math_2023_041

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЧЕРНІВЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ЮРІЯ ФЕДЬКОВИЧА

Факультет математики та інформатики
кафедра математичного моделювання

Розробка соціальної мережі сімейних зв'язків з використанням Node.js та React-Native

Кваліфікаційна робота

Рівень вищої освіти – другий (магістерський)

Виконав:
студент 6 курсу, 607 групи
Цуркан Віталій Євгенович
***Керівник**:*
асистент, к.т.н, Шкільнюк Д.В.

До захисту допущено
на засіданні кафедри
протокол № 9 від 5 грудня 2023 р
Зав. кафедрою _____ проф. Черевко І.М.

Чернівці – 2023

Анотація

В дипломній роботі розглядається питання розробки мобільного застосунку соціальної мережі сімейних зв’язків з використанням Node.js та React-Native.

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

_____ В.Є. Цуркан
(підпис)

Annotation

The thesis deals with the development of a mobile application for a social network of family connections using Node.js and React-Native.

The qualification work contains the results of my own research. The use of ideas, results, and texts of scientific research of other authors have a link to the appropriate source.

_____ В.Є. Цуркан
(підпис)

Зміст

Вступ

Сучасний світ переживає надзвичайно активний розвиток інформаційних технологій, які вплинули на усі аспекти людського життя. Однією з ключових сфер, що відчула вплив цього технологічного прориву, є спосіб, яким люди взаємодіють та підтримують соціальні зв'язки. Соціальні мережі стали не тільки платформами для обміну інформацією, але й важливими інструментами для підтримки зв'язків між людьми. Вони стали місцем, де формуються, зберігаються та підтримуються важливі соціальні зв'язки. З розвитком технологій та збільшенням доступності мобільних пристроїв соціальні мережі стали більш варіативними та різноманітними, відповідаючи різним потребам та інтересам користувачів.

Однак, в сучасному інформаційному середовищі велика увага приділяється спільнотам індивідів, які об'єднуються на основі родинних та особистих зв'язків. Сімейні зв'язки є важливими складовими частиною життя кожної людини, і створення платформи, яка дозволяє спростити та покращити комунікацію між членами сімей, може мати значущий вплив на якість їхнього життя.

Мета цієї роботи полягає в розробці інноваційної соціальної мережі, яка дозволить членам сімей більше часу проводити разом онлайн, обмінюватися фотографіями та відео, а також зберігати важливі спогади та інформацію. Для досягнення цієї мети використовуються сучасні технології розробки, зокрема Node.js[2] для створення серверної частини та React-Native[1] для розробки мобільних додатків. Node.js[2] дозволяє розробити потужний сервер для обробки запитів та зберігання даних, в той час як React-Native[1] надає можливість створювати мобільні додатки для різних платформ, забезпечуючи доступність розроблюваних сервісів для користувачів незалежно від їхнього пристрою.

У даній дипломній роботі докладно розглядається процес та основні принципи проектування та розробки соціальної мережі сімейних зв'язків, включаючи архітектурні рішення, вибір технологій, методи розробки а також інтеграція цієї платформи з мобільними пристроями для забезпечення максимальної доступності та зручності використання. Також розглядаються аспекти безпеки даних та захисту приватності користувачів, оскільки це є особливо важливими аспектами при розробці подібних платформ.

Розділ 1. Огляд і порівняльний аналіз існуючих рішень та сервісів в галузі соціальних мереж орієнтованих на сімейні групи.

1.1 Огляд існуючих рішень.

В сучасному світі широкого поширення та використання набули сервіси, які допомагають комунікувати та взаємодіяти між людьми за допомогою мобільних девайсів та в мережі Інтернет. Вони допомагають залишатися на зв’язку, та обмінюватися різноманітною інформацією, як між індивідами, так і між окремими групами користувачів. В даному розділі було проведено аналіз мобільних застосунків, які являються соціальними мережами, або схожими сервісами орієнтованими на користувачів в рамках однієї сім’ї; та визначено їхні функціональні особливості.

Facebook:

- *Переваги:*
    - Велика активна аудиторія та популярність серед користувачів усіх вікових груп.
    - Можливість створення сімейних груп та спільних подій для родини.
    - Функції обміну фотографіями та відео, чату, оголошень.
    - Користувач має можливість вказати своїх родичів серед існуючих користувачів.
- *Недоліки:*
    - Багатофункціональність, що може збити з пантелику користувача.
    - Проблеми з приватністю та безпекою даних.
    - Часто нерелевантна рекомендація контенту.

Рисунок 1.1 – Мобільний застосунок Facebook

Cozi Family Organizer:

- *Переваги:*
  - Спеціалізована платформа для організації сімейних справ та подій.
  - Можливість створення спільних календарів та списків справ для сім'ї.
  - Опція обміну спільними завданнями та списками покупок.
- *Недоліки:*
  - Обмежена спільність сервісу у порівнянні з більш загальними соціальними мережами.

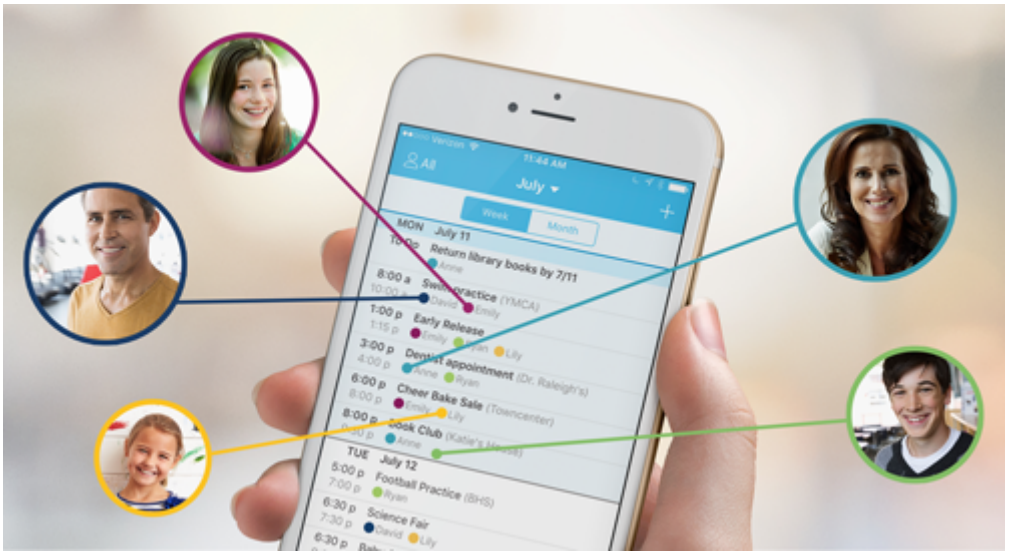

Рисунок 1.2 – Мобільний застосунок Cozy Family Organizer

FamilyAlbum:

- *Переваги:*
  - Спеціалізована платформа для зберігання і обміну фотографіями та відео в сімейному колі.
  - Зручна система тегування та сортування зображень.
  - Можливість обміну фотоальбомами з обмеженим доступом.
- *Недоліки:*
  - Спеціалізується на обміні фотографіями та відео, та не надає такі розширені можливості можливість залишати текстові коментарі або текстові повідомлення іншого типу.
  - Обмежує користувачів у різноманітних способах спілкування та співпраці.

Рисунок 1.3 – Мобільний застосунок FamilyAlbum.

Наведені приклади відображають, наскільки незаповненою та слоборозвинутою та перспективною є дана ніша, орієнтована саме на користувачів в одній сім’ї. В процесі розробки були враховані особливості інших застосунків, щоб додаток володів необхідними функціональними особливостями які повністю задовольняють користувацькі потреби.

1.2 Аналіз та постановка задачі на основі результатів досліджень.

За результатами огляду вже існуючих рішень можна краще зрозуміти сучасні підходи до поліпшення сімейних зв'язків через використання технологій. Було розглянуто різноманітні платформи та додатки, від великих соціальних мереж, таких як Facebook, до спеціалізованих сервісів, як Cozi Family Organizer та FamilyAlbum. Кожен з них має свої переваги та недоліки, і їх аналіз надав важливі вказівки для розробки соціальної

мережі сімейних зв'язків з використанням React-Native та Node.js. Однак враховуючи наявні обмеження та недоліки існуючих рішень, можна зробити висновок, що існує потреба у новому підході, який об'єднує найкращі аспекти цих рішень та надає користувачам зручний і безпечний інструмент для організації сімейних зв’язків, та обрано найбільш популярні мобільні операційні системи iOS та андроїд, найбільш підходящими для розробки соціальної мережі сімейних зв’язків.

Беручи за основу результати проведеного аналізу та враховуючи потреби користувачів, було визначено наступні основні завдання для проекту розробки соціальної мережі сімейних зв'язків:

- ➢ Мобільний застосунок: Розробити крос-платформенний мобільний застосунок під найпоширеніші операційні системи iOS та Android.
- ➢ Організація сімейних зв’язків: Забезпечити інструменти для організації та візуалізації сімейних зв’язків у соціальній мережі, включаючи сімейне дерево та можливість його оновлювати додаючи нових членів родини відповідно до їхніх сімейних зв’язків по відношенню до інших членів сім’ї на сімейному дереві. створення списків завдань, нагадувань та систему винагород за виконання.
- ➢ Обмін текстовим та медіа-контентом: Розробити функції для обміну текстом, фотографіями та відео в безпечному та приватному середовищі.
- ➢ Створення зручного інтерфейсу: Розробити простий, інтуїтивно-зрозумілий та легкий у використанні інтерфейс для користувачів, який зробить зручним взаємодію, як з іншими користувачами в межах сімейних груп так і з застосунком в цілому.
- ➢ Забезпечення безпеки та конфіденційності: Розробити заходи для захисту приватності користувачів та їхніх даних.

➢ Кросплатформеність й адаптивність: Застосунок повинен працювати на різних смартфонах на базі операційних систем iOS і Android з різними розширеннями екрану.

Поставлені завдання визначають основні цілі та напрями розробки проекту, який покликаний створити платформу, що допоможе організувати та візуалізувати сімейні зв’язки та забезпечує зручний і безпечний обмін медіа-контентом та текстовими повідомленнями в соціальній мережі сімейних зв’язків з урахуванням найкращих практик та функціональних особливостей.

Враховуючи поставлені завдання було сформульовано основні можливості користувача в застосунку:

- автентифікація з використанням соціальних провайдерів Apple та Google;
- редагування користувацького профілю;
- вихід з облікового запису (logout);
- візуалізація сімейних зв’язків з використанням сімейного дерева;
- створення та додавання нових користувачів у дерево;
- видалення створених користувачів з дерева;
- запрошення нових користувачів в застосунок з використанням запрошувальних посилань;
- перегляд профілів створених користувачів;
- редагування профілів створених користувачів;
- створення постів авторизованим користувачем;

- перегляд власних текстових постів або постів з медіа контентом та перегляд постів інших користувачів однієї сім’ї;
- видалення власних постів;
- створення коментарів під постами;
- видалення власних коментарів.

1.3 Висновки до розділу “Огляд і порівняльний аналіз існуючих рішень та сервісів в галузі соціальних мереж орієнтованих на сімейні групи”.

В першому розділі, даної роботи, було проведено порівняльний огляд вже існуючих відповідних аналогів та суміжних сервісів. Провівши огляд та аналіз вже наявних платформ, і їхніх недоліків й переваг, було прийнято рішення розробити соціальну мережу сімейних зв’язків саме у вигляді кросплатформенного мобільного додатку для операційних систем iOS та Android, оскільки саме смартфони на базі цих платформ, є найрозповсюдженішими на ринку смартфонів, а мобільні застосунки стають все популярнішими і все більша кількість користувачів надає перевагу використання додатків на смартфонах, а не сайтів.

Для забезпечення зручності користувачів, було вирішено, візуалізувати сімейні зв'язки, між користувачами, саме у вигляді сімейного дерева, яке забезпечить наглядність. Також, у даному розділі було визначено функціональні вимоги та завдання даного проекту, а також основні можливості користувачів у застосунку.

Розділ 2. Опис інструментів та технологій розробки.

2.1 Вибір архітектури для серверної частини та мобільного додатку

Щоб забезпечити зручність процесу розробки, ефективність і масштабованість, а також для успішної реалізації поставленого завдання, важливо вибрати відповідні архітектурні рішення. При проектуванні серверної частини, соціальної мережі сімейних зв'язків, було прийнято рішення використати підхід предметно-орієнтованого проектування (Domain-Driven Design, DDD)[16]. Domain-Driven Design[16] підхід надає можливість зосередитися на ключових аспектах предметної області, використовуючи ефективні та зручні інструменти для реалізації потреб користувачів.

Використання Node.js[2][15] дозволяє створити ефективний та масштабований сервер, особливо враховуючи асинхронну природу Node.js[2], яка дозволяє обробку багатьох запитів одночасно, що підходить для розробки високонавантажених додатків. Node.js[2] дає можливість використовувати JavaScript, як єдину мову на сервері та клієнті, що спрощує процес розробки та обслуговування коду. У поєднанні з TypeScript[14], Node.js[2] отримує статичну типізацію, що робить код більш надійним та полегшує виявлення помилок на етапі розробки. Як основний фреймворк, для платформи Node.js[2], було обрано Nest.js[3]. Даний фреймворк, який ґрунтується на Node.js, пропонує модульну архітектуру та використовує парадигму об’єктно-орієнтованого програмування (ООП). Це сприяє створенню легко розширюваних та підтримуваних додатків. Його використання узгоджено з TypeScript[14], надаючи підтримку статичної типізації та розширюючи можливості ООП. Заснований на стандартизованих підходах до структури коду, Nest.js[3] пропонує чітку організацію та полегшує розробку серверної частини

додатку. Відмінною підтримкою протоколів, таких як HTTP, WebSockets та GraphQL, Nest.js надає гнучкість у виборі оптимального способу взаємодії з сервером.

Для розробки мобільної частини, соціальної мережі сімейних зв’язків, було обрано React-Native[1] — фреймворк для кросплатформенної розробки з використанням React.js. Обраний фреймворк дає можливість розробляти нативні мобільні додатки під операційні системи iOS та Android використовуючи JavaScript/TypeScript[14] для обох платформ, залишаючи при цьому, можливість написання та використання нативного коду для відповідних операційних систем. Для пришвидшення розробки на React-Native, можна використати Expo SDK[13], що забезпечує великою кількістю необхідних API для роботи з нативними модулями. Expo SDK[13] за замовчуванням інкапсулює весь нативний код модулів платформ iOS та Android, але за необхідності дає, на вибір, можливість писати власний нативний код. Спільне використання “чистого” React-Native та окремих модулів Expo SDK[13] (“bare flow”), дозволяє значно спростити та прискорити процес розробки. Для написання логіки та керування станом (state management), було прийнято рішення використовувати Redux, а саме Redux-toolkit, який дозволяє помітно зменшити об’єм коду та зробити код біль структурованим та зрозумілим.

Обраний підхід до проектування та набір технологій забезпечує необхідну основу і надає необхідний інструментарій для розробки соціальної мережі сімейних зв’язків.

2.2 Опис архітектури серверної частини

Архітектура серверної частини соціальної мережі сімейних зв’язків ґрунтується на підході Domain-Driven Design або DDD[16].

Предметно-орієнтоване проектування[15] — це підхід до моделювання складного об'єктно-орієнтованого програмного забезпечення, що являє собою набір принципів і схем, спрямованих на створення оптимальних систем об'єктів. Зводиться до створення програмних абстракцій, які називаються моделями предметних областей. У ці моделі входить бізнес-логіка, що встановлює зв'язок між реальними умовами області застосування продукту і кодом. Предметно-орієнтоване проєктування не є будь-якою конкретною технологією або методологією. DDD[16] – це набір правил, які дають змогу ухвалювати правильні проєктні рішення. Цей підхід дає змогу значно прискорити процес проєктування програмного забезпечення в незнайомій предметній області.

Переваги DDD полягають в наступному:

- Концентрація основної уваги на предметній області;
- Створення програмних моделей, які відображають глибоке розуміння предметної області.

Основні визначення Domain-Driven Design:

1. **Домен (Domain)** — Предметна область, середовище, галузь. Предметна область, яку програміст використовує при створенні програмного забезпечення.
2. **Модель (Model)** — Система абстракцій, яка описує окремі аспекти предметної області.
3. **Загальна мова (Common language)** — Мова побудована навколо моделі предметної області. Використовується як програмістами при написанні програмного забезпечення, так і іншими членами команди (експертами обраної галузі). Загальна мова використовується для єдиного стилю опису домена і моделі

4. **Контекст (Context)** — Середовище, в якому предмет або дія означає своє значення.

Безперечно при проєктуванні бажано мати одну модель, яка повністю описує всю предметну область, але в реальності, для спрощення процесу розробки продукту, домен представляють у вигляді сполучення декількох взаємопов'язаних моделей. Стратегічно дизайн програмного продукту являє собою сукупність принципів для підтримки цілісності моделі, постійний рефакторинг як засіб дистиляції моделі, та поєднання декількох моделей в одну схему.

Концепція:

- **Обмежений контекст —** використання декількох моделей на різних рівнях проєкту. Даний підхід використовується для зменшення зв'язків між моделями, що виключає складність і заплутаність коду. Іноді буває незрозуміло, в якому саме контексті повинна використовуватися модель. Тому потрібно точно визначити контекст, в якому використовується модель. Визначити межі використання даної моделі та її характеристики.
- **Безперервна інтеграція** — існує тенденція фрагментування моделі у випадку коли в одному обмеженому контексті працюють одразу декілька людей. Це спричиняє розпад системи на дрібніші контексти, що в кінцевому результаті призводить до втрати цінності моделі. Тому потрібно постійно зливати код (мерджити), використовувати автоматизовані тести, приділяти увагу виробленню загальної мови в проєкті.
- **Карта контекстів** — При роботі над кількома окремими моделями у великій групі, різні члени команди можуть не знати про сутність інших моделей, що ускладнює процес загальної збірки кінцевого

продукту. Тому потрібно на етапі проєктування точно визначити, що саме виконує кожна модель і як вона взаємопов'язана з іншими моделями. У кінцевому результаті у вас повинна вийти мапа взаємозв'язків між моделями.

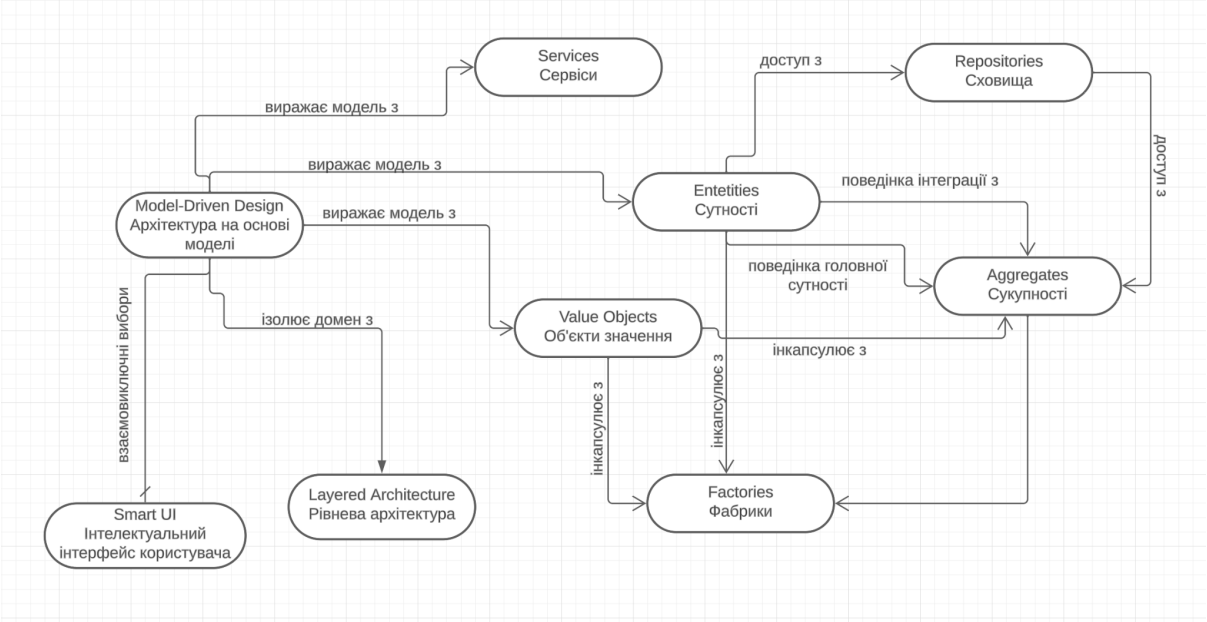

Рисунок 2.1.1 – Патерни в Domain Driven Design та відносини між ними.

При проєктуванні моделі предметної області необхідно сформувати спільну мову предметної області для опису вимог до системи, яка працює однаково добре як для бізнес-користувачів або спонсорів, так і для розробників програмного забезпечення. Ця мова визначається експертами в обраній галузі. У DDD є засоби для висловлення, створення та вилучення моделей предметної області:

- **Сутності** — Категорія індивідуальних об'єктів, які залишаються незмінними на різних етапах програми, для яких атрибути не грають великого значення, а послідовність та ідентичність, які поширюється в житті усієї системи називаються сутностями.

- **Об’єкти значення** — об'єкт, який містить атрибути, але не має концептуальної ідентичності. Він повинен розглядатися як незмінний об’єкт.
- **Сукупності** — Колекція об'єктів, які пов'язані між собою завдяки головній сутності (Root Entity), інакше відомій як Aggregate root. Коренева сутність колекції об'єктів гарантує узгодженість змін, що вносяться до сукупності, забороняючи зовнішнім об'єктам посилатися на членів колекції.
- **Сервіси** — Коли будь-яка операція концептуально не відноситься до будь-якого об'єкту, вона може бути реалізована в сервісі.
- **Сховища** — Отримання об'єктів предметної області повинно делегуватися в спеціалізовані сховища об'єктів. Це дає можливість поміняти місце збереження об'єктів.
- **Фабрики** — Створення об'єктів предметної області повинно бути делеговане до спеціалізованих фабрик. Це дає можливість підміняти реалізацію створення об'єктів.

Предметно-орієнтований підхід в проєктуванні є важливим методологічним підходом, який приносить безліч переваг у розробці програмних продуктів. Він полегшує розуміння вимог, сприяє виявленню ключових бізнес-процесів та робить розробку більш зорієнтованою на реальні потреби клієнта. Крім того, DDD[16] підтримує гнучке моделювання домену, що дозволяє легше адаптувати систему до змін у бізнес-логіці. Зменшення складності коду, фокус на стратегічно важливих аспектах системи та створення моделі як основи для проєктування архітектури — все це робить DDD потужним інструментом для успішної та зрозумілої розробки програмних продуктів.

2.3 Опис архітектури мобільного додатку

Керування станом та бізнес-логіка в мобільному додатку, соціальної мережі сімейних зв’язків, побудовано на Redux[9] архітектурі. Дана архітектура базується на парадигмі однонаправленого потоку даних та забезпечує централізоване управління станом, що полегшує розробку та підтримку складних додатків. Redux забезпечує простий та ефективний спосіб управління станом додатку, що полегшує відлагодження та розширення коду. Ця архітектура особливо корисна для великих та складних веб-додатків, де необхідно докладно відслідковувати стан та взаємодію між різними частинами додатку.

Основні концепції архітектури Redux включають:

1. **Store (Сховище)** — Це централізований об'єкт, який містить весь стан застосунку та використовується для доступу до нього. Весь стан, який використовується в додатку, є незмінним об'єктом, розташованим в одному місці, який визначає поточний стан всіх даних у вашому додатку.
2. **Actions (Дії)** — Дії є об'єктами, які вказують, що щось відбулося в додатку. Вони є єдиним джерелом інформації для зміни стану. Actions описують, що трапилося, і містять мінімальну необхідну інформацію про цю подію. Зазвичай об'єкт дії (action) має властивість type, яка вказує на тип дії, та інші властивості, які передають необхідні дані для виконання цієї дії. Redux відправляє їх у сховище, використовуючи метод store.dispatch().
3. **Reducers (Редьюсери)** — це функції, які приймають поточний стан та визначають, які зміни потрібно внести в стан додатку відповідно до отриманих дій. Редьюсери грають центральну роль у визначенні та управлінні станом в Redux-додатках. Вони є чистими функціями, тобто їх результат залежить тільки від їх вхідних даних. Редьюсер не

повинен мутувати вхідний стан. Замість цього він повинен створювати та повертати новий об'єкт стану, який представляє змінений стан. Коли ви відправляєте action до сховища (store), сховище передає цей action всім редьюсерам. Кожен редьюсер перевіряє тип action і відповідно реагує на нього, оновлюючи стан, який йому призначений.

4. **Store Subscribe (Підписка на Сховище)** — Компоненти додатку (react компоненти) можуть підписатися на зміни стану, щоб автоматично оновлювати себе, коли відбудуться зміни. Зміни, в свою чергу, можуть відбутися тільки внаслідок відправки дій (actions) до сховища.
5. **Middleware (Проміжний програмний засіб)** — Це шар програмного забезпечення, який розташований між відправленням дій (actions) та їх обробкою редьюсерами. Middleware може перехоплювати дії перед тим, як вони досягнуть редьюсера, і виконувати певні додаткові функції, такі як асинхронність, логування, трансформація дій тощо.

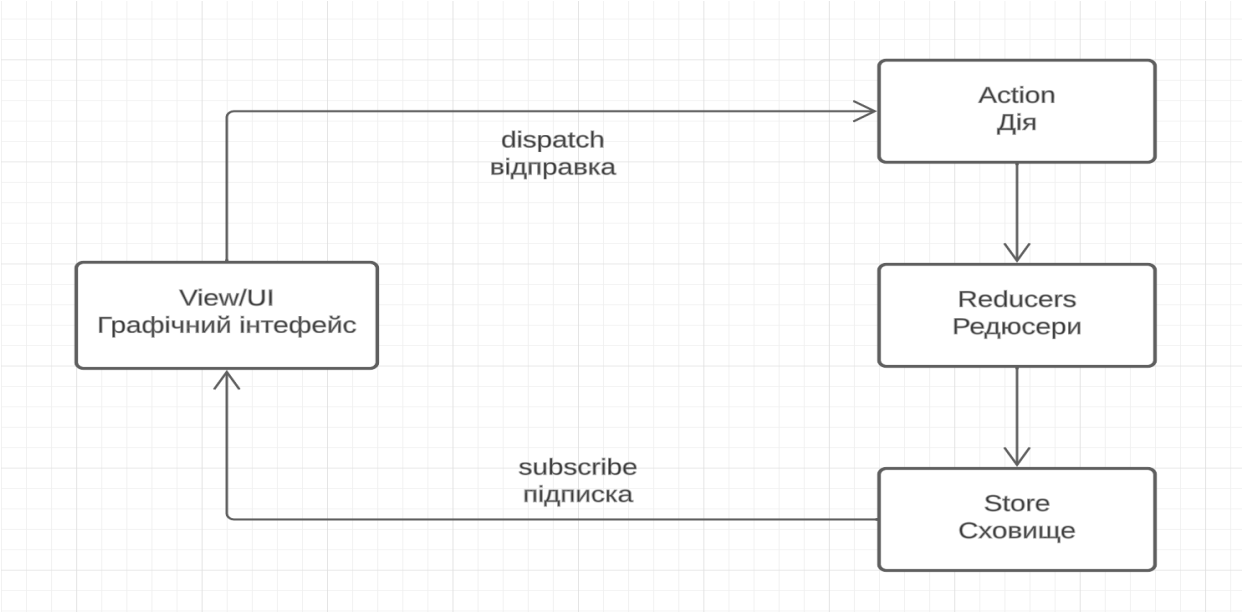

Рисунок 2.2.1 – Redux архітектура.

Для спрощення написання логіки керування станом за допомогою Redux, було вирішено використати Redux-Toolkit[8]. Redux-Toolkit[8] — це бібліотека, яка надає набір утиліт та шаблонів, які значно спрощують процес роботи з Redux. Створення Actions, Reducers та Сховища стає біль компактним та зрозумілим.

Основні особливості Redux-Toolkit:

1. **configureStore** — Redux-Toolkit надає функцію configureStore, призначеною для створення магазину Redux з налаштуваннями за замовчуванням. Вона об'єднує декілька налаштувань та розширень для полегшення роботи розробників та поліпшення продуктивності.
2. **createSlice** — Це функція Redux-Toolkit, яка дозволяє визначати редюсери та дії в одному місці. Це значно скорочує кількість шаблонного коду та спрощує процес додавання нових функціональностей.
3. **createAsyncThunk** — Для роботи з асинхронними операціями Redux-Toolkit надає createAsyncThunk, що робить обробку асинхронних дій більш простою та декларативною. Комбіновано з createSlice, це дозволяє розглядати асинхронні операції як частину слайсів стану.
4. **immer Integration** — Redux-Toolkit використовує immer для неізменної обробки стану. Це дозволяє змінювати стан без прямого мутування, що робить код більш безпечним та легшим для розуміння.

Використання Redux є корисним у багатьох випадках, особливо великих та складних додатках, де управління станом стає важливим завданням, оскільки надає однозначний та прогнозований спосіб управління станом додатка, який завдяку строгому управлінню змінами стану, спрощує процес відлагодження. Це дозволяє уникнути проблем,

пов'язаних із збереженням та оновленням стану великих додатків. Незважаючи на те, що використання Redux вимагає додаткового шаблонного коду, він пропонує ефективні та стабільні засоби управління станом, зокрема в комплексних або великих проектах, що сприяє покращенню обслуговуваності та розширюваності додатка.

2.4. Опис інструментів розробки та обґрунтування вибору технологій та інструментів

Сучасні вимоги до розробки серверних частин мобільних додатків визначають необхідність використання потужних та ефективних технологій. Розробляючи соціальну мережу сімейних зв’язків, було використано велике різноманіття технологій, інструментів та бібліотек, які забезпечують високий рівень продуктивності та масштабованості.

Основні використані інструменти та їх короткий опис:

Node.js[2] — Node.js є серверною платформою, побудованою на движку JavaScript V8 від Google. Його особливість полягає в асинхронному та подійно-орієнтованому програмуванні, що робить його ідеальним для створення швидких та ефективних додатків, особливо для тих, де велика кількість одночасних з'єднань важлива, як у випадку соціальних мереж. Однією з ключових переваг Node.js є його можливість працювати подійно-орієнтовано, що дозволяє легко взаємодіяти з вводом/виводом та обробляти тисячі одночасних з'єднань без створення багатьох потоків. Це особливо важливо для високопродуктивних та масштабованих веб-додатків, які взаємодіють з великою кількістю користувачів. Окрім того, Node.js[2] дозволяє використовувати JavaScript/TypeScript для написання серверного коду, що забезпечує єдність мови програмування між клієнтом та сервером. Це полегшує розуміння та

підтримку кодовій бази, що є важливим фактором при розробці та обслуговуванні проекту. Node.js має величезну кількість модулів та пакетів, доступних через пакетний менеджер npm, що робить його ідеальним вибором для екосистеми сучасних додатків.

Nest.js[3] — це модульний фреймворк для побудови серверних додатків на базі Node.js та Typescript. Обираючи Nest.js для розробки бекенду соціальної мережі, розробник переваги, які роблять процес розробки більш зручним та ефективним. Однією з ключових особливостей Nest.js[3] є використання концепції "провідного проектування" (leading design principles). Фреймворк пропонує використання декораторів та патернів проєктування, таких як Dependency Injection (DI), що дозволяє розробникам створювати чистий та добре структурований код. Одним із ключових компонентів Nest.js є модульність. Він дозволяє розділити функціональність додатка на невеликі та незалежні модулі, що полегшує розуміння та розширення кодової бази. Модульність також дозволяє легко підтримувати та тестувати окремі частини додатку. Важливою перевагою Nest.js є його інтеграція з Typescript.

TypeScript[14] — є сучасним мовою програмування, яка розширює стандартний JavaScript додаванням статичної типізації та інших корисних функцій. Обираючи Typescript для розробки бекенду соціальної мережі, розробник прагне досягти великої стабільності, безпеки та зручності у розробці. Однією з ключових переваг Typescript є можливість визначати типи для змінних, функцій та інших об'єктів. Це полегшує виявлення та усунення помилок ще на етапі розробки, забезпечуючи високий рівень безпеки коду. Завдяки статичній типізації, код стає більш читабельним та піддається ефективному рефакторингу. Typescript інтегрується безперешкодно з

Node.js[15] та Nest.js, що робить його природним вибором для розробки бекенду в контексті веб-додатків. Typescript забезпечує можливість використовувати новітні фічі ECMAScript та одночасно користуватися перевагами статичної типізації Typescript. рім того, Typescript підтримує концепцію класів, інтерфейсів та інших ООП-принципів, що робить код більш структурованим та об'єктно-орієнтованим. Це полегшує розуміння та підтримку великих кодових баз, які є характерними для розробки серверної частини соціальної мережі.

Firebase[11] — це платформа від Google, яка надає ряд сервісів для розробки та підтримки веб-додатків та мобільних застосунків. Однією з ключових переваг Firebase є його масштабовані та гнучкі сервіси. В даному випадку, серед усіх множинних та корисних функцій і сервісів Firebase цікавий як сервіс автентифікації. Firebase Authentication дозволяє зручно та безпечно реалізувати автентифікацію користувачів через різні методи, включаючи адресу електронної пошти, телефон та вже існуючі облікові записи Google, Facebook, Twitter та інші соціальні провайдери. Це важливий аспект для соціальної мережі, оскільки надає можливість користувачам зручно та безпечно увійти та взаємодіяти з додатком.

@nestjs/passport, passport.js — це модуль автентифікації та авторизації для Nest.js, який дозволяє легко інтегрувати різноманітні стратегії автентифікації в додаток. Використовуючи даний модуль, отримується можливість зручно налаштовувати різні стратегії, такі як локальна стратегія, стратегія JWT (JSON Web Token) та інші.

