МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики
(повне найменування назва факультету)

Кафедра програмування
(повна назва кафедри)

ДИПЛОМНА РОБОТА

на тему:

РОЗРОБКА ВОЛОНТЕРСЬКОЇ СОЦІАЛЬНОЇ МЕРЕЖІ

DEVELOPMENT OF A VOLUNTEER SOCIAL NETWORK

Виконав: студент групи ПМІ-41
спеціальності
122 Комп'ютерні науки та інформаційні технології (інформатика)
(шифр і назва спеціальності)

Плотіцин М.С.
(підпис) (прізвище та ініціали)

Керівник
проф. Заболоцький Т.М.
(підпис) (прізвище та ініціали)

Рецензент

(підпис) (прізвище та ініціали)

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет прикладної математики та інформатики

Кафедра програмування

Спеціальність 122 Комп'ютерні науки та інформаційні технології (інформатика)
(шифр і назва)

«ЗАТВЕРДЖУЮ»

Завідувач кафедри

_____

" " 20 року

ЗАВДАННЯ

НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

_____Плотіцина Максима Сергійовича_____
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка волонтерської соціальної мережі. Development of a volunteer social network

керівник роботи Заболоцький Тарас Миколайович, доктор економічних наук, професор_____,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені Вченою радою факультету від "13" вересня 2022 року протокол №15

2. Строк подання студентом роботи __13 червня 2023 року_____

3. Вихідні дані до роботи

*__Офіційні документації сервісів та бібліотек. Інтернет-форуми та обговорення на офіційних сторінках бібліотек у github*_____

_____

4. Зміст дипломної роботи (перелік питань, які потрібно розробити)

*__1) Визначити проблему, яка існує на даний момент. Виділити основні аспекти, оприділити складову безпеки_*

*__2) Дослідити та обрати технології для виконання даної дипломної роботи*_____

*__3) Продемонструвати серверну частину застосунку, показавши важливі прийняті рішення та загальну робочу складову*_____

*__4) Продемонструвати користувацьку частину, а саме які методи були обрані для роботи та як була реалізована комунікація з сервером*_____

*__5) Зробити візуальну презентацію виконаної роботи*_____

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

*__1) зображення роботи застосунку*_____

*__2) зображення діаграми бази даних*_____

3) зображення використання курсору за допомогою Prisma

4) зображення сторінки Swagger з описаними запитами

6. Консультанти розділів роботи

Розділ	**Прізвище, ініціали та посада консультанта**	**Підпис, дата**	
		Завдання видав	**Завдання прийняв**

7. Дата видачі завдання ____13 вересня 2022 року_____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	**Назва етапів дипломної роботи**	**Строк виконання етапів роботи**	**Примітка**
1	*Визначення проблеми та вибір технологій*	*жовтень*	*Виконано*
2	*Розробка серверної частини*	*жовтень-листопад*	*Виконано*
3	*Розробка користувацької частини*	*грудень-січень*	*Виконано*
4	*Перше тестування*	*січень*	*Виконано*
5	*Виправлення помилок*	*лютий*	*Виконано*
6	*Підключеення сервісу для зберігання фотознимків*	*березень*	*Виконано*
7	*Налаштування серверної частини та бази даних для роботи з фотознимками*	*березень*	*Виконано*
8	*Створення віртуальної машини, та запуск серверної частини у хмарі*	*квітень*	*Виконано*
9	*Оформлення дипломної роботи*	*травень-червень*	*Виконано*

Студент _____ Плотіцин М.С.

Керівник роботи _____ Заболоцький Т.М.

АВТОРЕФЕРАТ

Ця дипломна робота представляє новий мобільний застосунок (далі “Воло”), призначений для спрощення координації та співпраці між волонтерами, військовими, біженцями та іншими відповідними сторонами, залученими до допомоги у повномасштабній війні, що триває в Україні. “Воло” базується на найсучасніших технологіях і алгоритмах, що дозволяє користувачам ефективно створювати, переглядати та відповідати на запити про допомогу, забезпечуючи таким чином швидку та ефективну координацію зусиль.

На додаток до інтуїтивно зрозумілого інтерфейсу користувача, “Воло” включає надійні заходи безпеки та конфіденційності даних, можливості зв’язку в реальному часі та інші функції, які підвищують зручність і ефективність для користувачів. Інклюзивний підхід програми поширює допомогу за межі кордонів України, дозволяючи людям у всьому світі відкривати відповідні запити та ініціативи та робити свій внесок у них, створюючи глобальну мережу солідарності.

Війна в Україні, яка почалася у 2014 році, зумовила необхідність розробки такого інструменту для вирішення логістичних та координаційних проблем, які виникають під час доставки допомоги військовим і біженцям. Використовуючи сучасні технології та зосереджуючись на прозорості та адаптивності, “Воло” прагне сприяти ефективній комунікації, забезпечити централізовану платформу для координації допомоги, залучити світову спільноту, підвищити ефективність допомоги та представити інноваційне рішення поточної кризи.

Було вжито суттєвих заходів безпеки, щоб країни-агресори, зокрема росія та білорусь, не отримали доступу до програми з використанням геолокації та даних VPN. Простий процес програми сприяє ефективному розподілу та відповіді на запити в чотирьох категоріях: фінансові збори, збори для конкретних речей, матеріальна допомога та підтримка біженців.

Завдяки розробці “Воло” ця робота розглядає як практичні, так і теоретичні аспекти полегшення надання допомоги. Вона досліджує динаміку взаємодії волонтерів і жертводавців у цифровому контексті під час війни.

Посилання на користувацьку частину:

https://bitbucket.org/karrtopelka-max/volo_frontend/

Посилання на серверну частину:

https://bitbucket.org/karrtopelka-max/volo_server/

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

“Воло” - застосунок який розроблявся

VPN - Virtual Private Network - це сервіс, який дозволяє створити зашифроване тунельне з'єднання між пристроєм (наприклад, комп'ютером, смартфоном або планшетом) та інтернетом. Він працює, перехоплюючи вихідний трафік даних і пересилаючи його через сервери VPN, які розташовані у різних місцях по всьому світу.

Підхід mobile-first - це підхід до розробки веб-сайтів або застосунків, коли спочатку проектується та розробляється для мобільних пристроїв, а потім адаптується до більших екранів, таких як планшети та настільні комп'ютери.

Фреймворк - це набір інструментів і бібліотек, які надають розробникам готову і структуровану основу для побудови веб-застосунків або API .

ORM (Object-Relational Mapping) - це технологія, яка дозволяє розробникам працювати з базами даних, використовуючи об'єктно-орієнтований підхід, замість традиційного роботи з SQL-запитами та таблицями бази даних.

DTO (Data Transfer Object) - це патерн проектування, який використовується для передачі даних між компонентами системи. DTO - це об'єкт, який використовується для упаковки та передачі даних з одного місця в інше.

Refresh token - це механізм, який використовується для оновлення та продовження строку дії аутентифікаційного токену без необхідності повторної аутентифікації користувача.

Middleware - це проміжний шар програмного забезпечення, який обробляє запити та відповіді в мережевому стеку застосунку. В контексті веб-застосунків, middleware дозволяє виконувати функції перед обробкою запиту та після нього, або модифікувати поведінку запиту.

“Обгорткове пекло” ("wrapper hell") - відноситься до ситуації, коли компоненти в застосунку стають занадто глибоко вкладеними в інші компоненти. Це може

призводити до зайвої складності, погіршення читабельності коду та збільшення його об'єму.

Promise - це об'єкт в JavaScript, який використовується для асинхронного програмування. Він представляє результат асинхронної операції, яка може бути успішною або невдалим.

ВСТУП

У контексті дедалі складніших світових конфліктів, масових міграцій та нагальної потреби в гуманітарній допомозі, ефективні механізми координації та співпраці набувають особливого значення. Презентована в даній роботі розробка - мобільний застосунок "Воло" - створений відповідно до цих сучасних вимог, в основі яких лежать актуальні проблеми, пов'язані з триваючим в Україні з 2014 року військовим конфліктом.

Використовуючи передові IT-рішення, "Воло" функціонує як платформа, що дозволяє користувачам ініціювати, оглядати та реагувати на запити про допомогу. Центральним елементом дизайну застосунку є створення ефективного механізму координації дій у сфері надання матеріальної, фінансової та інформаційної допомоги. Для цього використовується комплексний, зручний для користувача інтерфейс, надійні інструменти забезпечення конфіденційності і безпеки, можливості взаємодії в реальному часі, а також додаткові функції, що сприяють підвищенню ефективності використання застосунку.

Стратегія "Воло", що ґрунтується на глобальному підході до надання допомоги, виходить за рамки географічних меж України, стимулюючи людей по всьому світу приєднуватися і вносити вклад в різноманітні проекти та ініціативи. Такий підхід підтримує формування універсальної мережі допомоги та співпраці.

Насущність такого інструменту, як "Воло", особливо висвітлюється у контексті війни в Україні, де він вирішує ключові проблеми матеріального забезпечення та координації доставки допомоги військовослужбовцям і біженцям. Базуючись на принципах адаптивності, "Воло" надає можливість динамічного оновлення запитів на допомогу та ресурсів відповідно до швидко змінюваних обставин. Крім того, застосунок сприяє прозорості, дозволяючи волонтерам та донорам відстежувати вплив їх підтримки.

Особлива увага приділяється забезпеченню безпеки в "Воло", де вживаються строгі заходи, включаючи перевірку геолокації та даних VPN, для

запобігання доступу з боку потенційно агресивних країн, зокрема росії та білорусії. Також програма має дизайн, що сприяє неперервному потоку запитів та відповідей на допомогу, охоплюючи чотири основні категорії: фінансова допомога, збір предметів першої необхідності, матеріальна допомога та підтримка біженців.

Мета

У світлі актуальної необхідності підвищення ефективності координації та співпраці в контексті волонтерської діяльності під час військового конфлікту в Україні, ця робота покликана розробити мобільний застосунок, призначений для синтезу зусиль волонтерів, благодійників та інших жертводавців, спрямованих на надання допомоги військовослужбовцям та біженцям. Основні аспекти цієї мети можна узагальнити наступним чином:

- Сприяння ефективній комунікації: Завданням розробки застосунку є фасилітування обміну інформацією між усіма зацікавленими сторонами, задіяними у процесі надання допомоги, з метою своєчасного та точного визначення потреб.
- Забезпечення централізованої платформи: Метою застосунку є створення централізованого середовища, де користувачі зможуть легко знаходити, організовувати та координувати допомогу, вчасно реагувати на зміну обставин та приоритетів.
- Підтримка глобальної спільноти: Застосунок призначений для створення доступної та зручної платформи для залучення в контекст української проблематики благодійників, волонтерів та інших осіб з усього світу, які прагнуть допомогти Україні під час війни.
- Збільшення ефективності допомоги: Застосунок покликаний оптимізувати розподіл ресурсів та витрат, забезпечуючи максимізацію ефекту від наданої допомоги на місцевому рівні.

- Створення інноваційного рішення: Метою розробки застосунку є використання сучасних технологій та підходів для створення новаторського продукту, який відповідає потребам користувачів.

