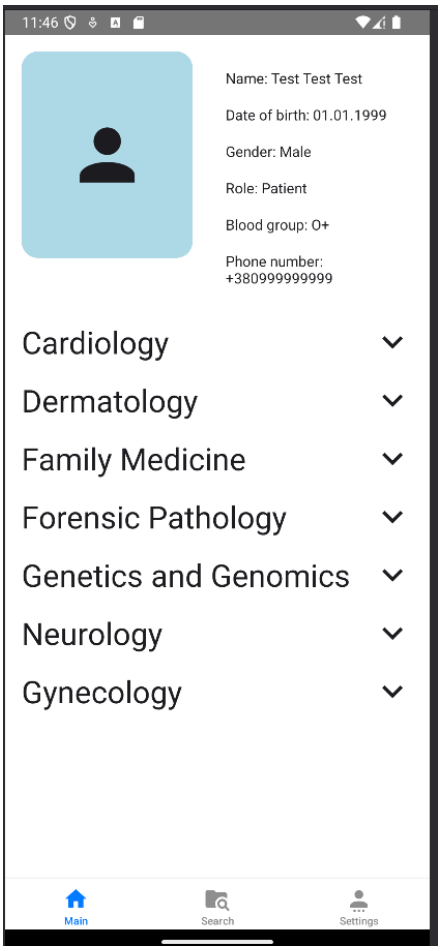

Рисунок 3.21 Головний екран

Він відображає персональну інформацію користувача, а також дані, пов’язані з медичною історією. Інтерфейс складається з кількох основних секцій:

Фотографія користувача — відображає аватар користувача, який може бути завантажений через додаток.

Особиста інформація — включає ім’я, прізвище, дату народження, контактні дані та інші персональні відомості.

Категорії медицини — перелік категорій в яких міститься інформація про минулі візити, медичні записи та рекомендації лікарів.

Користувач може взаємодіяти з цими даними, переглядати історію візитів, додавати нові записи або змінювати існуючу інформацію. Дизайн головного екрана забезпечує швидкий доступ до основної інформації та зручну навігацію між секціями.

Крім того, головний екран адаптований для відображення ролі користувача. Для ролі «Лікар» категорії будуть замінені на дані про досвід роботи, а також розділ з дипломами та сертифікатами.

3.5 Тестування клієнтської частини

Тестування клієнтської частини мобільного додатку є важливим етапом розробки, що дозволяє забезпечити стабільність роботи, відповідність функціоналу вимогам, а також зручність використання інтерфейсу для кінцевих користувачів. У процесі тестування застосовувались як автоматизовані, так і мануальні підходи [14].

Для автоматизації тестування клієнтської частини використовувались спеціальні інструменти, які дозволили перевірити роботу окремих компонентів, функцій і загальну логіку додатку [21]. Основні аспекти автоматизованого тестування:

Тести компонентів:

Написані модульні тести для основних компонентів за допомогою бібліотеки Jest [7] та React Native Testing Library [20].

Основний акцент був зроблений на перевірку правильності відображення інтерфейсу, валідації форм, коректності виклику GraphQL-запитів і обробки їх відповідей.

Приклад тестування форми авторизації: перевірка коректності валідації email, відображення повідомлень про помилки та функціональності кнопок.

Тести навігації:

Перевірено, чи правильно відбувається перенаправлення між екранами. Наприклад, після авторизації користувача система коректно перенаправляє його на головний екран.

Написані тести для перевірки умовної навігації залежно від ролі.

Тести інтеграційні:

Використано для перевірки взаємодії між компонентами та сервісами. Тести перевіряють правильність обробки відповіді від сервера та відображення відповідних повідомлень.

Перевірка GraphQL-запитів:

Проведено тестування взаємодії додатку з сервером через GraphQL [5]. Для цього використано мокові сервери, які імітують відповіді реального бекенду. Наприклад, перевірка запиту на авторизацію: тестувалося, чи коректно додається токен в заголовки, а також обробляються випадки з помилками, такі як невалідні дані.

Мануальне тестування виконувалось для перевірки додатку з точки зору реального користувача. Цей метод дозволив виявити та виправити потенційні проблеми, які могли залишитися непоміченими під час автоматизованого тестування.

Основні етапи мануального тестування:

Функціональне тестування:

Всі екрани додатку були протестовані на відповідність заявленому функціоналу. Наприклад, перевірено, чи коректно відображаються помилки

валідації на екрані авторизації, чи зберігається роль користувача після вибору на екрані вибору ролі.