Passport-firebase-jwt — це Passport стратегія, яка дозволяє використовувати Firebase JWT для автентифікації. Вона інтегрується

з Passport.js та дозволяє легко підключити Firebase до розроблюваного бекенду, використовуючи JWT для безпечної автентифікації.

Firebase-admin — це офіційний SDK для адміністрування Firebase проектів. Використовуючи цей SDK, отримується можливість легко взаємодіяти з різними сервісами Firebase, такими як база даних, автентифікація та зберігання файлів. Firebase Admin дозволяє розробнику використовувати серверні можливості Firebase безпосередньо в Nest.js додатку.

React-Native[1] — це фреймворк для розробки мобільних додатків, який дозволяє використовувати React та JavaScript для побудови нативних інтерфейсів для платформ iOS та Android. Основною перевагою React-Native[5] є можливість використовувати один код для обох платформ, що значно економить час та ресурси. Серед основних переваг React-Native, які обґрунтовують його вибір, варто відзначити наступні:

- **Кросплатформенність** — один і той же код може бути використаний для розробки додатків для обох платформ (iOS та Android), що спрощує та прискорює розробку.
- **Швидкість** розробки — за допомогою гарячого перезавантаження (hot-reloading), розробники можуть бачити зміни в реальному часі без перезапуску додатку, що робить процес розробки більш швидким та продуктивним.
- **Багатофункціональність** — підтримує велику кількість сторонніх бібліотек та модулів, що дозволяє легко використовувати різноманітні функціональності в додатках.

- **Нативний інтерфейс** — дозволяє створювати інтерфейс, який виглядає та працює так само, як нативний, оскільки використовує нативні компоненти для обох платформ.

React-Navigation — це бібліотека для навігації в React-Native додатках. Навігація в мобільних додатках — це важлива частина користувацького досвіду, і React-Navigation надає потужні та гнучкі інструменти для організації навігаційної структури додатку, що дозволяє зручно переміщатися між екранами. React-Navigation являє собою набір біоліотек з різними елементами навігації, дана декомпозиція дає змогу встановити тільки ті елементи навігації які безпосередньо необхідні і будуть використовуватися розробником, забезпечуючи при цьому необхідну гнучкість та кастомізацію.

Redux-Toolkit та Redux-Saga[10] — це інструментарій для розробки з Redux. Він забезпечує простий та потужний спосіб управління станом додатку, роблячи код більш компактним та зрозумілим. Використання Redux-Toolkit спрощує інтеграцію з Redux та забезпечує швидше та чистіше написання коду. Redux-Saga — це middleware для обробки побічних ефектів в Redux. Він дозволяє обробляти асинхронні операції, такі як запити до сервера чи обробка таймерів, в окремих потоках, використовуючи для цього функції-генератори. Redux-Saga дозволяє обробляти декілька побічних ефектів паралельно, забезпечуючи ефективну роботу з багатьма асинхронними операціями. Ще однією зручною особливістю є можливість відміняти або відкладати побічні ефекти, що полегшує управління асинхронною логікою. Використання redux-saga робить керування складною логікою обробки побічних ефектів більш простим та дозволяє підтримувати стан додатку в

синхронізованому стані. Комбінація з redux-toolkit надає потужний механізм для управління станом та асинхронними операціями в додатку.

Formik — це бібліотека для керування формами в React-Native додатках. Вона надає інтуїтивно зрозумілий та потужний спосіб обробки введення в формах, спрощуючи процес розробки та валідації. Formik надає простий та чистий API для керування формами, що полегшує їх розробку та обслуговування. Дана бібліотека корисна тим, що автоматично відслідковує стан форми, управляючи значеннями полів, їх валідацією та станом для поліпшення взаємодії з користувачем. Наявність можливості використовувати асинхронні функції для валідації дозволяє робити запити до сервера для перевірки унікальності даних, наприклад, електронної пошти. Formik спрощує використання валідації форм, в тому числі з іншою популярною бібліотекою, такою як yup.

NativeBase — це популярний UI-компонентний фреймворк для React-Native, який надає готові та стилізовані компоненти для швидкої та зручної розробки інтерфейсу користувача. NativeBase надає широкий набір готових компонентів, таких як кнопки, картки, іконки, списки, форми та багато інших. Це дозволяє розробникам швидко створювати функціональний та стильний інтерфейс. NativeBase підтримує темізацію, що дозволяє легко змінювати вигляд та стиль компонентів. Розробник може створювати власні теми або використовувати готові, щоб швидко адаптувати дизайн додатка. Компоненти NativeBase розроблені таким чином, щоб добре виглядати та працювати на обох основних мобільних платформах — iOS та Android. Це дозволяє забезпечити єдинообразний дизайн для всіх користувачів. NativeBase добре інтегрується з Redux, що

дозволяє легко керувати станом компонентів та взаємодіяти з ним через глобальний стан додатка. NativeBase дає розробникам можливість ефективно прискорити процес розробки мобільного інтерфейсу, користуючись готовими та стильними компонентами.

React-native-google-signin — це бібліотека для React-Native, яка дозволяє легко інтегрувати автентифікацію через сервіс Google в мобільному додатку. React-native-google-signin — використовує Google Sign-In бібліотеку від Google для безпечного та зручного механізму входу. Вона забезпечує можливість синхронізації з Google Акаунтом, що дозволяє користувачам використовувати свої дані та налаштування на різних пристроях. Використання протоколу OAuth 2.0 та OpenID Connect гарантує безпеку автентифікації та дозволяє користувачам надавати доступ до обраних ресурсів. Реалізована можливість отримання додаткової інформації від користувачів, такої як ім'я, електронна пошта та інші деталі профілю.

@react-native-firebase/auth — є модулем Firebase для React Native, який надає можливості для реалізації автентифікації користувачів в мобільних додатках. Модуль auth легко інтегрується з іншими модулями Firebase, забезпечуючи гармонійну роботу з усіма можливостями Firebase в React-Native додатках. Даний модуль надає можливість використовувати різні методи автентифікації, такі як електронна пошта/пароль, Google Sign-In, Facebook, Twitter, телефонний номер та інші. Firebase відповідає за безпечне зберігання та передачу конфіденційної інформації користувачів, що гарантує безпеку особистих даних. Auth використовує JWT-токени для автентифікації, що дозволяє безпечно передавати дані між клієнтом та сервером. Дана бібліотека зпечує можливість використання

модульного підходу до Firebase, що полегшує роботу з конкретними функціональними частинами, такими як автентифікація.

@react-native-firebase/dynamic-links — це модуль Firebase для React Native, який дозволяє реалізувати динамічні посилання в мобільних додатках. Динамічні посилання дозволяють створювати гнучкі та персоналізовані посилання, які можна використовувати для направлення користувачів на конкретні екрани або події вашого додатка. Модуль дозволяє створювати динамічні посилання, які можуть включати параметри та дані для персоналізації відвідувачів. Підтримується як для платформи Android, так і для iOS, що дозволяє однаково ефективно використовувати динамічні посилання на різних пристроях. Забезпечує можливість відстеження ефективності динамічних посилань за допомогою Firebase Analytics, дозволяючи оцінювати їх вплив та результативність. Динамічні посилання надають розробникам потужний інструмент для проведення рекламних та маркетингових кампаній зі зрозумілими результатами.

React-native-svg — це бібліотека для використання SVG (Scalable Vector Graphics) в React-Native додатках. SVG є векторним форматом зображень, який дозволяє створювати гнучкі та масштабовані графічні елементи. Дана бібліотека дозволяє розробникам ефективно використовувати потужні можливості векторної графіки для створення динамічних та високоякісних графічних елементів в мобільних додатках.

Розробляючи соціальну мережу сімейних зв'язків було використано систему управління базами даних (СУБД) Neo4j для зберігання та керування даними на стороні сервера. Neo4j[4] — це графова база даних, яка спеціалізується на зберіганні та обробці графових структур даних. Дані

зберігає у власному форматі, спеціалізовано пристосованому для представлення графової інформації, такий підхід, у порівнянні з моделюванням графової бази даних засобами реляційної СУБД, дає змогу застосовувати додаткову оптимізацію в разі даних зі складнішою структурою. У контексті розробки серверної частини застосунку з використанням Node.js та Nest.js, Neo4j використовується як основна база даних для зберігання та оптимізації сімейних зв'язків.

Основні особливості та переваги Neo4j:

1. **Графова структура даних —** Neo4j орієнтований на графову структуру даних, що ідеально підходить для моделювання та роботи зі схемою сімейних зв'язків. Оскільки основна ідея додатка полягає в роботі зі сімейними зв'язками, Neo4j, яка базується на графовій структурі даних, є природним вибором для зберігання та оптимізації таких взаємозв'язків.
2. **Запити мовою Cypher** — Для взаємодії з Neo4j використовується мова запитів Cypher, спеціально розроблена для роботи з графовою структурою даних, що спрощує взаємодію з базою даних та дозволяє зручно виконувати операції з графовою структурою. Cypher використовує декларативний синтаксис, що дозволяє описувати, що потрібно отримати, а не як це робити. Це спрощує створення зрозумілих та легких у виконанні запитів. Запити Cypher орієнтовані на графові операції, такі як знаходження шляхів між вузлами, фільтрація на основі зв'язків та взаємодія з графовою структурою.
3. **Швидкість та ефективність** — Neo4j забезпечує високий рівень швидкості та ефективності при операціях, пов'язаних з графовими запитами, особливо коли необхідно працювати зі складними взаємозв'язками. Головна перевага полягає в тому, що час читання залишається постійним як для 1 елемента, так і для 1 мільйона

елементів — тобто вузли можуть мати будь-яку кількість або тип відносин, не втрачаючи продуктивності.

4. **ACID властивості** — Neo4j гарантує підтримку стандарту ACID (Atomicity (Атомарність), Consistency (Консистентність), Isolation (Ізоляція), Durability (Стійкість)) властивості, що робить її надійною та стійкою до відмов — інтегрована стійкість забезпечує, що результати успішно виконаних транзакцій залишаються незмінними, навіть якщо виникне непередбачувана відмова апаратної або програмної системи. ACID властивості забезпечують надійність та цілісність графових даних в Neo4j. Транзакції в Neo4j виконуються атомарно, що означає, що будь-яке відмова чи помилка призведе до відміни всіх змін у транзакції.
5. **Візуалізація графів —** Neo4j має інструменти візуалізації, що дозволяють адміністраторам та розробникам візуалізувати структуру графових даних для легшого аналізу та кращого розуміння взаємозв'язків.
6. **Зручна інтеграція з Node.js та Nest.js** — Neo4j має підтримку для Node.js, що полегшує інтеграцію з серверним стеком, особливо якщо, як в даному випадку, використовується Nest.js. Інтеграція Neo4j з Node.js та Nest.js відбувається за допомогою офіційного драйвера Neo4j для JavaScript, який дозволяє взаємодіяти з базою даних Neo4j з кодом, написаним на JavaScript або TypeScript. У випадку Nest.js, який використовує TypeScript, інтеграція зручна та дозволяє забезпечити типізацію та інші переваги TypeScript.

Використання Node.js дозволяє створювати високопродуктивні та масштабовані серверні додатки, а Nest.js забезпечує структуровану та організовану архітектуру, полегшуючи розробку та підтримку. Обираючи Node.js для бекенду соціальної мережі, отримується потужний, швидкий та

ефективний інструмент для розробки та масштабування застосунку. Обравши Nest.js, було обрано потужний та гнучкий фреймворк, який полегшує створення та обслуговування бекенду для соціальної мережі сімейних зв’язків, забезпечуючи високий стандарт розробки та підтримку. База даних Neo4j дозволила оптимально вирішити завдання збереження та оптимізації сімейних зв'язків, використовуючи потужність графової бази даних. Її зручність інтеграції з Node.js та Nest.js забезпечує ефективність розробки. А візуалізація даних у вигляді графів, дає змогу легко та зручно відобразити дані у зрозумілій та доступній формі, беручи до уваги той факт, що сімейні відносини, можна зручно представити та відобразити у вигляді графів.

Використовуючи React-Native[6] для мобільної частини застосунку, забезпечується ефективність та високий рівень кросплатформенності, з використанням одного коду для обох платформ, зберігаючи при цьому нативний вигляд, швидкість реалізації та хороший рівень продуктивності. Redux та Redux-Toolkit забезпечує зручний та ефективний спосіб керування станом застосунку, підтримуючи чітку структуру взаємодії між компонентами. TypeScript[14] вносить перевагу типізації до проекту, що сприяє виявленню та виправленню помилок на етапі розробки, а також покращує документування та читабельність коду. Інтегрувавши Firebase в процес автентифікації користувачів, було підвищено рівень надійності та безпеки, оскільки в такому випадку Firebase відповідає за безпечне зберігання та передачу конфіденційної інформації користувачів, що гарантує безпеку особистих даних. Firebase вже довів свою надійність та захист даних, що є критичним для додатків, особливо тих, що працюють з особистою інформацією користувачів.

2.5. Висновок до розділу “Опис інструментів та технологій розробки”.

У цьому розділі було детально розглянуто інструментарій та технології, які було використано в процесі розробки застосунку соціальної мережі сімейних зв'язків. Вибір цього стеку технологій базується на низці ключових факторів, які сприятимуть успішному втіленню поставлених завдань та досягненню поставлених цілей зі створення продукту з високим рівнем якості. Застосування обраного стеку технологій та інструментів, які було обрано з урахуванням всіх особливостей та загальної специфіки проекту, дозволяє створити комплексний та високопродуктивний мобільний застосунок з ефективною серверною та мобільною частинами, враховуючи його предметну область та сферу застосування.

Розділ 3. Опис створення соціальної мережі сімейних зв’язків.

3.1. Створення нового мобільного застосунку на React-Native та TypeScript.

Створення нового мобільного застосунку на базі React-Native та TypeScript є важливим етапом у реалізації соціальної мережі сімейних зв'язків. Правильна організація та структура коду дозволить легко розширювати та підтримувати додаток в майбутньому.

Для початку розробки необхідно встановити деяке необхідне програмне забезпечення, яке буде використовуватися під час розробки. Оскільки розробляється мобільний застосунок під декілька платформ, потрібно встановити стандартні середовища розробки для кожної з платформ. Для iOS — це Xcode, а для Android — це Android Studio. Розробляючи мобільний додаток на React-Native, у нас немає жорсткої необхідності писати код в цих середовищах розробки, розробник може використати будь-який зручний для нього редактор коду, але вони необхідні для тестування застосунку на симуляторах iOS-девайсів та емуляторах Android-девайсів, а також для деяких додаткових налаштувань. Крім того є можливість, а подекуди необхідність, в окремих випадках, використовувати ці середовища розробки для створення білдів. Xcode можна скачати з офіційного магазину застосунків для mac OS — AppStore. Android Studio можна завантажити з офіційного сайту https://developer.android.com/studio.

Як для створення мобільного застосунку на React-Native, так і для бекенду на Node.js, необхідно встановити Node.js, який буде використовуватись, певним чином як для бекенду так і для мобільного застосунку, оскільки даний додаток розробляється з використанням мови

TypeScript. Node.js можна завантажиит з офіційного сайту — https://nodejs.org/, або з використанням різних пакетних менеджерів, наприклад brew.

Маючи встановлене все необхідне програмне забезпечення, можна приступати до ініціалізації проекту. З використанням пакетного менеджера npm, який встановлюється автоматично зі встановленням Node.js, а також інструменту командного рядка React-Native CLI, ініціалізується новий проект мобільного додатку з TypeScript. Для створення нового проекту, необхідно виконати наступну команду в терміналі в обраній розробником папці:

```
npx react-native@latest init AppName
```

Рисунок 3.1.1 — Створення нового React-Native проекту з використанням пакетного менеджера npm.

Всі нові версії React-Native[1] пропонують використання TypeScript за замовчуванням, оскільки він довів свою корисність розробникам та являє собою стандарт розробки на даний момент, хоча React-Native застосунки все ще можна розробляти використовуючи JavaScript. Налаштувати параметри типізації та компіляції для TypeScript можна після завершення ініціалізації нового проекту у файлі для конфігурації TypeScript — tsconfig.json.

Після завершення процесу ініціалізації проекту, можна спробувати перевірити роботоспроможність новоствореного проекту можна запустивши його. Для цього, по-перше, потрібно запустити Metro, JavaScript-бандлер, який постачається разом з React Native. Metro дуже схожий на веб-пакет для React Native додатків. На відміну від Swift чи Objective-C, JavaScript не компілюється, як і React Native. Бандлінг — це

не те саме, що компіляція, але він може допомогти покращити продуктивність при запуску та перевести деякі специфічні для платформи JavaScript у більш широко підтримуваний JavaScript. Metro отримує вхідний файл і різні опції, а повертає єдиний JavaScript-файл, який включає весь ваш код і його залежності. Зробити це можна виконавши в терміналі кореневого каталогу проекту наступну команду:

```
npm start
```

Рисунок 3.1.2 — Запуск JavaScript-бандлера Metro React-Native проекту.

Запустивши бандлер Metro, можна переходити до запуску проекту на симуляторі iOS, та емуляторі Android. Для запуску проекту на iOS симуляторі необхідно в терміналі, в кореневому каталозі проекту, виконати наступну команду:

```
npm run ios
```

Рисунок 3.1.3 – Запуск проекту на iOS симуляторі.

Паралельно, або замість запуску проекту в iOS симуляторі, можна запустити проект на Android емуляторі. Для цього необхідно виконати майже ідентичну команду, але вказати що запуск буде на платформі Android:

```
npm run android
```

Рисунок 3.1.4 – Запуск проекту на Android емуляторі.

Запуск проекту на Android емуляторі та iOS емуляторах, або справжніх девайсах, можливий і через середовища розробки Android

Studio та Xcode відповідно, але роблячи це через термінал, є можливість, виконати певні додаткові дії, які в такому випадку мають бути попередньо вказані у файлі package.json у полі scripts. Поле scripts у файлі package.json визначає набір команд, які можна викликати за допомогою пакетного менеджера npm. Кожна команда описується як ключ-значення у об'єкті scripts. Це поле надає спосіб визначити різні дії та задачі, пов'язані з розробкою, тестуванням, збіркою тощо.

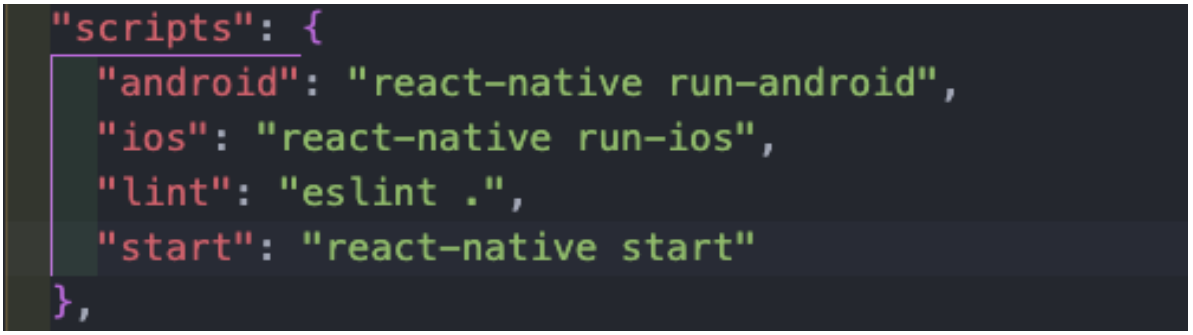

Рисунок 3.1.5 – Вигляд за замовчуванням поля scripts у файлі package.json новоствореного React-Native проекту.

В папці проекту присутні дві папки — ios та android, які включають в себе весь нативний код відповідних платформ проекту. Писати нативний код під будь-яку з платформ немає необхідності, але присутня можливість, а в деяких ситуаціях і необхідність зробити додаткові налаштування. Ініціалізований проект вже включає в себе кореневий файл App.tsx, який відповідає за ініціалізацію та рендеринг всіх інших компонентів. Будь-які інші нові компоненти мають бути додані в нього, або в один із компонентів який доданого в цей файл.

3.2. Створення нового бекенд-проекту на Node.js та Nest.js.

Створення бекенд-проекту є критичною частиною розробки соціальної мережі сімейних зв'язків, оскільки він відповідає за обробку запитів, управління базою даних та забезпечення взаємодії з мобільним

застосунком. Для ініціалізації нового бекенд-проекту на Node.js[2], так само як і для ініціалізації мобільної частини на React-Native необхідно мати встановлену Node.js, яка в свою чергу дає можливість використовувати пакетний менеджер npm.

Маючи встановлену Node.js можна виконати npm команду, яка встановить інтерфейс командного рядка Nest — Nest CLI. Для встановлення Nest CLI необхідно виконати в терміналі наступну команду:

```
$ npm i -g @nestjs/cli
```

Рисунок 3.2.1 – Npm команда встановлення Nest CLI.

Після успішного встановлення даної cli потрібно перейти, за допомогою терміналу, у зручну директорію, яка буде використовуватися для ініціалізації бекенд-проекту, і виконати там команду, яка і ініціалізує Nest.js проект, вказавши назву проекту:

```
$ nest new project-name
```

Рисунок 3.2.2 – Команда ініціалізації нового Node.js проекту на основі Nest.js фреймворка.

Після виконання даної команди необхідно буде відповісти на деякі конфігураційні запитання, наприклад яку мову використовувати JavaScript чи TypeScript. Після успішного завершення виконання даної команди буде створено початкову структуру проекту з встановленими залежностями, в папці node_modules, та відповідними конфігураційними файлами, наприклад tsconfig.json, в якому так само, як і при розробці мобільної частини на React-Native з використанням TypeScript, можна визначити та налаштувати параметри компіляції та типізації. Також буде створено кілька

інших шаблонних файлів, а також створено каталог src/, який буде заповнено кількома основними файлами:

Рисунок 3.2.3 – Структура папки src/ новоствореного Nest.js проекту.

src/: Це основна папка для розробки, де зберігається весь вихідний код проекту.

- **main.ts** — Основний файл, який використовується для запуску вашого додатку. Тут відбувається ініціалізація Nest.js застосунку.
- **app.controller.ts** — Контроллер який визначає маршрути та обробники для цих маршрутів. Основна відповідальність контролера полягає в обробці HTTP-запитів та відправці відповідей.
- **app.module.ts** — Основний модуль додатку, де ви визначаються всі компоненти, контролери, провайдери та інші модулі, необхідні для правильного функціонування застосунку. Всередині app.module.ts знаходиться клас AppModule. Зазвичай додатки мають кілька модулів, і AppModule виступає в ролі головного модуля, який об'єднує інші модулі та компоненти.
- **app.services.ts** — Відіграє роль сервісу в архітектурі Nest.js. Сервіси використовуються для виконання бізнес-логіки, обробки даних та виконання різних операцій, які не пов'язані безпосередньо з обробкою HTTP-запитів чи взаємодією з базою даних.

Для запуску серверної частини проекту необхідно в терміналі виконати наступну команду:

```
$ npm run start
```

Рисунок 3.2.4 – Команда для запуску серверної частини проекту.

Створення нового бекенд-проекту на Node.js та Nest.js дозволяє побудувати потужний та ефективний сервер, який буде відповідати за обробку та управління даними для соціальної мережі сімейних зв'язків.

3.3. API Документація

API документація є важливою складовою будь-якого проекту, особливо у великих проектах над якими працюють великі команди. Основна мета API документації - надати зрозумілу та консистентну інформацію для розробників, які планують використовувати API. Документація забезпечує зрозумілість та легкість використання вашого API. Вона надає чіткі та конкретні інструкції щодо того, як взаємодіяти з різними ендпоінтами та як обробляти дані. Документація служить як важливий ресурс для підтримки та розвитку проекту. Нові розробники, приєднуючись до проекту, можуть швидко зорієнтуватися в API та почати використовувати його.

Створення Swagger API документації для проекту на базі Node.js[2] та Nest.js[3] може бути виконано за допомогою бібліотеки @nestjs/swagger. Дана бібліотека дозволяє автоматично генерувати Swagger-специфікацію на основі анотацій та метаданих у імплементованому коді.

Щоб створити API-документацію з використанням Swagger Node.js проекту з використанням Nest.js потрібно за допомогою пакетного

менеджера npm встановити, ви можете використовувати бібліотеку @nestjs/swagger використавши наступну команду:

```
npm install --save @nestjs/swagger
```

Рисунок 3.3.1 – Встановлення бібліотеки @nestjs/swagger.

Додавши наступний код зі вказаними параметрами в ініціалізаційному файлі Nest.js проекту Swagger згенерує документацію для API з заголовком "Family Social Network API" та відповідним описом описом:

```
const versionInfo = await import('./version.json');
const versionDate = new Date().toLocaleDateString(locales: 'en-US', options: {
  day: 'numeric',
  month: 'short',
  year: 'numeric',
  minute: '2-digit',
  hour: 'numeric',
});
const config = new DocumentBuilder()
  .setTitle(title: 'Family Social Network API')
  .setDescription(description: 'API for managing family trees and members')
  .setVersion(version: `v.${versionInfo.service} ${versionDate}`)
  .addTag(name: 'Users', description: 'Operations related to managing users')
  .addTag(name: 'Trees', description: 'Operations related to managing family trees')
  .addTag(name: 'Files', description: 'Upload and delete files')
  .addTag(name: 'Comments')
  .addTag(name: 'Posts')
  .addBearerAuth()
  .build();

const document = SwaggerModule.createDocument(app, config);
SwaggerModule.setup(path: 'api', app, documentOrFactory: document);
```

Рисунок 3.3.2 – Код з налаштуваннями для ініціалізації Swagger.

Вказані значення можна налаштувати відповідно до розроблюваного проекту. Для того щоб побачити Swagger, необхідно запустити сервер виконавши, в терміналі проекту наступну команду:

```
npm run start
```

Рисунок 3.3.3 – Запуск сервера Node.js.

Після виконання всіх попередніх кроків та запуску сервера, Swagger UI буде доступний за адресою http://localhost:300/api. Перехід за цією адресою надасть змогу використовувати Swagger для легкого перегляду та тестування розробленого API.

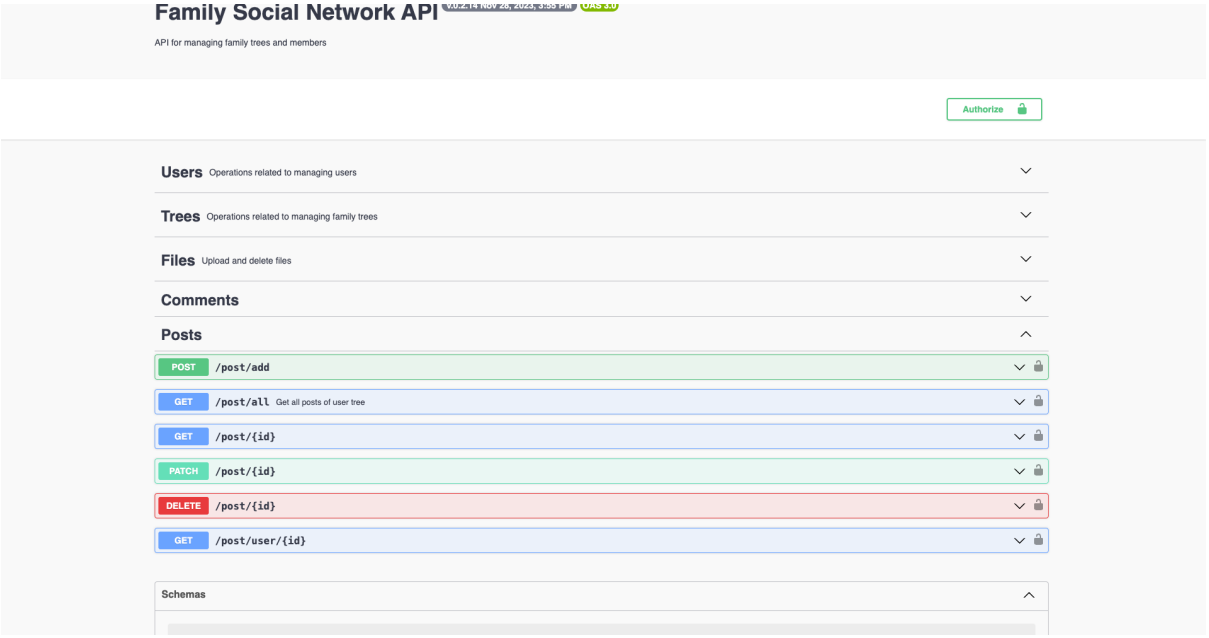

Рисунок 3.3.4 – Документація Swagger UI.

3.4. Функціонал сервісу.

3.4.1. Процес автентифікації.

Одразу після відкриття застосунку користувач потрапляє на екран автентифікації. Для того щоб зареєструватися в системі, або зайти у вже наявний профіль, необхідно автентифікуватися за допомогою свого профілю Google натиснувши єдину кнопку на екрані:

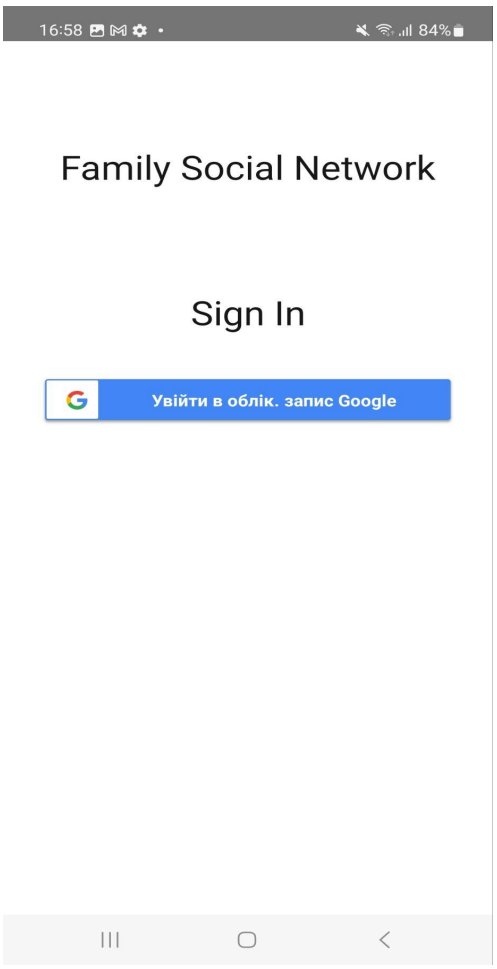

Рисунок 3.4.1.1 — Перший екран який бачить користувач коли заходить в застосунок та вхід з використанням профілю Google.

Після натискання кнопки “Увійти в облік. запис Google” буде відкрито вікно входу в обліковий запис користувача Google, де також буде доступна можливість створити новий профіль за його відсутності.

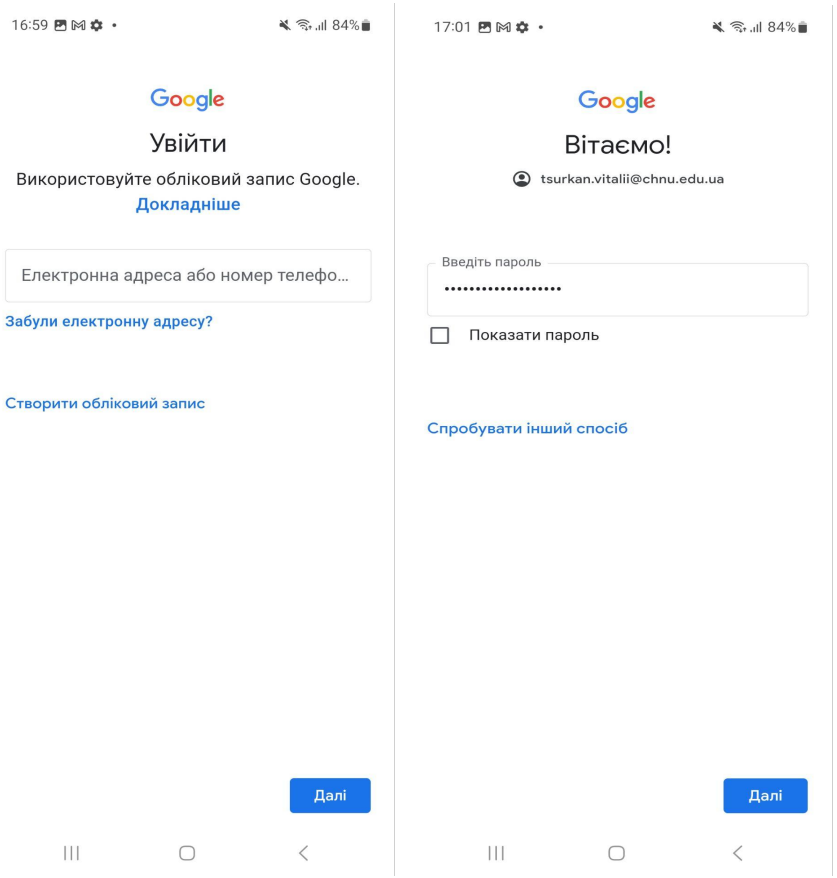

Рисунок 3.4.1.2 — Графічний інфтерфейс під час входу в обліковий запис користувача Google.