Обгрунтування дослідницької проблеми

- Військовий конфлікт: З вибухом військових дій в Україні у 2022 році, нація зазнала впливу множини викликів, зокрема відчутної потреби у наданні допомоги військовослужбовцям та біженцям. Це викликало необхідність створення застосунку, що координує і сприяє співпраці у наданні такої допомоги.
- Логістика та координація: Велика кількість людей, які потребують допомоги, а також обсяги військових дій ускладнюють ефективну координацію зусиль волонтерів та благодійників. Отже, мобільний застоунок "Воло" виступає актуальним інструментом для забезпечення цього процесу координації та логістики.
- Розширення допомоги: Мобільний застосунок надає можливість простягнути руку допомоги за межі України, залучаючи людей з усього світу до спільної мети. Це підтверджує актуальність розробки застосунку, що може залучити міжнародні ресурси та мережі.
- Використання сучасних технологій: Великий спектр сучасних технологій та комунікаційних засобів створює умови для розробки ефективних інструментів, які спрощують процес надання допомоги. Актуальність застосунку "Воло" полягає в тому, що він використовує ці технології для досягнення своєї мети.
- Забезпечення прозорості: Волонтерство та благодійність часто зіткнені з викликами прозорості та контролю за використанням фондів та ресурсів. Актуальність застосунку "Воло" полягає в тому, що він сприяє створенню прозорого та контрольованого процесу, що дозволяє благодійникам та волонтерам відслідковувати результати своєї підтримки.

- Адаптація до змінливих обставин: У військовий час ситуація може швидко змінюватися, і потреби людей та організацій також можуть різко змінитися. Актуальність застосунку "Воло" полягає в тому, що він дозволяє легко адаптуватися до змінюваних обставин, оновлюючи запити допомоги та відповідні ресурси у реальному часі.
- Включення різноманітних груп населення: Застосунок "Воло" також важливий, оскільки він забезпечує включення різноманітних груп населення, таких як особи з інвалідністю, літні люди та діти, надаючи їм доступ до допомоги та можливість активної участі в процесі.

Цілі та завдання дослідження

- Аналізувати відповідні технології та методології, що застосовуються при розробці мобільних застосунків для волонтерської допомоги, і відібрати найбільш ефективні з них для впровадження у рамках даного проекту.
- Розробити інтуїтивно зрозумілий та зорієнтований на користувача інтерфейс для застосунку, який надасть користувачам зручність у створенні, перегляді та відгуку на запити допомоги.
- Реалізувати систему захисту та конфіденційності даних користувачів, яка враховує потенційні ризики та забезпечує захист персональних даних та інформації про фінансові транзакції.
- Інтегрувати можливість комунікації в реальному часі між користувачами, що сприяє обміну контактами, ресурсами та інформацією щодо надання допомоги.
- Здійснити тестування застосунку з погляду зручності користування, швидкодії, надійності та безпеки з метою виявлення та усунення можливих проблем та недоліків перед офіційним запуском.

Аспект безпеки

У контексті безпеки найважливішою задачею є вжиття всіх можливих заходів з метою запобігання доступу до застосунку з боку агресора (конкретно, громадян російської федерації та республіки білорусь). Для цього застосунок повинен безвідмовно мати доступ до геолокаційних даних користувача, а також до даних про використання віртуальної приватної мережі (VPN). Якщо система виявить, що користувач здійснює вхід з території росії та білорусії, або використовує активне VPN-з'єднання, доступ до застосунку буде тимчасово обмежено. В разі використання VPN, доступ буде відновлено після вимкнення цієї служби.

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ ПРОБЛЕМ ТА ВИБІР ТЕХНОЛОГІЙ

1.1 Проблема

Аналізуючи сучасний контекст, можемо виявити наявність офіційних фондів, до яких люди мають можливість здійснювати внески. Однак, окрім них, в системі благодійності важливу роль відіграють волонтери, що здійснюють збір фінансових чи матеріальних ресурсів, спрямованих на задоволення конкретних потреб різних бригад.

Розглянемо репрезентативний приклад, що демонструє типову ситуацію. Михайло, який активно залучений у волонтерську діяльність, вступає в контакт з 81 штурмовою бригадою. Згідно з їх потребами, їм необхідний позашляховик для швидкого перевезення людей до певних місць. Михайло публікує в "stories" на Instagram, розгортає проблему, додає посилання на mono Банку та опціонально номери карт, на які відбувається збір коштів, і просить поширити дану інформацію. У Михайла 240 підписників, але виникають дві основні проблеми:

1. Не всі з 240 підписників побачать згаданий пост
2. Не всі, хто побачив, поширить даний запит

На практиці, в середньому пост бачить близько 140-170 людей, при цьому менше 30 з них поширять цю інформацію далі. Відзначається, що не всі, хто побачив, здійснюють внески, але цей аспект не може бути включений у аналіз, оскільки він визначається персональними рішеннями, на які неможливо вплинути.

Якщо збір коштів завершиться (за умови його успішного завершення), це відбудеться лише після тривалого періоду часу, коли необхідний позашляховик може вже не бути актуальним, оскільки ситуація на фронті не чекає завершення фінансових зборів, а реагує на поточні обставини.

Це поставило на порядок денний важливе питання: "Як забезпечити те, щоб інформація досягала більшої кількості людей, ніж особисті знайомства

волонтера?" Відповіддю стала розробка спеціалізованого сервісу, який був би відповідальний за організацію волонтерських ініціатив (збір коштів та ін.).

Враховуючи тенденції сучасного світу, де, згідно з дослідженнями, все більшу роль відіграє мобільна першість (mobile-first approach), вирішено розробити мобільний застосунок.

1.2 Вибрані технології

В рамках розробки цієї ініціативи, процес розробки застосунку можна розділити на дві основні частини: серверну (backend) та клієнтську (frontend).

Серверна частина включає в себе наступні компоненти:

1. База даних:
    - PostgreSQL - Вибір цієї системи реляційної бази даних обумовлений її високою надійністю, масштабованістю та гнучкістю у питаннях забезпечення цілісності даних. PostgreSQL відрізняється підтримкою розширених типів даних, складних запитів та можливістю ефективної роботи з великими наборами даних. Крім того, PostgreSQL може масштабуватися як горизонтально, так і вертикально.
    - Prisma (ORM) - У поєднанні з PostgreSQL, Prisma була вибрана через сучасний підхід до роботи з базами даних, який дозволяє взаємодіяти з базою даних в безпечний для типів спосіб. Prisma дозволяє підвищити продуктивність та знизити ймовірність помилок при моделюванні схем.
2. Фреймворк:
    - NestJS - Цей фреймворк, який використовує архітектурні патерни "MVC" або "MVVM", використовується для розробки серверних застосунків на JavaScript або TypeScript. NestJS базується на Node.js та Express.js, але надає більш структурований та модульний підхід до розробки.
3. Socket.IO - Ця бібліотека JavaScript дозволяє реалізувати багатонаправлену комунікацію в режимі реального часу між

клієнтом і сервером через веб-сокети. Вона забезпечує підтримку двостороннього обміну даними, подій та каналів комунікації.

4. Ngrok - Цей сервіс дозволяє створити тимчасову зовнішню адресу (тунель) до локального сервера, розташованого за файрволом або внутрішньою мережею.
5. JWT (JSON Web Token) - це стандарт для створення безпечних токенів автентифікації в форматі JSON.
6. Swagger - це набір інструментів для розробки, документування та використання API. Він надає зручність у створенні, описі та візуалізації RESTful API.
7. Google Cloud Storage - це послуга зберігання об'єктів у хмарі, надана Google. Вона дозволяє завантажувати, зберігати та керувати великими об'ємами даних, такими як зображення, відео, аудіофайли тощо.

Клієнтська частина має наступні компоненти:

1. Фреймворк
    a. React Native - фреймворк для розробки мобільних застосунків, який дозволяє використовувати JavaScript та React для створення кросплатформових застосунків, що працюють на iOS та Android.
2. Expo - Набір інструментів та сервісів, що спрощує розробку мобільних застосунків з використанням React Native.
3. Axios - це бібліотека JavaScript для виконання HTTP-запитів з клієнтського або серверного коду. Вона дозволяє легко взаємодіяти з веб-серверами та отримувати або надсилати дані через протокол HTTP.
4. React Query - це бібліотека для керування станом та кешування даних в React застосунках. Вона надає простий та потужний спосіб взаємодії з сервером, отримання даних та автоматичного оновлення інтерфейсу користувача.

5. Yup - це бібліотека JavaScript для валідації даних. Вона дозволяє визначити правила та умови для перевірки введених користувачем даних.

РОЗДІЛ 2. СЕРВЕРНА ЧАСТИНА

2.1 Вигляд бази даних

Перед вступом до виконання завдання важливим є визначення сутностей, що вимагають зберігання, та формату моделей.

Ключові сутності, які будуть використані в застосунку, включають "Користувач" та "Запит".

Відповідно до сутності "Користувач", визначено наступні атрибути (атрибути з * будуть додатково пояснені в подальшому тексті роботи):

- Електронна пошта (повинна бути унікальною)
- Ім'я та прізвище
- Основна фотографія
- Роль (обирається з п'яти варіантів: Адміністратор, Волонтер, Донор, Військовий, Біженець)
- Номер телефону
- Репутація (або рейтинг)
- Статус доступу*
- Статус верифікації*

З урахуванням сутності "Запит", визначено такі атрибути (атрибути з * будуть додатково пояснені в подальшому тексті роботи):

- Назва
- Опис
- Категорія*
- Тип (обирається з чотирьох варіантів: Фінансовий, Збір, Матеріальний, Речі)
- Статус (обирається з чотирьох варіантів: Відкритий, В процесі, Завершений, Закритий)
- Кількість переглядів
- Загальна сума зібраних коштів

- Фінансова ціль запиту
- Статус модерації запиту*
- Посилання на monoБанк

Додатково, для покращення функціональності застосунку, було розроблено наступні моделі:

- Категорія запиту
- Локація
- Перегляд запиту
- Теги
- Відгуки (рецензії)
- Чат
- Повідомлення
- Додаток (зображення)
- Коментар до запиту
- Інтереси людини
- Сповіщення
- А також моделі, які не були включені в початковий запуск: Пожертва, Віха

Схему готової бази даних можна побачити на рисунку 1.