Тестування інтерфейсу:

Особливу увагу було приділено дизайну та зручності використання.

Випробувано взаємодію з кнопками, текстовими полями та іншими елементами інтерфейсу на різних пристроях із різними розмірами екрану.

Перевірка навігації:

Тестувалося перемикання між екранами та їхня взаємодія. Наприклад, після успішного входу в систему користувач завжди потрапляє на головний екран, а після виходу — повертається на екран авторизації.

Тестування ролей:

Для кожної ролі проведено перевірку доступного функціоналу. Наприклад, для доктора було протестовано відображення карток пацієнтів, а для пацієнта — відображення категорій медичних даних.

Кросплатформне тестування:

Додаток перевірено на Android та iOS, щоб переконатися у стабільній роботі на обох платформах [12].

Перевірка продуктивності:

Виконано тестування на затримки під час виконання запитів до серверу, відкриття екрану чи завантаження зображень [6].

Тестування помилок:

Перевірено, як додаток поводиться в разі збоїв, таких як відсутність підключення до Інтернету, неправильний формат даних або помилки сервера.

ВИСНОВКИ

У ході розробки та реалізації проєкту інформаційної технології моніторингу здоров'я для підтримки прийняття рішень та забезпечення персоналізованого підходу було виконано низку важливих завдань. Завдяки ретельному підходу до кожного етапу, вдалося створити функціональний, зручний та безпечний застосунок. Основні результати роботи та підсумки наведені нижче.

1. Розробка серверної частини

Було створено бекенд на основі Node.js із використанням TypeScript для забезпечення типізації коду та підвищення надійності. Структура сервера була розроблена з використанням модульного підходу, що спрощує підтримку та розширення функціоналу. Використання GraphQL дозволило значно оптимізувати обмін даними між сервером та клієнтом завдяки чітко структурованим запитам та мутаціям. Також було реалізовано механізми авторизації через токени доступу, що забезпечує безпеку даних користувачів.

2. Інтеграція MongoDB для зберігання даних

Для зберігання медичних записів, профілів користувачів та іншої важливої інформації було обрано MongoDB. Ця база даних забезпечує гнучкість у зберіганні складних структур даних, що є критично важливим для зберігання різнопланової медичної інформації. Завдяки використанню ODM-бібліотеки Mongoose, процес створення та управління моделями даних був значно спрощений.

3. Реалізація клієнтської частини

Клієнтська частина була реалізована за допомогою React Native, що дозволило створити кросплатформенний мобільний застосунок для iOS та Android. Було впроваджено модульну систему навігації за допомогою React Navigation, що забезпечує інтуїтивний інтерфейс користувача. Для реалізації функціоналу зйомки та завантаження фото та відео була використана бібліотека React Native Camera, яка забезпечує високу якість роботи з

камерою. Завантаження файлів на сервер реалізовано через GraphQL-запити, що оптимізує процес передачі даних.

4. Забезпечення захисту даних

Система авторизації була створена із застосуванням JWT-токенів, що дозволяє забезпечити захист персональної інформації користувачів та їхніх медичних записів. Механізм токенів з інтерцепторами на клієнтській частині забезпечує автоматичне оновлення токенів та захищає застосунок від несанкціонованого доступу.

5. Тестування функціоналу

Серверна частина була ретельно протестована за допомогою Postman, що дозволило перевірити працездатність запитів та мутацій, а також впевнитися в коректній роботі системи безпекових механізмів та передачі даних. Було проведено тестування таких функцій, як реєстрація, авторизація, зміна пароля та електронної пошти, а також завантаження файлів.

Клієнтська частина була протестована програмно та мануально. Всі основні функції працюють стабільно. Додаток повністю відповідає технічним вимогам і є готовим для використання. Додаток забезпечує комфортну взаємодію з користувачем, зручну навігацію між екранами та належну візуалізацію інтерфейсу. Усі виявлені проблеми були виправлені на етапі тестування, що дозволило досягти високої якості кінцевого продукту.