Після завершення процесу входу у обліковий запис Google, користувача буде перенаправлено на екран де він може заповнити деяку базову інформацію про себе — ім’я, прізвище, дату народження та стать:

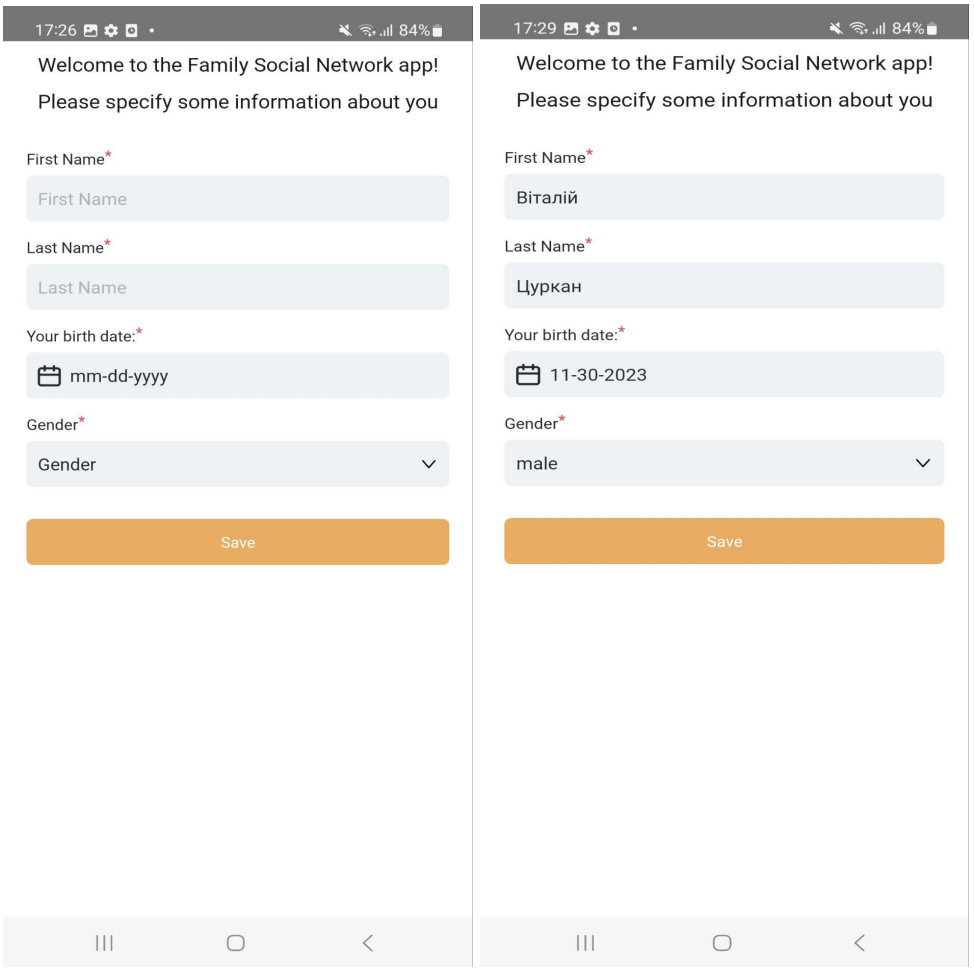

Рисунок 3.4.1.3 — Екран на якому користувач має змогу заповнити інформацію про себе із пустими полями та вже заповненими.

3.4.2. Опитуванням про структуру сімейних зв’язків.

Після введення всіх необхідних даний, всі поля обов’язкові до заповнення, користувач потрапить на екран де побачить повідомлення про те, що йому пропонують пройти коротке опитування про структуру його сім’ї щоб згенерувати та відобразити сім'ю користувача у вигляді сімейного дерева:

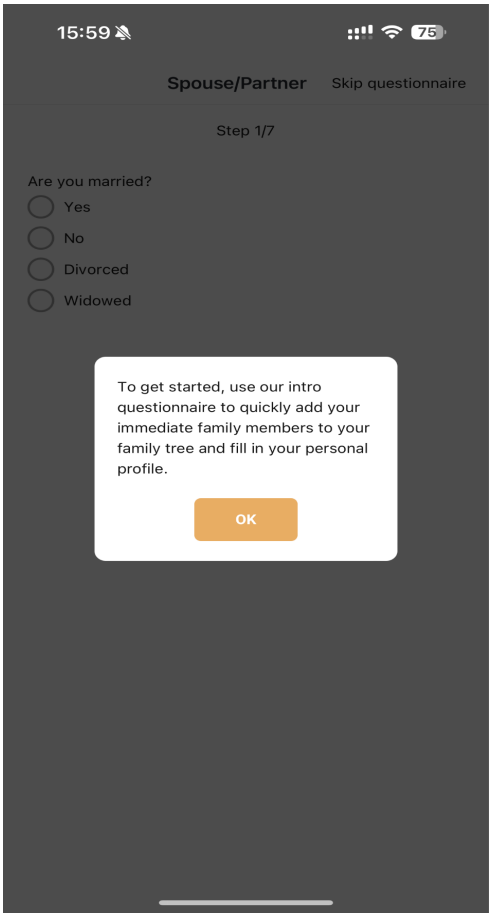

Рисунок 3.4.2.1 — Екран з повідомленням про можливість проходження опитування про найближчих родичів.

Натиснувши кнопку “ОК” дане повідомлення зникає, і користувач отримує можливість взаємодіяти з інтерфейсом. Проходити дане опитування необов’язково, але пройшовши його для користувача буде згенероване сімейне дерево на основі введених ним даних. Якщо користувач відмовиться його проходити, зробити це він може на будь-якому з семи етапів, то у нього все одно залишиться можливість додати всіх своїх родичів у ручному режимі.

Першим питанням у опитуванні є — чи користувач одружений. За замовчування користувачу доступні 4 варіанти відповідей: “Yes” – так, “No” – ні, “Divorced” – розлучений, “Widowed” – вдова/вдівець. При обраному варіанті “No” користувач має можливість одразу натиснути

кнопку “Next” і перейти до наступного питання. При обраному варіанті “Divorced” або “Widowed” користувач отримає змогу відповісти на запитання чи хоче він додати свого померлого або колишнього партнера. Якщо обрати варіант “No” то кнопка “Next” стане одразу доступною і користувач може рухатися далі. Якщо ж користувач обрав хоча б в одному з запитань варіант “Yes”, то з’являться додаткові поля де користувач зможе заповнити інформацію про свого партнера/партнерку у стосунках:

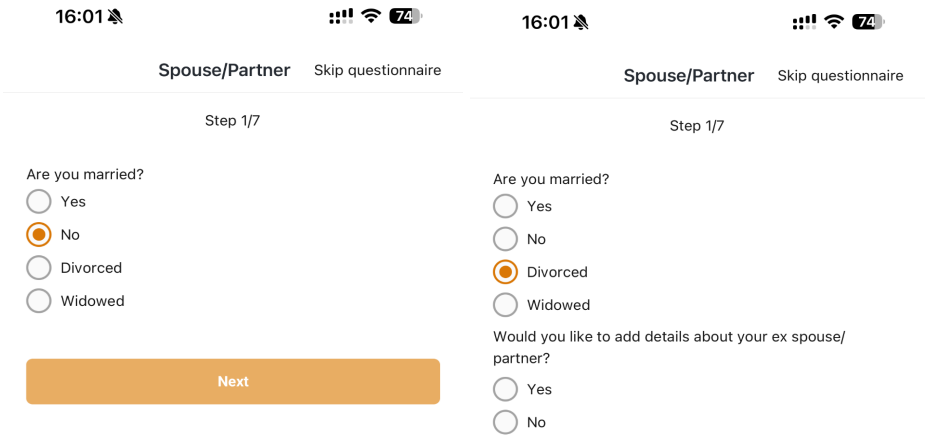

Рисунок 3.4.2.2 — Графічне відображення UI якщо користувач обрав варіанти “No” або “Divorced”.

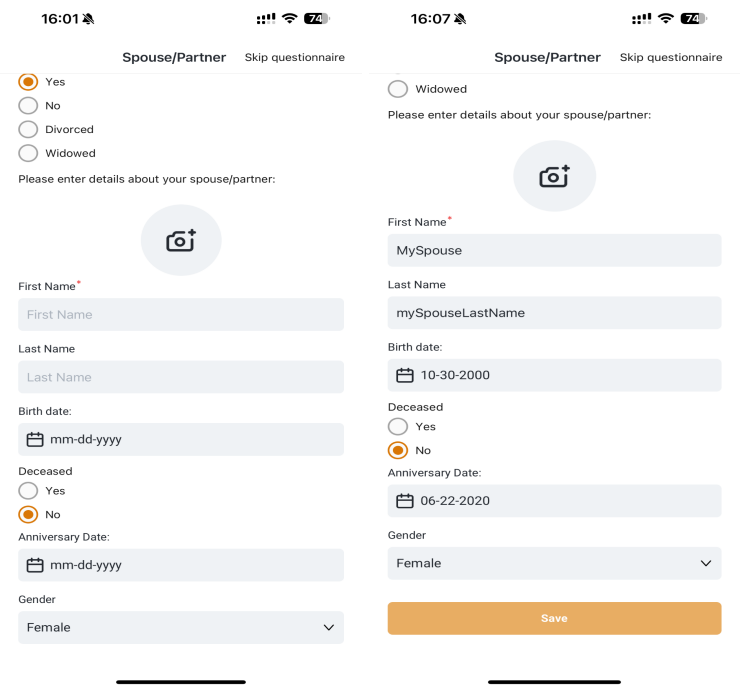

Рисунок 3.4.2.2 – UI–елементи для заповнення даних про партнера/партнерку за умови що користувач обрав варіант відповіді “Yes” хоча б на одне із поставлених запитань.

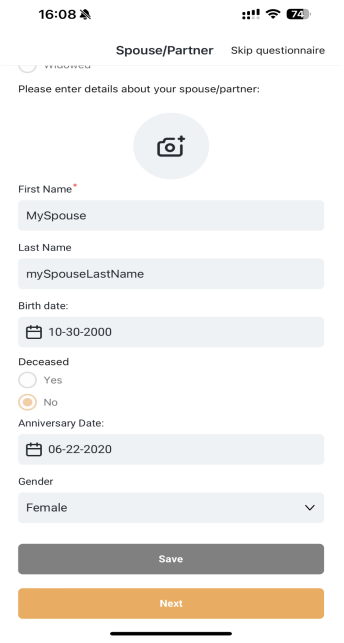

Рисунок 3.4.2.3 – Доступна кнопка “Next” після створення партнера/партнерки за умови що користувач обрав варіант відповіді “Yes” хоча б на одне із поставлених запитань.

На першому етапі опитування користувач має змогу його припинити, та перейти відразу до основної частини застосунку, натиснувши кнопку “Skip questionnaire”:

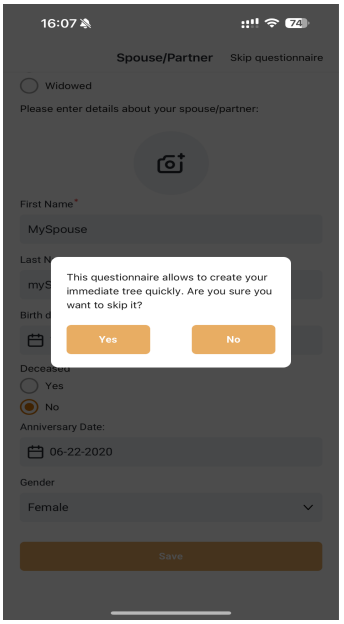

Рисунок 3.4.2.3 – Модальне вікно “Skip quesetionnaire”.

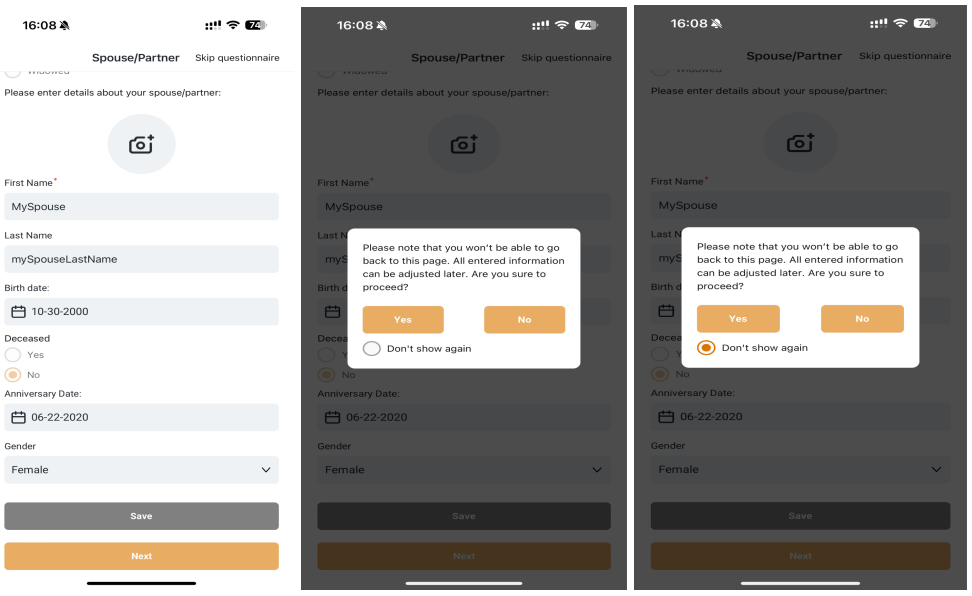

Рисунок 3.4.2.4 – Доступна кнопка “Next” після створення партнера/партнерки та модальне вікно, яке з’являється після натискання кнопки “Next”.

До того моменту поки користувач залишається на даному екрані він має змогу відредагувати введені дані про створеного родича, так само як і на всіх наступних екранах опитування.

Після натискання кнопки “Next”, користувач побачить модальне вікно, Рисунок 3.4.2.4, де буде проінформований, що більше не матиме змоги повернутися на цей екран і чи він впевнений що хоче продовжити. Дане модальне вікно буде з’являтися після натиснення кнопки “Next” на всіх етапах опитування, але вибравши опцію “Don’t show again” користувач може відмовитися від щоразового перегляду даного модального вікна. Натиснення кнопки “Yes” на модальному вікні призведе до переходу до наступного кроку опитування, де потрібно буде відповісти на запитання про дітей:

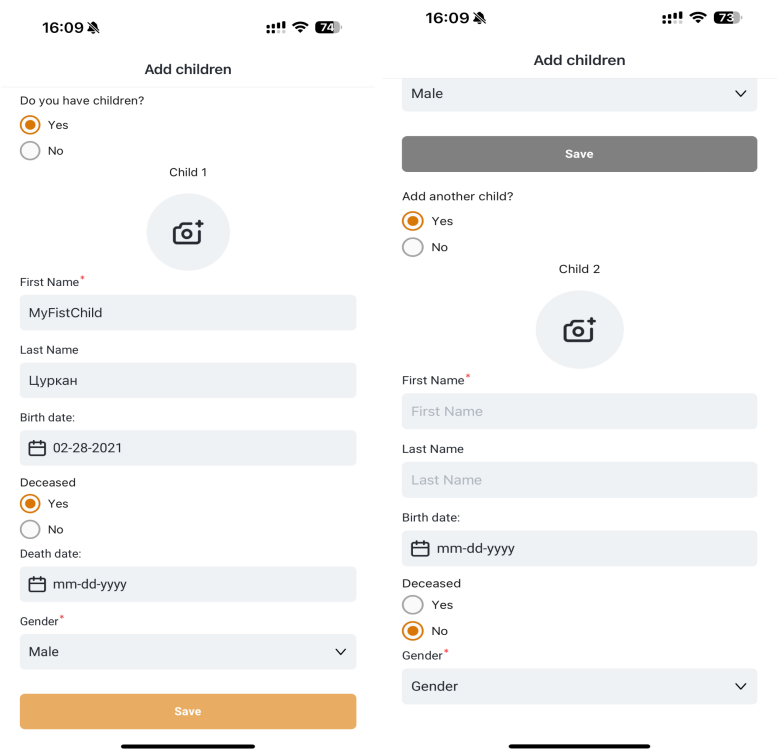

Рисунок 3.4.2.5 – Другий крок опитування — заповнення інформації про дітей.

На кроці додавання інформації про дітей, як і на попередньому, так і на кожному наступному, користувач може відповісти “No”, і перейти до наступного запитання. Обравши відповідь “Yes” користувач отримує змогу додати інформацію про будь-яку кількість дітей:

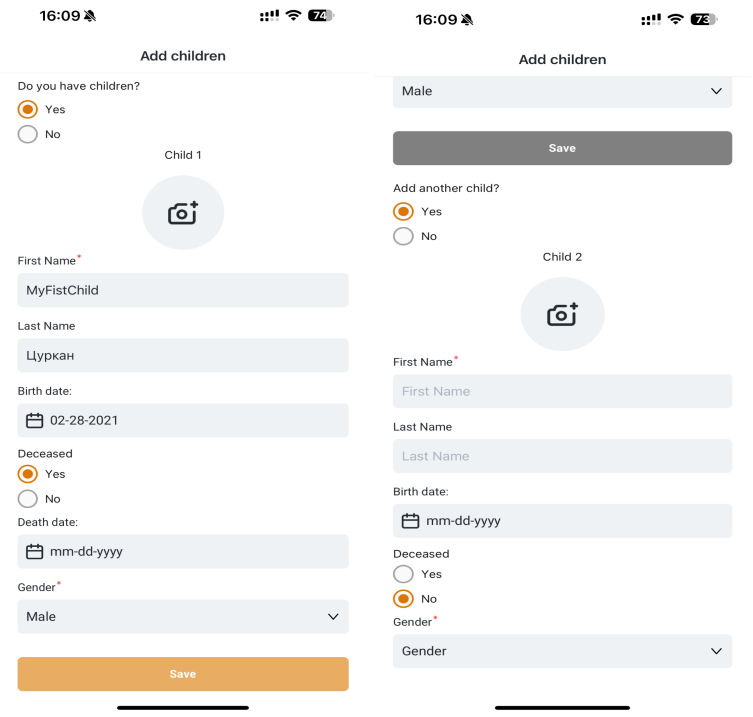

Рисунок 3.4.2.6 – Другий крок опитування — заповнення інформації про дітей.

Додавши всю необхідну інформацію про дітей, або відмовившись це зробити користувач потрапить на третій етап опитування, де потрібно буде заповнити інформацію про батьків Рисунок 3.4.2.7. На даному екрані, в процесі опитування, користувач зможе додати інформацію про батьків, відмовитись додавати інформацію про обох, або дадати інформацію тільки про одного з них. Від відповідей на дане питання залежить чи може користувач додати, в процесі опитування, своїх дідусів і бабусь, а також братів і сестер. Так, додавши лише одного з батьків, користувач зможе додати лише дідуся і бабусю в процесі опитування по лінії доданого з батьків. Таким чином крок додавання дідуся і бабусі по лінії іншого з

батьків буде пропущено. У будь-якому випадку у користувача залишається змога додати інформацію про них пізніше.

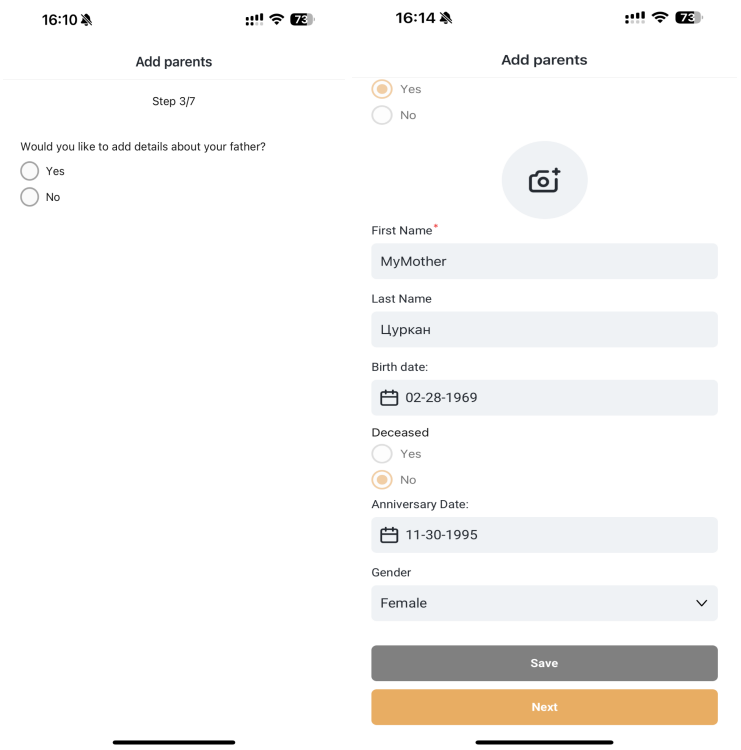

Рисунок 3.4.2.7 – Третій крок опитування — заповнення інформації про батьків.

У випадку заповнення інформації, хоча б про одного з батьків, користувача буде перенаправлено до наступного кроку проходження опитування де він зможе, за бажанням, додати інформацію про будь-яку кількість своїх братів і сестер:

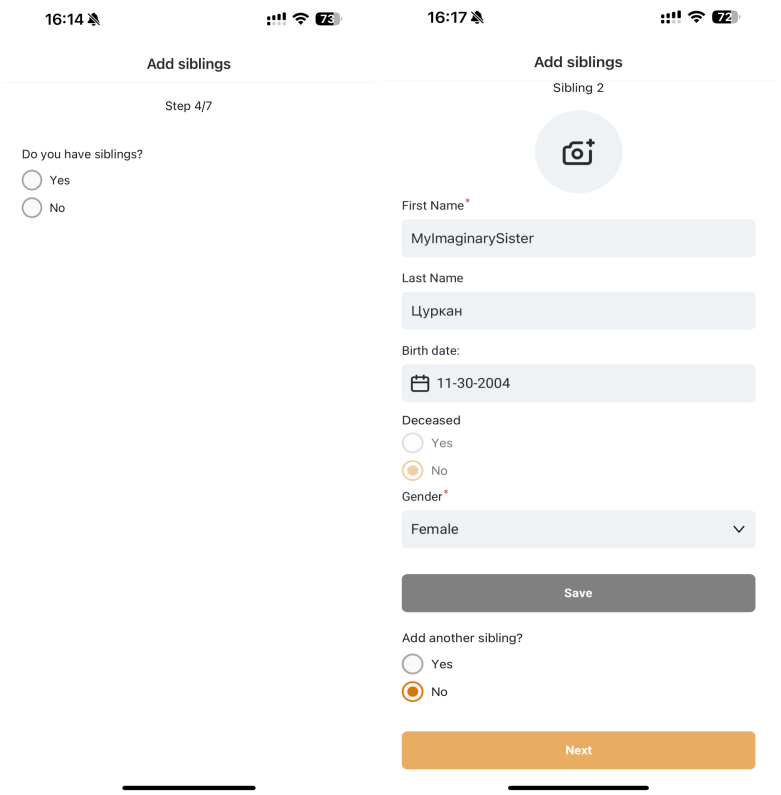

Рисунок 3.4.2.8 – Четвертий крок опитування — заповнення інформації про братів та сестер.

Пройшовши, будь-яким чином, етап заповнення інформації про братів і сестер, користувач потрапить на екран де він зможе додати інформацію про своїх дідусів і бабусь Рисунки 3.4.2.9. та 3.4.2.10. Дані екрани дуже схожі між собою та нагадують екран кроку заповнення інформації про батьків. Єдиними особливостями є те, що на них можна потрапити лише додавши інформацію про батьків. Не додавши інформацію про когось одного з батьків, користувач не отримає можливості додати своїх дідуся та бабуся по лінії цього з батьків.

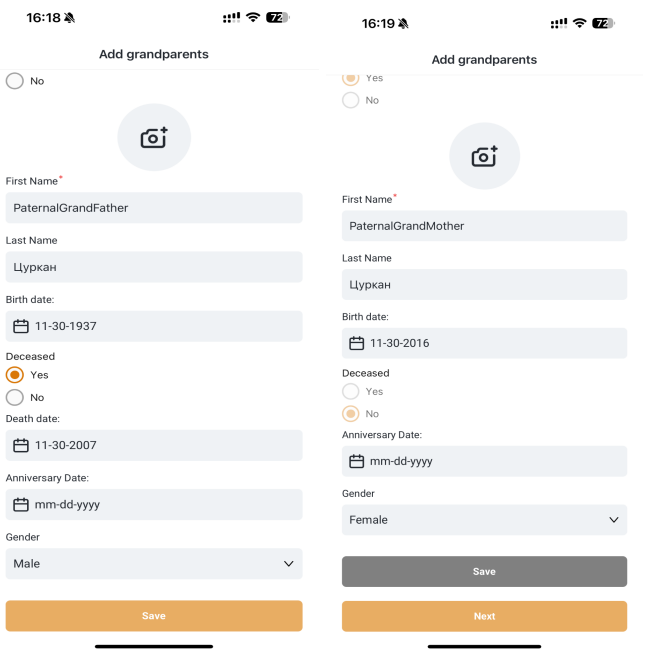

Рисунок 3.4.2.9 – П’ятий крок опитування — заповнення інформації про дідуся та бабусю по лінії батька.

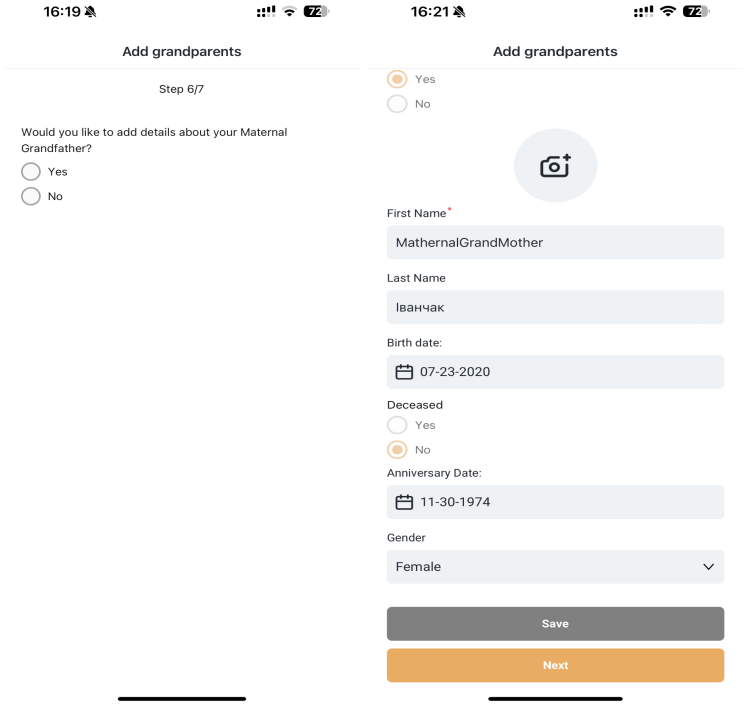

Рисунок 3.4.2.10 – Шостий крок опитування — заповнення інформації про дідуся та бабусю по лінії мами.

Останній крок при проходженні опитування — це заповнення додаткової інформації про себе: місця проживання, адреси, роботи а також домашніх улюбленців.

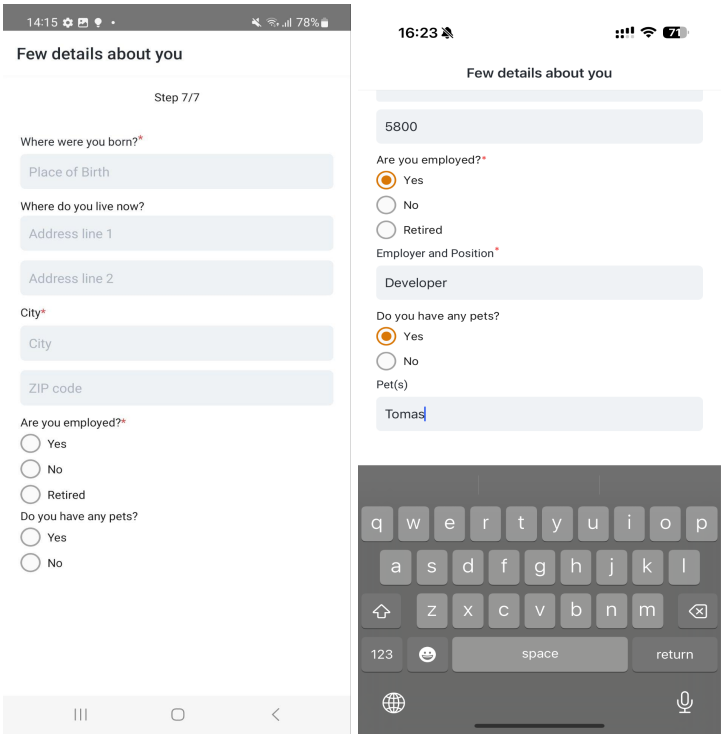

Рисунок 3.4.2.11 – Останній сьомий крок опитування — заповнення додаткової інформації про себе на Android та iOS.

3.4.3. Сімейне дерево.

Заповнивши всю необхідну інформацію на останньому етапі опитування, або відмовившись від проходження опитування користувач опиниться на екрані з сімейним деревом побудованим на основі інформації введеної користувачем при проходженні опитування. Структура сімейного дерева може відрізнятися в залежності від відповідей користувача при процесі опитування.

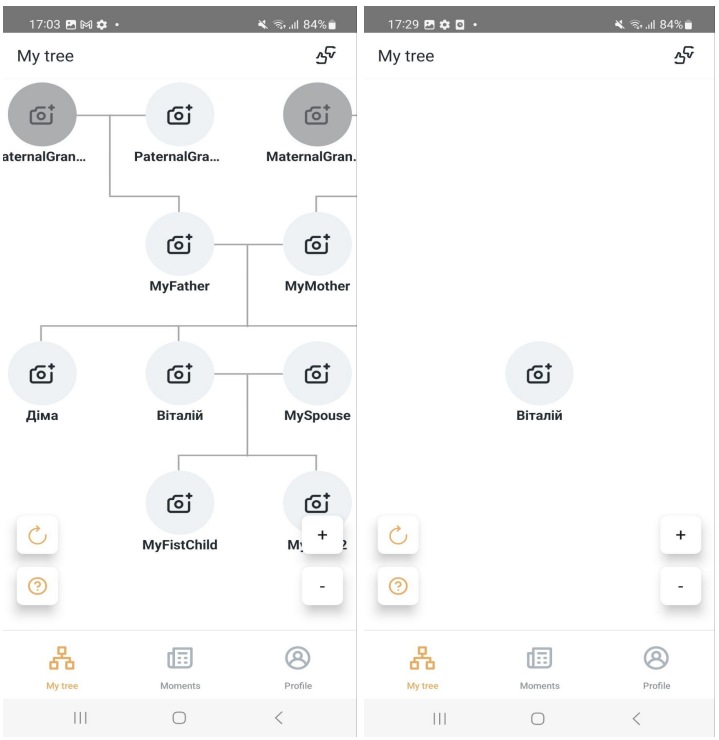

Рисунок 3.4.3.1 – Головний екран застосунку — екран з сімейним деревом у випадку повного проходження опитування і його пропуск.

При натисканні на елемент дерева відкривається допоміжне меню, яке з опціями для додавання нового родича до обраного члена в дерево, видалення обраного члене сімейного дерева, перегляд профілю з точки зору обраного родича, надсилання запрошення вибраному лену дерева або перехід до профілю обраного члена дерева. Список опцій може відрізнятися для кожного члена дерева в залежності від наявності, або відсутності, в нього тих чи інших сімейних зв’язків. Для прикладу — родич який вже має батька в дереві, не матиме доступної опції “Add father”.

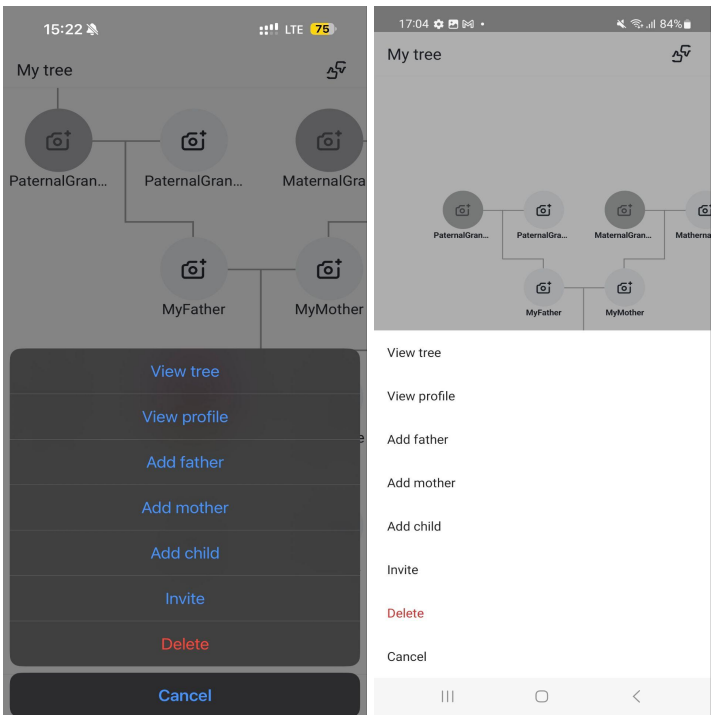

Рисунок 3.4.3.2 – Вигляд меню доступних опцій вибраного члена сімейного дерева на iOS та Android.

При переведенні телефону у горизонтальний (альбомний) режим, користувачем можуть бути помічені зміни — зникнення хедера ти нижньої панелі навігації Рисунок 3.4.3.3. Якщо нового члена дерева при створенні було помічено як померлого, то його зображення буде мати затемнений вигляд Рисунок 3.4.3.3.