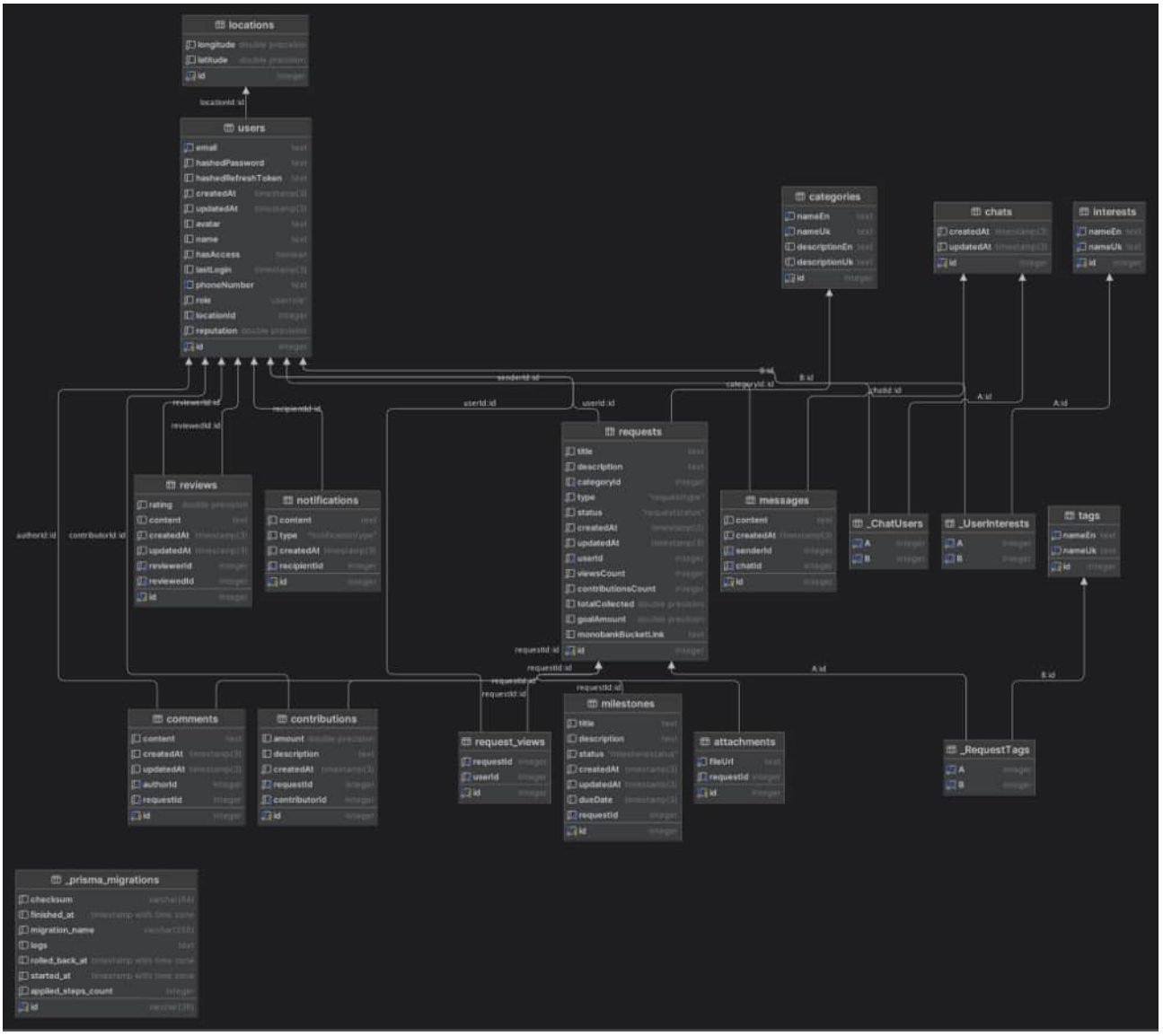

Рис. 1. Схема бази даних проєкту “Воло”

2.2 Ініціалізація проєкту

Як було вказано у попередніх розділах, для серверної частини було обрано фреймворк NestJS. Ініціалізацію базового проєкту можна реалізувати за допомогою команди 'nest new volo_server', після чого починається процес ініціалізації бази даних.

2.2.1 Prisma

Використовуючи Prisma для взаємодії з PostgresSQL, ініціалізація здійснюється за допомогою команди 'npx prisma'. Після цього Prisma генерує нові конфігураційні файли, які можна використовувати для подальшої роботи. Приклад опису моделі за допомогою Prisma приведений на Рисунку 2.

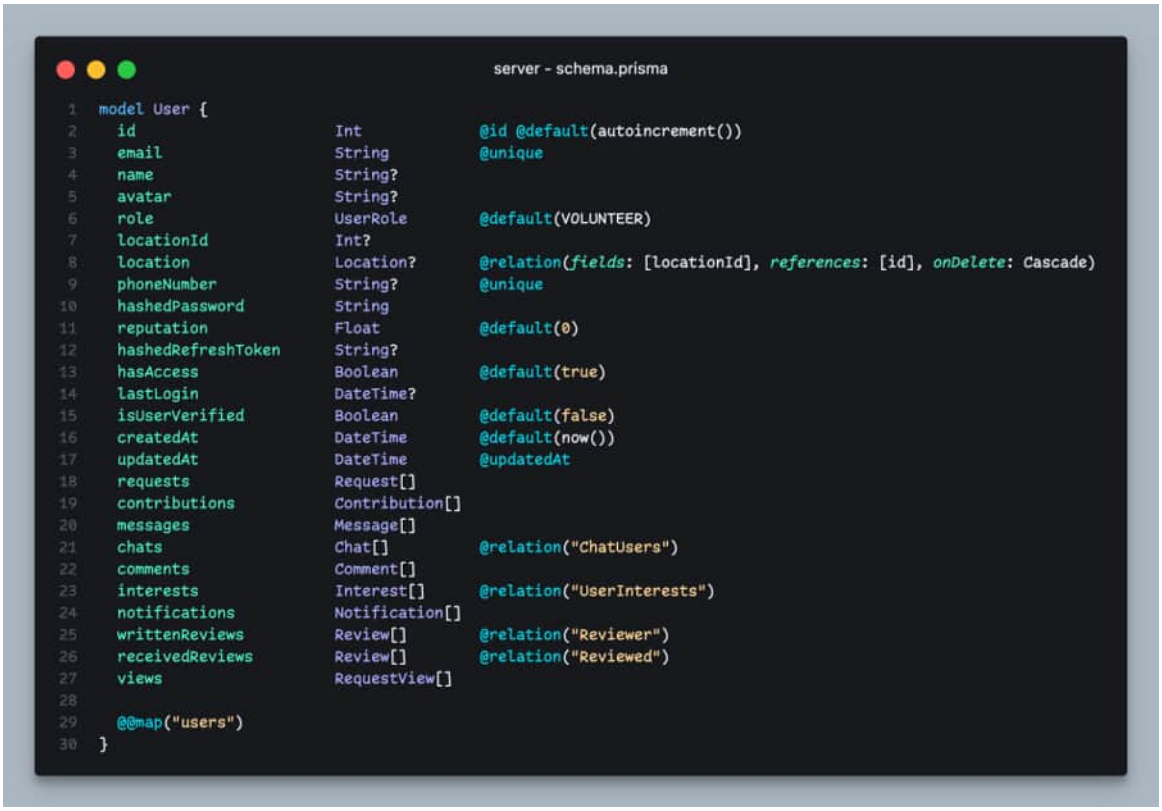

Рис. 2. Вигляд моделі Prisma.

2.2.2 NestJS та його специфічна структура

У контексті побудови застосунку на базі NestJS для формування ресурсів (груп запитів) використовується специфічний підхід. Розміщення ресурсів відбувається в папці "src", де створюються окремі папки у множинному форматі відповідно до типу ресурсу. Наприклад, для потреб автентифікації було сформовано папку auth.

Кожен ресурс, представлений у вигляді окремої папки, повинен включати:

- Контролер - Цей компонент відповідає за обробку HTTP-запитів та взаємодію з клієнтом. Він включає набір методів, які обслуговують різноманітні маршрути та HTTP-запити (GET, POST, PUT, DELETE тощо), та виконують відповідні дії. Декоратори у NestJS дозволяють визначити контролери, асоційовані з конкретними маршрутами або шляхами.
- Модуль - Цей елемент представляє основну структурну одиницю організації коду в NestJS. Модуль групує пов'язані контролери,

провайдери та інші компоненти. Він визначає контекст застосунку, структуру взаємодії між компонентами, а також може включати імпортовані зовнішні модулі, що надають додаткову функціональність, наприклад, роботу з базами даних або веб-сокетами.

- Сервіс - Цей компонент реалізує бізнес-логіку застосунку та специфічну функціональність. Він включає методи для роботи з даними, обробки логіки, взаємодії з базою даних або зовнішніми службами. За допомогою механізму ін’єкції залежностей (Dependency Injection) в NestJS, сервіси інтегруються в контролери або інші сервіси, що дозволяє декомпозувати логіку застосунку на відокремлені та перевикористовувані компоненти.
- DTO (Data Transfer Object)

Крім того, можливе створення додаткових компонентів відповідно до потреб конкретного проекту.

2.2.3 Безпека

Враховуючи строгі вимоги безпеки застосунку та намір забезпечити високий рівень довіри користувачів, було розроблено специфічний процес реєстрації. Під час реєстрації або входу в застосунок, на сервер відправляються дані геолокації - широта та довгота. На сервері зберігається статичний список країн, з яких не дозволено здійснювати вхід. На даний момент до таких країн входять росія, білорусь, Китай та Іран. Крім того, на сервері зберігається набір полігонів кордонів країн, які були отримані через API від Google.

Широта та довгота формують точку, яка може знаходитись всередині одного з цих полігонів. Якщо дані геолокації користувача відповідають одній з країн із забороненого списку, тоді в базі даних при створенні профілю користувача поле "Має доступ" буде встановлено в значення "Неправда" (False). В результаті, коли користувач намагається виконати вхід із заблокованого пристрою або з

електронною поштою, він буде перенаправлений на сторінку з повідомленням про неможливість доступу до застосунку.

Важливим аспектом роботи досліджуваного застосунку є створення запитів. В умовах вільного доступу до цього процесу, виникає питання про максимальне запобігання неправдивих запитів без ускладнення інтерфейсу для користувачів, які мають намір публікувати реальні запити.

Враховуючи цю потребу, було вирішено впровадити наступну логіку: коли користувач реєструється, він спочатку не верифікований, що відображено на сторінці профілю. Цей статус не заважає користувачу створювати запити, але такі запити перед публікацією перевіряються адміністрацією і менеджментом застосунку.

Адміністратори або менеджери мають можливість зв'язатися з користувачем за допомогою контактних даних, вказаних при реєстрації, з метою уточнення даних та інформації по запиту для підтвердження його автентичності.

Згідно з правилами застосунку, користувач має право на публікацію запиту лише за умови, що вказано його реальне ім'я та прізвище, а основна фотографія профілю відповідає зображенню користувача. У випадку відмови користувача надавати публічні дані, його акаунт може бути призупинено, а всі його запити - приховано до моменту їх перегляду.

Після одобрення запиту і оновлення даних користувача він отримує верифікаційний знак на сторінці профілю та має можливість створювати та відразу публікувати подальші запити.

Ці базові правила допомагають мінімізувати кількість потенційно неправдивих запитів, але вони не вичерпують всіх аспектів роботи застосунку.

Верифікований користувач має право створити запит без передбачуваної модерації. Однак, якщо адміністратори та менеджери виявлять необхідність уточнення інформації, вони можуть зв'язатися з користувачем і запитати додаткові дані щодо створеної публікації.

Також функціонує система відгуків. Якщо користувач отримав більше десяти відгуків з однією зіркою, адміністрація та менеджмент застосунку отримують

про це сповіщення і проводять аналіз цих відгуків. У випадку виявлення шахрайства або інших недоліків на основі відгуків, адміністрація та менеджмент застосунку мають право призупинити дію акаунту користувача та приховати всі його публікації до моменту розгляду ситуації.

Усі вищезгадані пункти корегуються у базі даних за допомогою двох полів:

Користувач (усі поля це булеві значення):

- Чи має доступ
- Чи верифікований
- Прихований

Запит (усі поля це булеві значення):

- Чи верифікований
- Прихований

2.3 Автентифікація

2.3.1 Контролер

Як було вже описано вище, контролер у NestJS приймає HTTP запити. Для потреб автентифікації були розроблені наступні методи запитів:

- POST методи включають: реєстрацію, вхід, оновлення токену (refresh token), а також вихід з системи.
- GET методи: в даному випадку представлений метод для отримання власних даних користувача.