У результаті розробки було створено повноцінний інформаційний мобільний застосунок моніторингу здоров'я для підтримки прийняття рішень та забезпечення персоналізованого підходу. Застосунок відповідає всім вимогам щодо безпеки та функціональності, забезпечує зручну роботу з медичними даними, дозволяє взаємодіяти з лікарями та ділитися медичними записами. Використання сучасних технологій, таких як React Native, GraphQL, Node.js та MongoDB, дозволило створити гнучку та надійну систему, яка легко масштабується і може бути розширена новими функціями у майбутньому.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Андреасян Г. Свіжий аналіз ринку Healthtech: 3 головні тренди 2023 року. Блоги про бізнес, політику, юридичну систему | LIGA.net. URL: https://blog.liga.net/user/handreasian/article/49919 (дата звернення: 04.12.2024).

2. Андрощук Г. Цифрова трансформація в охороні здоров'я: аналіз технологічних трендів - Юридична Газета. Юридична газета – онлайн версія. URL: https://yur-gazeta.com/publications/practice/informaciyne-pravo-telekomunikaciyi/cifrova-transformaciya-v-ohoroni-zdorovya-analiz-tehnologichnih-trendiv.html (дата звернення: 04.12.2024).

3. Як мобільні додатки трансформують галузь охорони здоров'я? | Stfalcon. Custom Software Development Company | Stfalcon.com. URL: https://stfalcon.com/uk/blog/post/How-Mobile-Apps-Are-Transforming-the-Healthcare-Industry (дата звернення: 04.12.2024).

4. Alawiye T. Advancements in AI applications for healthcare and user-centric digital health solutions. International journal of innovative science and research technology (IJISRT). 2024. С. 1578–1579. URL: https://doi.org/10.38124/ijisrt/ijisrt24jun1093 (дата звернення: 04.12.2024).

5. Apollo GraphQL Blog – latest insights and news on API and GraphQL solutions. Streamlining APIs, Databases, & Microservices | Apollo GraphQL. URL: https://www.apollographql.com/blog/ (дата звернення: 04.12.2024).

6. Atlassian documentation. Atlassian Documentation | Atlassian Support | Atlassian Documentation. URL: https://confluence.atlassian.com/enterprise/jira-performance-testing-available-tools-729743538.html (дата звернення: 04.12.2024).

7. Getting started · jest. Jest Delightful JavaScript Testing. URL: https://jestjs.io/docs/getting-started (дата звернення: 04.12.2024).

8. Google fit | google for developers. Google for Developers. URL: https://developers.google.com/fit (дата звернення: 04.12.2024).

9. GraphQL | A query language for your API. GraphQL | A query language for your API. URL: https://graphql.org/ (дата звернення: 04.12.2024).

10. Healthcare. Apple. URL: https://www.apple.com/healthcare/ (дата звернення: 04.12.2024).

11. Impact of mobile health and medical applications on clinical practice in gastroenterology / S. Kernebeck та ін. World journal of gastroenterology. 2020. Т. 26, № 29. С. 4182–4197. URL: https://doi.org/10.3748/wjg.v26.i29.4182 (дата звернення: 04.12.2024).

12. Interactive mobile app testing on 2000+ ios & android devices. BrowserStack. URL: https://www.browserstack.com/app-live (дата звернення: 04.12.2024).

13. JavaScript with syntax for types. TypeScript: JavaScript With Syntax For Types. URL: https://www.typescriptlang.org/ (дата звернення: 04.12.2024).

14. Knott D. Hands-On mobile app testing: a guide for mobile testers and anyone involved in the mobile app business. Addison-Wesley Longman, Incorporated, 2015. 256 с.

15. MongoDB: the developer data platform. MongoDB. URL: https://www.mongodb.com/ (дата звернення: 04.12.2024).

16. Node.js – Run JavaScript Everywhere. Node.js – Run JavaScript Everywhere. URL: https://nodejs.org/ (дата звернення: 04.12.2024).

17. Our software | epic. Epic | With the patient at the heart. URL: https://www.epic.com/software#PatientEngagement (дата звернення: 04.12.2024).

18. Postman documentation overview | postman learning center. Postman Learning Center. URL: https://learning.postman.com/docs/introduction/overview/ (дата звернення: 04.12.2024).

19. React Native · Learn once, write anywhere. React Native · Learn once, write anywhere. URL: https://reactnative.dev/ (дата звернення: 04.12.2024).

20. React native testing library. URL: https://callstack.github.io/react-native-testing-library/ (дата звернення: 04.12.2024).

21. Welcome - appium documentation. Redirecting. URL: https://appium.io/docs/en/latest/ (дата звернення: 04.12.2024).

22. WHO traditional medicine global summit 2023 meeting report: gujarat declaration. Journal of ayurveda and integrative medicine. 2023. С. 100821. URL: https://doi.org/10.1016/j.jaim.2023.100821 (дата звернення: 04.12.2024).