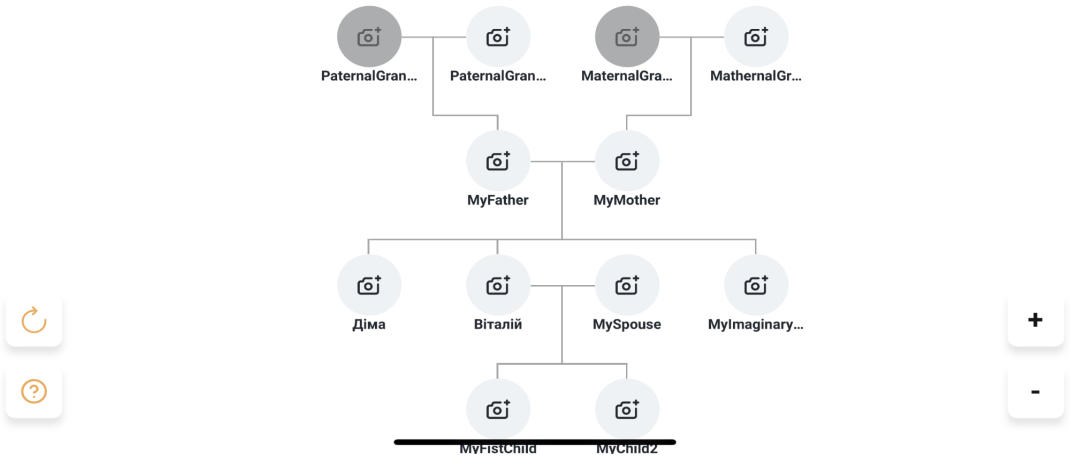

Рисунок 3.4.3.3 – Вигляд сімейного дерева у горизонтальному режимі.

Процес додаваннього нового члена сім’ї до сімейного дерева практично ідентичний — після вибору однієї з опцій для додавання нового родича з меню Рисунок 3.4.3.2, з'являється модальне вікно де необхідно ввести базову інформацію про нового члена дерева та натиснути кнопку “Add new user”. Новий родич автоматично з’явиться в дереві.

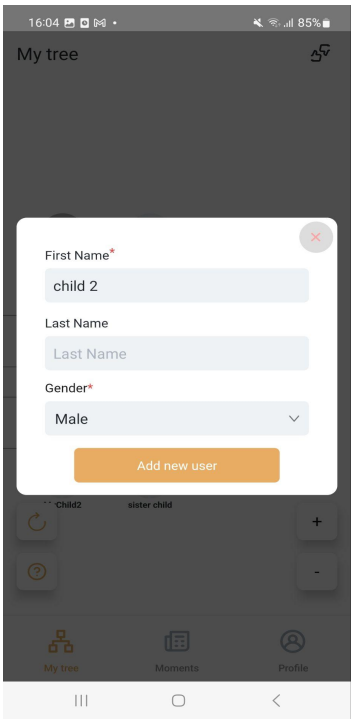

Рисунок 3.4.3.4 – Вигляд модального вікна при додаванні нового родича в дерево.

Екран з деревом володіє властивістю прокрутки та масштабування (скролінг). Скористатися можливістю масштабування можна з допомогою кнопок “+” та “-”, або використовуючи двома пальцями жест щипок (“pinch”) чи швидке подвійне натискання поза елементами дерева.

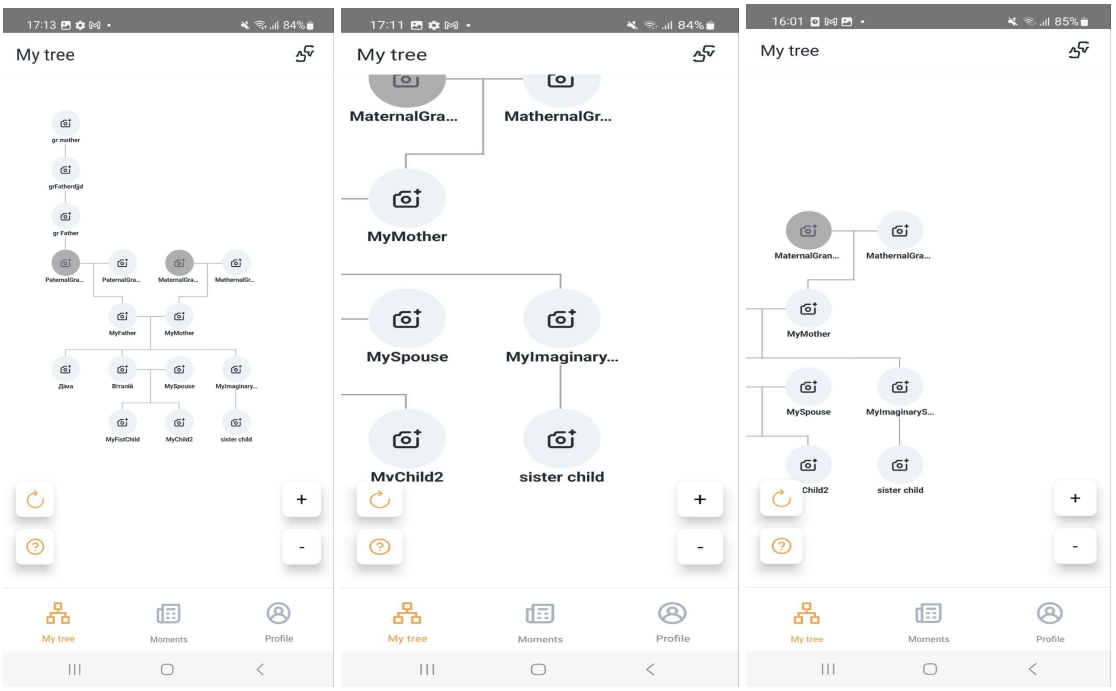

Рисунок 3.4.3.5 – Вигляд сімейного дерева при використанні різного масштабування та прокручування.

У випадку натискання кнопки зі знаком питання відкривається затемнене напівпрозоре модальне вікно з підказками для навігації.

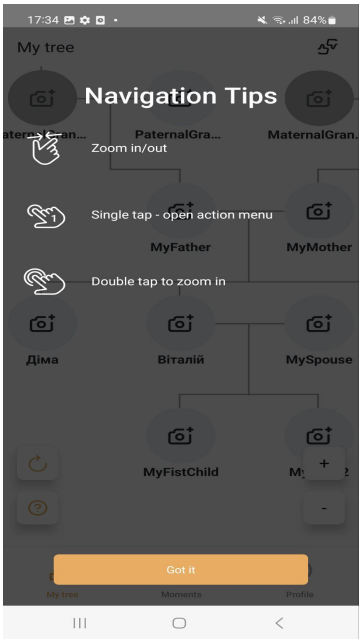

Рисунок 3.4.3.6 – Модальне вікно з підказками жестів для масштабування та користування деревом.

Верхня кнопка ліворуч (з заокругленою стрілкою) дає можливість повернутися до свого сімейного дерева у випадку, для випадку якщо користувач переглядатиме чуже дерево, на якому є відсутнім. Даний випадок можливий за умови переходу на дерево партнера/партнерки когось зі своїх родичів Рисунок 3.4.3.7. Натискання даної кнопки поверне користувача до перегляду власного сімейного дерева Рисунок 3.4.3.5.

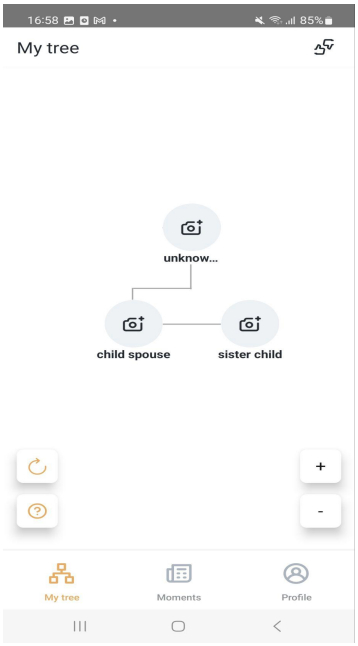

Рисунок 3.4.3.7 – Окремий випадок перегляду чужого дерева.

3.4.4. Стрічка сімейних моментів.

Стрічка сімейних моментів включає список моментів або постів від усіх членів сімейного дерева. Моменти можуть включати в себе текст, фото або відео, а також комбінації тексту з фото чи відео.

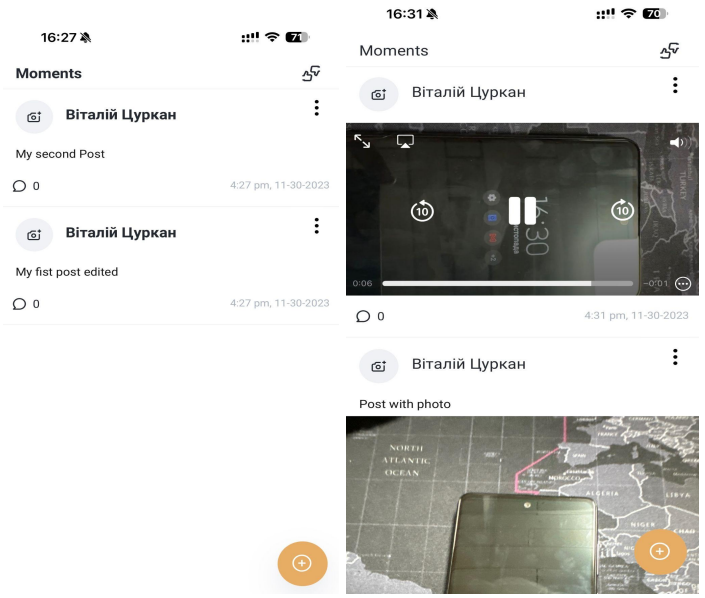

Рисунок 3.4.4.1 – Вигляд екрану з текстовими, фото та відео моментами.

Для створення моменту користувачу необхідно натиснути кнопку з символом + внизу екрану, яка перенаправить користувача на екран створення поста Рисунок 3.4.4.2. Фото та відео можуть бути як завантажені з галереї так і одразу зроблені в додатку з використанням допоміжного меню Рисунок 3.4.4.2.

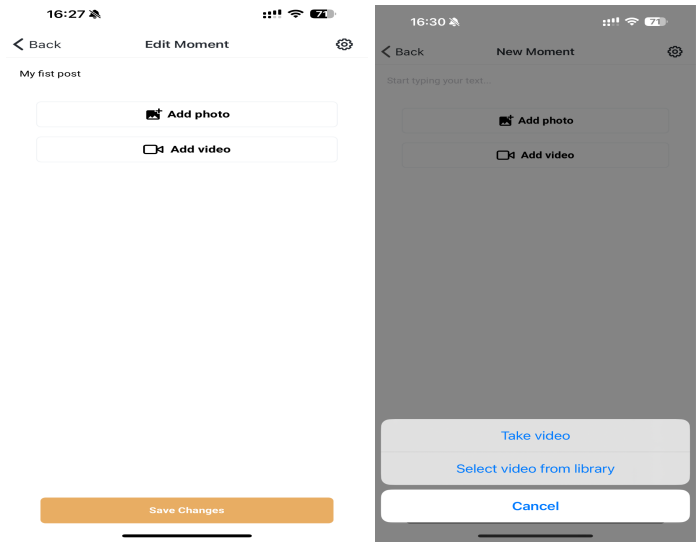

Рисунок 3.4.4.2 – Екран створення нового момента.

Для видалення або редагування поста, користувачем може бути натиснутий вертикальний символ трикрапок, що призведе до виклику допоміжного меню з опціями видалення та редагування Рисунок 3.4.4.3. При натисненні опції “Edit” користувача буде перенаправлено на екран редагування моментів Рисунок 3.4.4.4. При виборі варіанту “Delete”, момент буде одразу видалений.

Рисунок 3.4.4.3 – Вигляд допоміжного меню для видалення та редагування створених моментів.

Після завершення редагування та натиснення кнопки “Save changes” зміни внесені в момент буде збережено, а користувача буде перенаправлено на екран зі стрічкою моментів.

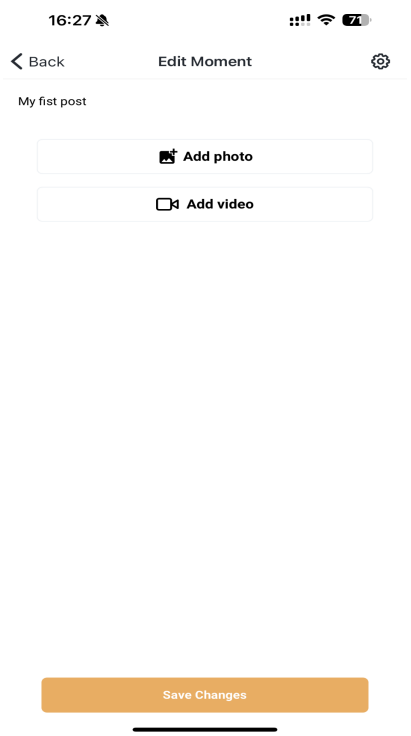

Рисунок 3.4.4.4 – Вигляд екрану для редагування створеного моменту.

До кожного моменту є можливість залишати коментарі Рисунок 3.4.4.5.

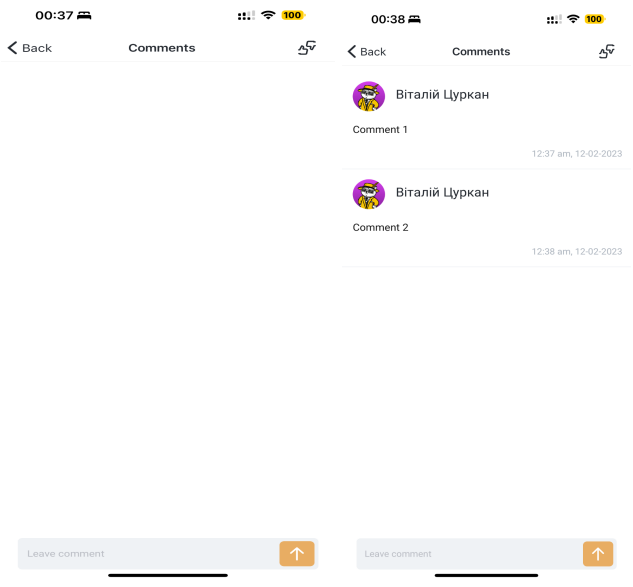

Рисунок 3.4.4.5 – Коментарі під моментом.

3.4.5. Профіль користувача

При переході на вкладку профілю, Рисунок 3.4.5.1 користувач може побачити екран зі своїм іменем та фото, а також 3 вкладки “Personal Info” з інформацією про профіль, “My Posts” із постами створеними виключно даним користувачем Рисунок 3.4.5.9 та “Family Members” з найближчими родичами Рисунок 3.4.5.10 та можливістю додати нових використовуючи те саме модальне вікно що й на сімейному дереві Рисунок 3.4.3.4. В хедері крім заголовку доступні дві кнопки — одна призначена для перенаправлення користувача на екран де можна залишити фідбек Рисунок 3.4.5.5, а інша переходу на екран де користувач може здійснити вихід з додатку Рисунок 3.4.5.11.

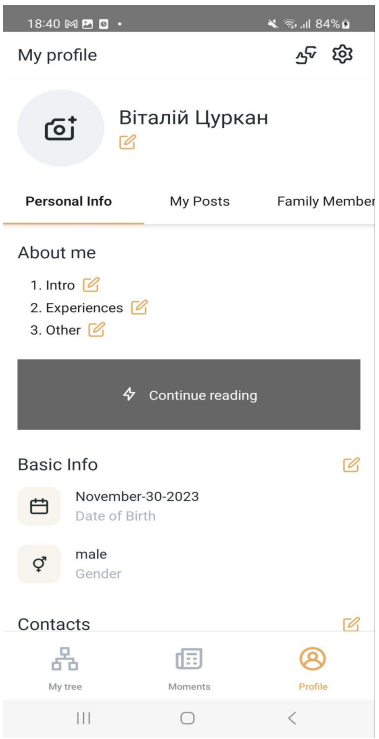

Рисунок 3.4.5.1 – Екран профілю користувача з персональною інформацією.

Натиснувши на іконку під власним ім’ям користувач потрапить на екран де зможе відредагувати власне ім’я та задати або змінити фото профілю. Оновлені ім’я та фото профілю також буде відображено на сімейному дереві Рисунок 3.4.5.3.

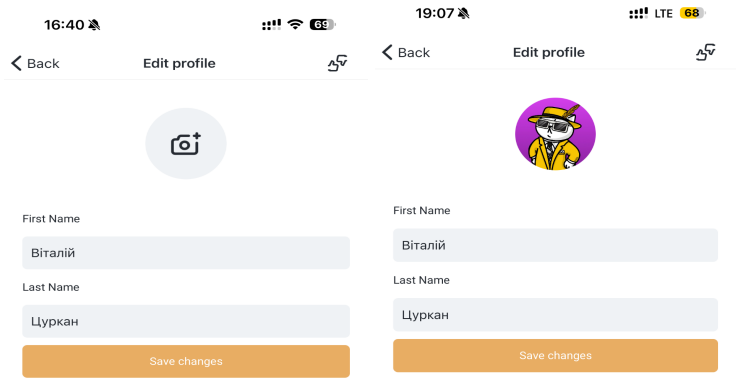

Рисунок 3.4.5.2 – Екран редагування імені та фото профілю.

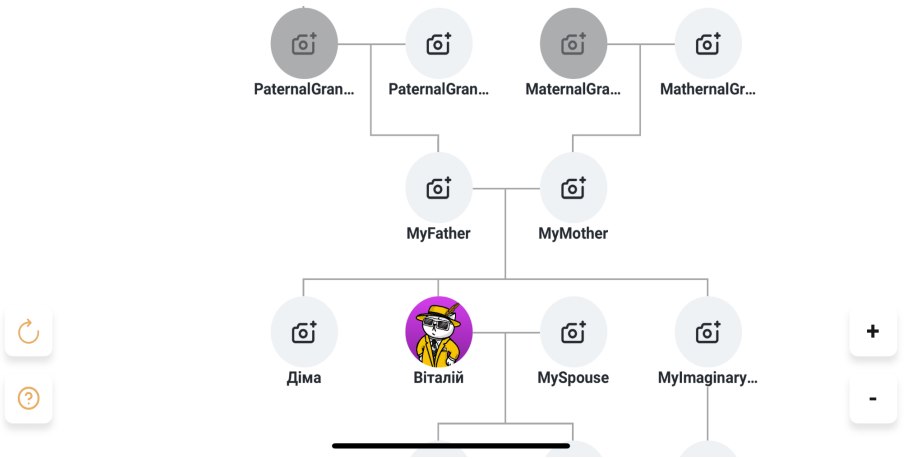

Рисунок 3.4.5.3 – Змінене фото профілю на сімейному дереві.

Перейшовши в розділ “Contacts” користувач може переглянути свою електронну поштову адресу та задати/змінити номер телефону Рисунок 3.4.5.4. У випадку якщо користувач введе номер телефону але не збереже його натиснувши кнопку “Save changes” і спробує повернутися на попередній екран, то побачить модальне вікно з пропозицією зберегти зміни, або повернутися на попередній екран втративши їх. Даний принцип валідації застовано на всіх екранах де доступна можливість редагування.

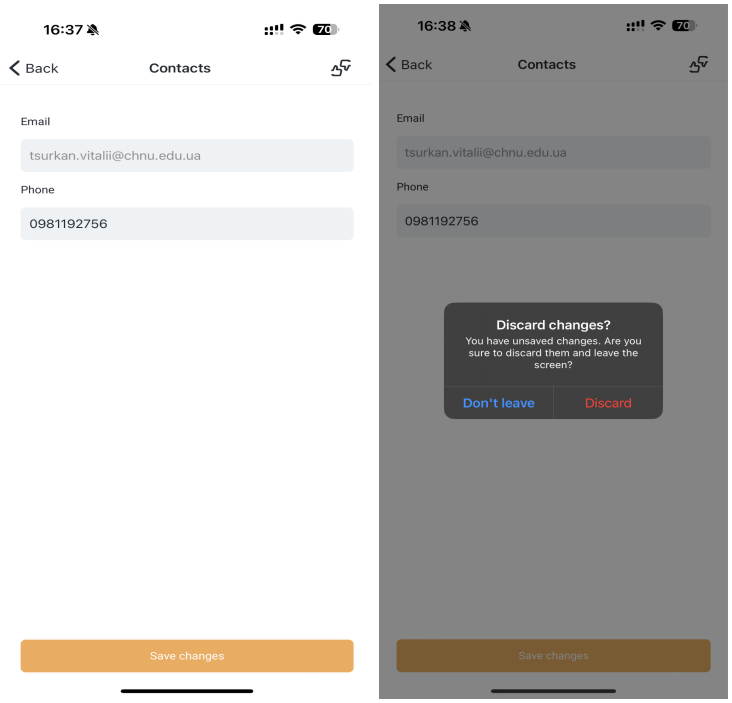

Рисунок 3.4.5.4 – Екран редагування контактної інформації.

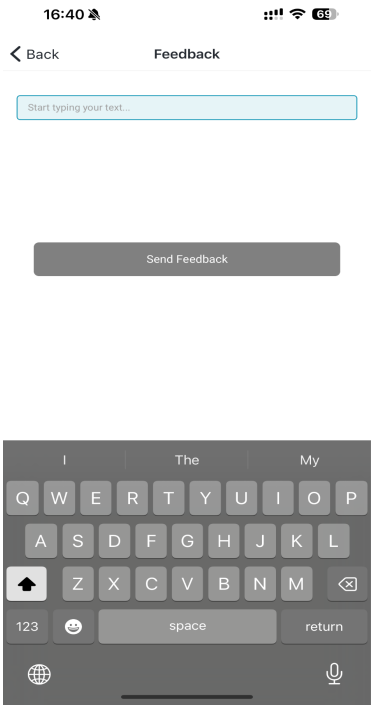

Рисунок 3.4.5.5 – Екран відправки відгуку.

Перейшовши на екран розділу About me користувач отримує можливість ввести інформацію про себе використовуючи текст, фото та відео Рисунок 3.4.5.6. Для зручності інформацію про користувача розділена на 3 ідентичні секції “About me”, “Experience” та “Others”.

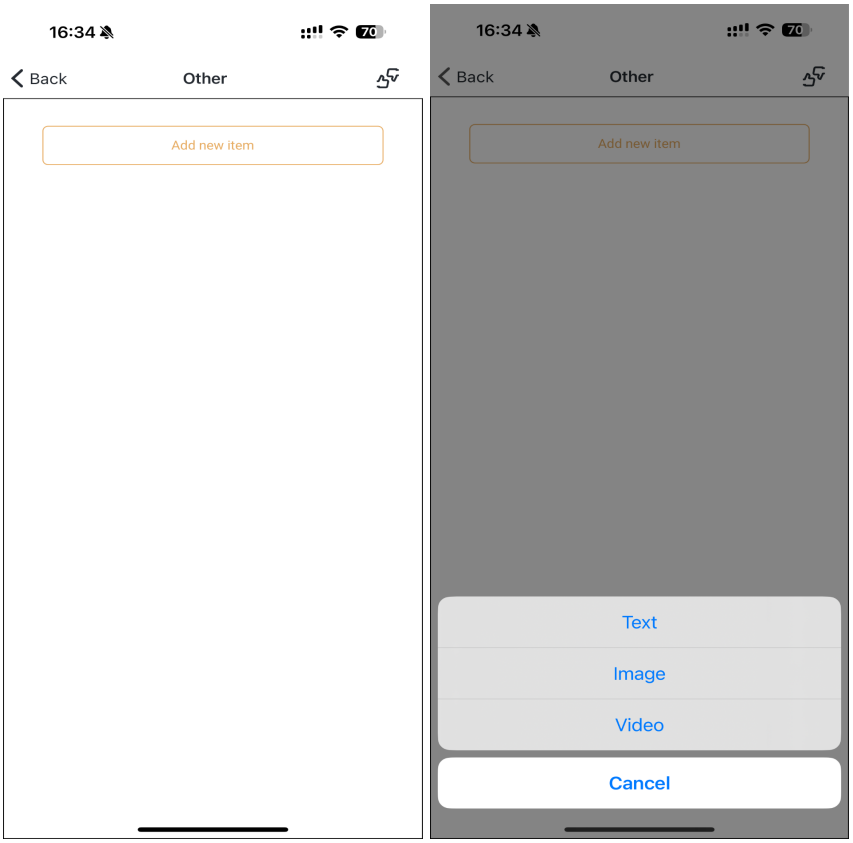

Рисунок 3.4.5.6 – Екран редагування інформації About me.

Перше введене речення та перше завантажене фото буде, крім екрану перегляду розділу About me буде доступне на екрані профілю користувача Рисунок 3.4.5.7.

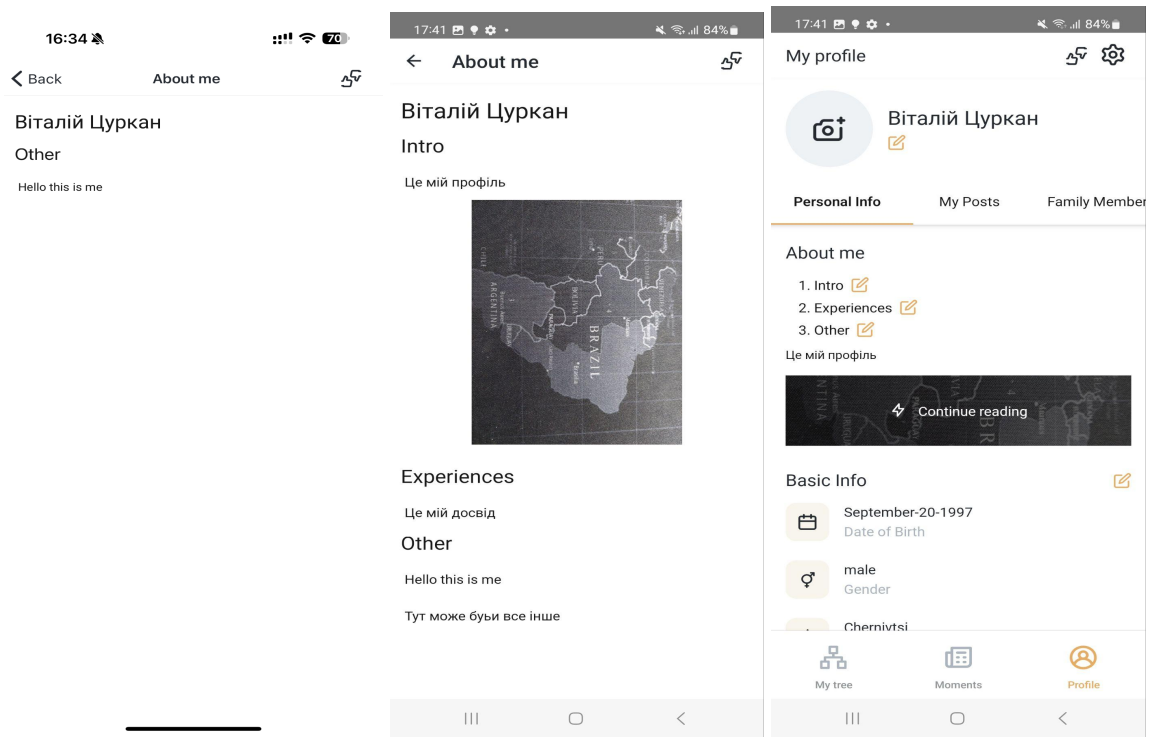

Рисунок 3.4.5.7 – Екран перегляду розділу інформації About me та відображення інформації з розділу About me на головному екрані профілю користувача.

Для внесення змін базової інформації “Basic info” користувачем здійснюється перехід на окремий екран редагування Рисунок 3.4.5.8, де є можливість задати зміни, або внести нову інформацію про адресу, роботу, та домашніх улюбленців. Внісши інформацію користувач обов’язково повинен зберегти внесені зміни натиснувши кнопку “Save changes”.

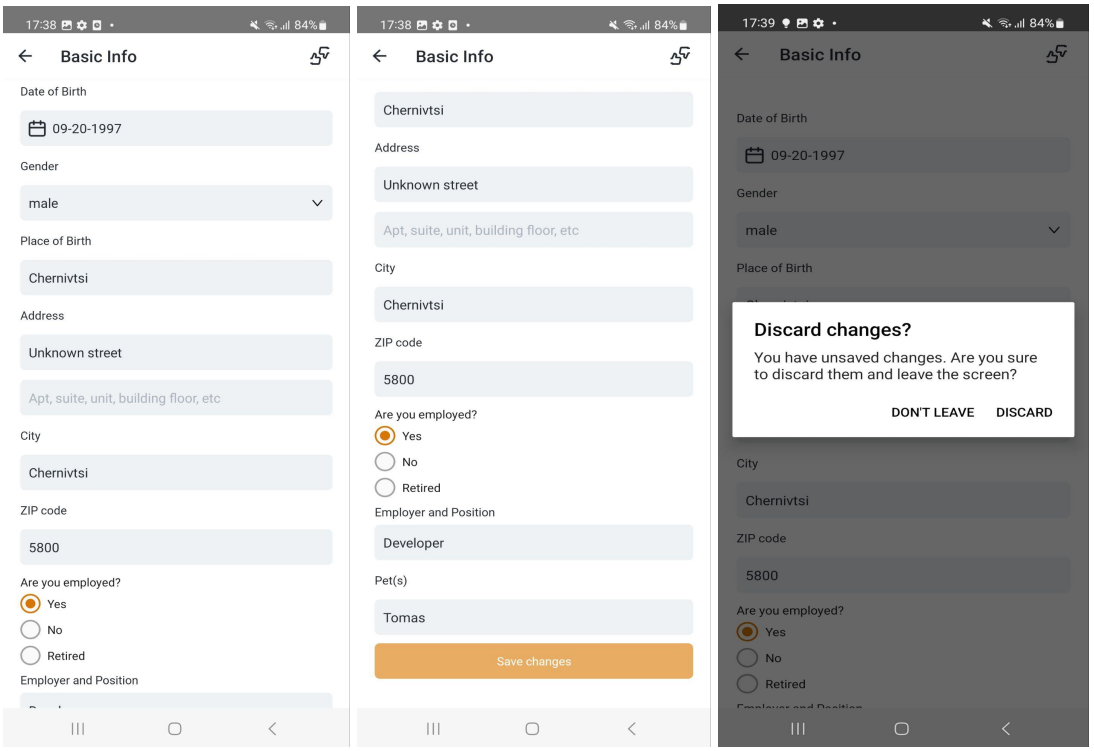

Рисунок 3.4.5.8 – Екран редагування розділу базової інформації Basic Info.

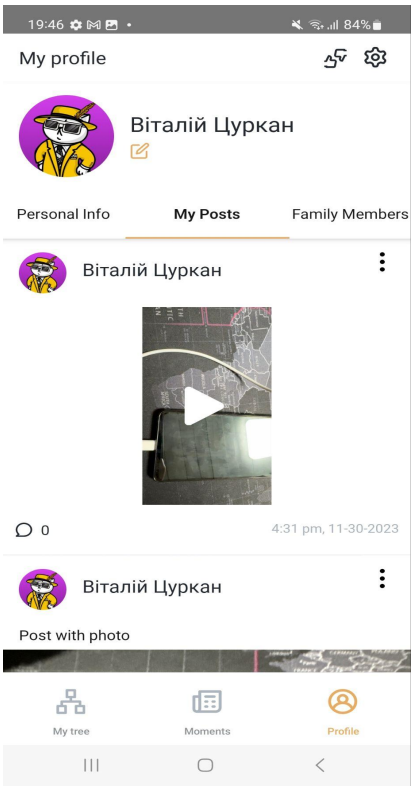

Рисунок 3.4.5.9 – Вкладка “My Posts” на головному екрані користувача.

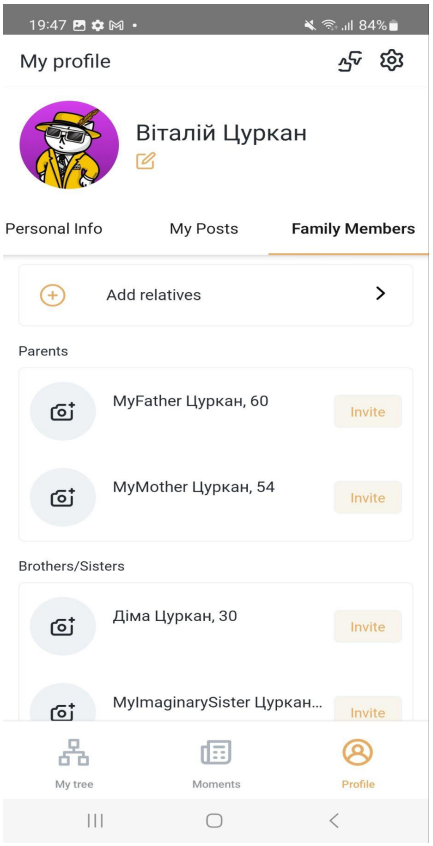

Рисунок 3.4.5.10 – Вкладка “Family Members” на головному екрані користувача.

Для виходу з додатку користувачу необхідно натиснути на крайню праву кнопку в хедері, яка перенаправить користувача на екран з доступною кнопкою “Log Out” Рисунок 3.4.5.11, натискання якої призведе до виходу з застосунку на екран автентифікації Рисунок 3.4.1.1.

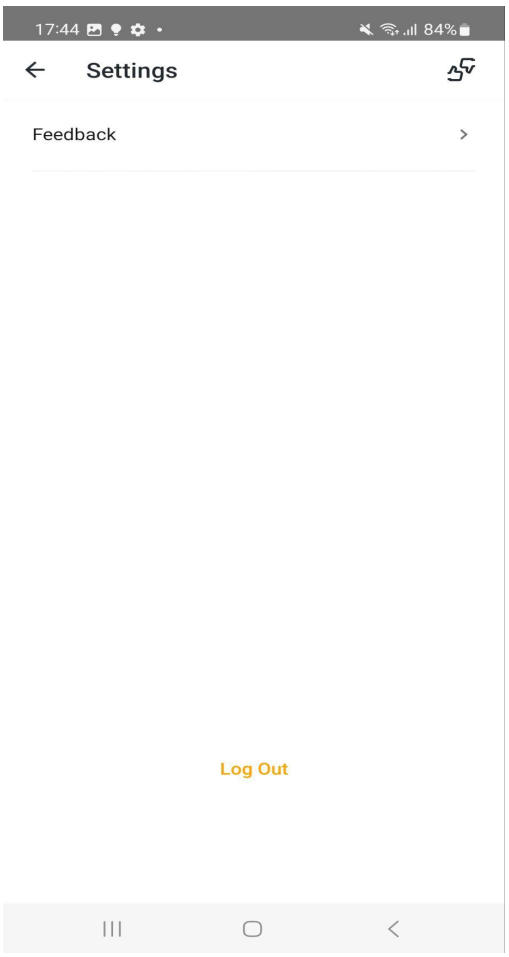

Рисунок 3.4.5.11 – Екран виходу з застосунку.