У контексті NestJS декоратори параметрів використовуються для розширення функціональності методів. Це дозволяє визначати спеціалізовані декоратори, що виконують певну логіку передачі значень параметрів до методу. Всі строки, що починаються з “@”, розглядаються як декоратори і виконуються перед початком обробки запиту. Приклад одного з обробників:

```
@Public()
@Post('local/signup/')
@HttpCode(HttpStatus.CREATED)
@ApiCreatedResponse({ type: AuthEntity })
```

```
signUpLocal(@Body() dto: SignUpDto): Promise<Tokens> {
  return this.authService.signUpLocal(dto);
}
```

Всі рядки, що ініціюються з символу “@”, класифікуються як декоратори та повинні бути виконані перед початком обробки власне запиту. Подібну методологію можна спостерігати у використанні інших фреймворків, де проміжні етапи (middlewares) виконуються перед запуском запиту.

Враховуючи, що більшість запитів можуть бути виконані виключно авторизованими користувачами системи, оптимальним рішенням було б створення модулю, який би автоматично захищав всі запити, за винятком тих, що позначені декоратором "@Public()".

```
import { SetMetadata } from '@nestjs/common';
export const Public = () => SetMetadata('isPublic', true);
```

Завдяки декораторам, усі запити, за винятком тих, які відмічені як “@Public()”, автоматично захищаються від доступу. Декоратор "@Public()" використовує функцію "SetMetadata", яка дозволяє встановити значення метаданих для певного елемента (наприклад, класу, методу або параметра).

Для забезпечення захисту використовується гвард (guard) на токен доступу (access token). Гварди в NestJS служать для захисту маршрутів або ресурсів від несанкціонованого доступу. В даному випадку, AccessTokenGuard перевіряє валідність токену доступу та автентифікує користувача, якщо ресурс не відмічено як публічний.

```
import { ExecutionContext, Injectable } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { AuthGuard } from '@nestjs/passport';
@Injectable()
export class AccessTokenGuard extends AuthGuard('jwt') {
  constructor(private reflector: Reflector) {
    super();
  }

  canActivate(context: ExecutionContext) {
```

```
    const isPublic = this.reflector.getAllAndOverride<boolean>('isPublic', [
      context.getHandler(),
      context.getClass(),
    ]);

    if (isPublic) {
      return true;
    }

    return super.canActivate(context);
  }
}
```

Було розроблено власний гвард, AccessTokenGuard, що реалізує функціональність автентифікації за допомогою токенів доступу (access tokens) з використанням бібліотеки @nestjs/passport. В рамках цього гварда використовується reflector з @nestjs/core для отримання метаданих ресурсу. У методі canActivate перевіряється значення isPublic. Якщо ресурс відмічений як публічний, повертається true, що дозволяє доступ до ресурсу без додаткової перевірки. В протилежному випадку, викликається метод canActivate з AuthGuard('jwt'), який перевіряє валідність JWT-токену та автентифікує користувача на його основі. Використання таких гвардів дозволяє з легкістю отримувати дані про користувача, не передаючи додаткових полів у запит.

```
import { ExecutionContext, createParamDecorator } from '@nestjs/common';
export const GetCurrentUserId = createParamDecorator(
  (_: undefined, ctx: ExecutionContext): number => {
    const request = ctx.switchToHttp().getRequest();

    return request.user['sub'];
  },
);
```

У цьому кодовому фрагменті створюється специфічний декоратор параметрів, названий **GetCurrentUserId**. Цей декоратор дозволяє втілити процедуру отримання ідентифікатора поточного користувача.

GetCurrentUserId використовує createParamDecorator з модуля @nestjs/common для креації декоратора параметрів. Ця функція приймає в

якості аргумента обробник параметрів, що викликається під час виконання методу, який застосовує декоратор.

Обробник параметрів, у даному випадку, приймає два аргументи: _: undefined та ctx: ExecutionContext. Перший аргумент немає використання (позначається як _, оскільки він є несуттєвим), в той час як другий аргумент, ctx, містить контекст виконання, надаючи доступ до поточного HTTP-запиту та інформації щодо виконання.

Усередині обробника параметрів, використовується ctx.switchToHttp().getRequest() для отримання об'єкту запиту. З цього об'єкту далі вилучається властивість user['sub'], яка відображає ідентифікатор користувача, збереженого в об'єкті запиту.

Таким чином, при застосуванні декоратора @GetCurrentUserId() до параметру методу, NestJS викличе визначений обробник параметрів, вилучить поточний об'єкт запиту та поверне ідентифікатор користувача, який згодом буде переданий у відповідний параметр методу.

2.3.2 Сервіс

Усі сервіси в NestJS мають підтримувати механізм ін’єкції залежностей (Dependency Injection), що вказується наявністю декоратора "@Injectable()".

Декоратор @Injectable() у NestJS використовується для анотації класів, що функціонують як сервіси та підлягають механізму ін’єкції залежностей. Коли до класу застосовується @Injectable(), NestJS ідентифікує цей клас як сервіс та може автономно забезпечити створення екземпляра даного класу та ін’єкції його залежностей в інші компоненти.

Це створює можливість використання цього сервісу в інших класах (наприклад, контролери чи інші сервіси) шляхом вказівки цього класу як залежності в конструкторі. Після цього NestJS автоматично ініціює створення екземпляра даного сервісу та передає його в конструктор. Декоратор @Injectable() дозволяє також застосування інших декораторів та механізмів

NestJS, як-от контейнер інжектування залежностей (Dependency Injection Container), модулі тощо.

Підсумовуючи, @Injectable() відіграє вирішальну роль в реалізації принципу інверсії керування (Inversion of Control) та ін'єкції залежностей в NestJS, що сприяє структурності, модульності та повторному використанню коду. Розглянемо приклад сервісу реєстрації:

```
async signUpLocal(dto: SignUpDto): Promise<Tokens> {
```

Цей метод використовується для реєстрації нового користувача використовуючи локальні облікові дані (електронна адреса, пароль тощо). Процес реєстрації включає наступні кроки:

```
const country = crg.get_country(dto.latitude, dto.longitude).name;
```

Визначення назви країни, заснованої на координатах dto.latitude та dto.longitude. Ця інформація отримується за допомогою функції country-reverse-geocoding.

```
const hashedPassword = await this.hashData(dto.password);
```

Шифрування пароля за допомогою алгоритму bcrypt перед збереженням його в базі даних.

```
if (RED_FLAG_COUNTRIES.includes(country)) {
      await this.prisma.user.create({
        data: {
          email: dto.email,
          hashedPassword,
          lastLogin: new Date(),
          hasAccess: false,
          locationId: location.id,
        },
      });

      throw new ForbiddenException('Слава Україні!', {
        description: country,
      });
    }
```

Перевірка, чи належить визначена країна до списку RED_FLAG_COUNTRIES, який містить перелік країн, доступ з яких обмежено. У випадку позитивного результату перевірки, в базі даних створюється

користувач з відміткою hasAccess = false, після чого генерується виключення ForbiddenException.

```
const userExists = await this.prisma.user.findUnique({
      where: {
        email: dto.email,
      },
    });

    if (userExists) {
      throw new ForbiddenException('Користувач вже зареєстрований');
    }
```

Перевірка, чи користувач з вказаною електронною адресою вже зареєстрований в базі даних. У випадку позитивного результату перевірки, генерується виключення ForbiddenException.

```
const newUser = await this.prisma.user.create({
      data: {
        email: dto.email,
        hashedPassword,
        lastLogin: new Date(),
        locationId: location.id,
        role: dto.role,
      },
    });

    const tokens = await this.getTokens(newUser.id, newUser.email);
    await this.updateRefreshToken(newUser.id, tokens.refreshToken);

    return tokens;
```

Створення нового користувача за допомогою this.prisma.user.create, генерація токенів за допомогою this.getTokens, оновлення оновлюваного токену за допомогою this.updateRefreshToken і повернення отриманих токенів.

```
async getTokens(userId: number, email: string) {
    const [accessToken, refreshToken] = await Promise.all([
      this.jwtService.signAsync(
        {
          sub: userId,
          email,
        },
        {
          secret: process.env.JWT_SECRET,
```

```
        expiresIn: 60 * 60 * 24 * 7,
      },
    ),
    this.jwtService.signAsync(
      {
        sub: userId,
        email,
      },
      {
        secret: process.env.JWT_RT_SECRET,
        expiresIn: 60 * 60 * 24 * 7,
      },
    ),
  ]);

  return { accessToken, refreshToken };
}
```

Метод getTokens використовує jwtService для генерації JWT-токенів. Цей метод приймає ідентифікатор користувача і електронну адресу, генерує access token та refresh token за допомогою this.jwtService.signAsync і повертає об'єкт з цими токенами.

```
async updateRefreshToken(userId: number, refreshToken: string) {
    const hash = await this.hashData(refreshToken);

    await this.prisma.user.update({
      where: {
        id: userId,
      },
      data: {
        hashedRefreshToken: hash,
      },
    });
  }
```

Метод updateRefreshToken отримує ідентифікатор користувача та оновлюваний токен, шифрує оновлюваний токен за допомогою this.hashData і оновлює поле hashedRefreshToken користувача в базі даних за допомогою this.prisma.user.update.

Загалом, цей код реалізує процедуру реєстрації користувача, генерації JWT-токенів та оновлення оновлюваного токену в середовищі NestJS з

використанням залежностей prisma та jwtService. Таким чином, досягається забезпечення конфіденційності пароля, а поверненні токени без секретного слова не можуть бути розшифровані.

2.3.3 JWT (JSON Web Token)

JSON Web Token (JWT) є стандартом відкритого коду для безпечної передачі даних між двома сторонами як JSON-об'єкт. Ця інформація може бути довірена та перевірена, оскільки JWT підписується за допомогою секретного ключа або пари ключів (публічний / приватний). JWT часто використовується для аутентифікації та авторизації користувачів в розподілених системах та мікросервісних архітектурах.

Кожен JWT містить три основні частини: заголовок, дані та підпис.

- Заголовок (Header) зазвичай містить два поля: тип токена, який є JWT, та алгоритм хешування, наприклад HMAC SHA256 або RSA.
- Дані (Payload) містить інформацію про суб'єкт та інші додаткові дані. Інформація можу бути зарезервована, загальна або приватна.
- Підпис (Signature) створюється шляхом кодування заголовку, даних та секрету через алгоритм, зазначений у заголовку.

Токени JWT використовуються для визначення того, ким ви є (аутентифікація) та що ви можете зробити (авторизація). Вони можуть зберігатися у веб-браузері користувача та автоматично включатися в кожний HTTP-запит до сервера.

Оскільки JWT підписуються, вони надають гарантію того, що вміст токена не був змінений після видачі. Однак, варто зазначити, що хоча вміст JWT може бути перевірений та довірений, це не означає, що він є закодованим.

У цілому, JSON Web Token надає гнучкий механізм для безпечної передачі даних між сторонами у дистрибутивному середовищі. Вони відіграють ключову роль у сучасних системах аутентифікації та авторизації, забезпечуючи ефективність, безпеку та надійність обміну даними.

2.4 Надсилання інформації за запитом

2.4.1 Надсилання одного об’єкту

В рамках архітектури веб-застосунку, що використовує фреймворк Nest.js, діяє специфічний механізм обробки клієнтських запитів. Відповідно до цього механізму, початковий запит, ініційований клієнтом, проходить через кілька послідовних етапів обробки, на виході з яких формується відповідь.

Сформовані відповіді містять дані, що генеруються в ході різноманітних операцій, включаючи пошук, маніпуляцію та створення даних. Кінцевий набір даних, що становить відповідь, базується на вимогах початкового клієнтського запиту, а також на актуальному стані даних, що зберігаються в застосунку.