ДОДАТОК А

Лістинг серверної частини:

Головний файл

```
import express from 'express'
import { graphqlUploadExpress } from 'graphql-upload-ts'
import { ApolloServer } from 'apollo-server-express'
import { typeDefs } from './graphql/typeDefs'
import { resolvers } from './graphql/resolvers'
import connectDB from './config/db'
import { authMiddleware } from './auth/authMiddleware'
import path from 'path'
import cors from 'cors'

const app = express()
const PORT = process.env.PORT || 3001

const corsOptions = {
  origin: '*',
  optionsSuccessStatus: 200
}

app.use((req, res, next) => {
  const origin = '*'
  res.setHeader('Access-Control-Allow-Origin', origin)
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT,
DELETE')
  res.setHeader(
    'Access-Control-Allow-Headers',
    'Origin, X-Requested-With, Content-Type, Accept,
Authorization'
  )
  res.setHeader('Access-Control-Allow-Credentials', 'false')
  next()
})
```

```
app.use(cors(corsOptions))

app.use('/uploads',           express.static(path.join(__dirname,
'../uploads')))

app.use(graphqlUploadExpress())

async function startServer() {
  const server = new ApolloServer({
    typeDefs,
    resolvers,
    context: ({ req }) => {
      try {
        const    user    =    req?.headers?.authorization    ?
authMiddleware(req) : null
        return { user }
      } catch (error) {
        throw new Error(`${error}`)
      }
    }
  })
  await server.start()
  // eslint-disable-next-line @typescript-eslint/ban-ts-comment
  // @ts-ignore
  server.applyMiddleware({ app, path: '/graphql' })

  app.listen(PORT, () => {
    console.log(
      `Server                      running                      on
http://localhost:${PORT}${server.graphqlPath}`
    )
  })
}

connectDB()
startServer()
```

Обробник токену авторизації

```
import jwt from 'jsonwebtoken'

// eslint-disable-next-line @typescript-eslint/no-explicit-any
export const authMiddleware = (req: any) => {
  const authHeader = req.headers.authorization

  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    throw new Error('No token provided')
  }

  const token = authHeader.split(' ')[1]

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET!)
    return decoded
  } catch {
    throw new Error('Invalid token')
  }
}
```

Підключення до бази даних

```
import mongoose from 'mongoose'

const connectDB = async () => {
  try {
    await mongoose.connect(process.env.MONGODB_URI!)
    console.log('MongoDB connected')
  } catch (error) {
    console.error('Error connecting to MongoDB', error)
    process.exit(1)
  }
}

export default connectDB
```

Сервіс керування відправки емейлів

```
import nodemailer from 'nodemailer'
import dotenv from 'dotenv'

dotenv.config()

const transporter = nodemailer.createTransport({
  host: 'smtp.ukr.net',
  port: 465,
  secure: true,
  auth: {
    user: process.env.EMAIL_USER,
    pass: process.env.EMAIL_PASS
  }
})

export const sendVerificationCode = async (email: string, code:
string) => {
  const mailOptions = {
    from: process.env.EMAIL_USER,
    to: email,
    subject: 'Your Verification Code',
    text: `Your verification code is ${code}`
  }

  await transporter.sendMail(mailOptions)
}
```

Сервіс роботи з токенами

```
import jwt from 'jsonwebtoken'

export const generateAccessToken = (id: string) => {
  return jwt.sign({ id }, process.env.JWT_SECRET!, { expiresIn:
'15m' })
}
```

```
export const generateRefreshToken = (id: string) => {
  return jwt.sign({ id }, process.env.JWT_REFRESH_SECRET!, {
    expiresIn: '7d'
  })
}

export const verifyToken = (token: string, secret: string) => {
  return jwt.verify(token, secret)
}
```