Висновки

У ході виконання дипломної роботи було розглянуто та здійснено розробку мобільного застосунку соціальної мережі сімейних зв’язків з використанням Node.js та React-Native. Було проведено детальний аналіз існуючих соціальних мереж та додатків для організації сімейних зв'язків.

При проведенні аналізу було виділено переваги, недоліки та функціональні особливості кожного застосунку, що надало підґрунтя для розробки нового продукту. На основі результатів аналізу було визначено ключові завдання для розробки соціальної мережі сімейних зв’язків.

Сформульовано чіткі та конкретні цілі проєкту, враховуючи технічні вимоги та очікувані переваги та особливості для користувачів соціальної мережі сімейних зв’язків, а саме організація та візуалізація сімейних зв’язків у вигляді деревоподібної структури, можливість додавати (та за необхідності видаляти) нових користувачів із дерева, особистий профіль користувача та можливість його редагування, запрошення нових користувачів доєднатися до дерева а також базовий функціонал будь-якого застосунку соціальної мережі — простота автентифікації та можливість створювати, редагувати та видаляти пости з різноманітним контентом, таким як тестові повідомлення, фото та відео контент, та можливість залишати коментарі під постами.

Для реалізації поставленого завдання було використано оптимальний стек технологій, що включає Nest.js на базі Node.js, React-Native та мови програмування TypeScript для забезпечення ефективної і надійної розробки та високоякісної взаємодії з користувачем. При розробці архітектури серверної частини застосунку, було обрано та використано підхід предметно-орієнтованого проєктування для створення ефективної та гнучкої архітектури. При розробці мобільної частини додатку, було

розглянуто та реалізовано архітектурні переваги використання Redux для керування станом додатка, а також впроваджено Redux-Toolkit для більш ефективного управління станом та виконання асинхронних операцій. Вибір технологій та інструментів базувався на вимогах проекту та доцільності їхнього застосуваня для розроблюваного проекту.

За результатами роботи вдалося створити ергономічний застосунок високопродуктивної та зручну соціальну мережу сімейних зв'язків. Розроблений мобільний застосунок повністю відповідає закладеним вимогам. Імплементовані функції, оптимальний стек технологій та підхід проєктно-орієнтованого проектування вказують на високий потенціал для подальшого розвитку та впровадження нового функціоналу даного програмного продукту.

Список використаної літератури

1. React-Native Documentation. [Онлайн]. Доступно: https://reactnative.dev/docs/
2. Node.js Documentation. [Онлайн]. Доступно: https://nodejs.org/en/docs
3. Nest.js Documentation. [Онлайн]. Доступно: https://docs.nestjs.com/
4. Neo4j Documentation. [Онлайн]. Доступно: https://neo4j.com/docs/
5. Nader Dabit, React native in action : Mainings, 2020. 320 сторінок, ISBN : 978-1242753471.
6. Джон Сміт. (2020). "Розробка мобільних додатків за допомогою React-Native." Journal of Mobile Development, 15(2), 123-145.
7. Boyd, D., & Ellison, N. (2008). "Social network sites: Definition, history, and scholarship." Journal of Computer-Mediated Communication, 13(1), 210-230.
8. Redux-Toolkit Documentation. [Онлайн]. Доступно: https://redux-toolkit.js.org/api
9. React-Redux Documentation. [Онлайн]. Доступно: https://react-redux.js.org/api/hooks
10. Redux-Saga Documentation. [Онлайн]. Доступно: https://redux-saga.js.org/docs/api
11. Firebase Documentation. [Онлайн]. Доступно: https://firebase.google.com/docs
12. React-Native-Firebase Documentation. [Онлайн]. Доступно: https://rnfirebase.io/
13. Expo SDK Documentation. [Онлайн]. Доступно: https://docs.expo.dev/
14. TypeScript Documentation. [Онлайн]. Доступно: https://www.typescriptlang.org/docs/
15. Майк Кантелон, та Натан Райліх, та Марк Гартер Node.js in Action, 3 edition, Manning, 2019 року, 392 сторінки, ISBN-78 1778961923.

16. Mario Casciaro, Luciano Mammino Node.js Design Patterns, 3 edition, Packt Publishing, 2020 року, 660 сторінки, ISBN-13: 978-1839214110.

Додатки

Файл **app.module.ts**

```
import { NodemailerService } from "./services/email/provider/nodemailer.service";
import { Module } from "@nestjs/common";
import { AppController } from "./app.controller";
import { AppService } from "./app.service";
import { TreeController } from "./domain/tree/tree.controller";
import { TreeService } from "./domain/tree/tree.service";
import { UserController } from "./domain/user/user.controller";
import { UserService } from "./domain/user/user.service";
import { ConfigModule, ConfigService } from "@nestjs/config";
import { createDatabaseConfiguration,  DatabaseModule, } from
"./services/database/database.module";
import { DatabaseCypherModule } from
"./services/database/database.cypher.module";
import configuration from "src/services/config/env.config";
import { DatabaseConfig } from "./services/database/database.service";
import { UserRepository } from "./domain/user/user.repository";
import { TreeRepository } from "./domain/tree/tree.repository";
import { FileService } from "./domain/file/file.service";
import { FileController } from "./domain/file/file.controller";
import { PostController } from "./domain/post/post.controller";
import { PostService } from "./domain/post/post.service";
import { PostRepository } from "./domain/post/post.repository";
import { CommentController } from "./domain/comment/comment.controller";
import { CommentRepository } from "./domain/comment/comment.repository";
import { CommentService } from "./domain/comment/comment.service";
import { FirebaseAuthStrategy } from
"./services/auth/firebase/firebase-auth.strategy";
import { EmailController } from "./services/email/email.controller";
import { EmailService } from "./services/email/email.service";
import { EmailProvider } from "./services/email/email.interface";
@Module({
 imports: [
  ConfigModule.forRoot({
   envFilePath: `src/env/.env.${process.env.NODE_ENV || "local"}`,
   isGlobal: true,
   load: [configuration],
  }),
  DatabaseCypherModule.forRootAsync({
   imports: [ConfigModule],
   inject: [ConfigService],
   useFactory: async (
    configService: ConfigService
   ): Promise<DatabaseConfig> => {
    return createDatabaseConfiguration(configService);
   },
  }),
  DatabaseModule.forRootAsync({
   imports: [ConfigModule],
   inject: [ConfigService],
   useFactory: (configService: ConfigService): DatabaseConfig =>
    createDatabaseConfiguration(configService),
  }),
 ],
 controllers: [
  AppController,
  UserController,
  TreeController,
  FileController,
  PostController,
  CommentController,
  EmailController,
 ],
 providers: [
  NodemailerService,
  AppService,
  FirebaseAuthStrategy,
  UserService,
  UserRepository,
  TreeService,
  TreeRepository,
  FileService,
  PostService,
  PostRepository,
  CommentService,
  CommentRepository,
```

```
  EmailService,
  {
    provide: EmailProvider,
    useClass: NodemailerService,
  },
 ],
})
export class AppModule {}
```

Файл **main.ts**

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';

async function bootstrap() {
 const app = await NestFactory.create(AppModule);

 const config = new DocumentBuilder()
  .setTitle('Family Links API')
  .setDescription('API for the Family Links social network project')
  .setVersion('1.0')
  .build();

 const document = SwaggerModule.createDocument(app, config);
 SwaggerModule.setup('api', app, document);

 await app.listen(process.env.PORT || 3000);
}
bootstrap();
```

Файл **utils.repository.ts**

```
import { Injectable, Scope } from "@nestjs/common";
import { ConfigService } from "@nestjs/config";
@Injectable({ scope: Scope.REQUEST })
export class UtilsRepository {
 constructor(private readonly configService: ConfigService) {}
 static getStringVersion(param): string {
  return typeof param === "string" ? param : JSON.stringify(param);
 }
}
```

Файл **user.controller.spec.ts**

```
import { Test, TestingModule } from '@nestjs/testing';
import { UserController } from './user.controller';
import { UserService } from './user.service';

describe('UserController', () => {
 let controller: UserController;
 beforeEach(async () => {
  const module: TestingModule = await Test.createTestingModule({
   controllers: [UserController],
   providers: [UserService],
  }).compile();

  controller = module.get<UserController>(UserController);
 });

 it('should be defined', () => {
  expect(controller).toBeDefined();
 });
});
```

Файл **user.controller.ts**

```
import {
 Controller,
 Get,
 Post,
 Body,
 Patch,
 Param,
 Delete,
 UseGuards,
 UploadedFile,
 UseInterceptors,
} from "@nestjs/common";
import { UserService } from "./user.service";
import { CreateUserDto } from "./dto/create-user.dto";
import { UpdateUserDto } from "./dto/update-user.dto";
import { FirebaseAuthGuard } from "src/services/auth/firebase/firebase-auth.guard";

@Controller("user")
export class UserController {
```

```
constructor(private readonly userService: UserService) {}

@Post("add")
create(@Body() createUserDto: CreateUserDto) {
  return this.userService.create(createUserDto);
}

@Get()
findAll() {
  return this.userService.findAll();
}

@Get(":id")
findOne(@Param("id") id: string) {
  return this.userService.findOne(+id);
}

@Get("email/:email")
findOneByEmail(@Param("email") email: string) {
  return this.userService.findOneByEmail(email);
}

@Get("initUser/:token")
initUser(@Param("token") jwt: string) {
  return this.userService.initUser(jwt);
}

@Patch(":id")
update(@Param("id") id: string, @Body() updateUserDto: UpdateUserDto) {
  return this.userService.update(+id, updateUserDto);
}

@Delete("remove/:id")
remove(@Param("id") id: string) {
  return this.userService.remove(+id);
}
}
```

Файл **user.module.ts**

```
import { Module } from '@nestjs/common';
import { UserService } from './user.service';
import { UserController } from './user.controller';

@Module({
 controllers: [UserController],
 providers: [UserService]
})
export class UserModule {}
```

Файл **user.module.ts**

```
import { BadRequestException, Inject, Injectable, Scope } from "@nestjs/common";
import { Connection } from "cypher-query-builder";
import * as cypher from "src/services/database/repository.utils";
import { ConfigService } from "@nestjs/config";
import {
 CUSTOM_ERROR_MESSAGE,
 DATABASE_CONNECTION,
} from "src/services/database/database.constants";
import { User } from "./entities/user.entity";
import { UtilsRepository } from "src/utils/utils.repository";

@Injectable({ scope: Scope.REQUEST })
export class UserRepository {
 constructor(
  @Inject(DATABASE_CONNECTION) private readonly connection: Connection,
  private readonly configService: ConfigService
 ) {}
 private query(): cypher.RepositoryQuery {
  return new cypher.RepositoryQuery(this.connection);
 }

 async addNewUser(userData: any): Promise<User> {
  const result = await this.query()
   .createEntity<{ [key in keyof Partial<User>]: any }>("User", userData)
   .commitWithReturnEntity();

  if (result) {
   const data = result.data;

   return {
```

```
      id: data.User.identity,
      ...data.User.properties,
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }

 async deleteUser(id: number): Promise<any> {
  const result = await this.query()
    .deleteEntityById("User", id)
    .commitWithReturnEntity();
  if (result) {
    const data = result.data;
    return {
      response: "done",
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }

 async getUserEntity(id: number): Promise<any> {
  const result = await this.query()
    .fetchUserByUserId(id)
    .commitWithReturnEntity();
  const parent = await this.query()
    .fetchUserByUserId(id)
    .resolveUsersParentsByRelation(
      result.data.User.properties.myTreeIdByParent1
    )
    .commitWithReturnEntities();
  let parents = [];
  if (parent && parent?.length && parent[0].data.UserP) {
    parents.push(parent[0].data.UserP);
    if (parent[0].data.UserM) {
      parents.push(parent[0].data.UserM);
    }
  }
  const spouses = await this.query()
   .fetchUserByUserId(id)
   .resolveUsersSpouseByRelationByTreeId(
    result.data.User.properties.myTreeIdByParent1
   )
   .commitWithReturnEntities();
  let spouse = [];
  if (spouses && spouses?.length && spouses[0].data.UserS) {
   spouse.push(spouses[0].data.UserS);
  }

  let siblings = [];
  if (parent && parent?.length && parent[0].data.UserP) {
   const siblingsArr = await this.query()
    .fetchUserByUserId(parent[0].data.UserP.identity)
    .resolveUsersChildrenByRelation(
     parent[0].data.UserP.properties.myTreeIdByParent1
    )
    .commitWithReturnEntities();

   if (siblingsArr && siblingsArr?.length && siblingsArr[0].data.UserKList) {
    let famalyLine = siblingsArr[0].data.UserKList;
    if (famalyLine.length > 1) {
     famalyLine = famalyLine.filter((object) => {
      return object.identity != id;
     });
     siblings = famalyLine;
    }
   }
  }

  const childrens = await this.query()
   .fetchUserByUserId(id)
   .resolveUsersChildrenByRelation(
    result.data.User.properties.myTreeIdByParent1
   )
   .commitWithReturnEntities();

  let kids = [];
  if (childrens && childrens.length && childrens[0].data.UserKList) {
```

```
    kids = childrens[0].data.UserKList;
  }

  if (
    !childrens[0].data.UserKList.length &&
    spouses &&
    spouses[0].data.UserS
  ) {
    const spouseChildrens = await this.query()
      .fetchUserByUserId(spouses[0].data.UserS.identity)
      .resolveUsersChildrenByRelation(
        spouses[0].data.UserS.properties.myTreeIdByParent1
      )
      .commitWithReturnEntities();

    if (
      spouseChildrens &&
      spouseChildrens.length &&
      spouseChildrens[0].data.UserKList
    ) {
      kids = spouseChildrens[0].data.UserKList;
    }
  }

  if (result) {
    const data = result.data;
    data.User.properties.parents = parents.length ? parents : null;
    data.User.properties.siblings = siblings.length ? siblings : null;
    data.User.properties.spouse = spouse.length ? spouse : null;
    data.User.properties.kids = kids.length ? kids : null;
    return {
      id: data.User.identity,
      ...data.User.properties,
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
}

async updateUserEntity(userId, userParams): Promise<any> {
  const params = userParams;
  // @ts-ignore
  let result= null;
  const user = await this.query()
  .fetchUserByUserId(userId)
  .commitWithReturnEntity();
  if(params && params.isInvitedUser && user &&
user.data.User.properties.isActivated) {
    return {
      response: {
        isActivated: true
      },
    };
  } else{
  result = await this.query()
  .fetchUserByUserId(userId)
  .updateEntity(
    "User",
    Object.entries({
      "User.userPictureLink": params?.userPictureLink,
      "User.userPictureKey": params?.userPictureKey,
      "User.firstName": params?.firstName,
      "User.maidenName": params?.maidenName,
      "User.lastName": params?.lastName,
      "User.introduction": params?.introduction
        ? UtilsRepository.getStringVersion(params?.introduction)
        : null,
      "User.setting": params?.setting
        ? UtilsRepository.getStringVersion(params?.setting)
        : null,
      "User.birthdate": params?.birthdate,
      "User.dateOfDeath": params?.dateOfDeath,
      "User.anniversaryDate": params?.anniversaryDate,
      "User.isDeceased": params?.isDeceased,
      "User.isActivated": params?.isActivated,
      "User.gender": params?.gender,
      "User.hometown": params?.hometown,
      "User.homeCountry": params?.homeCountry,
      "User.email": params?.email,
      "User.phone": params?.phone,
```

```
    "User.address": params?.address,
    "User.spouseTreeId": params?.spouseTreeId,
    "User.myTreeIdByParent1": params?.myTreeIdByParent1,
    "User.myTreeIdByParent2": params?.myTreeIdByParent2,
    "User.storageFolderId": params?.storageFolderId,
    "User.spouse": params?.spouse
      ? UtilsRepository.getStringVersion(params?.spouse)
      : null,
    "User.kids": params?.kids
      ? UtilsRepository.getStringVersion(params?.kids)
      : null,
    "User.pets": params?.pets,
    "User.bornAddress": params?.bornAddress
      ? UtilsRepository.getStringVersion(params?.bornAddress)
      : null,
    "User.parents": params?.parents
      ? UtilsRepository.getStringVersion(params?.parents)
      : null,
    "User.siblings": params?.siblings
      ? UtilsRepository.getStringVersion(params?.siblings)
      : null,
    "User.socialNetworks": params?.socialNetworks
      ? UtilsRepository.getStringVersion(params?.socialNetworks)
      : null,
    "User.employerAndPosition": params?.employerAndPosition,
    "User.education": params?.education
      ? UtilsRepository.getStringVersion(params?.education)
      : null,
  }).reduce((valuesAcc, [key, value]) => {
    return value !== undefined && value !== null
      ? {
        ...valuesAcc,
        [key]: value,
      }
      : valuesAcc;
  }, {})
 )
 .commitWithReturnEntity();
}
```

```
 if (result !== null) {
  const data = result.data;
  return {
   id: data.User.identity,
   ...data.User.properties,
  };
 }
 throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
}
public async getUserFromEmail(email: string): Promise<User> {
 const result = await this.query()
  .findUserByEmail(email)
  .commitWithReturnEntity();

 if (result) {
  const data = result.data;
  return {
   id: data.User.identity,
   ...data.User.properties,
  };
 }
}

public async getUsersWithStorageFileId(): Promise<User[]> {
 const result = await this.query()
  .raw(
   `MATCH(User:User)
 WHERE User.storageFolderId IS NOT NULL
 RETURN COLLECT(User{.*, id: id(User)}) as Users`
  )
  .commit();

 if (result[0]?.Users) {
  return result[0]?.Users as User[];
 }
 return null;
}
}
```

Файл **user.service.spec.ts**

```
import { Test, TestingModule } from '@nestjs/testing';
import { UserService } from './user.service';
describe('UserService', () => {
 let service: UserService;
 beforeEach(async () => {
  const module: TestingModule = await Test.createTestingModule({
   providers: [UserService],
  }).compile();

  service = module.get<UserService>(UserService);
 });

 it('should be defined', () => {
  expect(service).toBeDefined();
 });
});
```

Файл **user.service.ts**

```
import { Injectable } from "@nestjs/common";
import { FirebaseAuthStrategy } from
"src/services/auth/firebase/firebase-auth.strategy";
import { UpdateUserDto } from "./dto/update-user.dto";
import { User } from "./entities/user.entity";
import { UserRepository } from "./user.repository";

@Injectable()
export class UserService {
 constructor(
  private readonly firebaseService: FirebaseAuthStrategy,
  private readonly userRepository: UserRepository
 ) {}

 async create(usersProperties): Promise<User> {
  const result = await this.userRepository.addNewUser(usersProperties);
  return result;
 }

 findAll() {
  return `This action returns all user`;
 }

 async findOne(id: number): Promise<User> {
  const result = await this.userRepository.getUserEntity(id);
  return result;
 }

 async findOneByEmail(email: string): Promise<User> {
  const result = await this.userRepository.getUserFromEmail(email);
  return result;
 }

 async update(id: number, updateUserDto: UpdateUserDto): Promise<User> {
  const result = await this.userRepository.updateUserEntity(
   id,
   updateUserDto
  );
  return result;
 }
 async remove(id: number) {
  const result = await this.userRepository.deleteUser(id);
  return result;
 }
 public async initUser(jwt: string) {
  const jwtParsed = await this.firebaseService.validate(jwt);
  const result = await this.userRepository.getUserFromEmail(jwtParsed.email);
  return result;
 }
}
```

Файл **tree.service.ts**

```
import { Injectable } from "@nestjs/common";
import { CreateTreeDto } from "./dto/create-tree.dto";
import { UpdateTreeDto } from "./dto/update-tree.dto";
import { JoinToTreeDto } from "./dto/join-to-tree.dto";
import { TreeRepository } from "./tree.repository";
import { Tree } from "./entities/tree.entity";

@Injectable()
export class TreeService {
 constructor(private readonly treeRepository: TreeRepository) {}
```

```
async create(treeProperties: CreateTreeDto): Promise<Tree> {
  const result = await this.treeRepository.addNewTree(treeProperties);
  return result;
}

findAll() {
  return `This action returns all tree`;
}

async findOne(id: number) {
  const result = await this.treeRepository.getTree(id);
  return result;
}

async findOneByUUID(id: string) {
  const result = await this.treeRepository.getTreeByUUID(id);
  return result;
}

async getTreeMembers(id: number) {
  const result = await this.treeRepository.getTreeMembers(id);
  return result;
}

async getPartTreeByUserId(treeId: number, userId: number) {
  const result = await this.treeRepository.getPartTreeByUserId(
    treeId,
    userId
  );
  return result;
}

async getTreeInPartsUserId(treeId: number, userId: string) {
  const result = await this.treeRepository.getTreeInPartsUserIdNew(
    treeId,
    userId
  );
  return result;
}

async update(id: number, updateTreeDto: UpdateTreeDto) {
  const result = await this.treeRepository.updateTreeEntity(
    id,
    updateTreeDto
  );
  return result;
}

async join(id: number, joinToTreeProperty: JoinToTreeDto) {
  let result = null;
  if (joinToTreeProperty.relation == "MARRIED") {
    result = await this.treeRepository.joinToTreeMarried(
      id,
      joinToTreeProperty
    );
  }

  if (joinToTreeProperty.relation == "DESCENDANT") {
    result = await this.treeRepository.joinToTreeDescendant(
      id,
      joinToTreeProperty
    );
  }

  if (joinToTreeProperty.relation == "MARRIEDSUBTREE") {
    result = await this.treeRepository.joinToTreeMarriedSubTree(
      id,
      joinToTreeProperty
    );
  }
  return result;
}

async remove(id: number) {
  const result = await this.treeRepository.removeTree(id);
  return result;
}
```

```
 async removeUserFromTree(id: number) {
  const result = await this.treeRepository.removeUserFromTree(id);
  return result;
 }
}
```

Файл **tree.repository.ts**

```
import { BadRequestException, Inject, Injectable, Scope } from "@nestjs/common";
import { Connection } from "cypher-query-builder";
import * as cypher from "src/services/database/repository.utils";
import { ConfigService } from "@nestjs/config";
import {
 CUSTOM_ERROR_MESSAGE,
 DATABASE_CONNECTION,
} from "src/services/database/database.constants";
import { Tree } from "./entities/tree.entity";
import { UtilsRepository } from "src/utils/utils.repository";
import { HusbandTreeUserType } from "./tree.types";

@Injectable({ scope: Scope.REQUEST })
export class TreeRepository {
 constructor(
  @Inject(DATABASE_CONNECTION) private readonly connection: Connection,
  private readonly configService: ConfigService
 ) {}
 private query(): cypher.RepositoryQuery {
  return new cypher.RepositoryQuery(this.connection);
 }

 async addNewTree(treeData: { name: string; userId: number }): Promise<Tree> {
  const result = await this.query()
   .createEntity<{ [key in keyof Partial<Tree>]: any }>(
    "Tree",
    { name: treeData.name },
    true
   )
   .commitWithReturnEntity();

  await this.query()
   .createMemberRelation(treeData.userId, result.data.Tree.identity)
   .commitWithReturnEntity();

  await this.updateUserParamTreeOwner(treeData.userId);

  await this.query()
   .fetchUserByUserId(treeData.userId)
   .updateEntity(
    "User",
    Object.entries({
     "User.myTreeIdByParent1": UtilsRepository.getStringVersion(
      result.data.Tree.identity
     ),
    }).reduce((valuesAcc, [key, value]) => {
     return value !== undefined && value !== null
      ? {
        ...valuesAcc,
        [key]: value,
       }
      : valuesAcc;
    }, {})
   )
   .commitWithReturnEntity();

  if (result) {
   const data = result.data;
   return {
    id: data.Tree.identity,
    ...data.Tree.properties,
   };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }

 async getTreeMembers(id: any): Promise<Tree[]> {
  const result = await this.query()
   .fetchAllByEntityId(id, "Tree")
   .commitWithReturnEntity();

  if (result) {
```

```
    const data = result.data;
    return {
      id: data.Tree.identity,
      ...data.Tree.properties,
      treeMembers: data.nList,
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
}

async getTree(id: any): Promise<Tree[]> {
  const result = await this.query()
    .fetchAllByEntityId(id, "Tree")
    .commitWithReturnEntity();

  if (result) {
    const data = result.data;
    const tree = cypher.buildTree(data, id);
    return {
      id: data.Tree.identity,
      ...data.Tree.properties,
      tree: tree[0],
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
}

async removeUserFromTree(id: number): Promise<any> {
  const result = await this.query()
    .deleteUserFromTree("User", id)
    .commitWithReturnEntity();
  if (result) {
    return {
      response: "done",
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
}

async removeTree(id: number): Promise<any> {
  const result = await this.query()
    .deleteEntityById("Tree", id)
    .commitWithReturnEntity();
  if (result) {
    return {
      response: "done",
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
}

async getTreeByUUID(id: any): Promise<Tree[]> {
  const result = await this.query()
    .fetchAllByEntityUUUID(id, "Tree")
    .commitWithReturnEntity();

  if (result) {
    const data = result.data;
    const tree = cypher.buildTree(data, data.Tree.identity);
    return {
      id: data.Tree.identity,
      ...data.Tree.properties,
      tree: tree[0],
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
}

async getPartTreeByUserId(treeId: number, userId: number): Promise<Tree[]> {
  const spouses = await this.query()
    .fetchUserByUserId(userId)
    .resolveUsersSpouseByRelationByTreeId(treeId.toString())
    .commitWithReturnEntities();
  let spouseId = null;
  if (spouses && spouses?.length && spouses[0].data.UserS) {
    spouseId = spouses[0].data.UserS.identity;
```

```
  }

  const result = await this.query()
    .fetchAllByEntityId(treeId, "Tree")
    .commitWithReturnEntity();

  if (result) {
    const data = result.data;
    const tree = cypher.buildPartTree(data, spouseId, treeId.toString());
    return {
      id: data.Tree.identity,
      ...data.Tree.properties,
      tree: tree[0],
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }

 async getDataFromTree(treeId: number, userId: string) {
  const parent = await this.query()
    .fetchUserByUserId(+userId)
    .resolveUsersParentsByRelation(treeId.toString())
    .commitWithReturnEntities();

  let parentId = null;
  if (parent && parent?.length && parent[0].data.UserP) {
    parentId = parent[0].data.UserP.identity;
  }

  const resultForSubTree = await this.query()
    .fetchAllByEntityId(treeId, "Tree")
    .commitWithReturnEntity();
  const result = await this.getOverwriteTreeForSpouse(treeId);

  if (parentId === null) {
    parentId = userId;
  }

  const spouses = await this.query()
    .fetchUserByUserId(+parentId)
    .resolveUsersSpouseByRelationByTreeId(treeId.toString())
    .commitWithReturnEntities();
  let spouseId = null;
  if (spouses && spouses?.length && spouses[0].data.UserS) {
    spouseId = spouses[0].data.UserS.identity;
  }

  const currentSubTree = await this.getMarriedSubTreeUsersByUserId(spouseId);
  return { result, resultForSubTree, currentSubTree, parentId, spouseId };
 }

 async getOverwriteTreeForSpouse(bottomTreeId) {
  const resForBottomTree = await this.query()
    .raw(
      `MATCH(Tree:Tree)
  WHERE id(Tree)=${bottomTreeId}
  OPTIONAL MATCH(tree)<-[:TREE_MEMBER_USER]-(user:User)
  OPTIONAL MATCH(user)-[rList*1]-(nList)
  RETURN {Tree:Tree,rList:collect(distinct rList),nList:collect(distinct nList)} as
data
  `
    )
    .commit();

  resForBottomTree[0].data.nList = resForBottomTree[0].data.nList.map(
    (user) => {
      let res = {
        ...user,
        properties: {
          ...user.properties,
          myTreeIdByParent1: bottomTreeId,
          myTreeIdByParent2: bottomTreeId,
        },
      };
      if (user.properties.spouseTreeId) {
        res = {
          ...res,
          properties: { ...user.properties, spouseTreeId: bottomTreeId },
        };
```

```
    }
    return res;
  }
 );

 resForBottomTree[0].data.rList = resForBottomTree[0].data.rList.map(
  (rel) => {
   return rel.map((item) => ({
    ...item,
    label: item.label.replace(/([\d]+)/, bottomTreeId),
   }));
  }
 );

 return resForBottomTree[0];
}

async getDataFromTreeSpouseNoSubtree(
 treeId: number,
 userId: string,
 spouseRels
) {
 const topParentId = userId;
 const topSpouseId = null;
 const topTreeId = treeId;
 const bottomParentId = spouseRels[0].CurrentSpouseMarriedTo.identity;
 const bottomSpouseId = userId;
 const bottomTreeId =
  spouseRels[0].CurrentSpouseMarriedTo.properties.myTreeIdByParent1;

 const result = await this.query()
  .fetchAllByEntityId(treeId, "Tree")
  .commitWithReturnEntity();

 const resForBottomTree = await this.getOverwriteTreeForSpouse(bottomTreeId);

 const currentSubTree = await this.getMarriedSubTreeUsersByUserId(
  topSpouseId
 );

 return {
  result,
  currentSubTree,
  topParentId,
  topSpouseId,
  topTreeId,
  bottomParentId,
  bottomSpouseId,
  bottomTreeId,
  resForBottomTree,
 };
}

async getDataFromTreeSpouseWithSubtree(
 treeId: number,
 userId: string,
 spouseRels
) {
 let topParentId: string;
 let topSpouseId;

 const topTreeId = treeId;
 const bottomParentId = spouseRels[0].CurrentSpouseMarriedTo.identity;
 const bottomSpouseId = userId;
 const bottomTreeId =
  spouseRels[0].CurrentSpouseMarriedTo.properties.myTreeIdByParent1;

 const parent = await this.query()
  .fetchUserByUserId(+userId)
  .resolveUsersParentsByRelation(topTreeId.toString())
  .commitWithReturnEntities();

 if (parent && parent?.length && parent[0].data.UserP) {
  topParentId = parent[0].data.UserP.identity;
 }

 const result = await this.query()
  .fetchAllByEntityId(topTreeId, "Tree")
```

```
    .commitWithReturnEntity();

  const resForBottomTree = await this.getOverwriteTreeForSpouse(bottomTreeId);

  const spouses = await this.query()
    .fetchUserByUserId(+topParentId)
    .resolveUsersSpouseByRelationByTreeId(topTreeId.toString())
    .commitWithReturnEntities();

  if (
    spouses &&
    spouses?.length &&
    (spouses[0].data.UserS || spouses[0].data.User)
  ) {
    topSpouseId =
      spouses[0]?.data?.UserS?.identity || spouses[0].data.User.identity;
  }

  const currentSubTree = await this.getMarriedSubTreeUsersByUserId(
    topSpouseId
  );

  return {
    result,
    resForBottomTree,
    currentSubTree,
    topParentId,
    topSpouseId,
    topTreeId,
    bottomParentId,
    bottomSpouseId,
    bottomTreeId,
  };
}

async getTreeInPartsUserIdNew(treeId: number, userId: string): Promise<any> {
  const userDataRaw = await this.query()
    .findEntityById("User", +userId)
    .commitWithReturnEntities();

  const userData = userDataRaw[0].data.User;

  const myTreeIdByParent1 = userData?.properties?.myTreeIdByParent1;

  treeId = myTreeIdByParent1;

  const spouseTreeId = userData?.properties?.spouseTreeId;

  if (myTreeIdByParent1 && spouseTreeId) {
    var spouseRels = await this.query()
      .findEntityById("User", +userId)
      .raw(
        `MATCH(User:User)
      WHERE id(User)=${userId}
      OPTIONAL
MATCH(User)-[:USER_MARRIED_SUB_TREE_USER_TREE_${spouseTreeId}]-
>(Parent:User)<-[:USER_DESCENDANT_USER_TREE_${myTreeIdByParent1}]-(
User)
          WITH User, Parent
          OPTIONAL
MATCH(User)-[:USER_MARRIED_USER_TREE_${spouseTreeId}]-(CurrentSpous
eMarriedTo:User)
          RETURN User, Parent, CurrentSpouseMarriedTo`
      )
      .commit();
  }

  const firstSpouse = spouseRels?.length > 0 && spouseRels[0];

  if (firstSpouse?.Parent?.identity) {
    var userType = HusbandTreeUserType.SPOUSE_WITH_SUBTREE;
  } else if (firstSpouse?.CurrentSpouseMarriedTo?.identity) {
    var userType = HusbandTreeUserType.SPOUSE_NO_SUBTREE;
  } else {
    var userType = HusbandTreeUserType.HUSBAND;
  }

  switch (userType) {
    case HusbandTreeUserType.HUSBAND:
      var { currentSubTree, parentId, result, resultForSubTree, spouseId } =
        await this.getDataFromTree(treeId, userId);
```