Серверний аспект програми Nest.js відповідає за обробку вхідних клієнтських запитів, активуючи відповідні методи в рамках своїх служб. Кожен з цих методів реалізує потрібну бізнес-логіку, взаємодіє з головною базою даних через інструмент об'єктно-реляційного відображення (ORM), такий як Prisma, та кінцево формує відповідь.

По завершенні підготовки відповіді, програма Nest.js ініціює процес її відправки клієнту. Це відбувається автоматично в момент завершення методу, відповідального за обробку запиту клієнта. Дані відповіді перетворюються в серіалізований формат, який оптимізований для передачі через мережу (зазвичай JSON), а потім відправляються клієнту.

2.4.2 Курсор в контексті об'єктно-реляційного відображення Prisma

В об'єктно-реляційному відображенні (ORM) Prisma використовується поняття "курсор". Курсор, в цьому контексті, відноситься до механізму, який дозволяє здійснювати послідовну навігацію по наборах даних в базі даних.

Курсори є критично важливими при реалізації стратегій сторінкування, особливо при роботі з великими наборами даних. Вони вказують на конкретне місце в наборі даних, що вже було оброблено, та слугують посиланням для отримання наступних частин даних.

У Prisma, курсори можуть бути використані для ефективного отримання та обробки даних, при чому зазвичай вони використовуються у комбінації з операціями **take** та **skip**, що дозволяють вказати кількість даних для отримання, та пропускати певну кількість записів відповідно.

У загальному випадку, курсор в Prisma представляє собою об'єкт, який містить пари ключ-значення, де ключ відповідає полю моделі, а значення відображає конкретний запис в полі, що вказує на місце в наборі даних, де була зупинена попередня операція зчитування.

На рисунку 4 показано перші 4 записи даних після запису з ідентифікатором 29. У цьому прикладі новий курсор має значення 52:

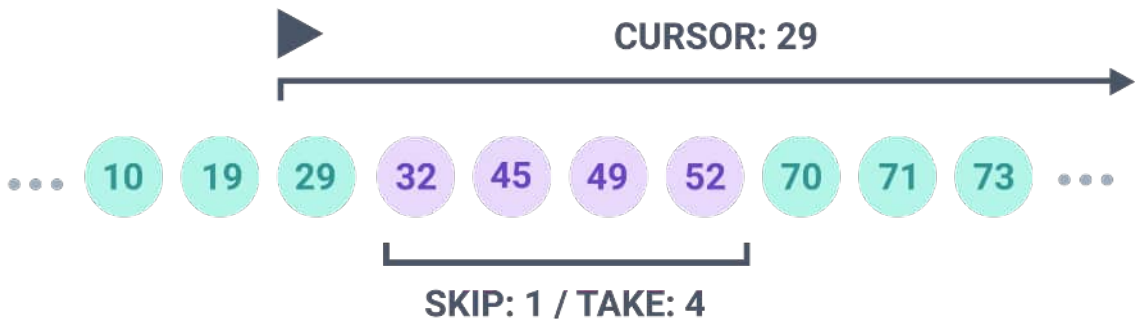

Рис.3 Робота курсору у Prisma

2.4.3 Надсилання обє'кту з курсором

```
async findAll(query: PaginationQuery) {
    const { limit, cursor, status, type, fromDate, category,
tags } =
      transformQuery(query);

    const result = await this.prisma.request.findMany({
      skip: cursor ? 1 : 0,
      take: limit ?? 10,
      cursor: cursor ? { id: cursor } : undefined,
      orderBy: {
        id: 'desc',
      },
```

```
      where: {
        isOnModeration: false,
        createdAt: {
          gte: fromDate,
        },
      });
    const nextCursor = result.length > 0 ?
result[result.length - 1].id : null;
    return {
      data: result,
      next: nextCursor,
    };
  }
```

У контексті зазначеного методу **findAll()**, параметри запиту, які надаються клієнтом, ініційно проходять процедуру перетворення за допомогою функції **transformQuery()**. Дана функція здійснює обробку параметрів з метою забезпечення коректності повернутих значень, які відповідають специфікаціям моделі **Request**.

Викликана в контексті даного методу, функція **findMany()** отримує колекцію запитів, базуючись на перетворених параметрах запиту. У разі наявності курсора, цей останній використовується як вихідна точка для пошуку даних. Параметр **take** застосовується з метою обмеження кількості записів, що повертаються. Додатково, ця функція забезпечує упорядкування результатів за зменшенням **id**, гарантуючи при цьому пріоритетність повернення найновіших записів.

Після отримання запитів, **nextCursor** визначається як **id** останнього запиту у відповідному наборі результатів. Даний курсор надалі може бути використаний клієнтом у рамках наступного запиту для отримання додаткової сторінки даних.

Процес виконання методу **findAll()** завершується поверненням об'єкту, який включає як набір даних (**result**), так і **nextCursor**. Дана реалізація забезпечує те, що клієнт отримує не лише запитувані дані, але й інформацію про те, який

курсор має бути використаний для наступного запиту, тим самим формуючи систему сторінкування на основі курсору.

2.5 Swagger

Swagger є набором інструментів відкритого коду для проектування, будівництва, документування та використання RESTful веб-сервісів. Swagger включає автоматичне документування API та генерацію коду, що сприяє швидкому розробленню та використанню веб-сервісів.

NestJS, в свою чергу, надає вбудовану підтримку для Swagger через пакет @nestjs/swagger. Цей пакет дозволяє автоматично генерувати Swagger документацію для NestJS проектів, розроблених за допомогою декораторів NestJS.

Через використання різних декораторів, такі як @ApiProperty(), @ApiTags(), @ApiResponse(), можна анотувати різні аспекти API (наприклад, ендпойнти, моделі відповідей, параметри запиту тощо) для автоматичної генерації документації Swagger.

Отримана документація Swagger доступна через веб-інтерфейс, який зазвичай доступний за адресою /api на сервері застосунку. Цей веб-інтерфейс не тільки відображає деталі API, але також дозволяє інтерактивно використовувати API безпосередньо з браузера, що полегшує тестування та розробку.

Приклад опису DTO (Data Transfer Object) для того щоб він правильно був відображений у документації:

```
export class CreateUserDto {
  @ApiProperty({ required: true })
  @IsEmail()
  @IsNotEmpty()
  email: string;

  @ApiProperty({ required: false })
  @IsOptional()
  @IsString()
  name?: string;
```

```
  @ApiProperty({ required: false, default: UserRole.VOLUNTEER, enum: UserRole
})
  @IsEnum(UserRole)
  role?: UserRole;
}
```

Після запуску сервера, за посиланням “/api” ми отримуємо сторінку Swagger (Рисунок 4), описані DTO (Рисунок 5) та описані запити (Рисунок 6)

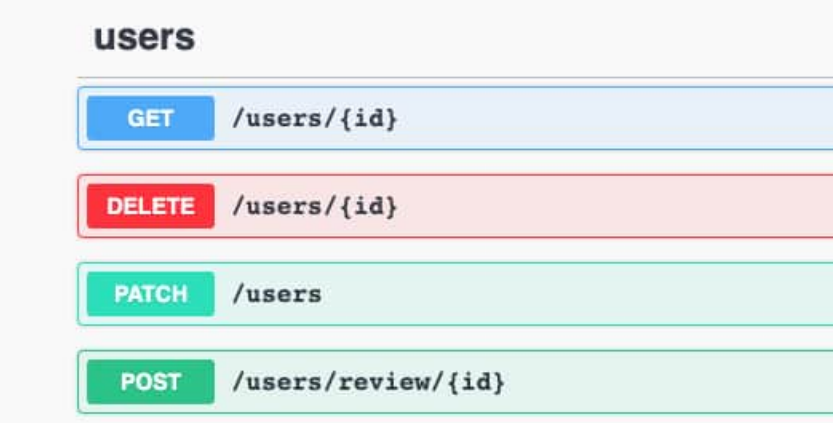

Рис.4 Відображення описаних запитів у Swagger

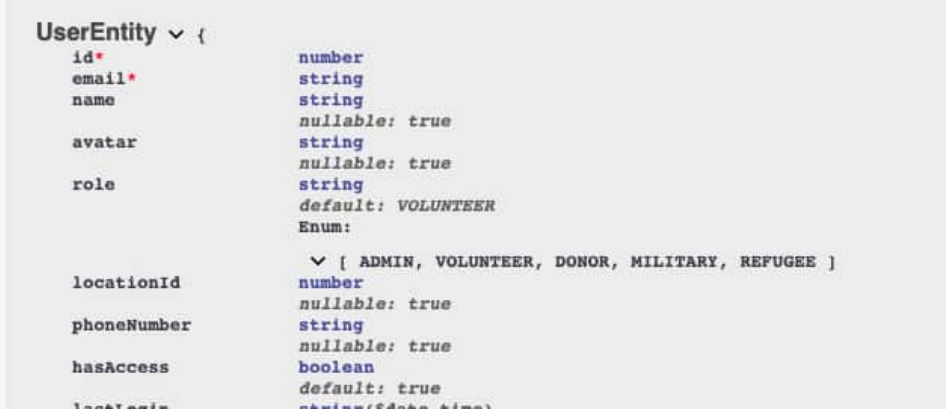

Рис.5 DTO у Swagger

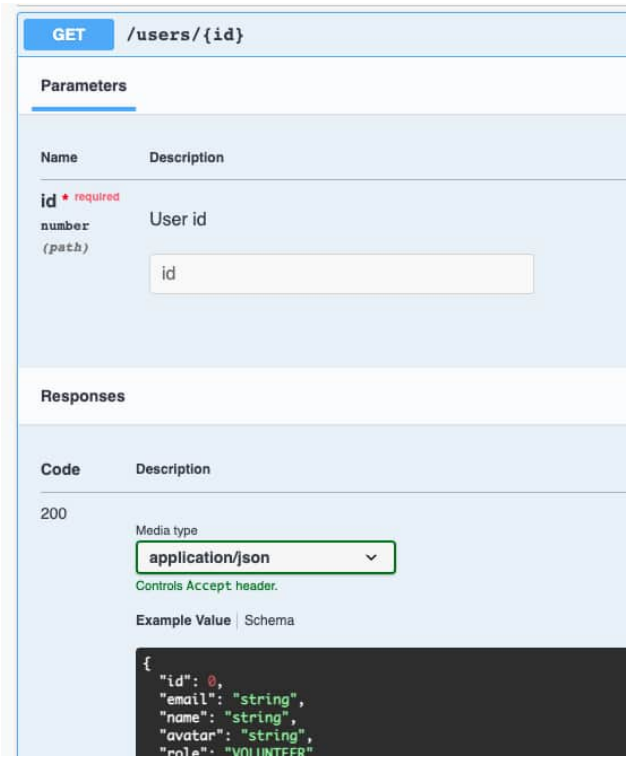

Рис.6 Деталізований вигляд запиту у Swagger

РОЗДІЛ 3. КОРИСТУВАЦЬКА ЧАСТИНА

3.1 Розвиток React: від класових до функціональних компонентів

React.js, відомий також як React, є JavaScript бібліотекою для розробки користувацьких інтерфейсів, створеною Facebook. Хоча у React від самого початку була можливість створювати компоненти як функціональні, так і класові, раніше переважно використовувалися класові компоненти.

Класові компоненти давали можливість використовувати ряд методів життєвого циклу, таких як componentDidMount, componentDidUpdate та componentWillUnmount, для управління різними етапами життєвого циклу компонентів. Класові компоненти також дозволяли використовувати стан (state) в компонентах, що було важливим для побудови інтерактивних інтерфейсів.