Файл з контролерами запитів

```
import bcrypt from 'bcryptjs'
import User from '../../models/User'
import VerificationCode from '../../models/VerificationCode'
import { sendVerificationCode } from '../../utils/emailService'
import {
  generateAccessToken,
  generateRefreshToken,
  verifyToken
} from '../../utils/tokenService'
import fs from 'fs'
import { FileUpload } from 'graphql-upload-ts'

export const resolvers = {
  Query: {
    getUser: async (
      _: unknown,
      { _id }: { _id: string },
      context: { user: { id: string } }
    ) => {
      try {
        if (!context.user) {
          throw new Error('Not authenticated')
        }
```

```
    const user = await User.findById(_id)
    if (!user) throw new Error('User not found')
    return user
  } catch (error) {
    throw new Error(`Error fetching user: ${error}`)
  }
},

getUsers: async (
  _: unknown,
  {
    role,
    position,
    search,
    page = 1,
    limit = 10
  }: {
    role?: string
    position?: string
    search?: string
    page?: number
    limit?: number
  },
  context: { user: { id: string } }
) => {
  try {
    if (!context.user) {
      throw new Error('Not authenticated')
    }

    const filter: Record<string, unknown> = {}
    if (role) filter.role = role
    if (position) filter.position = position

    if (search) {
```

```
          filter.$or = [
            { firstName: { $regex: search, $options: 'i' } },
            { lastName: { $regex: search, $options: 'i' } },
            { middleName: { $regex: search, $options: 'i' } }
          ]
        }

        const skip = (page - 1) * limit

        const users = await User.find(filter).skip(skip).limit(limit)

        return users
      } catch (error) {
        throw new Error(`Error fetching users: ${error}`)
      }
    },

    getSharedCards: async (
      _: unknown,
      {
        doctorId,
        search,
        page = 1,
        limit = 10
      }: { doctorId: string; search?: string; page?: number; limit?: number },
      context: { user: { id: string } }
    ) => {
      if (!context.user) {
        throw new Error('Not authorized')
      }

      const user = await User.findById(context.user.id)

      if (user?.role !== 'Doctor') {
```

```
        throw new Error('You have to be a doctor')
      }

      if (context.user.id !== doctorId) {
        throw new Error('Access denied')
      }

      const query = { sharedWith: doctorId }
      if (search) {
        //  eslint-disable-next-line  @typescript-eslint/ban-ts-
comment
        // @ts-ignore
        query.$or = [
          { firstName: { $regex: search, $options: 'i' } },
          { lastName: { $regex: search, $options: 'i' } }
        ]
      }

      const patients = await User.find(query)
        .skip((page - 1) * limit)
        .limit(limit)

      return patients
    }
  },

  Mutation: {
    sendCode: async (_: unknown, { email }: { email: string }) =>
{
      const  code  =  Math.floor(100000  +  Math.random()  *
900000).toString()
      await VerificationCode.findOneAndDelete({ email })
      await new VerificationCode({ email, code }).save()
      await sendVerificationCode(email, code)
      return 'Verification code sent to your email'
    },
```

```
    verifyCodeAndRegister: async (
      _: unknown,
      {
        email,
        code,
        password
      }: { email: string; code: string; password: string }
    ) => {
      const verificationRecord = await VerificationCode.findOne({
email, code })
      if (!verificationRecord) throw new Error('Invalid or expired
code')

      const existingUser = await User.findOne({ email })
      if (existingUser) throw new Error('User already exists')

      const hashedPassword = await bcrypt.hash(password, 10)
      const user = new User({ email, password: hashedPassword })
      await user.save()
      await VerificationCode.findOneAndDelete({ email })

      const accessToken = generateAccessToken(user.id)
      const refreshToken = generateRefreshToken(user.id)
      return { ...user.toObject(), accessToken, refreshToken }
    },

    login: async (
      _: unknown,
      { email, password }: { email: string; password: string }
    ) => {
      const user = await User.findOne({ email })
      if    (!user    ||    !(await    bcrypt.compare(password,
user.password)))
        throw new Error('Invalid email or password')
```

```
      const accessToken = generateAccessToken(user.id)
      const refreshToken = generateRefreshToken(user.id)
      return { ...user.toObject(), accessToken, refreshToken }
    },

    refreshToken: async (_: unknown, { token }: { token: string })
=> {
      try {
        const payload = verifyToken(token,
process.env.JWT_REFRESH_SECRET!)