```
if (result) {
  const data = result.data;
  const rootUserForTopData = await this.query()
    .findEntityById("User", +parentId)
    .resolveHusbandTreeUntillEnd(treeId.toString())
    .commit();

  const rootUser = rootUserForTopData[0]?.RootParent;

  if (rootUser) {
    var rootPartTreeUser = [rootUser];
  } else {
    const res = await this.query()
      .findEntityById("User", +parentId)
      .commitWithReturnEntities();
    var rootPartTreeUser = [res[0]?.data.User];
  }

  const bottomPartTree = await cypher.buildPartTreeWithoutSubTreeRel(
    data,
    parentId,
    treeId.toString()
  );

  if (!bottomPartTree[0][0]?.user) {
    bottomPartTree[0][0].user = rootPartTreeUser[0];
  }
  const rootPartTree = await cypher.buildRootPartTree(
    data,
    parentId,
    treeId.toString(),
    currentSubTree,
    rootPartTreeUser
  );
  if (!rootPartTree[0][0]?.identity && !rootPartTree[0][0]?.user) {
    rootPartTree[0][0] = rootPartTreeUser[0];
  }
  const subTree = await cypher.buildSubTree(
    resultForSubTree.data,
    treeId.toString(),
    spouseId,
    currentSubTree
  );
  return {
    id: data.Tree.identity,
    ...data.Tree.properties,
    rootPartTree: rootPartTree ? rootPartTree[0] : null,
    subTree: subTree ? subTree[0] : null,
    bottomPartTree: bottomPartTree ? bottomPartTree[0] : null,
  };
}
break;

case HusbandTreeUserType.SPOUSE_NO_SUBTREE:
  var {
    result,
    currentSubTree,
    topParentId,
    topSpouseId,
    topTreeId,
    bottomParentId,
    bottomSpouseId,
    bottomTreeId,
    resForBottomTree,
  } = await this.getDataFromTreeSpouseNoSubtree(
    treeId,
    userId,
    spouseRels
  );

  if (result) {
    const data = result.data;
    const [[bottomPartTreeInit]] =
      await cypher.buildPartTreeWithoutSubTreeRel(
        resForBottomTree.data,
        bottomParentId,
        bottomTreeId.toString()
      );
```

```
    const married = [...bottomPartTreeInit.married];
    const user = { ...bottomPartTreeInit.user };

    const bottomPartTree = [
      { ...bottomPartTreeInit, user: married[0], married: [user] },
    ];

    const rootPartTree = await cypher.buildRootPartTree(
      data,
      topParentId,
      topTreeId.toString(),
      currentSubTree,
      null
    );

    return {
      id: data.Tree.identity,
      ...data.Tree.properties,
      rootPartTree: rootPartTree ? rootPartTree[0] : null,
      subTree: null,
      bottomPartTree: bottomPartTree || null,
    };
  }
  break;

case HusbandTreeUserType.SPOUSE_WITH_SUBTREE:
  var {
    result,
    resForBottomTree,
    currentSubTree,
    topParentId,
    topSpouseId,
    topTreeId,
    bottomParentId,
    bottomSpouseId,
    bottomTreeId,
  } = await this.getDataFromTreeSpouseWithSubtree(
    treeId,
    userId,
    spouseRels
  );

  if (result) {
    const data = result.data;

    const [[bottomPartTreeInit]] =
      await cypher.buildPartTreeWithoutSubTreeRel(
        resForBottomTree.data,
        topParentId,
        bottomTreeId.toString()
      );

    const rootPartTree = await cypher.buildRootPartTree(
      data,
      topParentId,
      topTreeId.toString(),
      currentSubTree
    );

    const subTree = await cypher.buildSubTree(
      data,
      topTreeId.toString(),
      topSpouseId,
      currentSubTree
    );

    return {
      id: data.Tree.identity,
      ...data.Tree.properties,
      rootPartTree: rootPartTree ? rootPartTree[0] : null,
      subTree: subTree ? subTree[0] : null,
      bottomPartTree: bottomPartTreeInit ? [bottomPartTreeInit] : null,
    };
  }
  break;

default:
```

```
      break;
    }
  }

  async getTreeInPartsUserId(treeId: number, userId: string): Promise<any> {
    const { currentSubTree, parentId, result, spouseId } =
      await this.getDataFromTree(treeId, userId);

    if (result) {
      const data = result.data;
      const bottomPartTree = await cypher.buildPartTreeWithoutSubTreeRel(
        data,
        parentId,
        treeId.toString()
      );
      const rootPartTree = await cypher.buildRootPartTree(
        data,
        spouseId || parentId,
        treeId.toString(),
        currentSubTree
      );
      const subTree = await cypher.buildSubTree(
        data,
        treeId.toString(),
        spouseId,
        currentSubTree
      );
      return {
        id: data.Tree.identity,
        ...data.Tree.properties,
        rootPartTree: rootPartTree ? rootPartTree[0] : null,
        subTree: subTree ? subTree[0] : null,
        bottomPartTree: bottomPartTree ? bottomPartTree[0] : null,
      };
    }
    throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
  }

  async updateTreeEntity(treeId, treeParams): Promise<any> {
    const params = treeParams;
    // @ts-ignore
    const result = await this.query()
      .findEntityById("Tree", treeId)
      .updateEntity(
        "Tree",
        Object.entries({
          "Tree.name": params.name,
        }).reduce((valuesAcc, [key, value]) => {
          return value !== undefined && value !== null
            ? {
                ...valuesAcc,
                [key]: value,
              }
            : valuesAcc;
        }, {})
      )
      .commitWithReturnEntity();

    if (result !== null) {
      const data = result.data;
      return {
        id: data.Tree.identity,
        ...data.Tree.properties,
      };
    }
    throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
  }

  async joinToTreeDescendant(id: number, treeProperties: any): Promise<any> {
    const [{ Tree: treeGoingDown }] = await this.query()
      .fetchUserByUserId(treeProperties.toUserId)
      .raw(
        `OPTIONAL
MATCH(User)-[TreeUserRelations:TREE_MEMBER_USER]->(Tree:Tree)
      WHERE id(Tree)=${id}`
      )
      .customReturn(`Tree`)
      .commit();
```

```
  if (treeGoingDown?.identity) {
    var childId = treeProperties.userId;
  }

  if (childId) {
    const parentData = await this.query()
      .fetchUserByUserId(treeProperties.toUserId)
      .commitWithReturnEntity();

    var husbandTreeId = parentData?.data?.User?.properties?.spouseTreeId;

    if (husbandTreeId) {
      const [
        {
          Husband: { identity, properties },
        },
      ] = await this.query()
        .fetchUserByUserId(treeProperties.toUserId)
        .raw(
          `OPTIONAL
MATCH(User)-[:USER_MARRIED_USER_TREE_${husbandTreeId}]->(Husband:
User)`
        )
        .customReturn("Husband")
        .commit();

      var husband = { id: identity, ...properties };
      treeProperties.toUserId = husband.id;
      id = husbandTreeId;
    }
  }
  const spouses = await this.query()
    .fetchUserByUserId(treeProperties.toUserId)
    .resolveUsersSpouseByRelationByTreeId(id.toString())
    .commitWithReturnEntities();
  let spouseId = null;
  if (spouses && spouses?.length && spouses[0].data.UserS) {
    spouseId = spouses[0].data.UserS.identity;
  }

  const arrayOfSpouses = await this.getArrayOfSpouses(
    treeProperties.toUserId,
    id,
    spouseId
  );
  if (arrayOfSpouses?.length) {
    for (let item of arrayOfSpouses) {
      await this.joinUserToTreeDescendantParent2(
        treeProperties.userId,
        treeProperties.toUserId,
        item.data.UserS.properties.myTreeIdByParent1
      );
    }
  }

  if (spouseId) {
    const childrenCurrentParent = await this.query()
      .fetchUserByUserId(+treeProperties.toUserId)
      .resolveUsersChildrenByRelation(UtilsRepository.getStringVersion(id))
      .commitWithReturnEntities();

    let kidsCurrent;
    kidsCurrent = childrenCurrentParent[0].data.UserKList;

    const childrenMarriedParent = await this.query()
      .fetchUserByUserId(+treeProperties.userId)
      .resolveUsersChildrenByRelation(
        spouses[0].data.UserS.properties.myTreeIdByParent1
      )
      .commitWithReturnEntities();

    if (!spouses[0].data.UserS.properties.myTreeIdByParent1) {
      const spouseTree = await this.addNewTree({
        name: spouses[0].data.UserS.properties.firstName,
        userId: +spouseId,
      });

      await this.query()
```

```
  .createMemberAndMarriedRelations(
    treeProperties.toUserId,
    spouseId,
    spouseTree["id"]
  )
  .commitWithReturnEntity();
await this.updateUserParamSpouseTreeId(spouseId, id);
await this.updateUserParamSpouseTreeId(
  treeProperties.toUserId,
  spouseTree["id"]
);

await this.joinUserToTreeDescendantParent2(
  +treeProperties.userId,
  +spouseId,
  +spouseTree["id"]
);
if (childrenCurrentParent?.length && spouseTree["id"]) {
  if (kidsCurrent?.length == 1) {
    await this.joinUserToTreeDescendantParent2(
      +kidsCurrent[0].identity,
      +spouseId,
      spouseTree["id"]
    );
  }

  if (kidsCurrent?.length > 1) {
    for (let item of kidsCurrent) {
      await this.joinUserToTreeDescendantParent2(
        +item.identity,
        +spouseId,
        spouseTree["id"]
      );
    }
  }
 }
} else {

 let spouseInMarriedTree;
 let userInMarriedTree;
 spouseInMarriedTree = await this.query()
  .fetchUserInTree(
    spouseId,
    spouses[0].data.UserS.properties.myTreeIdByParent1
  )
  .commitWithReturnEntity();
 if (!spouseInMarriedTree) {
  await this.query()
    .createMemberAndMarriedRelations(
      treeProperties.toUserId,
      spouseId,
      spouses[0].data.UserS.properties.myTreeIdByParent1
    )
    .commitWithReturnEntity();
  await this.updateUserParamSpouseTreeId(spouseId, id);
 }
 if (kidsCurrent && kidsCurrent?.length) {
  userInMarriedTree = await this.query()
    .fetchUserInTree(
      +kidsCurrent[0].identity,
      spouses[0].data.UserS.properties.myTreeIdByParent1
    )
    .commitWithReturnEntity();
 }
 if (!userInMarriedTree && kidsCurrent?.length) {
  await this.joinUserToTreeDescendantParent2(
    +kidsCurrent[0].identity,
    +spouseId,
    spouses[0].data.UserS.properties.myTreeIdByParent1
  );
 }

 await this.joinUserToTreeDescendantParent2(
  +treeProperties.userId,
  +spouseId,
  +spouses[0].data.UserS.properties.myTreeIdByParent1
 );
}
```

```
  }


  const result = await this.query()
    .createMemberAndDescendantRelations(
      treeProperties.userId,
      treeProperties.toUserId,
      id
    )
    .commitWithReturnEntity();


  await this.updateUserParamMyTreeIdByParent1(treeProperties.toUserId, id);
  await this.query()
    .fetchUserByUserId(treeProperties.userId)
    .updateEntity(
      "User",
      Object.entries({
        "User.myTreeIdByParent1": UtilsRepository.getStringVersion(id),
      }).reduce((valuesAcc, [key, value]) => {
        return value !== undefined && value !== null
          ? {
              ...valuesAcc,
              [key]: value,
            }
          : valuesAcc;
      }, {})
    )
    .commitWithReturnEntity();


  // [KT] add child to every tree
  if (childId) {
    const [vals] = await this.query()
      .raw(
        `MATCH(Tree:Tree)
        WHERE id(Tree) = ${id}
        OPTIONAL MATCH(Tree)<-[:TREE_MEMBER_USER]-(User:User)
        WHERE User.myTreeIdByParent1 <> "${id}"


        WITH COLLECT(DISTINCT toInteger(User.myTreeIdByParent1)) as
treeIdsToAttach
        RETURN treeIdsToAttach
        `
      )
      .commit();


    const { treeIdsToAttach } = vals;


    if (treeIdsToAttach?.length > 0) {
      await this.query()
        .raw(
          `OPTIONAL MATCH (Tree:Tree) WHERE id(Tree) IN [${treeIdsToAttach}]
WITH Tree MATCH (User:User) WHERE id(User)=${childId} WITH User, Tree
MERGE (Tree)<-[:TREE_MEMBER_USER]-(User) RETURN Tree, User`
        )
        .commit();
    }
  } else {
    await this.query()
      .raw(
        `MATCH (User:User) WHERE ID(User) = ${treeProperties.toUserId}
    MATCH (Tree:Tree) WHERE ID(Tree) = ${id}
    MERGE (User)-[TreeUserRelations:TREE_MEMBER_USER]->(Tree)
  RETURN {} as data`
      )
      .commit();
  }


  if (result) {
    const output = result.data;
    return {
      response: "done",
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }


 async joinToTreeMarried(id: number, treeProperties: any): Promise<any> {
  const spouseId = treeProperties.userId;
  const husbandId = treeProperties.toUserId;


  const husbandUser = await this.query()
```

```
  .fetchUserByUserId(+husbandId)
  .commitWithReturnEntity();

const marriedUser = await this.query()
.fetchUserByUserId(+treeProperties.userId)
.commitWithReturnEntity();

let spouseTree = null;
if(marriedUser && !marriedUser.data.User.properties.myTreeIdByParent1){
  spouseTree = await this.addNewTree({
    name: `SPOUSE_TREE_${spouseId}`,
    userId: spouseId,
  });

  await this.query()
  .createMemberAndMarriedRelations(
    treeProperties.toUserId,
    treeProperties.userId,
    spouseTree.id
  )
  .commitWithReturnEntity();
}

if(marriedUser && marriedUser.data.User.properties.myTreeIdByParent1){
  spouseTree = marriedUser.data.User.properties.myTreeIdByParent1
  await this.query()
  .createMemberAndMarriedRelations(
    treeProperties.toUserId,
    treeProperties.userId,
    marriedUser.data.User.properties.myTreeIdByParent1
  )
  .commitWithReturnEntity();
}

id = husbandUser.data.User.properties.myTreeIdByParent1;

const result = await this.query()
  .createMemberAndMarriedRelations(
    treeProperties.userId,
    husbandId,
    id
  )
  .commitWithReturnEntity();

const spouseUser = await this.query()
  .fetchUserByUserId(+spouseId)
  .commitWithReturnEntity();

if (husbandUser && husbandUser?.data?.User?.properties.subTreeTargetUser) {
  const subTreeUser = await this.query()
    .fetchUserByUserId(husbandUser.data.User.properties.subTreeTargetUser)
    .commitWithReturnEntity();

  if (subTreeUser && subTreeUser.data.User.properties.myTreeIdByParent1) {
    await this.updateUserParamMyTreeIdByParent1(
      husbandId,
      subTreeUser.data.User.properties.myTreeIdByParent1
    );
    await this.query()
      .createMemberAndMarriedRelations(
        spouseId,
        husbandId,
        subTreeUser.data.User.properties.myTreeIdByParent1
      )
      .commitWithReturnEntity();
  }

  if (subTreeUser && subTreeUser?.data?.User?.properties.spouseTreeId) {
    await this.query()
      .createMemberAndMarriedRelations(
        spouseId,
        husbandId,
        subTreeUser.data.User.properties.spouseTreeId
      )
      .commitWithReturnEntity();
  }
}
const childrenMarriedParent = await this.query()
```

```
    .fetchUserByUserId(+spouseId)
    .resolveUsersChildrenByRelation(
      spouseUser.data.User.properties.myTreeIdByParent1
    )
    .commitWithReturnEntities();

  let kidsCurrent;
  let kidsMarried;
  const childrenCurrentParent = await this.query()
    .fetchUserByUserId(+husbandId)
    .resolveUsersChildrenByRelation(UtilsRepository.getStringVersion(id))
    .commitWithReturnEntities();

  const descendantsCurrentParent = await this.query()
    .fetchUserByUserId(+husbandId)
    .raw(
      `OPTIONAL
MATCH(User)-[:TREE_MEMBER_USER]->(Tree:Tree)<-[:TREE_MEMBER_USE
R]-(Kids:User)
  RETURN COLLECT(DISTINCT ID(Kids)) as data
  `
    )
    .commit();

  const husbandTreeMembers = descendantsCurrentParent[0]?.data;
  if (husbandTreeMembers) {
    await this.query()
      .raw(
        `MATCH(User:User)
    WHERE ID(User) IN [${husbandTreeMembers}]
    WITH User
    MATCH(Tree:Tree)
    WHERE ID(Tree)=${spouseTree.id}
    WITH User, Tree
    MERGE(Tree)<-[:TREE_MEMBER_USER]-(User)
    RETURN Tree, User`
      )
      .commit();
  } else {
    const descendantsCurrentParent = await this.query()
      .fetchUserByUserId(+husbandId)
      .resolveUsersChildrenByRelationUtilEnd(
        UtilsRepository.getStringVersion(id)
      )
      .commitWithReturnEntities();

    kidsCurrent = descendantsCurrentParent[0]?.data?.UserKList;
    if (kidsCurrent?.length > 0) {
      for (let item of kidsCurrent) {
        await this.query()
          .createMemberRelation(
            item.identity,
            spouseUser.data.User.properties.myTreeIdByParent1
          )
          .commitWithReturnEntity();
      }
    }
  }
  if (
    childrenCurrentParent?.length &&
    !childrenMarriedParent?.length &&
    spouseUser.data.User.properties.myTreeIdByParent1
  ) {
    kidsCurrent = childrenCurrentParent[0].data.UserKList;
    if (kidsCurrent?.length == 1) {
      await this.joinUserToTreeDescendantParent2(
        +kidsCurrent[0].identity,
        +spouseId,
        spouseUser.data.User.properties.myTreeIdByParent1
      );
    }

    if (kidsCurrent?.length > 1) {
      for (let item of kidsCurrent) {
        await this.joinUserToTreeDescendantParent2(
          +item.identity,
          +spouseId,
          spouseUser.data.User.properties.myTreeIdByParent1
        );
```

```
    }
  }
}

if (childrenMarriedParent?.length && !childrenCurrentParent?.length) {
  kidsMarried = childrenMarriedParent[0].data.UserKList;

  if (kidsMarried?.length == 1) {
    await this.joinUserToTreeDescendantParent2(
      +kidsMarried[0].identity,
      husbandId,
      id
    );

    if (kidsMarried?.length > 1) {
      for (let item of kidsCurrent) {
        await this.joinUserToTreeDescendantParent2(
          +item.identity,
          husbandId,
          id
        );
      }
    }
  }
}

if (childrenMarriedParent?.length && childrenCurrentParent?.length) {
  kidsMarried = childrenMarriedParent[0].data.UserKList;
  kidsCurrent = childrenCurrentParent[0].data.UserKList;
  const diffKidsForP = kidsMarried.filter(
    (e) => !kidsCurrent.find((a) => e.firstName == a.firstName)
  );
  const diffKidsForM = kidsCurrent.filter(
    (e) => !kidsMarried.find((a) => e.firstName == a.firstName)
  );
  for (let item of diffKidsForP) {
    await this.joinUserToTreeDescendantParent2(
      +item.identity,
      +husbandId,
      id
    );
  }
  for (let item of diffKidsForM) {
    if (spouseUser.data.User.properties.myTreeIdByParent1) {
      await this.joinUserToTreeDescendantParent2(
        +item.identity,
        +spouseId,
        spouseUser.data.User.properties.myTreeIdByParent1
      );
    }
  }
}

await this.query()
  .fetchUserByUserId(spouseId)
  .updateEntity(
    "User",
    Object.entries({
      "User.spouseTreeId": UtilsRepository.getStringVersion(id),
    }).reduce((valuesAcc, [key, value]) => {
      return value !== undefined && value !== null
        ? {
            ...valuesAcc,
            [key]: value,
          }
        : valuesAcc;
    }, {})
  )
  .commitWithReturnEntity();

if (result) {
  const output = result.data;
  return {
    response: "done",
  };
}
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
}
```

```
async joinToTreeMarriedSubTree(
  id: number,
  treeProperties: any
): Promise<any> {

  const result = await this.query()
  .createMemberAndMarriedSubTreeRelations(
    treeProperties.userId,
    treeProperties.toUserId,
    id
  )
  .commitWithReturnEntity();

  const targetUser = await this.query()
    .fetchUserByUserId(treeProperties.userId)
    .commitWithReturnEntities();
  const targetUserProps = targetUser[0]?.data?.User?.properties;

  if (targetUserProps.spouseTreeId) {
    id = targetUserProps.spouseTreeId;
  }

  await this.updateUserParamMyTreeIdByParent2(treeProperties.userId, id);

  const resultToSubTreeUser = await this.query()
    .fetchUserByUserId(treeProperties.toUserId)
    .commitWithReturnEntities();
  if (
    !targetUserProps?.myTreeIdByParent1 ||
    +targetUserProps?.myTreeIdByParent1 == id
  ) {
    const targetUserTree = await this.addNewTree({
      name: `treeName${treeProperties.userId}`,
      userId: +treeProperties.userId,
    });
    const spouses = await this.query()
      .fetchUserByUserId(treeProperties.userId)
      .resolveUsersSpouseByRelationByTreeId(UtilsRepository.getStringVersion(id))
      .commitWithReturnEntities();

  let spouseId = null;
  if (spouses && spouses?.length && spouses[0].data.UserS) {
    spouseId = spouses[0].data.UserS.identity;
  }

  if (spouseId) {
    await this.query()
      .createMemberAndMarriedRelations(
        spouses[0].data.UserS.identity,
        treeProperties.userId,
        +targetUserTree["id"]
      )
      .commitWithReturnEntity();

    const childrenCurrentParent = await this.query()
      .fetchUserByUserId(spouses[0].data.UserS.identity)
      .resolveUsersChildrenByRelation(UtilsRepository.getStringVersion(id))
      .commitWithReturnEntities();

    let kidsCurrent;

    if (childrenCurrentParent?.length) {
      kidsCurrent = childrenCurrentParent[0].data.UserKList;
      if (kidsCurrent?.length == 1) {
        await this.joinUserToTreeDescendantParent2(
          +kidsCurrent[0].identity,
          +treeProperties.userId,
          +targetUserTree["id"]
        );
      }

      if (kidsCurrent?.length > 1) {
        for (let item of kidsCurrent) {
          await this.joinUserToTreeDescendantParent2(
            +item.identity,
            +treeProperties.userId,
            +targetUserTree["id"]
```

```
        );
      }
    }
  }
}

await this.updateUserParamTreeOwner(treeProperties.userId);
await this.updateUserParamMyTreeIdByParent1(
  treeProperties.toUserId,
  +targetUserTree["id"]
);
await this.updateUserParamMyTreeIdByParent1(
  treeProperties.userId,
  +targetUserTree["id"]
);
await this.joinUserToTreeDescendantParent1(
  treeProperties.userId,
  treeProperties.toUserId,
  +targetUserTree["id"]
);
} else {
await this.updateUserParamMyTreeIdByParent1(
  treeProperties.toUserId,
  +targetUserProps?.myTreeIdByParent1
);
await this.joinUserToTreeDescendantParent1(
  treeProperties.userId,
  treeProperties.toUserId,
  +targetUserProps?.myTreeIdByParent1
);

await this.query()
  .createMemberRelation(
    treeProperties.toUserId,
    +targetUserProps?.myTreeIdByParent1
  )
  .commitWithReturnEntity();

await this.updateUserParamMyTreeIdByParent1(
  treeProperties.userId,
  +targetUserProps?.myTreeIdByParent1
);
}

let ToSubTreeUser = treeProperties.userId;

if (
  resultToSubTreeUser.length &&
  resultToSubTreeUser[0].data.User.properties.subTreeTargetUser
) {
  ToSubTreeUser =
    resultToSubTreeUser[0].data.User.properties.subTreeTargetUser;
}
await this.updateUserParamSubTreeTargetUser(
  treeProperties.toUserId,
  ToSubTreeUser
);

await this.query()
  .createMemberRelation(treeProperties.userId, id)
  .commitWithReturnEntity();

if (result) {
  const output = result.data;
  return {
    response: "done",
  };
}
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
}

async joinUserToTreeDescendantParent2(
  userId: number,
  toUserId: number,
  treeId: number
): Promise<any> {
  // >>DO ALL IN 1 QUERY
  await this.query()
```

```
    .createMemberAndDescendantRelations(userId, toUserId, treeId)
    .commitWithReturnEntity();

  await this.query()
    .createMemberRelation(userId, treeId)
    .commitWithReturnEntity();

  await this.query()
    .fetchUserByUserId(userId)
    .updateEntity(
      "User",
      Object.entries({
        "User.myTreeIdByParent2": UtilsRepository.getStringVersion(treeId),
      }).reduce((valuesAcc, [key, value]) => {
        return value !== undefined && value !== null
          ? {
              ...valuesAcc,
              [key]: value,
            }
          : valuesAcc;
      }, {})
    )
    .commitWithReturnEntity();
}

async joinUserToTreeDescendantParent1(
  userId,
  toUserId,
  treeId
): Promise<any> {
  await this.query()
    .createMemberAndDescendantRelations(userId, toUserId, treeId)
    .commitWithReturnEntity();

  await this.query()
    .createMemberRelation(userId, treeId)
    .commitWithReturnEntity();

  await this.query()
    .fetchUserByUserId(userId)
    .updateEntity(
      "User",
      Object.entries({
        "User.myTreeIdByParent1": UtilsRepository.getStringVersion(treeId),
      }).reduce((valuesAcc, [key, value]) => {
        return value !== undefined && value !== null
          ? {
              ...valuesAcc,
              [key]: value,
            }
          : valuesAcc;
      }, {})
    )
    .commitWithReturnEntity();
}

async updateUserParamSpouseTreeId(userId, treeId): Promise<any> {
  await this.query()
    .fetchUserByUserId(userId)
    .updateEntity(
      "User",
      Object.entries({
        "User.spouseTreeId": UtilsRepository.getStringVersion(treeId),
      }).reduce((valuesAcc, [key, value]) => {
        return value !== undefined && value !== null
          ? {
              ...valuesAcc,
              [key]: value,
            }
          : valuesAcc;
      }, {})
    )
    .commitWithReturnEntity();
}
async updateUserParamMyTreeIdByParent1(userId, treeId): Promise<any> {
  await this.query()
    .fetchUserByUserId(userId)
    .updateEntity(
```

```
      "User",
      Object.entries({
        "User.myTreeIdByParent1": UtilsRepository.getStringVersion(treeId),
      }).reduce((valuesAcc, [key, value]) => {
        return value !== undefined && value !== null
          ? {
              ...valuesAcc,
              [key]: value,
            }
          : valuesAcc;
      }, {})
    )
    .commitWithReturnEntity();
}
async updateUserParamMyTreeIdByParent2(userId, treeId): Promise<any> {
  await this.query()
    .fetchUserByUserId(userId)
    .updateEntity(
      "User",
      Object.entries({
        "User.myTreeIdByParent2": UtilsRepository.getStringVersion(treeId),
      }).reduce((valuesAcc, [key, value]) => {
        return value !== undefined && value !== null
          ? {
              ...valuesAcc,
              [key]: value,
            }
          : valuesAcc;
      }, {})
    )
    .commitWithReturnEntity();
}
async updateUserParamSubTreeTargetUser(userId, targetUserId): Promise<any> {
  await this.query()
    .fetchUserByUserId(userId)
    .updateEntity(
      "User",
      Object.entries({
        "User.subTreeTargetUser": +targetUserId,
      }).reduce((valuesAcc, [key, value]) => {
        return value !== undefined && value !== null
          ? {
              ...valuesAcc,
              [key]: value,
            }
          : valuesAcc;
      }, {})
    )
    .commitWithReturnEntity();
}

async updateUserParamTreeOwner(userId): Promise<any> {
  await this.query()
    .fetchUserByUserId(userId)
    .updateEntity(
      "User",
      Object.entries({
        "User.treeOwner": true,
      }).reduce((valuesAcc, [key, value]) => {
        return value !== undefined && value !== null
          ? {
              ...valuesAcc,
              [key]: value,
            }
          : valuesAcc;
      }, {})
    )
    .commitWithReturnEntity();
}

async getArrayOfSpouses(userId, treeId, spouseId): Promise<any> {
  let result = await this.getRecursiveSpouses(userId, treeId);
  if (spouseId) {
    result?.filter((object) => {
      return object.identity !== spouseId;
    });
  }
```

```
  // to each wife's personal tree add child -> userId
  return result || [];
 }

async getRecursiveSpouses(userId: number, treeId: string): Promise<any> {
  let result = [];

  const parent = await this.query()
    .fetchUserByUserId(userId)
    .resolveUsersParentsByRelation(treeId)
    .commitWithReturnEntities();

  let spouse = null;

  if (parent && parent?.length && parent[0].data.UserP) {
    spouse = await this.query()
      .fetchUserByUserId(parent[0].data.UserP.identity)
      .resolveUsersSpouseByRelationByTreeId(treeId.toString())
      .commitWithReturnEntities();
  }

  if (spouse && spouse.length && spouse[0].data.UserS) {
    result.push(spouse[0].data.UserS);
  }

  // base case
  if (parent && parent?.length && parent[0].data.UserP) {
    await this.getRecursiveSpouses(parent[0].data.UserP.identity, treeId);
  } else {
    return result;
  }
}

//=========================================================>
async getMarriedSubTreeUsersByUserId(userId: number): Promise<any> {
  let testMethod = [];
  testMethod = await this.query()
    .findAllUsersByParam(userId, "User")
    .commitWithReturnEntities();

  if (testMethod) {
    return testMethod.map(({ data }) => {
      const testMethod = data;
      return {
        id: testMethod.User.identity,
        ...testMethod.User.properties,
      };
    });
  }
 }
}
```

Файл **tree.module.ts**

```
import { Module } from '@nestjs/common';
import { TreeService } from './tree.service';
import { TreeController } from './tree.controller';

@Module({
 controllers: [TreeController],
 providers: [TreeService]
})
export class TreeModule {}
```

Файл **tree.controller.ts**

```
import {
 Controller,
 Get,
 Post,
 Body,
 Patch,
 Param,
 Delete,
 UseGuards,
} from "@nestjs/common";
import { TreeService } from "./tree.service";
import { CreateTreeDto } from "./dto/create-tree.dto";
import { UpdateTreeDto } from "./dto/update-tree.dto";
import { JoinToTreeDto } from "./dto/join-to-tree.dto";
import { FirebaseAuthGuard } from "src/services/auth/firebase/firebase-auth.guard";

@Controller("tree")
```

```
export class TreeController {
 constructor(private readonly treeService: TreeService) {}

 @Post("add")
 create(@Body() createTreeDto: CreateTreeDto) {
  return this.treeService.create(createTreeDto);
 }

 @Get()
 findAll() {
  return this.treeService.findAll();
 }

 @Get(":id")
 findOne(@Param("id") id: string) {
  return this.treeService.findOne(+id);
 }

 @Get("all/:id")
 findOneByUUID(@Param("id") id: string) {
  return this.treeService.findOneByUUID(id);
 }

 @Get("members/:id")
 getTreeMembers(@Param("id") id: string) {
  return this.treeService.getTreeMembers(+id);
 }

 @Get("partTree/:treeId/:userId")
 getPartTreeByUserId(
  @Param("treeId") treeId: string,
  @Param("userId") userId: string
 ) {
  return this.treeService.getPartTreeByUserId(+treeId, +userId);
 }

 @Get("treeInParts/:treeId/:userId")
 getTreeInPartsUserId(
  @Param("treeId") treeId: string,
  @Param("userId") userId: string
 ) {
  return this.treeService.getTreeInPartsUserId(+treeId, userId);
 }

 @Patch(":id")
 update(@Param("id") id: string, @Body() updateTreeDto: UpdateTreeDto) {
  return this.treeService.update(+id, updateTreeDto);
 }

 @Patch("jointo/:id")
 join(@Param("id") id: string, @Body() joinToTreeDto: JoinToTreeDto) {
  return this.treeService.join(+id, joinToTreeDto);
 }

 @Delete(":id")
 remove(@Param("id") id: string) {
  return this.treeService.remove(+id);
 }

 @Delete("user/:id")
 removeUserFromTree(@Param("id") id: string) {
  return this.treeService.removeUserFromTree(+id);
 }
}
```

Файл **tree.controller.spec.ts**

```
import { Test, TestingModule } from '@nestjs/testing';
import { TreeController } from './tree.controller';
import { TreeService } from './tree.service';

describe('TreeController', () => {
 let controller: TreeController;

 beforeEach(async () => {
  const module: TestingModule = await Test.createTestingModule({
   controllers: [TreeController],
   providers: [TreeService],
  }).compile();
```

```
  controller = module.get<TreeController>(TreeController);
 });

 it('should be defined', () => {
  expect(controller).toBeDefined();
 });
});
```

Файл **post.controller.spec.ts**

```
import { Test, TestingModule } from '@nestjs/testing';
import { PostController } from './post.controller';
import { PostService } from './post.service';

describe('PostController', () => {
 let controller: PostController;

 beforeEach(async () => {
  const module: TestingModule = await Test.createTestingModule({
   controllers: [PostController],
   providers: [PostService],
  }).compile();

  controller = module.get<PostController>(PostController);
 });

 it('should be defined', () => {
  expect(controller).toBeDefined();
 });
});
```

Файл **post.controller.ts**

```
import {
 Controller,
 Get,
 Post,
 Body,
 Patch,
 Param,
 Delete,
 UseGuards,
} from "@nestjs/common";
import { PostService } from "./post.service";
import { CreatePostDto } from "./dto/create-post.dto";
import { UpdatePostDto } from "./dto/update-post.dto";
import { FirebaseAuthGuard } from "src/services/auth/firebase/firebase-auth.guard";

@Controller("post")
export class PostController {
 constructor(private readonly postService: PostService) {}

 @Post("add")
 create(@Body() createPostDto: CreatePostDto) {
  return this.postService.create(createPostDto);
 }

 @Get("all/:id")
 findAll(@Param("id") id: string) {
  return this.postService.findAll(id);
 }

 @Get(":id")
 findOne(@Param("id") id: string) {
  return this.postService.findOne(+id);
 }

 @Get("user/:id")
 findAllByUserId(@Param("id") id: string) {
  return this.postService.findAllByUserId(id);
 }

 @Patch(":id")
 update(@Param("id") id: string, @Body() updatePostDto: UpdatePostDto) {
  return this.postService.update(+id, updatePostDto);
 }