Однак, з часом, React почав рухатися в бік функціональних компонентів. Це було здійснено завдяки впровадженню так званих "хуків" (hooks) у версії React 16.8. Хуки - це функції, які дозволяють використовувати стан та інші особливості React без написання класу. Два основні хуки - useState та useEffect - дозволяють використовувати стан та методи життєвого циклу в функціональних компонентах відповідно.

useState дозволяє додати стан до функціонального компонента, а useEffect використовується для виконання певних дій при монтуванні, оновленні та демонтуванні компонента, подібно до методів життєвого циклу в класових компонентах.

3.1.1 Переваги використання хуків у React.js

Хуки, представлені в React 16.8, внесли істотні поліпшення в парадигму розробки React, переважно у три основні сфери: зрозумілість коду, повторне використання логіки та розділення відповідальності між компонентами. Ось більш детальний опис цих переваг:

1. Зрозумілість коду: Однією з ключових переваг хуків є їхня здатність спрощувати код, роблячи його більш зрозумілим та читабельним. В порівнянні з класовими компонентами, які вимагають знання

складних концепцій JavaScript, таких як this, bind, конструктори та систему наслідування, хуки дозволяють писати більш прямолінійний код без необхідності використання згаданих концепцій.

2. Повторне використання логіки: Раніше, при використанні класових компонентів, повторне використання логіки стану чи ефектів було складним і здебільшого обмежувалося паттернами вищого порядку (HOC) та рендер-пропами. Ці паттерни могли призвести до "обгорткового пекла" та складностей в підтримці коду. З введенням хуків, React включив можливість використовувати так званих custom hooks - хуків, які дозволяють розділяти компонентну логіку на перевикористовувані функції.
3. Розділення відповідальності між компонентами: Класові компоненти часто вимушують сплітати код, що базується на методах життєвого циклу, а не на схожій функціональності. Це може ускладнити розуміння того, як код взаємодіє. Хуки дозволяють розділити код, групуючи його за функціональністю, що полегшує розуміння та підтримку коду.

Отже, перехід до використання хуків у React дозволив значно підвищити продуктивність розробки, упростивши написання та читання коду, забезпечивши можливість повторного використання логіки, і впровадивши ефективне розділення відповідальності між компонентами.

3.2 Структура проєкту

- **Кореневий рівень**: Сюди входять всі глобальні конфігураційні файли, такі як package.json, tsconfig.json, metro.config.js, babel.config.js і expo.plist, які необхідні для налаштування роботи проекту. Також тут розміщені README.md (файл документації проекту), LICENSE (ліцензійний договір) та app.json (конфігурація застосунку Expo).

- **assets**: В цій папці знаходяться всі медіа файли, такий як зображення і анімації, які використовуються в застосунку.
- **src**: Головний каталог коду програми. Він поділений на підкаталоги згідно функціональної відповідальності.
    - **App.tsx**: Головний файл застосунку, який включає в себе кореневий компонент.
    - **components**: Каталог, що містить усі реакт-компоненти, які використовуються в застосунку.
    - **constants**: Каталог, що містить константи, які використовуються в застосунку.
    - **contexts**: Каталог, який містить файли контексту React.
    - **features**: Каталог, в якому розташовані всі компоненти які відносяться до конкретних функціоналів, наприклад для роботи з особистою сторінкою користувача, для роботи з запитами.
    - **hooks**: Каталог, що містить власні хуки для роботи з даними, та зв’язком між компонентами.
    - **navigation**: Каталог, що містить налаштування навігації проекту.
    - **screens**: Каталог, що містить всі екрани застосунку.
    - **types**: Каталог, що містить всі типи TypeScript, які використовуються в проекті.
    - **utils**: Каталог, який містить допоміжні функції або утиліти.

Чому дана структура є однією з найкращих практик? Ось декілька аргументів:

- **Модульність**: Проєкт чітко розбитий на логічні модулі та компоненти (такі як components, …, hooks, screens, utils), що сприяє більш ефективному управлінню кодом. Кожний з цих модулів має свою конкретну функцію, що полегшує розробку, тестування та підтримку коду.

- **Масштабованість**: Завдяки модульності, структура проекту легко масштабується. Нові функції або компоненти можуть бути додані відповідно до відповідних модулів без значного впливу на інші частини системи.
- **Узгодженість**: Всі компоненти, типи, хуки, і т.д. знаходяться у відповідних місцях. Ця уніфікація підтримує консистентність коду і сприяє його читабельності.
- **Утримання**: Чітка структура дозволяє легше виконувати обслуговування і оновлення коду. Це особливо важливо в довгостроковій перспективі, коли проект розростається.
- **Розділення відповідальностей (Separation of Concerns)**: Зрозуміла розбивка на рівні модулів відображає принцип розділення відповідальностей, що є ключовим принципом проектування програмного забезпечення.

3.3 Робота з axios

Код, представлений в цьому розділі, показує приклад створення API контексту у React.js за допомогою бібліотеки axios та JSON Web Tokens. Основний аспект цього прикладу - управління доступом до API за допомогою JWT та впровадження механізму відновлення токену.

3.3.1 Axios API клієнт

```
function getApiClient(): AxiosInstance {
  return axios.create({
    baseURL: API_ENDPOINT ?? '',
    paramsSerializer: {
      serialize: (params) => qs.stringify(params, { arrayFormat: 'comma' }),
    },
  })
}

export function setAuthHeaders(token?: string | null) {
  if (token) {
    apiClient.defaults.headers.common.Authorization = `Bearer ${token}`
  }
}
```

```
export const apiClient = getApiClient()
```

apiClient є інстанцією axios, що створюється за допомогою функції *getApiClient()*. Вона використовує константу *API_ENDPOINT* для встановлення базового URL для API-запитів. Функція *setAuthHeaders(token)* встановлює заголовки авторизації для всіх наступних HTTP-запитів, що здійснюється за допомогою *apiClient*.

3.3.2 Конфігурація перехоплювачів (interceptors)

```
const refreshAccessToken = (
  client: AxiosInstance,
  refresh: string,
  request: AxiosRequestConfig
) =>
  client
    .post<UserAuthInfo>('/auth/refresh/', { refresh })
    .then((response) => {
      const token = response.data.accessToken

      AsyncStorage.setItem(ACCESS_TOKEN_KEY, token)

      if (request.headers) {
        request.headers['Authorization'] = `Bearer ${token}`
      }

      client(request)
    })
    .catch((err) => {
      console.warn(err)
    })

export const configureInterceptors = (
  client: AxiosInstance,
  clearStorageValue: () => void
): void => {
  client.interceptors.response.use(
    (response) => response,
    async (error: AxiosError<{ statusCode?: string }>) => {
      const { response: errorResponse, config: originalRequest } = error

      const isAuthorizationError = errorResponse?.status === 401

      if (!isAuthorizationError || !originalRequest) {
```

```
        return Promise.reject(error)
      }
      if (originalRequest?.url === API_ENDPOINT + '/auth/refresh/') {
        clearStorageValue()

        return Promise.reject(error)
      }
      if (errorResponse?.data?.statusCode === 'token_not_valid') {
        const refreshToken = await AsyncStorage.getItem(REFRESH_TOKEN_KEY)

        if (refreshToken && isValidToken(refreshToken)) {
          return refreshAccessToken(client, refreshToken, originalRequest)
        } else {
          clearStorageValue()
        }
      }

      return Promise.reject(error)
    }
  )
}
```

Перехоплювачі дозволяють виконувати певні операції перед тим, як запит відправлено або коли відповідь отримана. *configureInterceptors* використовується для встановлення перехоплювачів відповіді, що слідкують за кодом статусу 401 (неавторизовано). Якщо код статусу 401 отримано, код спробує відновити токен за допомогою endpoint */auth/refresh/*. Якщо відновлення токену не вдалося, здійснюється вихід з системи.

3.3.3 Контекст зберігання AsyncStorage

```
export const useAsyncStorage = (key: string, defaultValue?: string) => {
  const [value, setValue] = useState<undefined | null | string>()

  const updateValue = (newValue: string) => {
    AsyncStorage.setItem(key, newValue)
    setValue(newValue)
  }

  const clearValue = () => {
    AsyncStorage.removeItem(key)
    setValue(null)
  }

  useEffect(() => {
```

```
    const getStorageValue = async () => {
      const storageValue = await AsyncStorage.getItem(key)

      setValue(
        storageValue === null && defaultValue ? defaultValue : storageValue
      )
    }

    getStorageValue()
  }, [defaultValue, key])

  return { value, setValue: updateValue, removeValue: clearValue }
}
```

```
export const useAsyncStorageContext = (): AsyncStorageContextType =>
  useContext(AsyncStorageContext)
```

```
export type AsyncStorageContextType = {
  accessToken: undefined | null | string
  setAccessToken: (userToken: string) => void
  removeAccessToken: () => void
}

const defaultValues: AsyncStorageContextType = {
  accessToken: undefined,
  setAccessToken: () => undefined,
  removeAccessToken: () => undefined,
}

export const AsyncStorageContext =
  createContext<AsyncStorageContextType>(defaultValues)

export const AsyncStorageProvider = ({ children }: ReactChildren) => {
  const [token, setToken] = useState<undefined | null | string>()

  const {
    value: accessToken,
    setValue: setAccessToken,
    removeValue: removeAccessToken,
  } = useAsyncStorage(ACCESS_TOKEN_KEY)

  useEffect(() => {
    setToken(accessToken)
  }, [accessToken])
```

```
  return (
    <AsyncStorageContext.Provider
      value={{ accessToken: token, setAccessToken, removeAccessToken }}
    >
      {children}
    </AsyncStorageContext.Provider>
  )
}
```

AsyncStorageContext[13] використовується для зберігання *accessToken*. *useAsyncStorage* - це хук, який використовується для читання, запису та видалення значень із *AsyncStorage*. *AsyncStorageProvider* використовується для зберігання *accessToken* і надає методи *setAccessToken* та *removeAccessToken*.

3.3.4 API контекст

```
export const ApiContext: Context<{
  client: AxiosInstance
}> = createContext({
  client: apiClient,
})

export const ApiProvider = ({ children }: ReactChildren): JSX.Element => {
  const { accessToken, removeAccessToken } = useAsyncStorageContext()

  useEffect(() => {
    configureInterceptors(apiClient, removeAccessToken)
  }, [removeAccessToken])

  useEffect(() => {
    setAuthHeaders(accessToken)
  }, [accessToken])

  return (
    <ApiContext.Provider
      value={{
        client: apiClient,
      }}
    >
      {children}
    </ApiContext.Provider>
  )
}
```

ApiContext використовується для надання інстанції *apiClient* для всіх компонентів застосунку. *ApiProvider* використовує *ApiContext* для надання

значень контексту. Він також конфігурує перехоплювачі та встановлює заголовки авторизації при зміні *accessToken*.

Імплементація, представлена в даному розділі, показує сучасний та ефективний спосіб управління аутентифікацією та авторизацією в React.js застосунках. Її ключовими особливостями є:

1. Централізоване управління API-запитами: Використання контексту для надання інстанції apiClient дає можливість управляти всіма API-запитами в одному місці. Це полегшує відслідковування, відлагодження і модифікацію поведінки API-запитів.
2. Інтеграція з JWT: Використання JSON Web Tokens (JWT) як основного механізму авторизації, що дає нам безпечний і стандартизований спосіб обміну інформацією між сервером і клієнтом.
3. Механізм відновлення токену: Вбудований механізм відновлення токену дозволяє автоматично відновлювати сесію користувача, коли токен доступу закінчує свій термін дії, без необхідності здійснювати повторний вхід в систему.
4. Постійне зберігання токену: Застосування AsyncStorage дозволяє зберігати токен доступу між сеансами, що підвищує зручність використання застосунку.