        // eslint-disable-next-line @typescript-eslint/ban-ts-
comment
        // @ts-ignore
        const newAccessToken =
generateAccessToken(payload.userId)
        return newAccessToken
      } catch {
        throw new Error('Invalid refresh token')
      }
    },

    changePassword: async (
      _: unknown,
      {
        email,
        currentPassword,
        newPassword
      }: { email: string; currentPassword: string; newPassword:
string }
    ) => {
      const user = await User.findOne({ email })
      if (!user || !(await bcrypt.compare(currentPassword,
user.password)))
        throw new Error('Current password is incorrect')

      user.password = await bcrypt.hash(newPassword, 10)
```

```
      await user.save()
      return 'Password updated successfully'
    },

    changeEmail: async (
      _: unknown,
      {
        currentEmail,
        newEmail,
        code
      }: { currentEmail: string; newEmail: string; code: string }
    ) => {
      const verificationRecord = await VerificationCode.findOne({
        email: newEmail,
        code
      })
      if (!verificationRecord) throw new Error('Invalid or expired
code')

      const user = await User.findOne({ email: currentEmail })
      if (!user) throw new Error('User not found')

      user.email = newEmail
      await user.save()
      await  VerificationCode.findOneAndDelete({  email:  newEmail
})

      return 'Email updated successfully'
    },

    updateUser: async (
      _: unknown,
      {
        _id,
        input
      }: {
```

```
  _id: string
  input: {
    role?: 'User' | 'Doctor'
    firstName?: string
    lastName?: string
    middleName?: string
    bloodGroup?: string
    birthDate?: Date
    phone?: string
    gender?: string
    allergies?: string[]
    operations?: Array<{
      date: Date
      description: string
      photos?: string[]
    }>
    medicalCategories?: {
      category: string
      diagnoses: string[]
      visits: Array<{
        date: Date
        diagnosis?: string
        description: string
        files?: string[]
      }>
    }[]
    certificates?: string[]
    experience?: Array<{
      description: string
      startDate: string
      endDate: string
    }>
    position?: string
  }
},
context: { user: { id: string } }
```

```
    ) => {
      try {
        if (!context.user) {
          throw new Error('Not authenticated')
        }

        const user = await User.findById(_id)
        if (!user) throw new Error('User not found')

        Object.assign(user, input)

        await user.save()

        return user
      } catch (error) {
        throw new Error(`Error updating user ${error}`)
      }
    },

    shareCard: async (
      _: unknown,
      { patientId, doctorId }: { patientId: string; doctorId:
string },
      context: { user: { id: string } }
    ) => {
      if (!context.user) {
        throw new Error('Not authenticated')
      }

      const user = await User.findById(context.user.id)

      if (user?.role !== 'User') {
        throw new Error('Only patients can share their cards')
      }

      if (context.user.id !== patientId) {
```

```
      throw new Error('Access denied')
    }

    const patient = await User.findById(patientId)
    const doctor = await User.findById(doctorId)

    if (!patient || !doctor || doctor.role !== 'Doctor') {
      throw new Error('Invalid patient or doctor ID')
    }

    if (!patient.sharedWith) {
      patient.sharedWith = []
    }

    if (!patient.sharedWith.includes(doctorId)) {
      patient.sharedWith.push(doctorId)
    }

    await patient.save()
    return 'Card shared successfully'
  },

  uploadFiles: async (
    _: unknown,
    { files }: { files: Promise<FileUpload>[] },
    context: { user: { id: string } }
  ) => {
    try {
      if (!context.user) {
        throw new Error('Not authenticated')
      }

      const uploadedPaths: string[] = []

      for (const filePromise of files) {
        const fileConfig = await filePromise
```

```
          // eslint-disable-next-line @typescript-eslint/ban-ts-
comment
          // @ts-ignore
          const { createReadStream, filename } = fileConfig.file

          const path = `uploads/${Date.now()}-${filename}`
          const stream = createReadStream()

          await new Promise((resolve, reject) => {
            const out = fs.createWriteStream(path)
            stream.pipe(out)
            out.on('finish', () => resolve(true))
            out.on('error', (err) => reject(err))
          })

          uploadedPaths.push(path)
        }

        return uploadedPaths
      } catch (error) {
        throw new Error(`Error uploading files: ${error}`)
      }
    }
  }
}
```