 @Delete(":id")
 remove(@Param("id") id: string) {
  return this.postService.remove(+id);
 }
}
```

Файл **post.module.ts**

```
import { Module } from '@nestjs/common';
import { PostService } from './post.service';
import { PostController } from './post.controller';

@Module({
 controllers: [PostController],
 providers: [PostService]
})
export class PostModule {}
```

Файл **post.repository.ts**

```
import { BadRequestException, Inject, Injectable, Scope } from '@nestjs/common';
import { Connection } from 'cypher-query-builder';
import * as cypher from 'src/services/database/repository.utils';
import { ConfigService } from '@nestjs/config';
import {CUSTOM_ERROR_MESSAGE, DATABASE_CONNECTION } from
'src/services/database/database.constants';
import { Post } from './entities/post.entity';
import { UtilsRepository } from 'src/utils/utils.repository';

@Injectable({ scope: Scope.REQUEST })
export class PostRepository {
 constructor(
  @Inject(DATABASE_CONNECTION) private readonly connection: Connection,
  private readonly configService: ConfigService,
 ) {
 }

 private query(): cypher.RepositoryQuery {
  return new cypher.RepositoryQuery(this.connection);
 }

 async getPostEntity(id: number): Promise<any> {
  const result = await this.query()
  .findEntityByIdWithUserOptionalMatch('Post', id)
  .commitWithReturnEntity();
  if (result) {
   const data = result.data;
   return {
    id: data.Post.identity,
    publishedByUser: data.User,
    ...data.Post.properties,
   };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }

 async getAllPostsByTreeUUID(uuid: string): Promise<any> {
  const treeUUID = UtilsRepository.getStringVersion(uuid);
  const result = await this.query()
  .fetchAllByEntityUUUIDWithUsers(treeUUID, 'Post')
  .commitWithReturnEntitiesRow();
  if (result) {
   return result[0].Posts;
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }

 async findAllByUserId(id: number): Promise<any> {
  const result = await this.query()
  .findAllPostsByUserId(+id, 'Post')
  .commitWithReturnEntities();
  if (result) {
   return result.map(({ data }) => {
    const result = data;
    return {
     id: result.Post.identity,
     ...result.Post.properties,
    };
   });
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }

 async addNewPost(postData: any): Promise<Post[]> {
  postData.postBody = UtilsRepository.getStringVersion(postData?.postBody);
  postData.publishedById = +postData.publishedById;
```

```
  const result = await this.query()
  .createEntity<{ [key in keyof Partial<Post>]: any }>('Post',
   postData
  )
  .commitWithReturnEntity();

  if (result) {
   const data = result.data;

   return {
    id: data.Post.identity,
    ...data.Post.properties,
   };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }

 async deletePost(id: number): Promise<any> {
  const result = await this.query()
  .deleteEntityById('Post', id)
  .commitWithReturnEntity();
  if (result) {
   const data = result.data;
   return  {
    "response": "done"
   };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }

 async updatePostEntity(
  postId,
  postParams,
 ): Promise<any> {
  const params = postParams;
  const result = await this.query()
  .findEntityById('Post', postId)
  .updateEntity(
   'Post',
   Object.entries({
    'Post.postType': params?.postType,
    'Post.postBody': params?.postBody ?
UtilsRepository.getStringVersion(params?.postBody) : null,
    'Post.comments': params?.comments,
   }).reduce((valuesAcc, [key, value]) => {
    return value !== undefined && value !== null
     ? {
      ...valuesAcc,
      [key]: value,
     }
     : valuesAcc;
   }, {}),
  )
  .commitWithReturnEntity();

  if (result !== null) {
   const data = result.data;
   return {
    id: data.Post.identity,
    ...data.Post.properties,
   };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }
}
```

Файл **post.service.spec.ts**

```
import { Test, TestingModule } from '@nestjs/testing';
import { PostService } from './post.service';

describe('PostService', () => {
 let service: PostService;

 beforeEach(async () => {
  const module: TestingModule = await Test.createTestingModule({
   providers: [PostService],
  }).compile();
```

```
  service = module.get<PostService>(PostService);
});

it('should be defined', () => {
  expect(service).toBeDefined();
});
});
```

Файл **post.service.ts**

```
import { Injectable } from '@nestjs/common';
import { CreatePostDto } from './dto/create-post.dto';
import { UpdatePostDto } from './dto/update-post.dto';
import { PostRepository } from './post.repository';
import { FileService } from "../file/file.service";

@Injectable()
export class PostService {
 constructor(
  private readonly postRepository: PostRepository,
  private readonly fileService: FileService
 ) {}

 async create(createPostDto: CreatePostDto) {
  const result = await this.postRepository.addNewPost(createPostDto);
  return result;
 }

 async findAll(uuid: string) {
  const result = await this.postRepository.getAllPostsByTreeUUID(uuid);
  return result;
 }

 async findOne(id: number) {
  const result = await this.postRepository.getPostEntity(id);
  return result;
 }

 async findAllByUserId(id: string) {
  const result = await this.postRepository.findAllByUserId(+id);
  return result;
 }

 async update(id: number, updatePostDto: UpdatePostDto) {
  const result = await this.postRepository.updatePostEntity(id, updatePostDto);
  return result;
 }

 async remove(id: number) {
  const file = await this.postRepository.getPostEntity(id);
  console.log('file', file);
  await this.fileService.removeFileFromS3(fielsInForm3);
  const result = await this.postRepository.deletePost(id);
  return result;
 }
}
```

Файл **comment.service.ts**

```
import { Injectable } from '@nestjs/common';
import { CommentRepository } from './comment.repository';
import { CreateCommentDto } from './dto/create-comment.dto';
import { GetAllCommentDto } from './dto/get-all-comment.dto';
import { UpdateCommentDto } from './dto/update-comment.dto';
@Injectable()
export class CommentService {
 constructor(
  private readonly commentRepository: CommentRepository,
 ) {}
  async create(createCommentDto: CreateCommentDto) {
  const result = await
this.commentRepository.addNewComment(createCommentDto);
  return result;
 }
 async findAll(id: number, type: string) {
  const result = await this.commentRepository.getAllCommentsByIds(id, type);
  return result;
 }
 async findOne(id: number) {
  const result = await this.commentRepository.getCommentEntity(id);
  return result;
 }
 async update(id: number, updateCommentDto: UpdateCommentDto) {
```

```
  const result = await this.commentRepository.updateCommentEntity(id,
updateCommentDto);
  return result;
 }
 async remove(id: number) {
  const result = await this.commentRepository.deleteComment(id);
  return result;
 }
}
```

Файл **comment.service.spec.ts**

```
import { Test, TestingModule } from '@nestjs/testing';
import { CommentService } from './comment.service';

describe('CommentService', () => {
 let service: CommentService;
 beforeEach(async () => {
  const module: TestingModule = await Test.createTestingModule({
   providers: [CommentService],
  }).compile();
  service = module.get<CommentService>(CommentService);
 });
 it('should be defined', () => {
  expect(service).toBeDefined();
 });
});
```

Файл **comment.repository.ts**

```
import { BadRequestException, Inject, Injectable, Scope } from "@nestjs/common";
import { Connection } from "cypher-query-builder";
import * as cypher from "src/services/database/repository.utils";
import { ConfigService } from "@nestjs/config";
import {
 CUSTOM_ERROR_MESSAGE,
 DATABASE_CONNECTION,
} from "src/services/database/database.constants";
import { Comment } from "./entities/comment.entity";
import { UtilsRepository } from "src/utils/utils.repository";
import { ENTITY_TYPE_COMMENT, ENTITY_TYPE_POST } from
"./comment.constants";
import { PostRepository } from "../post/post.repository";
@Injectable({ scope: Scope.REQUEST })
export class CommentRepository {
 constructor(
  @Inject(DATABASE_CONNECTION) private readonly connection: Connection,
  private readonly configService: ConfigService,
  private readonly postRepository: PostRepository
 ) {}
 private query(): cypher.RepositoryQuery {
  return new cypher.RepositoryQuery(this.connection);
 }
 async addNewComment(commentData: any): Promise<Comment[]> {
  commentData.commentBody = UtilsRepository.getStringVersion(
   commentData?.commentBody
  );
  commentData.publishedById = +commentData.publishedById;
  const result = await this.query()
   .createEntity<{ [key in keyof Partial<Comment>]: any }>(
    "Comment",
    commentData
   )
   .commitWithReturnEntity();

  const entityForComment = await this.query()
   .findEntityById(
    commentData.commentForEntityType,
    commentData.commentForEntityId
   )
   .commitWithReturnEntity();

  let comments = [];
  if (commentData.commentForEntityType === ENTITY_TYPE_POST) {
   if (
    entityForComment?.data?.Post?.properties?.comments &&
    entityForComment?.data?.Post?.properties?.comments?.length
   ) {
    comments = entityForComment?.data?.Post?.properties?.comments;
    comments.push(+result?.data?.Comment?.identity);
   } else {
    comments = [+result.data.Comment.identity];
   }
```

```
    await this.postRepository.updatePostEntity(
      commentData.commentForEntityId,
      { comments: comments }
    );
  }

  if (commentData?.commentForEntityType === ENTITY_TYPE_COMMENT) {
    if (
      entityForComment?.data?.Comment?.properties?.comments &&
      entityForComment?.data?.Comment?.properties?.comments?.length
    ) {
      comments = entityForComment?.data?.Comment?.properties?.comments;
      comments.push(+result?.data?.Comment?.identity);
    } else {
      comments = [+result?.data?.Comment?.identity];
    }
    this.updateCommentEntity(commentData.commentForEntityId, {
      comments: comments,
    });
  }

  if (result) {
    const data = result.data;
    return {
      id: data.Comment.identity,
      ...data.Comment.properties,
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }

 async getCommentEntity(id: number): Promise<any> {
  const result = await this.query()
    .findEntityByIdWithUserOptionalMatch("Comment", id)
    .commitWithReturnEntity();
  if (result) {
    const data = result.data;
    return {
      id: data.Comment.identity,
      publishedByUser: data.User,
      ...data.Comment.properties,
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }

 async getAllCommentsByIds(id: number, type: string): Promise<any> {
  const entityResult = await this.query()
    .findEntityById(type, id)
    .commitWithReturnEntity();
  let result = null;
  if (type === ENTITY_TYPE_COMMENT) {
    result = await this.query()
      .findEntityByIdsWithUsers(
        "Comment",
        entityResult.data.Comment.properties.comments
      )
      .commitWithReturnEntitiesRow();
  }
  if (type === ENTITY_TYPE_POST) {
    result = await this.query()
      .findEntityByIdsWithUsers(
        "Comment",
        entityResult.data.Post.properties.comments
      )
      .commitWithReturnEntitiesRow();
  }
  if (result) {
    return result[0].Comments;
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }

 async deleteComment(id: number): Promise<any> {
  const result = await this.query()
    .deleteEntityById("Comment", id)
    .commitWithReturnEntity();
```

```
  if (result) {
    const data = result.data;
    return {
      response: "done",
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }

 async updateCommentEntity(commentId, commentParams): Promise<any> {
  const params = commentParams;
  const result = await this.query()
    .findEntityById("Comment", commentId)
    .updateEntity(
      "Comment",
      Object.entries({
        "Comment.commentType": params?.commentType,
        "Comment.commentBody": params?.commentBody,
        "Comment.comments": params?.comments,
      }).reduce((valuesAcc, [key, value]) => {
        return value !== undefined && value !== null
          ? {
              ...valuesAcc,
              [key]: value,
            }
          : valuesAcc;
      }, {})
    )
    .commitWithReturnEntity();

  if (result !== null) {
    const data = result.data;
    return {
      id: data.Comment.identity,
      ...data.Comment.properties,
    };
  }
  throw new
BadRequestException(CUSTOM_ERROR_MESSAGE.DB_QUERY_ERROR);
 }
}
```

Файл **comment.module.ts**

```
import { Controller, Get, Post, Body, Patch, Param, Delete } from '@nestjs/common';
import { CommentService } from './comment.service';
import { CreateCommentDto } from './dto/create-comment.dto';
import { UpdateCommentDto } from './dto/update-comment.dto';

@Controller('comment')
export class CommentController {
 constructor(private readonly commentService: CommentService) {}

 @Post('add')
 create(@Body() createCommentDto: CreateCommentDto) {
  return this.commentService.create(createCommentDto);
 }

 @Get('all/:id/:type')
 findAll(
  @Param('id') id: string,
  @Param('type') type: string,
 ) {
  return this.commentService.findAll(+id, type);
 }

 @Get(':id')
 findOne(@Param('id') id: string) {
  return this.commentService.findOne(+id);
 }

 @Patch(':id')
 update(@Param('id') id: string, @Body() updateCommentDto: UpdateCommentDto)
{
  return this.commentService.update(+id, updateCommentDto);
 }

 @Delete(':id')
 remove(@Param('id') id: string) {
  return this.commentService.remove(+id);
 }
```

```
}
```

Файл **comment.controller.spec.ts**

```
import { Test, TestingModule } from '@nestjs/testing';
import { CommentController } from './comment.controller';
import { CommentService } from './comment.service';

describe('CommentController', () => {
 let controller: CommentController;

 beforeEach(async () => {
  const module: TestingModule = await Test.createTestingModule({
    controllers: [CommentController],
    providers: [CommentService],
  }).compile();

  controller = module.get<CommentController>(CommentController);
 });

 it('should be defined', () => {
  expect(controller).toBeDefined();
 });
});
```

Файл **App.tsx**

```
import 'react-native-gesture-handler';
import React from 'react';
import {Provider} from 'react-redux';
import {NativeBaseProvider} from 'native-base';
import {ActionSheetProvider} from '@expo/react-native-action-sheet';
import {SafeAreaProvider} from 'react-native-safe-area-context';
import store from '~redux/store';
import AppNavigator from '~navigation/AppNavigator';
export default function App() {
 return (
  <Provider store={store}>
   <ActionSheetProvider>
    <SafeAreaProvider>
     <NativeBaseProvider>
      <AppNavigator />
     </NativeBaseProvider>
    </SafeAreaProvider>
   </ActionSheetProvider>
  </Provider>
 );
}
```

Файл **useKeyboard.ts**

```
import {useState, useEffect} from 'react';
import {Keyboard} from 'react-native';

const useKeyboard = (): [boolean, () => void] => {
 const [isKeyboardVisible, setIsKeyboardVisible] = useState<boolean>(false);

 const keyboardDismiss = (): void => {
  Keyboard.dismiss();
  setIsKeyboardVisible(false);
 };

 useEffect(() => {
  const onKeyboardShow = (): void => {
   setIsKeyboardVisible(true);
  };

  const onKeyboardHide = (): void => {
   setIsKeyboardVisible(false);
  };

  const showSubscription = Keyboard.addListener('keyboardDidShow',
onKeyboardShow);
  const hideSubscription = Keyboard.addListener('keyboardDidHide',
onKeyboardHide);

  return () => {
   showSubscription.remove();
   hideSubscription.remove();
  };
 }, []);
```

```
 return [isKeyboardVisible, keyboardDismiss];
};
export {useKeyboard};
```

Файл **useOverlaySpinner.ts**

```
import {useState} from 'react';
const useOverlaySpinner = (isVisible: boolean): boolean => {
 const [isSpinnerVisible, setIsSpinnerVisible] = useState<boolean>(isVisible);

 if (!isVisible) {
  setTimeout(() => {
   setIsSpinnerVisible(false);
  }, 300);
 } else {
  setTimeout(() => {
   setIsSpinnerVisible(true);
  }, 300);
 }

 return isSpinnerVisible;
};

export {useOverlaySpinner};
```

Файл **useAppDispatch.ts**

```
import {useDispatch} from 'react-redux';
import {AppDispatch} from '~redux/store';
export const useAppDispatch: () => AppDispatch = useDispatch;
```

Файл **store.ts**

```
import {configureStore} from '@reduxjs/toolkit';
import createSagaMiddleware from 'redux-saga';

import rootSaga from '~redux/sagas/rootSaga';

import postsSlice from './slices/posts';
import userSlice from './slices/user';
import treeSlice from './slices/tree';
import fileUploadsSlice from './slices/fileUploads';
import commentsSlice from './slices/comments';
import authSlice from './slices/auth';

const sagaMiddleware = createSagaMiddleware();

const middleware = [sagaMiddleware];

const store = configureStore({
 reducer: {
  posts: postsSlice,
  user: userSlice,
  tree: treeSlice,
  fileUploads: fileUploadsSlice,
  comments: commentsSlice,
  auth: authSlice,
 },
 middleware: getDefaultMiddleware =>
  getDefaultMiddleware({serializableCheck: false}).concat(middleware),
});

sagaMiddleware.run(rootSaga);

export type RootState = ReturnType<typeof store.getState>;
export type AppDispatch = typeof store.dispatch;
export default store;
```

Файл **user.ts**

```
import {createSlice, createAsyncThunk} from '@reduxjs/toolkit';
import get from 'lodash/get';

import api from '~api/api';
import {parseStringToJSONdata} from '~utils';
import {RootState} from '~redux/store';
import {CreateRelativeAsyncPayload, UpdateUserByIdPayload} from
'~redux/@types';

import {getUserPostsByUserIdAsync} from './posts';
import {addRelativesByTreeIdAsync, getTreeInPartsByIdAsync} from './tree';
import {saveAuthUser} from './auth';

type InitialState = {
 userLoading: boolean;
 spouseLoading: boolean;
```

```
 multipleRequestsLoading: boolean;
 errors: string;
 user: any;
};

const initialState: InitialState = {
 userLoading: false,
 spouseLoading: false,
 multipleRequestsLoading: false,
 errors: '',
 user: {
  data: {},
  newUserData: {},
  recentlyDeletedUserId: null,
  spouseData: null,
  showSpouseActionSheet: false,
  newRelativeData: null,
  relationToNewUser: null,
 },
};

export const getUserByIdAsync = createAsyncThunk<any,
string>('user/getUserById', async userId => {
 const userResponse = await api.get(`/user/${userId}`);
 return userResponse.data;
});

export const getUserAndItsPostsByIdAsync = createAsyncThunk<any, string |
undefined>(
 'user/getUserAndItsPostsById',
 async (userId, {dispatch}) => {
  const userResponse = await api.get(`/user/${userId}`);
  dispatch(getUserAndItsPostsByIdOnlyUserSuccess(userResponse.data));
  await dispatch(getUserPostsByUserIdAsync(userResponse.data.id));
  return userResponse.data;
 }
);

export const updateUserByIdAsync = createAsyncThunk<any,
UpdateUserByIdPayload, {state: RootState}>(
 'user/updateUserSetting',
 async ({userId, userData}, {dispatch, getState}) => {
  const userResponse = await api.patch(`/user/${userId}`, userData);
  const currentAuthState = getState().auth;

  if (userResponse.data && currentAuthState.auth?.data?.id === userId) {
   dispatch(saveAuthUser(userResponse.data));
  }

  return userResponse.data;
 }
);

export const createRelativeAsync = createAsyncThunk<
 any,
 CreateRelativeAsyncPayload,
 {state: RootState}
>(
 'user/createRelativeAsync',
 async (
  {
   userId,
   treeId,
   parents,
   authUserId,
   newRelativeData,
   addRelativeViaModal,
   questionnaireCase,
   setting,
  },
  {dispatch, getState}
 ) => {
  const response = await api.post('/user/add', newRelativeData);

  const newUserId = response.data.id;
  const currentState = getState().user;
  const addRelativeActionSheetOption = currentState.user.relationToNewUser;

  await dispatch(
   addRelativesByTreeIdAsync({
```

```
      treeId,
      userId,
      newUserId,
      addRelativeActionSheetOption,
      parents,
      addRelativeViaModal,
      questionnaireCase: questionnaireCase ?? null,
    })
  );

  if (addRelativeViaModal) {
    // eslint-disable-next-line no-use-before-define
    dispatch(setFamilyRelationFromCurrentUserToNewUser(null));
    await dispatch(getUserAndItsPostsByIdAsync(userId));
    await dispatch(getTreeInPartsByIdAsync({treeId, userId}));
  } else {
    await dispatch(
      updateUserByIdAsync({
        userId: authUserId ?? userId,
        userData: {setting},
      })
    );
    const authUserResponse = await dispatch(getUserByIdAsync(authUserId ??
userId));
    if (authUserResponse) {
      dispatch(saveAuthUser(authUserResponse.payload));
    }
    if (authUserId) {
      await dispatch(getUserByIdAsync(userId));
    }
  }

  return response.data;
 }
);

export const getSpouseByIdAsync = createAsyncThunk<any, string>(
 'user/getSpouseById',
 async spouseId => {
  const response = await api.get(`/user/${spouseId}`);
  return response.data;
 }
);

const userSlice = createSlice({
 name: 'user',
 initialState,
 reducers: {
  resetUserState: state => {
   state.user = {};
  },
  resetNewUserData: state => {
   state.user.newUserData = {};
  },
  getUserAndItsPostsByIdOnlyUserSuccess: (state, {payload}) => {
   state.user.data = payload;
  },

  createUser: (_, __) => {},
  createUserLoading: state => {
   state.userLoading = true;
  },
  createUserSuccess: (state, {payload}) => {
   state.userLoading = false;
   state.user.newUserData = payload;
  },
  createUserFailure: (state, {payload}) => {
   state.userLoading = false;
   state.errors = payload;
  },

  updateUserById: (_, __) => {},
  updateUserByIdLoading: state => {
   state.userLoading = true;
  },
  updateUserByIdSuccess: (state, {payload}) => {
   state.userLoading = false;
   state.user.data = payload;
  },
```

```
  updateUserByIdFailure: (state, {payload}) => {
   state.userLoading = false;
   state.errors = payload;
  },

  deleteUserById: (_, __) => {},
  deleteUserByIdLoading: state => {
   state.userLoading = true;
  },
  deleteUserByIdSuccess: (state, {payload}) => {
   state.userLoading = false;
   state.user.recentlyDeletedUserId = payload;
  },
  deleteUserByIdFailure: (state, {payload}) => {
   state.userLoading = false;
   state.errors = payload;
  },

  resetSpouseData: state => {
   state.user.spouseData = null;
  },
  hideSpouseActionSheet: state => {
   state.user.showSpouseActionSheet = false;
  },
  setFamilyRelationFromCurrentUserToNewUser: (state, {payload}) => {
   state.user.relationToNewUser = payload;
  },
 },
 extraReducers: builder => {
  builder
   .addCase(getUserAndItsPostsByIdAsync.pending, state => {
    state.userLoading = true;
   })
   .addCase(getUserAndItsPostsByIdAsync.fulfilled, (state, action) => {
    state.userLoading = false;
    state.user.data = action.payload;
   })
   .addCase(getUserAndItsPostsByIdAsync.rejected, (state, action) => {
    state.userLoading = false;
    state.errors = action.error.message || '';
   })
   .addCase(getUserByIdAsync.pending, state => {
    state.userLoading = true;
   })
   .addCase(getUserByIdAsync.fulfilled, (state, action) => {
    state.userLoading = false;
    state.user.data = action.payload;
   })
   .addCase(getUserByIdAsync.rejected, (state, action) => {
    state.userLoading = false;
    state.errors = action.error.message || '';
   })
   .addCase(createRelativeAsync.pending, state => {
    state.multipleRequestsLoading = true;
   })
   .addCase(createRelativeAsync.fulfilled, (state, action) => {
    state.multipleRequestsLoading = false;
    state.user.newRelativeData = action.payload;
   })
   .addCase(createRelativeAsync.rejected, (state, action) => {
    state.multipleRequestsLoading = false;
    state.errors = action.error.message || '';
   })
   .addCase(getSpouseByIdAsync.pending, state => {
    state.spouseLoading = true;
   })
   .addCase(getSpouseByIdAsync.fulfilled, (state, action) => {
    state.spouseLoading = false;
    state.user.spouseData = action.payload;
    state.user.showSpouseActionSheet = true;
   })
   .addCase(getSpouseByIdAsync.rejected, state => {
    state.spouseLoading = false;
   })
   .addCase(updateUserByIdAsync.pending, state => {
    state.userLoading = true;
   })
   .addCase(updateUserByIdAsync.fulfilled, (state, action) => {
```

```
      state.userLoading = false;
      state.user.data = action.payload;
    })
    .addCase(updateUserByIdAsync.rejected, state => {
      state.userLoading = false;
    });
 },
});

export const {
 createUser,
 createUserLoading,
 createUserSuccess,
 createUserFailure,

 getUserAndItsPostsByIdOnlyUserSuccess,
 updateUserById,
 updateUserByIdLoading,
 updateUserByIdSuccess,
 updateUserByIdFailure,
 deleteUserById,
 deleteUserByIdLoading,
 deleteUserByIdSuccess,
 deleteUserByIdFailure,
 resetNewUserData,
 resetSpouseData,
 hideSpouseActionSheet,
 setFamilyRelationFromCurrentUserToNewUser,
 resetUserState,
} = userSlice.actions;

export default userSlice.reducer;
export const userSelector = {
 getUser: (state: RootState) => ({
   data: get(state.user, 'user.data', {}) || {},
   newUserData: get(state.user, 'user.newUserData', {}) || {},
   recentlyDeletedUserId: get(state.user, 'user.recentlyDeletedUserId', null) || null,
   showSpouseActionSheet: get(state.user, 'user.showSpouseActionSheet', false) ||
false,
   newRelativeData: get(state.user, 'user.newRelativeData', null) || null,
   relationToNewUser: get(state.user, 'user.relationToNewUser', null) || null,
 }),
 getUserLoading: (state: RootState) => state.user.userLoading,
 getSpouseLoading: (state: RootState) => state.user.spouseLoading,
 getMultipleRequestsLoading: (state: RootState) =>
state.user.multipleRequestsLoading,
 getErrors: (state: RootState) => state.user.errors,
};
export const userIntroductionFirstTextSelector = (state: RootState) => {
 const parsedData = parseStringToJSONdata(state.user.user.data?.introduction);

 if (parsedData?.stories) {
   delete parsedData?.stories;
 }

 const preparedData = parsedData && Object.values(parsedData)?.flat();
 const onlyText =
   preparedData?.length && preparedData?.filter((item: any) => item?.type ===
'text');

 return onlyText;
};

export const userIntroductionFirstImageSelector = (state: RootState) => {
 const parsedData = parseStringToJSONdata(state.user.user.data?.introduction);

 if (parsedData?.stories) {
   delete parsedData?.stories;
 }

 const preparedData = parsedData && Object.values(parsedData)?.flat();
 const onlyImage =
   preparedData?.length && preparedData?.filter((item: any) => item?.type ===
'images');

 return onlyImage;
};
```

Файл **tree.ts**

```
import {createSlice, createAsyncThunk} from '@reduxjs/toolkit';
import get from 'lodash/get';
```

```
import api from '~api/api';
import {TreeMemberActionSheetOptions, TreeRelations} from '~utils';
import {
 AddRelativesByTreeIdAsyncPayload,
 GetTreeInPartsByIdAsyncPayload,
 InitializeTreeData,
 UpdateTreeByIdAsyncPayload,
} from '~redux/@types';

import {RootState} from '~redux/store';
import {
 getAddRelativeToTreeInQuestionnaireBodyData,
 getAddRelativeToTreeInUserProfileBodyData,
} from '~screens/TreeScreens/Helpers/AddRelativeToTreeOutsideOfTreeHelper';

const initialState = {
 treeLoading: false,
 errors: '',
 tree: {
  data: [],
  treeMembersData: [],
  newUserJoinedId: '',
  newTreeData: [],
  treeViewer: null,
 },
};

export const getTreeInPartsByIdAsync = createAsyncThunk<any,
GetTreeInPartsByIdAsyncPayload>(
 'tree/getTreeInPartsById',
 async ({treeId, userId}) => {
  const treeResponse = await api.get(`/tree/treeInParts/${treeId}/${userId}`);
  return treeResponse.data;
 }
);

export const createTreeByIdAsync = createAsyncThunk<any, InitializeTreeData>(
 'tree/createTreeById',
 async treeData => {
  const treeResponse = await api.post('/tree/add', treeData);
  return treeResponse.data;
 }
);

export const getTreeMembersByIdAsync = createAsyncThunk<any, string>(
 'tree/getTreeMembersById',
 async treeId => {
  const treeResponse = await api.get(`/tree/members/${treeId}`);
  return treeResponse.data;
 }
);

export const updateTreeByIdAsync = createAsyncThunk<any,
UpdateTreeByIdAsyncPayload>(
 'tree/updateTreeById',
 async ({
  treeId,
  toUserId,
  newUserId,
  roleType,
  isRootUser,
  isWifeOfRootUser,
  treeMemberThatHoldsChildren,
 }) => {
  let userRelationInTree;
  let bodyData;
  if (treeMemberThatHoldsChildren && roleType !==
TreeMemberActionSheetOptions.AddParent) {
   bodyData = {
    userId: newUserId,
    toUserId: treeMemberThatHoldsChildren?.user?.identity,
    relation: TreeRelations.Descendant,
   };
  } else if (
   treeMemberThatHoldsChildren &&
   roleType === TreeMemberActionSheetOptions.AddParent &&
   !isWifeOfRootUser
  ) {
   bodyData = {
    userId: toUserId,
```

```
      toUserId: newUserId,
      relation: TreeRelations.MarriedSubtree,
    };
  } else {
    userRelationInTree =
      roleType === TreeMemberActionSheetOptions.AddSpouse
        ? TreeRelations.Married
        : TreeRelations.Descendant;

    bodyData = {
      userId:
        (isRootUser && roleType === TreeMemberActionSheetOptions.AddParent) ||
        (isWifeOfRootUser && roleType ===
TreeMemberActionSheetOptions.AddParent)
          ? toUserId
          : newUserId,
      toUserId:
        (isRootUser && roleType === TreeMemberActionSheetOptions.AddParent) ||
        (isWifeOfRootUser && roleType ===
TreeMemberActionSheetOptions.AddParent)
          ? newUserId
          : toUserId,
      relation: isWifeOfRootUser ? TreeRelations.MarriedSubtree : userRelationInTree,
    };
  }

  const response = await api.patch(`/tree/jointo/${treeId}`, bodyData);
  if (response?.data?.response === 'done') {
    return newUserId;
  }
 }
);

export const addRelativesByTreeIdAsync = createAsyncThunk<any,
AddRelativesByTreeIdAsyncPayload>(
 'tree/addRelativesByTreeId',
 async ({
  treeId,
  userId,
  newUserId,
  addRelativeActionSheetOption,
  parents,
  addRelativeViaModal,
  questionnaireCase,
 }) => {
 let firstParentId: string = '';
 if (parents && parents?.length > 0) {
  firstParentId = parents[0]?.identity;
 }

 let bodyData = {
  userId: newUserId,
  toUserId: userId,
  relation: TreeRelations.Descendant,
 };

 if (addRelativeViaModal) {
  bodyData = getAddRelativeToTreeInUserProfileBodyData(
   userId,
   newUserId,
   addRelativeActionSheetOption,
   parents,
   firstParentId
  );
 } else {
  if (questionnaireCase) {
   bodyData = getAddRelativeToTreeInQuestionnaireBodyData(
    userId,
    newUserId,
    questionnaireCase,
    parents,
    firstParentId
   );
  }
 }

 const response = await api.patch(`/tree/jointo/${treeId}`, bodyData);
 if (response?.data?.response === 'done') {
  return;
 }
```

```
 }
);

const treeSlice = createSlice({
 name: 'tree',
 initialState,
 reducers: {
  setTreeViewer: (state, {payload}) => {
   state.tree.treeViewer = payload;
  },
  clearTree: state => {
   state.tree = initialState.tree;
  },
 },
 extraReducers: builder => {
  builder
   .addCase(getTreeInPartsByIdAsync.pending, state => {
    state.treeLoading = true;
   })
   .addCase(getTreeInPartsByIdAsync.fulfilled, (state, action) => {
    state.treeLoading = false;
    state.tree.data = action.payload;
   })
   .addCase(getTreeInPartsByIdAsync.rejected, (state, action) => {
    state.treeLoading = false;
    state.errors = action.error.message || '';
   })
   .addCase(getTreeMembersByIdAsync.pending, state => {
    state.treeLoading = true;
   })
   .addCase(getTreeMembersByIdAsync.fulfilled, (state, action) => {
    state.treeLoading = false;
    state.tree.treeMembersData = action.payload;
   })
   .addCase(getTreeMembersByIdAsync.rejected, (state, action) => {
    state.treeLoading = false;
    state.errors = action.error.message || '';
   })
   .addCase(createTreeByIdAsync.pending, state => {
    state.treeLoading = false;
   })
   .addCase(createTreeByIdAsync.fulfilled, (state, action) => {
    state.treeLoading = false;
    state.tree.newTreeData = action.payload;
   })
   .addCase(createTreeByIdAsync.rejected, (state, action) => {
    state.treeLoading = false;
    state.errors = action.error.message || '';
   })
   .addCase(updateTreeByIdAsync.pending, state => {
    state.treeLoading = true;
   })
   .addCase(updateTreeByIdAsync.fulfilled, (state, action) => {
    state.treeLoading = false;
    state.tree.newUserJoinedId = action.payload;
   })
   .addCase(updateTreeByIdAsync.rejected, (state, action) => {
    state.treeLoading = false;
    state.errors = action.error.message || '';
   });
 },
});

export const {setTreeViewer, clearTree} = treeSlice.actions;

export const treeSelector = {
 getTree: (state: RootState) => ({
  data: get(state.tree, 'tree.data', []) || [],
  treeMembersData: get(state.tree, 'tree.treeMembersData', []) || [],
  newUserJoinedId: get(state.tree, 'tree.newUserJoinedId', '') || '',
  newTreeData: get(state.tree, 'tree.newTreeData', []) || [],
  treeViewer: get(state.tree, 'tree.treeViewer', null) || null,
  isNewUserAddedToAnotherTree: get(state.tree,
'tree.isNewUserAddedToAnotherTree', null) || null,
  spouseThatAddParentToAnotherTree:
   get(state.tree, 'tree.spouseThatAddParentToAnotherTree', null) || null,
 }),
 getTreeLoading: (state: RootState) => state.tree.treeLoading,
 getErrors: (state: RootState) => state.tree.errors,
```