3.4 Утримання користувача (активна сесія)

Після успішного входу в систему користувач отримує JWT, який зберігається в AsyncStorage, персистентному механізмі зберігання даних в React Native. Цей токен містить інформацію про користувача та має часовий термін дії. Він використовується для авторизації користувача при подальших запитах до API. Токен також можна використовувати для відновлення сеансу користувача при повторному запуску застосунку.

У разі закінчення часу дії токену, запит до API поверне помилку 401 (Unauthorized). При цьому, замість того, щоб вимагати від користувача повторного входу в систему, запускається процедура відновлення токену. Це досягається за допомогою axios-інтерцепторів, які виявляють помилку 401 та автоматично відсилають запит на відновлення токену. Після отримання нового токену, він зберігається в AsyncStorage, а запит до API повторюється автоматично з новим токеном.

3.5 Отримання даних з серверу

3.5.1 React Query

Це бібліотека[10] для управління станом та кешування даних в React застосунках. Вона надає прості та потужні засоби для взаємодії з сервером, отримання даних та автоматичного оновлення інтерфейсу користувача.

Основні функції React Query включають:

1. Запити (Queries): React Query дозволяє виконувати запити до сервера для отримання даних. Вона автоматично кешує дані та надає можливість оновлювати їх за потребою.
2. Мутації (Mutations): Бібліотека дозволяє виконувати мутації - запити для зміни даних на сервері. Вона автоматично керує відправкою запитів та оновленням кешу.
3. Автоматичне кешування (Automatic Caching): React Query кешує дані, отримані від сервера, та оновлює їх автоматично відповідно до конфігурації кешування. Це дозволяє зберігати актуальні дані та поліпшує продуктивність застосунку.
4. Пагінація (Pagination): React Query надає можливість роботи з пагінацією даних, що дозволяє легко отримувати та керувати великими наборами даних.
5. Оптимістичне оновлення (Optimistic Updates): Бібліотека підтримує оптимістичне оновлення інтерфейсу користувача, що дозволяє

швидко оновлювати інтерфейс, навіть до отримання підтвердження від сервера.

3.5.2 Структура отримання даних

```
export type UseRequestProps = {
  id: number | undefined
}

export const useRequest = ({
  id,
}: UseRequestProps): UseQueryResult<Request, Error> => {
  const client = useApiClient()

  return useQuery(
    REACT_QUERY_KEYS.REQUEST.map((key) =>
      key === ':id' && id ? id.toString() : key
    ),
    () =>
      client
        .get(`/requests/${id}/`)
        .then(unwrapResponse)
        .catch(unwrapErrorResponse),
    { enabled: !!id }
  )
}

export const REACT_QUERY_KEYS = {
    REQUEST: ['request', ':id'],
}
```

useQuery, який використовується для створення запитів на читання. Цей хук надає можливість забезпечити асинхронне отримання даних з API.

Хук **useRequest**, представлений в коді, це приклад використання **useQuery** для отримання даних про конкретний запит на основі його ідентифікатора (**id**). Цей ідентифікатор передається як аргумент у **UseRequestProps**.

Першим кроком у хуку **useRequest** є отримання екземпляра API клієнта за допомогою хука **useApiClient()**. Цей API клієнт буде використовуватися для відправки запиту на отримання даних, який було описано раніше.

Наступним кроком є виклик **useQuery**, що приймає три аргументи:

1. Масив ключів, що ідентифікує цей конкретний запит в контексті всіх інших запитів, що виконуються за допомогою React Query. У даному випадку використовується константа **REACT_QUERY_KEYS.REQUEST**, заміна ключа **':id'** на конкретний ідентифікатор запиту.
2. Функція, яка повертає Promise, виконуючи асинхронний запит на отримання даних за допомогою API клієнта. Відповідь, отримана від сервера, обробляється за допомогою функції **unwrapResponse**, яка витягує фактичні дані з відповіді. У разі помилки, вона буде оброблена за допомогою **unwrapErrorResponse**.
3. Об'єкт опцій, який дозволяє керувати поведінкою хука. В даному випадку використовується опція **enabled**, яка дозволяє умовно активувати або деактивувати виконання запиту, залежно від наявності **id**.

Результатом виконання цього хука є об'єкт **UseQueryResult**, який містить детальну інформацію про стан запиту, включаючи отримані дані, статус запиту, помилки та інші корисні властивості.

3.5.3 “Нескінченне” прокручування (Infinite Scroll)

Нескінечне прокручування - це техніка веб-дизайну, яка автоматично завантажує наступний блок контенту, коли користувач досягає кінця сторінки. Ця технологія дозволяє забезпечити неперервний потік контенту без потреби явної користувацької взаємодії для завантаження більше даних, таких як натискання на кнопку "завантажити більше". Замість цього, новий контент автоматично завантажується і додається до кінця видимої частини сторінки в момент, коли користувач досягає її кінця.

Цей підхід відомий своєю здатністю покращити користувацький досвід на веб-сайтах і мобільних застосунках, оскільки він забезпечує безперервний потік даних без необхідності очікувати завантаження додаткових сторінок. Особливо він ефективний для веб-сайтів і застосунків, які використовують дані, що

орієнтовані на прокручування, такі як соціальні мережі, блоги або сторінки з результатами пошуку.

3.5.4 FlatList як допоміжний компонент для імплементації "нескінченного прокручування"

FlatList - це вбудований компонент в React Native, який використовується для відображення прокручуваних списків елементів. FlatList має декілька особливостей, які роблять його ідеальним для використання в контексті "нескінченного прокручування".

Одна з ключових властивостей FlatList - це ефективність. Він використовує техніку "лінивого завантаження" (lazy loading), де елементи списку провантажуються лише тоді, коли вони стають видимими. Це означає, що великі списки даних можуть бути відображені без впливу на продуктивність.

FlatList також має вбудовану підтримку обробки події "кінець списку досягнуто" (onEndReached). Ця подія викликається, коли користувач доходить до кінця відображуваних даних.

Додатково, FlatList дозволяє розробникам налаштувати індикатор завантаження, який відображається під час завантаження наступної сторінки даних (через властивість ListFooterComponent).

Таким чином, FlatList у React Native не тільки забезпечує оптимальне виконання і гнучкість при відображенні прокручуваних списків даних, але і вбудовує підтримку для паттерну "нескінченного прокручування".

3.5.5 Використання “нескінченного” прокручування

```
const { data, isLoading, fetchNextPage, hasNextPage, isFetchingNextPage } =
    useRequests({})
const handleLoadMore = () => hasNextPage && fetchNextPage()
return (
<FlatList
        data={data?.pages.map((page) => page.data).flat()}
        {/* ... */}
        onEndReached={handleLoadMore}
        ListFooterComponent={isFetchingNextPage ? <Spinner /> : null}
      />
)
```

В цьому коді **useRequests({})** - це використання React Query для взаємодії з API, що витягує список запитів. В результаті, ми отримуємо декілька значень, які допомагають нам керувати нескінченним прокручуванням:

- **data**: Масив сторінок, які були завантажені до цього моменту. Кожна сторінка включає дані, отримані від API.
- **fetchNextPage**: Функція, яку можна викликати, коли користувач доходить до кінця списку, щоб завантажити наступну сторінку даних.
- **hasNextPage**: Прапорець, що вказує, чи є доступні для завантаження наступні сторінки даних.
- **isFetchingNextPage**: Прапорець, що вказує, чи відбувається наразі завантаження наступної сторінки.

Функція **handleLoadMore** визначена так, що вона викликає **fetchNextPage**, якщо є доступні для завантаження наступні сторінки (**hasNextPage** = true). Ця функція передається в компонент **FlatList** як обробник події **onEndReached**, що викликається, коли користувач доходить до кінця списку.

У **ListFooterComponent** встановлено умовний вираз, який відображає компонент **Spinner (компонент прокрутки)**, коли відбувається завантаження наступної сторінки (**isFetchingNextPage** = true), а в іншому випадку відображається null (тобто нічого).