ДОДАТОК Б

Лістинг клієнтської частини:

Екран логіну:

```
import React, { FC, useMemo, useState } from 'react'
import { StyleSheet, View } from 'react-native'

import { useNavigation } from '@react-navigation/native'
import { NativeStackNavigationProp } from '@react-navigation/native-stack'
import { Button, Icon, TextInput } from 'react-native-paper'
import { SafeAreaView } from 'react-native-safe-area-context'

import { RootStackParamList } from '@navigation/AppNavigator'

type AuthScreenNavigationProp = NativeStackNavigationProp<
  RootStackParamList,
  'Auth'
>

const AuthScreen: FC = () => {
  const [email, setEmail] = useState('')
  const [password, setPassword] = useState('')
  const [eyeEnabled, setEyeEnabled] = useState(true)

  const { navigate, replace } = useNavigation<AuthScreenNavigationProp>()

  const onEyePress = () => {
    setEyeEnabled((prev) => !prev)
  }

  const onSignUpPress = () => {
    navigate('Registration')
  }
```

```
  const onSignInPress = () => {
    // TODO change navigation to main screen
    replace('Home')
  }

  const eyeIcon = useMemo(() => (eyeEnabled ? 'eye' : 'eye-off'),
[eyeEnabled])

  return (
    <SafeAreaView style={styles.container}>
      <View style={styles.box}>
        <View style={styles.iconContainer}>
          <Icon      source="hospital-building"      color="blue"
size={100} />
        </View>

        <TextInput
          label="Email"
          autoComplete="email"
          inputMode="email"
          keyboardType="email-address"
          mode="outlined"
          value={email}
          onChangeText={setEmail}
        />

        <TextInput
          label="Password"
          autoComplete="password"
          mode="outlined"
          value={password}
          onChangeText={setPassword}
          secureTextEntry={eyeEnabled}
          right={
            <TextInput.Icon      icon={eyeIcon}      color="blue"
onPress={onEyePress} />
```

```
          }
        />

        <Button
          icon="account-cowboy-hat-outline"
          mode="elevated"
          onPress={onSignInPress}
        >
          Sign In
        </Button>

        <Button
          icon="account-multiple-plus-outline"
          mode="text"
          onPress={onSignUpPress}
        >
          Sign Up
        </Button>
      </View>
    </SafeAreaView>
  )
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff'
  },
  box: {
    flex: 1,
    paddingHorizontal: 16,
    justifyContent: 'center',
    gap: 16
  },
  iconContainer: {
    alignItems: 'center',
```

```
    width: '100%'
  }
})

export default AuthScreen
```

Головний файл навігації

```
import React from 'react'

import { createBottomTabNavigator } from '@react-navigation/bottom-tabs'
import { NavigationContainer } from '@react-navigation/native'
import { createNativeStackNavigator } from '@react-navigation/native-stack'
import { Icon } from 'react-native-paper'

import AuthScreen from '@screens/AuthScreen'
import MainScreen from '@screens/MainScreen'
import ProfileScreen from '@screens/ProfileScreen'
import RegistrationScreen from '@screens/RegistrationScreen'
import RoleSelectionScreen from '@screens/RoleSelectionScreen'
import SearchScreen from '@screens/SearchScreen'
import SettingsScreen from '@screens/SettingsScreen'

export type RootStackParamList = {
  Auth: undefined
  Registration: undefined
  RoleSelection: undefined
  Home: undefined
  Search: undefined
  Profile: { userId: string }
}

const Stack = createNativeStackNavigator<RootStackParamList>()
const Tab = createBottomTabNavigator()
```

```
function SearchStackNavigator() {
  return (
    <Stack.Navigator screenOptions={{ headerShown: false }}>
      <Stack.Screen name="Search" component={SearchScreen} />
      <Stack.Screen name="Profile" component={ProfileScreen} />
    </Stack.Navigator>
  )
}

function BottomTabNavigator() {
  return (
    <Tab.Navigator>
      <Tab.Screen
        name="Main"
        component={MainScreen}
        options={{
          headerShown: false,
          tabBarLabel: 'Main',
          tabBarIcon: ({ color, size }) => (
            <Icon source="home" color={color} size={size} />
          )
        }}
      />
      <Tab.Screen
        name="Search"
        component={SearchStackNavigator}
        options={{
          headerShown: false,
          tabBarLabel: 'Search',
          tabBarIcon: ({ color, size }) => (
            <Icon      source="folder-search"      color={color}
size={size} />
          )
        }}
      />
```

```
      <Tab.Screen
        name="Settings"
        component={SettingsScreen}
        options={{
          headerShown: false,
          tabBarLabel: 'Settings',
          tabBarIcon: ({ color, size }) => (
            <Icon     source="account-settings"     color={color}
size={size} />
          )
        }}
      />
    </Tab.Navigator>
  )
}

const AppNavigator: React.FC = () => {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen
          name="Auth"
          component={AuthScreen}
          options={{ headerShown: false }}
        />
        <Stack.Screen
          name="Registration"
          component={RegistrationScreen}
          options={{
            headerTitle: 'Registration',
            headerTitleAlign: 'center'
          }}
        />
        <Stack.Screen
          name="RoleSelection"
          component={RoleSelectionScreen}
```

```
          options={{
            headerTitle: 'Role Selection',
            headerTitleAlign: 'center'
          }}
        />
        <Stack.Screen
          name="Home"
          component={BottomTabNavigator}
          options={{ headerShown: false }}
        />
      </Stack.Navigator>
    </NavigationContainer>
  )
}

export default AppNavigator
```