РОЗДІЛ 4. ДЕМОНАСТРАЦІЯ ЗАСТОСУНКУ

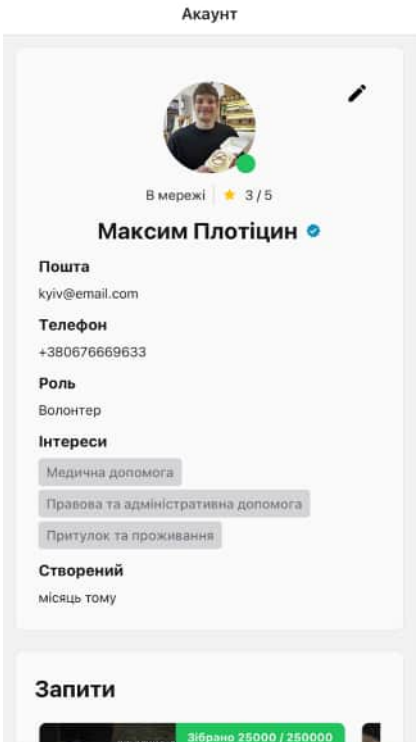

рис. 7 Сторінка користувача

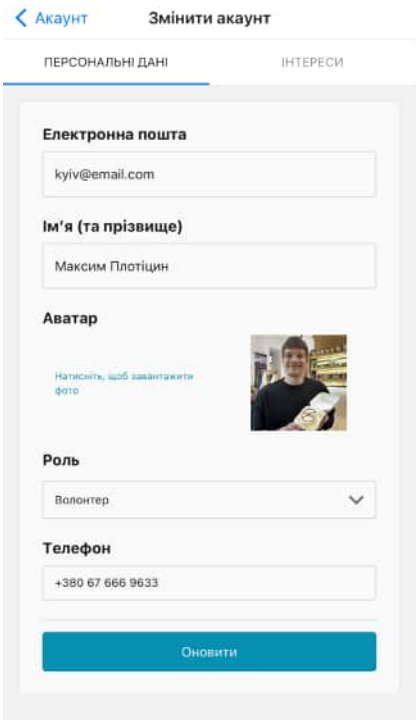

рис. 8 Редагування інформації про користувача

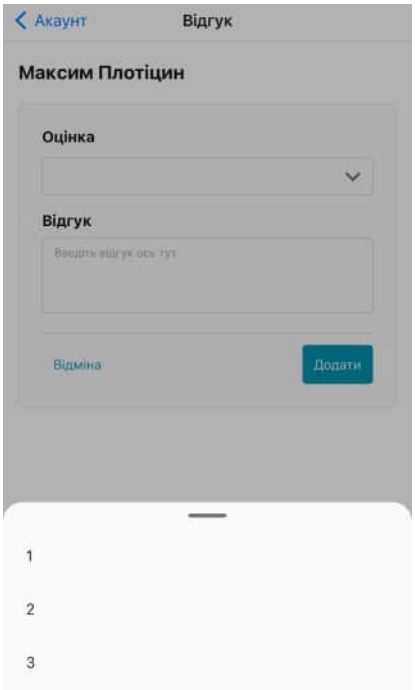

рис. 9 Відгук про користувача

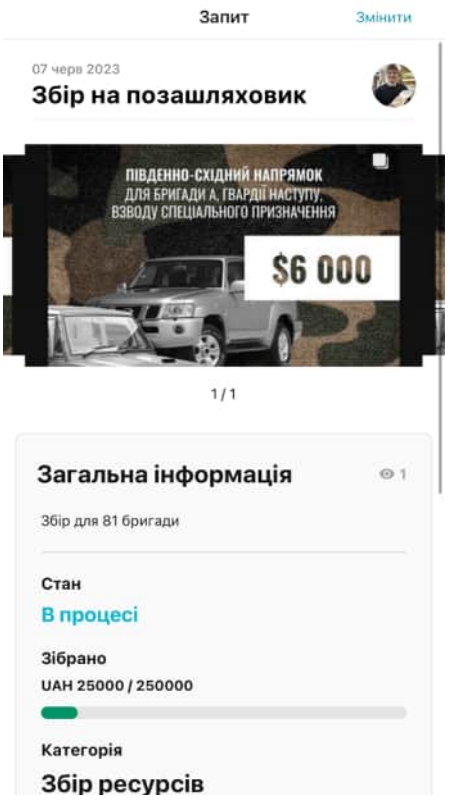

рис. 10 Вигляд створеного запиту

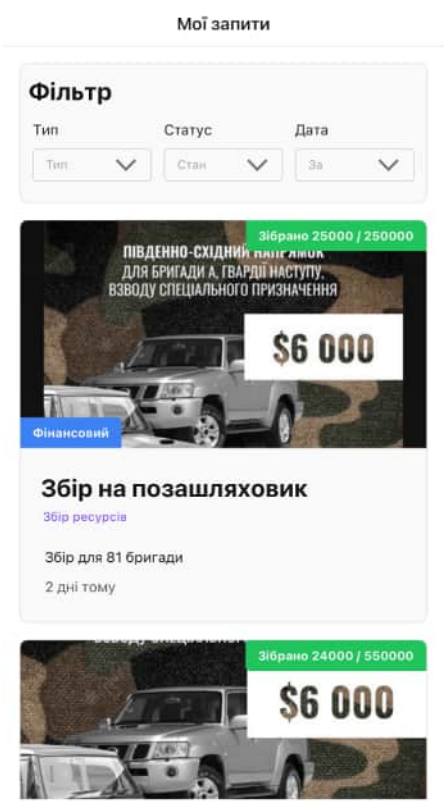

рис. 11 Фільтрація запитів

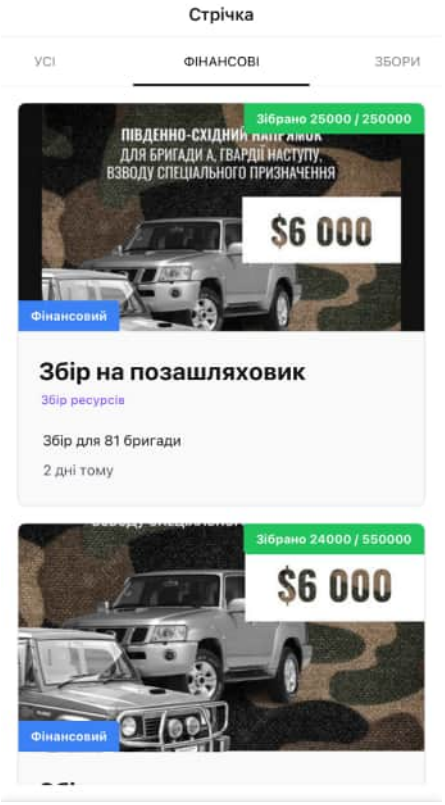

рис. 12 головна сторінка застосунку

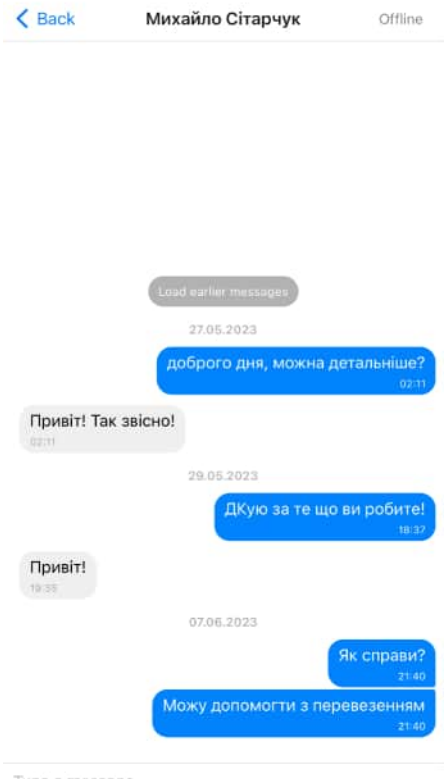

рис. 13 Спілкування з людьми

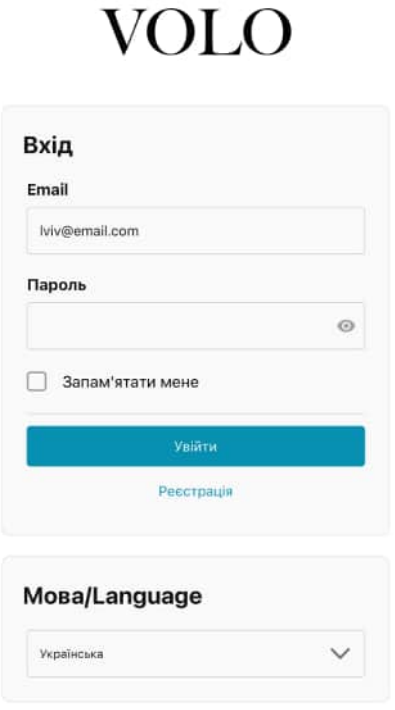

рис. 14 Сторінка входу та реєстрації

На рисунках 7 та 8, продемонстровано користувацьку сторінку, та можливість міняти та оновлювати персональну інформацію. На рисунку 9 представлена можливість залишати відгук людині, а саме виставивши оцінку від одного до п’яти, та залишивши коментар.

На рисунку 10 представлено вигляд сторінки запиту, також у верхньому куті, в залежності від того чи запит був створений користувачем, можна побачити кнопку “Змінити”, що дає змогу тримати в актуальності дані про збір.

Рисунки 12 та 13 показують можливість фільтрації запитів за такими критеріями: Тип запиту (фінансовий, матеріальний тощо), Статус запиту (відкритий, завершений, в процесі, закритий) та дата створення запиту (за останні 24 години, за останній тиждень/місяць/рік).

Рисунок 13 демонструє функціонал чату між користувачами, де також є інформація чи людина є у мережі чи ні. А також, якщо людина почне писати, то відповідний індикатор з’явиться замість мережевого статусу людини.

Рисунок 14 показує сторінку входу у застосунок, та можливість відразу міняти стандартну мову застосунку. Сторінка реєстрації подібна до сторінки входу, з невеликими відмінностями, такими як підтвердження паролю, для забезпечення послідовності дизайну.

ВИСНОВОК

У ході виконання дипломної роботи було розроблено мобільний застосунок "Воло", призначений для спрощення координації та співпраці між волонтерами, військовими, біженцями та іншими сторонами, залученими до допомоги у повномасштабній війні в Україні.

Застосунок базується на сучасних технологіях і алгоритмах, що дозволяють користувачам ефективно створювати, переглядати та відповідати на запити про допомогу. Це забезпечує швидку та ефективну координацію зусиль.

"Воло" включає надійні заходи безпеки та конфіденційності даних, можливості зв’язку в реальному часі та інші функції, які підвищують зручність і ефективність для користувачів. Інклюзивний підхід програми поширює допомогу за межі кордонів України, дозволяючи людям у всьому світі відкривати відповідні запити та ініціативи та робити свій внесок у них, створюючи глобальну мережу солідарності.

Таким чином, розроблений застосунок "Воло" відповідає актуальним потребам суспільства та сприяє ефективній взаємодії та координації зусиль різних сторін, залучених до допомоги в умовах війни.

У процесі розробки мобільного застосунку "Воло" було використано ряд сучасних технологій. Для створення серверної частини застосунку було використано NestJS - потужний фреймворк, який дозволяє створювати надійні та ефективні серверні застосунки. Для роботи з базою даних було використано Prisma - високопродуктивний ORM, який забезпечує безпечний та зручний доступ до даних.

Для автентифікації користувачів було використано JWT (JSON Web Token), що дозволяє забезпечити безпечну та надійну автентифікацію. Swagger було використано для документації API, що полегшує розробку та тестування.

На стороні клієнта було використано React Query для управління станом застосунку, Axios API для взаємодії з сервером, та AsyncStorage для зберігання даних на пристрої користувача.

В ході розробки було враховано потреби різних груп користувачів. Застосунок має інтуїтивно зрозумілий інтерфейс, що дозволяє користувачам легко створювати, переглядати та відповідати на запити про допомогу.

Особливу увагу було приділено безпеці та конфіденційності даних. Застосунок включає надійні заходи безпеки, що захищають дані користувачів від несанкціонованого доступу.

Всі ці елементи разом дозволили створити ефективний та надійний інструмент для координації допомоги в умовах війни, що відповідає актуальним потребам суспільства.

Проєкт рахується з відкритим вихідним кодом (open source), тому долучитись до його покращення може кожен бажаючий.

Посилання на користувацьку частину:

https://bitbucket.org/karrtopelka-max/volo_frontend/

Посилання на серверну частину:

https://bitbucket.org/karrtopelka-max/volo_server/

ВИКОРИСТАНІ ДЖЕРЕЛА

1. Axios docs. Axios. URL: https://axios-http.com/docs/intro.
2. “Custom animated bottom tab bar in react native with react navigation, reanimated 2, and typescript”. Medium. URL: https://medium.com/@taitasciore/custom-animated-bottom-tab-bar-in-react-native-with-react-navigation-and-reanimated-2-and-33b91851e6f1.
3. “Deploy nest JS app with postgres in VPS”. Medium. URL: https://medium.com/swlh/deploy-nest-js-app-with-postgres-in-vps-e1ce4abd2cad.
4. “Documentation | NestJS - A progressive Node.js framework”. NestJS. URL: https://docs.nestjs.com/.
5. “JWT.IO - JSON web tokens introduction”. JSON Web Tokens - jwt.io. URL: https://jwt.io/introduction.
6. “KeyboardAvoidingView not working properly? Check your Flexbox Layout first”. Medium. URL: https://medium.com/@nickopops/keyboardavoidingview-not-working-properly-c413c0a200d4.
7. "Mobile-first vs. Desktop content". Medium. URL: https://medium.com/my-great-learning/mobile-first-vs-desktop-content-dbc3632d5e85.
8. “Nativebase | universal components for react and react native”. NativeBase. URL: https://docs.nativebase.io/?utm_source=HomePage&utm_medium=header&utm_campaign=NativeBase_3.
9. “Prisma documentation | concepts, guides, and reference”. Prisma. URL: https://www.prisma.io/docs/.
10. “Queries | tanstack query docs”. TanStack. URL: https://tanstack.com/query/latest/docs/react/guides/queries.
11. “React navigation”. React Navigation. URL: https://reactnavigation.org/docs/getting-started.
12. “Swagger | NestJS”. NestJS | Swagger. URL: https://docs.nestjs.com/migration-guide#nestjsswagger-package.

13. “Usage | async storage”. Async Storage. URL: https://react-native-async-storage.github.io/async-storage/docs/usage/.