Файл з обробкою запитів на серверну частину

```
import {
  ApolloClient,
  ApolloLink,
  HttpLink,
  InMemoryCache,
  Observable
} from '@apollo/client'
import { setContext } from '@apollo/client/link/context'
import { onError } from '@apollo/client/link/error'

import {
  clearTokens,
  getAccessToken,
  getRefreshToken,
  saveToken
} from '@store/index'
```

```
const URL = 'http://192.168.0.111:3001/graphql/'

const httpLink = new HttpLink({
  uri: URL
})

const authLink = setContext(async (_, { headers }) => {
  const token = getAccessToken()
  return {
    headers: {
      ...headers,
      Authorization: token ? `Bearer ${token}` : ''
    }
  }
})

const errorLink = onError(
  ({ graphQLErrors, networkError, operation, forward }) => {
    if (graphQLErrors) {
      for (const err of graphQLErrors) {
        if (err.extensions?.code === 'UNAUTHENTICATED') {
          const refreshToken = getRefreshToken()

          if (refreshToken) {
            // Return an Observable to retry the operation
            return new Observable((observer) => {
              fetch(`${URL}/refresh-token`, {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify({ refreshToken })
              })
                .then((response) => response.json())
                .then((data) => {
                  if (data.accessToken && data.refreshToken) {
                    saveToken(data.accessToken,
data.refreshToken)
```

```
                    const                oldHeaders                =
operation.getContext().headers
                    operation.setContext({
                      headers: {
                        ...oldHeaders,
                        Authorization:                        `Bearer
${data.accessToken}`
                      }
                    })

                    // Retry the operation
                    return forward(operation).subscribe({
                      next: observer.next.bind(observer),
                      error: observer.error.bind(observer),
                      complete: observer.complete.bind(observer)
                    })
                  } else {
                    clearTokens()
                    observer.error(new  Error('Failed  to  refresh
token'))
                  }
                })
                .catch((error) => {
                  clearTokens()
                  observer.error(error)
                })
            })
          } else {
            clearTokens()
          }
        }
      }
    }
    if (networkError) {
      console.error(`[Network error]: ${networkError}`)
```

```
    }
  }
)

export const apolloClient = new ApolloClient({
  link: ApolloLink.from([authLink, errorLink, httpLink]),
  cache: new InMemoryCache()
})
```

Файл роботи з токенами

```
import { MMKV } from 'react-native-mmkv'

export const storage = new MMKV()

export const StorageKeys = {
  ACCESS_TOKEN: 'accessToken',
  REFRESH_TOKEN: 'refreshToken'
}

export const saveToken = (accessToken: string, refreshToken:
string) => {
  storage.set(StorageKeys.ACCESS_TOKEN, accessToken)
  storage.set(StorageKeys.REFRESH_TOKEN, refreshToken)
}

export const getAccessToken = () =>
  storage.getString(StorageKeys.ACCESS_TOKEN) || null

export const getRefreshToken = () =>
  storage.getString(StorageKeys.REFRESH_TOKEN) || null

export const clearTokens = () => {
  storage.delete(StorageKeys.ACCESS_TOKEN)
  storage.delete(StorageKeys.REFRESH_TOKEN)
}
```

Головний екран з провайдерами

```
import React from 'react'

import { ApolloProvider } from '@apollo/client'
import ErrorBoundary from 'react-native-error-boundary'
import { PaperProvider } from 'react-native-paper'
import { SafeAreaProvider } from 'react-native-safe-area-context'

import { apolloClient } from '@api/client'
import AppNavigator from '@navigation/AppNavigator'

export default function App() {
  return (
    <ErrorBoundary>
      <ApolloProvider client={apolloClient}>
        <SafeAreaProvider>
          <PaperProvider>
            <AppNavigator />
          </PaperProvider>
        </SafeAreaProvider>
      </ApolloProvider>
    </ErrorBoundary>
  )
}
```