```
};

export default treeSlice.reducer;
```

Файл **posts.ts**

```
import {createSlice, createAsyncThunk} from '@reduxjs/toolkit';

import get from 'lodash/get';


import api from '~api/api';

import {CreatePostAsyncPayload, UpdatePostByIdAsyncPayload} from
'~redux/@types';

import {RootState} from '~redux/store';


const initialState = {
 postsLoading: false,
 errors: '',
 posts: {
  data: [],
  treePostsData: [],
  newPostsData: {},
  recentlyDeletedPostId: null,
  singlePost: {},
 },
};


export const getUserPostsByUserIdAsync = createAsyncThunk<any, string>(
 'posts/getUserPostsByUserId',
 async userId => {
  const postsResponse = await api.get(`/post/user/${+userId}`);
  return postsResponse.data;
 }
);


export const getTreeMembersPostsByTreeIdAsync = createAsyncThunk<any,
string>(
 'posts/getTreeMembersPostsByTreeId',
 async treeId => {
  const treeMembersPostsResponse = await api.get(`/post/all/${treeId}`);
  return treeMembersPostsResponse.data;
 }
);


export const getSinglePostByIdAsync = createAsyncThunk<any, string>(
 'posts/getSinglePostById',
 async postId => {
  const singlePostResponse = await api.get(`/post/${postId}`);
  return singlePostResponse.data;
 }
);


export const createPostAsync = createAsyncThunk<any, CreatePostAsyncPayload>(
 'posts/createPost',
 async ({userId = '', treeId, postType, postBody}) => {
  const postData = {
   publishedById: +userId,
   treeUUID: treeId,
   postType,
   postBody,
  };

  const postResponse = await api.post('/post/add', postData);
  return postResponse.data;
 }
);


export const updatePostByIdAsync = createAsyncThunk<any,
UpdatePostByIdAsyncPayload>(
 'posts/updatePostById',
 async ({postId, userId, postBody, comments}) => {
  const postData = {
   publishedById: userId,
   postBody,
   comments,
  };
  const updatePostResponse = await api.patch(`/post/${postId}`, postData);
  return updatePostResponse.data;
 }
);


export const deletePostByIdAsync = createAsyncThunk<any, string | undefined>(
 'posts/deletePostById',
 async postId => {
```

```
  const deletePostResponse = await api.delete(`/post/${postId}`);
  return deletePostResponse.data;
 }
);

const postsSlice = createSlice({
 name: 'posts',
 initialState,
 reducers: {
  cleanSinglePost: state => {
   state.posts.singlePost = {};
  },
  cleanNewPostData: state => {
   state.posts.newPostsData = {};
  },
  cleanPosts: state => {
   state.posts.data = [];
   state.posts.treePostsData = [];
  },
 },
 extraReducers: builder => {
  builder
   .addCase(getUserPostsByUserIdAsync.fulfilled, (state, action) => {
    state.posts.data = action.payload;
   })
   .addCase(getUserPostsByUserIdAsync.rejected, (state, action) => {
    state.errors = action.error.message || '';
   })
   .addCase(getTreeMembersPostsByTreeIdAsync.pending, (state, _) => {
    state.postsLoading = true;
   })
   .addCase(getTreeMembersPostsByTreeIdAsync.fulfilled, (state, action) => {
    state.postsLoading = false;
    state.posts.treePostsData = action.payload;
   })
   .addCase(getTreeMembersPostsByTreeIdAsync.rejected, (state, action) => {
    state.postsLoading = false;
    state.errors = action.error.message || '';
   })
  .addCase(getSinglePostByIdAsync.pending, (state, _) => {
   state.postsLoading = true;
  })
  .addCase(getSinglePostByIdAsync.fulfilled, (state, action) => {
   state.postsLoading = false;
   state.posts.singlePost = action.payload;
  })
  .addCase(getSinglePostByIdAsync.rejected, (state, action) => {
   state.postsLoading = false;
   state.errors = action.error.message || '';
  })
  .addCase(createPostAsync.pending, (state, _) => {
   state.postsLoading = true;
  })
  .addCase(createPostAsync.fulfilled, (state, action) => {
   state.postsLoading = false;
   state.posts.newPostsData = action.payload;
  })
  .addCase(createPostAsync.rejected, (state, action) => {
   state.postsLoading = false;
   state.errors = action.error.message || '';
  })
  .addCase(updatePostByIdAsync.pending, (state, _) => {
   state.postsLoading = true;
  })
  .addCase(updatePostByIdAsync.fulfilled, (state, action) => {
   state.postsLoading = false;
   state.posts.newPostsData = action.payload;
  })
  .addCase(updatePostByIdAsync.rejected, (state, action) => {
   state.postsLoading = false;
   state.errors = action.error.message || '';
  })
  .addCase(deletePostByIdAsync.pending, (state, _) => {
   state.postsLoading = true;
  })
  .addCase(deletePostByIdAsync.fulfilled, (state, action) => {
   state.postsLoading = false;
   state.posts.recentlyDeletedPostId = action.payload;
```

```
    })
    .addCase(deletePostByIdAsync.rejected, (state, action) => {
      state.postsLoading = false;
      state.errors = action.error.message || '';
    });
 },
});

export const {cleanSinglePost, cleanNewPostData, cleanPosts} = postsSlice.actions;
export const postsSelector = {
 getPosts: (state: RootState) => {
  const posts = get(state.posts, 'posts.data', []) || [];

  return posts.map((post: any) => {
   return {
    id: get(post, 'id', '') || '',
    labels: get(post, 'labels', '') || '',
    postBody: get(post, 'postBody', '') || '',
    postType: get(post, 'postType', '') || '',
    treeUUID: get(post, 'treeUUID', '') || '',
    createDate: get(post, 'createDate', 0) || 0,
    publishedById: get(post, 'publishedById', '') || '',
    comments: get(post, 'comments', []) || [],
   };
  });
 },
 getTreePostsData: (state: RootState) => state.posts.posts.treePostsData,
 getNewPostsData: (state: RootState) => state.posts.posts.newPostsData,
 getRecentlyDeletedPostId: (state: RootState) =>
state.posts.posts.recentlyDeletedPostId,
 getSinglePost: (state: RootState) => state.posts.posts.singlePost,
 getPostsLoading: (state: RootState) => state.posts.postsLoading,
};
export default postsSlice.reducer;
```

Файл **comments.ts**

```
import {createSlice, createAsyncThunk} from '@reduxjs/toolkit';
import {updatePostByIdAsync} from './posts';
import get from 'lodash/get';

import api from '~api/api';
import {
 CreateCommentAsyncPayload,
 DeleteCommentByIdAsyncPayload,
 UpdateCommentByIdAsyncPayload,
} from '~redux/@types';
import {RootState} from '~redux/store';

const initialState = {
 commentsLoading: false,
 errors: '',
 comments: {
  data: [],
  postComments: [],
  newCommentData: {},
  recentlyDeletedCommentId: null,
  singleComment: {},
 },
};

export const getCommentsByPostIdAsync = createAsyncThunk<any, string>(
 'comments/getCommentsByPostId',
 async postId => {
  const commentsResponse = await api.get(`/comment/all/${postId}/Post`);
  return commentsResponse.data;
 }
);

export const getCommentsByCommentIdAsync = createAsyncThunk<any, string>(
 'comments/getCommentsByCommentId',
 async commentId => {
  const commentsByCommentIdResponse = await api.get(`/post/all/${commentId}`);
  return commentsByCommentIdResponse.data;
 }
);

export const getUserSingleCommentIdAsync = createAsyncThunk<any, string>(
 'comments/getUserSingleCommentId',
 async commentId => {
  const singleCommentResponse = await api.get(`/comment/${commentId}`);
```

```
  return singleCommentResponse.data;
 }
);
export const createCommentAsync = createAsyncThunk<any,
CreateCommentAsyncPayload>(
 'comments/createComment',
 async ({userId, entityId, entityType, commentType, commentData}) => {
  const commentBody = {
   publishedById: +userId,
   commentForEntityType: entityType,
   commentForEntityId: entityId,
   commentType,
   commentBody: commentData,
  };
  const commentResponse = await api.post('/comment/add', commentBody);
  return commentResponse.data;
 }
);

export const updateCommentByIdAsync = createAsyncThunk<any,
UpdateCommentByIdAsyncPayload>(
 'comments/updateCommentById',
 async ({commentId, userId, entityId, entityType, commentType, commentData}) =>
{
  const commentBody = {
   publishedById: +userId,
   commentForEntityType: entityType,
   commentForEntityId: entityId,
   commentType,
   commentBody: JSON.stringify(commentData),
  };

  const updateCommentResponse = await api.patch(`/comment/${commentId}`,
commentBody);
  return updateCommentResponse.data;
 }
);

export const deleteCommentByIdAsync = createAsyncThunk<any,
DeleteCommentByIdAsyncPayload>(
 'comments/deleteCommentById',
 async (
  {commentId, postId, postAuthorId, updatedPostBody,
commentsIdThatShouldDelete},
  {dispatch}
 ) => {
  const deleteCommentResponse = await api.delete(`/comment/${commentId}`);
  if (deleteCommentResponse.data.response === 'done') {
   await dispatch(
    updatePostByIdAsync({
     postId,
     userId: postAuthorId,
     postBody: updatedPostBody,
     comments: commentsIdThatShouldDelete,
    })
   );
  }

  return deleteCommentResponse.data;
 }
);

const commentsSlice = createSlice({
 name: 'comments',
 initialState,
 reducers: {
  cleanSingleComment: state => {
   state.comments.singleComment = {};
  },
  cleanComments: state => {
   state.comments.data = [];
  },
  cleanNewPostData: state => {
   state.comments.newCommentData = {};
  },
 },
 extraReducers: builder => {
  builder
   .addCase(getCommentsByPostIdAsync.pending, (state, _) => {
    state.commentsLoading = true;
   })
   .addCase(getCommentsByPostIdAsync.fulfilled, (state, action) => {
```

```
  state.commentsLoading = false;
  state.comments.data = action.payload;
 })
 .addCase(getCommentsByPostIdAsync.rejected, (state, action) => {
  state.commentsLoading = false;
  state.errors = action.error.message || '';
 })
 .addCase(getCommentsByCommentIdAsync.pending, (state, _) => {
  state.commentsLoading = true;
 })
 .addCase(getCommentsByCommentIdAsync.fulfilled, (state, action) => {
  state.commentsLoading = false;
  state.comments.postComments = action.payload;
 })
 .addCase(getCommentsByCommentIdAsync.rejected, (state, action) => {
  state.commentsLoading = false;
  state.errors = action.error.message || '';
 })
 .addCase(getUserSingleCommentIdAsync.pending, (state, _) => {
  state.commentsLoading = true;
 })
 .addCase(getUserSingleCommentIdAsync.fulfilled, (state, action) => {
  state.commentsLoading = false;
  state.comments.singleComment = action.payload;
 })
 .addCase(getUserSingleCommentIdAsync.rejected, (state, action) => {
  state.commentsLoading = false;
  state.errors = action.error.message || '';
 })
 .addCase(createCommentAsync.pending, (state, _) => {
  state.commentsLoading = true;
 })
 .addCase(createCommentAsync.fulfilled, (state, action) => {
  state.commentsLoading = false;
  state.comments.newCommentData = action.payload;
 })
 .addCase(createCommentAsync.rejected, (state, action) => {
  state.commentsLoading = false;
  state.errors = action.error.message || '';
 })
 .addCase(updateCommentByIdAsync.pending, (state, _) => {
  state.commentsLoading = true;
 })
 .addCase(updateCommentByIdAsync.fulfilled, (state, action) => {
  state.commentsLoading = false;
  state.comments.newCommentData = action;
 })
 .addCase(updateCommentByIdAsync.rejected, (state, action) => {
  state.commentsLoading = false;
  state.errors = action.error.message || '';
 })
 .addCase(deleteCommentByIdAsync.pending, (state, _) => {
  state.commentsLoading = true;
 })
 .addCase(deleteCommentByIdAsync.fulfilled, (state, action) => {
  state.commentsLoading = false;
  state.comments.recentlyDeletedCommentId = action.payload;
 })
 .addCase(deleteCommentByIdAsync.rejected, (state, action) => {
  state.commentsLoading = false;
  state.errors = action.error.message || '';
 });
 },
});

export const {cleanSingleComment, cleanComments} = commentsSlice.actions;
export const commentsSelector = {
 getComments: (state: RootState) => {
  const comments = get(state.comments, 'comments.data', []) || [];
  return comments.map((comment: any) => {
   return {
    Comment: {
     identity: get(comment, 'Comment.identity', '') || '',
     labels: get(comment, 'Comment.labels', '') || '',
     properties: {
      commentBody: get(comment, 'Comment.properties.commentBody', '') || '',
      commentForEntityId: get(comment,
'Comment.properties.commentForEntityId', '') || '',
```

```
          commentForEntityType: get(comment,
'Comment.properties.commentForEntityType', '') || '',
          commentType: get(comment, 'Comment.properties.commentType', '') || '',
          createDate: get(comment, 'Comment.properties.createDate', 0) || 0,
          publishedById: get(comment, 'Comment.properties.publishedById', '') || '',
        },
      },
      User: {
        identity: get(comment, 'User.identity', '') || '',
        labels: get(comment, 'User.labels', '') || '',
        properties: {
          birthdate: get(comment, 'User.properties.birthdate', '') || '',
          createDate: get(comment, 'User.properties.createDate', 0) || 0,
          email: get(comment, 'User.properties.email', '') || '',
          firstName: get(comment, 'User.properties.firstName', '') || '',
          gender: get(comment, 'User.properties.gender', '') || '',
          isActivated: get(comment, 'User.properties.isActivated', true) || false,
          lastName: get(comment, 'User.properties.lastName', '') || '',
          myTreeIdByParent1: get(comment, 'User.properties.myTreeIdByParent1', '') ||
'',
          treeOwner: get(comment, 'User.properties.treeOwner', true) || false,
          updateDate: get(comment, 'User.properties.updateDate', 0) || 0,
          userPictureLink: get(comment, 'User.properties.userPictureLink', 0) || 0,
        },
      },
    };
  });
},
getPostComments: (state: RootState) => state.comments.comments.postComments,
getNewCommentData: (state: RootState) =>
state.comments.comments.newCommentData,
getRecentlyDeletedCommentId: (state: RootState) =>
  state.comments.comments.recentlyDeletedCommentId,
getSingleComment: (state: RootState) => state.comments.comments.singleComment,
getCommentsLoading: (state: RootState) => state.comments.commentsLoading,
};
export default commentsSlice.reducer;
```

Файл **auth.ts**

```
import {createAsyncThunk, createSlice, PayloadAction} from '@reduxjs/toolkit';
import auth from '@react-native-firebase/auth';
import get from 'lodash/get';
import {GoogleSignin} from '@react-native-google-signin/google-signin';
import {RootState} from '~redux/store';
import {AuthData} from '~redux/@types';
import {parseStringToJSONdata, removeAuthToken} from '~utils';
import {QuestionnaireSettings} from '~types/QuestionnaireSettings';
type InitialStateType = {
 authLoading: boolean;
 errors: string;
 auth: {
  data: AuthData | null;
  isAuthorized: boolean;
 };
};
const initialState: InitialStateType = {
 authLoading: false,
 errors: '',
 auth: {
  data: null,
  isAuthorized: false,
 },
};
export const signOutAsync = createAsyncThunk('auth/signOut', async () => {
 await GoogleSignin.signOut();
 await auth().signOut();
 await removeAuthToken();
 return;
});
const authSlice = createSlice({
 name: 'auth',
 initialState,
 reducers: {
  signInUser: (_, __: PayloadAction<string | undefined>) => {},
  signInWithTokenLoading: state => {
   state.authLoading = true;
  },
  signInWithTokenSuccess: (state, {payload}) => {
   state.authLoading = false;
   state.auth.data = payload;
   state.auth.isAuthorized = true;
```

```
  },
  signInWithTokenFailure: (state, {payload}) => {
    state.authLoading = false;
    state.errors = payload;
  },
  saveAuthUser: (state, {payload}) => {
    if (state.auth.data) {
      state.auth.data = {
        ...state.auth.data,
        ...payload,
      };
    } else {
      state.auth.data = payload;
    }
  },
  setDoNotShowQuestionnaireModal: (state, {payload}) => {
    if (state.auth.data) {
      state.auth.data.setting = {
        ...parseStringToJSONdata(state.auth.data.setting),
        doNotShowQuestionnaireModal: payload,
      };
    }
  },
},
extraReducers: builder => {
  builder
    .addCase(signOutAsync.pending, state => {
      state.authLoading = true;
    })
    .addCase(signOutAsync.fulfilled, state => {
      state.authLoading = false;
      state.auth.data = null;
      state.auth.isAuthorized = false;
    })
    .addCase(signOutAsync.rejected, (state, {error}) => {
      state.authLoading = false;
      state.errors = error.message || '';
    });
},
});
export const {
 saveAuthUser,
 signInUser,
 signInWithTokenLoading,
 signInWithTokenSuccess,
 signInWithTokenFailure,
 setDoNotShowQuestionnaireModal,
} = authSlice.actions;
export const authSelector = {
 getAuthLoading: (state: RootState) => state.auth.authLoading,
 getIsAuthorized: (state: RootState) => state.auth.auth.isAuthorized,
 getAuthUser: (state: RootState) => ({
   data: get(state.auth, 'auth.data', null) || null,
 }),
};
export const questionnaireSettingsSelector = (state: RootState) => {
 const parsedData: QuestionnaireSettings =
parseStringToJSONdata(state.auth.auth.data?.setting);
 return parsedData;
};
```

Файл **LoginScreen.tsx**

```
import React, {useEffect} from 'react';
import {Alert} from 'react-native';
import {Box, Text} from 'native-base';
import auth, {FirebaseAuthTypes} from '@react-native-firebase/auth';
import {
 GoogleSignin,
 GoogleSigninButton,
 statusCodes,
} from '@react-native-google-signin/google-signin';
import {useDispatch} from 'react-redux';

import {LogoIcon} from '~shared/Icons';
import {storeAuthToken, WEB_CLIENT_ID} from '~utils';
import {signInUser} from '~redux/slices/auth';
import styles from './styles';
const LoginScreen = () => {
 const dispatch = useDispatch();
 useEffect(() => {
```

```
  GoogleSignin.configure({
    scopes: ['email'], // what API you want to access on behalf of the user, default is email and profile
    // client ID of type WEB for your server (needed to verify user ID and offline access)
    webClientId: WEB_CLIENT_ID,
    offlineAccess: true, // if you want to access Google API on behalf of the user FROM YOUR SERVER
  });
 }, []);
 const onGoogleButtonPress = async (): Promise<FirebaseAuthTypes.UserCredential | undefined> => {
  try {
    // Check if your device supports Google Play
    await GoogleSignin.hasPlayServices({showPlayServicesUpdateDialog: true});
    // Get the users ID token
    const {idToken} = await GoogleSignin.signIn();
    // Create a Google credential with the token
    const googleCredential = auth.GoogleAuthProvider.credential(idToken);
    // Sign-in the user with the credential
    const authResponse = auth().signInWithCredential(googleCredential);
    return authResponse;
  } catch (error: any) {
    console.log('error :>> ', error);
    if (error.code === statusCodes.SIGN_IN_CANCELLED) {
      // user cancelled the login flow
      Alert.alert('Cancel');
    } else if (error.code === statusCodes.IN_PROGRESS) {
      Alert.alert('Signin in progress');
      // operation (f.e. sign in) is in progress already
    } else if (error.code === statusCodes.PLAY_SERVICES_NOT_AVAILABLE) {
      Alert.alert('PLAY_SERVICES_NOT_AVAILABLE');
      // play services not available or outdated
    } else {
      // some other error happened
      Alert.alert('SOME_UNUSUAL_ERROR_HAPPENED');
      console.log('error :>> ', error);
    }
  }
 };
 const onAuthStateChanged = async (userAuth: FirebaseAuthTypes.User | null) => {
  if (!userAuth) {
   return;
  }
  if (userAuth) {
   try {
    const currentUser = auth().currentUser;
    if (currentUser) {
     const userToken = await currentUser.getIdToken(true);
     await storeAuthToken(userToken);
     if (currentUser?.email) {
      dispatch(signInUser(currentUser?.email));
     }
    }
   } catch (error) {
    console.log('error :>> ', error);
   }
  }
 };
 useEffect(() => {
  const subscriber = auth().onAuthStateChanged(onAuthStateChanged);
  return () => {
   subscriber;
  };
 }, []);
 const onGoogleSigninButton = () =>
  onGoogleButtonPress().then(() => console.log('Signed in with Google!'));
 return (
  <Box width="100%">
   <Box style={styles.logo}>
    <LogoIcon />
   </Box>
   <Text style={styles.signInTitle}>Sign In</Text>

   <Box alignItems="center">
    <GoogleSigninButton
     style={styles.socialButton}
     size={GoogleSigninButton.Size.Wide}
     color={GoogleSigninButton.Color.Dark}
     onPress={onGoogleSigninButton}
```

```
      />
    </Box>
  </Box>
 );
};
export default LoginScreen;
```

Файл **WelcomeScreen.tsx**

```
import React from 'react';
import {useSelector} from 'react-redux';
import {Formik} from 'formik';
import {Box, Text} from 'native-base';
import {KeyboardAwareScrollView} from
'@codler/react-native-keyboard-aware-scroll-view';
import {updateUserByIdAsync} from '~redux/slices/user';
import {authSelector, questionnaireSettingsSelector} from '~redux/slices/auth';
import {useAppDispatch} from '~hooks/redux';
import {CustomTextInput, CustomButton} from '~shared';
import {CalendarIcon} from '~shared/Icons';
import {InputType, signUpSchema} from '~utils';
import {QuestionnaireSettings} from '~types/QuestionnaireSettings';
import {styles} from './styles';
type InitialValues = {
 firstName: string;
 lastName: string;
 maidenName?: string;
 birthdate: string;
 gender: string;
};
interface IUserDataDependingOnGender extends InitialValues {
 isActivated: boolean;
 setting: QuestionnaireSettings;
}
const WelcomeScreen = () => {
 const dispatch = useAppDispatch();
 const authUser = useSelector(authSelector.getAuthUser);
 const questionnaireSettings = useSelector(questionnaireSettingsSelector);
 const getUserDataDependingOnGender = (values: InitialValues):
IUserDataDependingOnGender => {
  if (
   values.gender.toLocaleLowerCase() === 'female' ||
   values.gender.toLocaleLowerCase() === 'non-binary'
  ) {
   return {
    firstName: values?.firstName?.trim(),
    lastName: values?.lastName?.trim(),
    maidenName: values?.maidenName?.trim(),
    birthdate: values?.birthdate,
    gender: values?.gender,
    isActivated: true,
    setting: {
     ...questionnaireSettings,
     isUserUpdatedOnWelcomeScreen: true,
    },
   };
  }
  return {
   firstName: values?.firstName?.trim(),
   lastName: values?.lastName?.trim(),
   birthdate: values?.birthdate,
   gender: values?.gender,
   isActivated: true,
   setting: {
    ...questionnaireSettings,
    isUserUpdatedOnWelcomeScreen: true,
   },
  };
 };
 const initialValues: InitialValues = {
  firstName: '',
  lastName: '',
  maidenName: '',
  birthdate: '',
  gender: '',
 };

 const onSubmit = async (values: InitialValues): Promise<void> => {
  await dispatch(
   updateUserByIdAsync({
    userId: authUser.data?.id,
```

```
      userData: getUserDataDependingOnGender(values),
    })
  );
};

return (
  <KeyboardAwareScrollView
    enableOnAndroid
    extraHeight={0}
    contentContainerStyle={styles.scrollContainer}
  >
    <Box
      width="100%"
      height="100%"
      alignItems="center"
      justifyContent="center"
      backgroundColor="white"
      pb={8}
    >
      <Text marginTop={3} style={styles.welcomeTitle}>
        Welcome to the Family Social Network!
      </Text>
      <Text style={styles.askInfoAbout}>Please specify some information about
you</Text>
      <Formik initialValues={initialValues} validationSchema={signUpSchema}
onSubmit={onSubmit}>
        {props => (
          <>
            <Box alignItems="center" justifyContent="center" mt={15} mb={5}
width="100%">
              <Box width="90%">
                <CustomTextInput
                  title="First Name"
                  placeholder="First Name"
                  marginTop={2}
                  marginTopErrorLabel={-8}
                  marginBottomErrorLabel={0}
                  onChange={props.handleChange('firstName')}
                  value={props.values.firstName}
                  error={
                    props.touched.firstName && props.errors.firstName
                      ? props.errors.firstName
                      : null
                  }
                  onBlur={props.handleBlur('firstName')}
                />
                <CustomTextInput
                  title="Last Name"
                  placeholder="Last Name"
                  marginTop={2}
                  marginTopErrorLabel={-8}
                  marginBottomErrorLabel={0}
                  onChange={props.handleChange('lastName')}
                  value={props.values.lastName}
                  error={
                    props.touched.lastName && props.errors.lastName ?
props.errors.lastName : null
                  }
                  onBlur={props.handleBlur('lastName')}
                />
                {(props.values.gender.toLocaleLowerCase() === 'female' ||
                  props.values.gender.toLocaleLowerCase() === 'non-binary') && (
                  <CustomTextInput
                    title="Maiden Name"
                    placeholder="Maiden Name"
                    onChange={props.handleChange('maidenName')}
                    value={props.values.maidenName}
                    error={
                      props.touched.maidenName && props.errors.maidenName
                        ? props.errors.maidenName
                        : null
                    }
                    onBlur={props.handleBlur('maidenName')}
                  />
                )}
                <CustomTextInput
                  title="Your birth date:"
                  IconLeft={CalendarIcon}
                  type={InputType.date}
                  placeholder="Birth date"
                  marginTop={2}
```

```
                marginTopErrorLabel={-8}
                marginBottomErrorLabel={0}
                onChange={props.handleChange('birthdate')}
                value={props.values.birthdate}
                error={
                  props.touched.birthdate && props.errors.birthdate
                    ? props.errors.birthdate
                    : null
                }
                onBlur={props.handleBlur('birthdate')}
              />
              <CustomTextInput
                dropdownIcon
                title="Gender"
                type={InputType.gender}
                placeholder="Gender"
                marginTop={2}
                marginTopErrorLabel={-8}
                marginBottomErrorLabel={0}
                onChange={props.handleChange('gender')}
                value={props.values.gender}
                error={props.touched.gender && props.errors.gender ?
props.errors.gender : null}
                onBlur={props.handleBlur('gender')}
              />
            </Box>
          </Box>
          <CustomButton onPress={props.handleSubmit} title="Save" />
        </>
      )}
    </Formik>
  </Box>
 </KeyboardAwareScrollView>
);
};
export default WelcomeScreen;
```

Файл **UserProfileScreen.tsx**

```
import React, {FC, useCallback} from 'react';
import {useSelector} from 'react-redux';
import {useFocusEffect} from '@react-navigation/native';
import {getUserAndItsPostsByIdAsync, userSelector} from '~redux/slices/user';
import {authSelector} from '~redux/slices/auth';
import {useAppDispatch} from '~hooks/redux';
import {Tabs, UserAvatarWithTitles} from '~shared';
import {outeStackList, StackScreenPropsWithParams} from
'~types/NavigationTypes';
type UserProfileScreenProps =
StackScreenPropsWithParams<RouteStackList.UserProfileScreen>;
const UserProfileScreen: FC<UserProfileScreenProps> = ({route}) => {
 const dispatch = useAppDispatch();
 const authUser = useSelector(authSelector.getAuthUser);
 const user = useSelector(userSelector.getUser);
 const userData = user?.data;
 useFocusEffect(
  useCallback(() => {
    dispatch(getUserAndItsPostsByIdAsync(route?.params?.userId ||
authUser.data?.id));
  }, [dispatch, route?.params?.userId, authUser.data?.id])
 );

 return (
  <>
   <UserAvatarWithTitles
    username={
     (userData?.firstName || '') +
     (userData?.maidenName ? ` (${userData?.maidenName})` : '') +
     (userData?.lastName ? ` ${userData?.lastName}` : '')
    }
    image={userData?.userPictureLink}
    userId={userData?.id}
   />
   <Tabs />
  </>
 );
};
export default UserProfileScreen;
```

Файл **TreeScreen.tsx**

```
import React, {useEffect, useState, useCallback, FC} from 'react';

import {
 ActivityIndicator,
```

```
 LayoutChangeEvent,
 TouchableOpacity,
 useWindowDimensions,
} from 'react-native';

import {HStack, Text, VStack, Box} from 'native-base';
import {useSelector} from 'react-redux';
import Svg, {Line} from 'react-native-svg';
import FastImage from 'react-native-fast-image';
import {useActionSheet} from '@expo/react-native-action-sheet';
import {useFocusEffect} from '@react-navigation/native';
import Orientation, {useDeviceOrientationChange} from
'react-native-orientation-locker';
import {scale} from 'react-native-utils-scale';
import Spinner from 'react-native-loading-spinner-overlay/lib';
import {StackScreenProps} from '@react-navigation/stack';

import {authSelector, saveAuthUser} from '~redux/slices/auth';
import {
 getTreeInPartsByIdAsync,
 setTreeViewer,
 treeSelector,
 updateTreeByIdAsync,
} from '~redux/slices/tree';
import {
 deleteUserById,
 getSpouseByIdAsync,
 getUserByIdAsync,
 hideSpouseActionSheet,
 resetNewUserData,
 resetSpouseData,
 userSelector,
} from '~redux/slices/user';

import {ZoomableContainer} from '~shared';
import {TreeTypes, TreeMemberActionSheetOptions} from '~utils';
import {noGenderAvatarImage} from '~images';
import {RootTabParamList} from '~navigation/TabNavigator';

import {ITreeMemberDescendant} from '~interfaces/ITreeMemberDescendant';
import {IRelative} from '~interfaces/IRelative';
import {IUser} from '~interfaces/IUser';
import {TreePart} from '~types/TreePart';
import {AddRelativesActionSheetOptions} from
'~types/AddRelativesActionSheetOptions';

import AddUserToTreeModal from './components/AddUserToTreeModal';
import treeStyles from './treeStyles';

import navigateToUserProfile from './Helpers/NavigationHelper';
import {
 isRootMember,
 hasChildren,
 isMarried,
 isDescendantOrSpouse,
 isParentLineDuplication,
 renderDescendantNameDependingOnGender,
 renderDescendantAvatarDependingOnGender,
 renderSpouseNameDependingOnGender,
 renderSpouseAvatarDependingOnGender,
 isMotherOfRootTreeMember,
 shouldRenderSubTree,
 shouldRenderRootTree,
 shouldSwapSubTreeAndRootTree,
 isSpouseOfAuthUser,
} from './Helpers/RenderTreeMembersHelpers';
import {
 getRootTreeMemberActionSheetOptions,
 getDefaultTreeMemberOptions,
 getTreeMemberSpouseOptions,
 getFinalTreeMemberOptions,
} from './Helpers/ActionSheetHelpers';
import {
 getVerticalTopLineColorOfTreeMember,
 getHorizontalLinesFromTopTreesToMainTree,
 getTopHorizontalLineOfSpouse,
 getTopHorizontalLineOfDescendant,
} from './Helpers/RenderTreeLinesHelpers';
import {swapWidthAndHeightInPlaces} from './Helpers/TopTreesRenderHelpers';
import {
```

```
 getDescendantIDDependingOnGender,
 getSpouseIDDependingOnGender,
} from './Helpers/GetTreeMemberIdHelpers';
import {renderChildren} from './Helpers/RenderChildren';
import {getGetTreeInPartsByIdAsyncUserIdOfAuthUserTreeParamHelper} from
'./Helpers/GetRequestsParamsHelpers';
import {useAppDispatch} from '~hooks/redux';

type TreeScreenProps = StackScreenProps<RootTabParamList, 'tree'>;

const siblingGap = 3;
const nodeTitleColor = '#252A31';
const pathColor = '#E4E4E4';
const strokeWidth = 3;

const TreeScreen: FC<TreeScreenProps> = ({navigation}) => {
 const {nodeStyle, nodeTitleStyle, imageStyle} = treeStyles;

 const dispatch = useAppDispatch();

 const authUser = useSelector(authSelector.getAuthUser);

 const tree = useSelector(treeSelector.getTree);
 const treeLoading = useSelector(treeSelector.getTreeLoading);
 const user = useSelector(userSelector.getUser);
 const spouseLoading = useSelector(userSelector.getSpouseLoading);

 const [isScreenFocused, setIsScreenFocused] = useState(false);

 useFocusEffect(
  useCallback(() => {
   setIsScreenFocused(true);
   return () => {
    setIsScreenFocused(false);
   };
  })
);

 useDeviceOrientationChange(() => {
  Orientation.getAutoRotateState(rotationEnabled => {
   isScreenFocused && rotationEnabled
    ? Orientation.unlockAllOrientations()
    : Orientation.lockToPortrait();
  });
 });

 const getAuthUser = useCallback(async () => {
  if (authUser?.data?.id) {
   const authUserResponse = await dispatch(getUserByIdAsync(authUser?.data?.id));
   if (authUserResponse) {
    dispatch(saveAuthUser({...authUserResponse.payload, isInitialFetchFinished:
true}));
   }
  }
 }, [authUser.data?.id, dispatch]);

 useEffect(() => {
  getAuthUser();
 }, [getAuthUser]);

 const [isModalVisible, setIsModalVisible] = useState<boolean>(false);

 const [selectedUserId, setSelectedUserId] = useState<string>('');

 const [selectedTreeMemberThatHoldsChildren,
setSelectedTreeMemberThatHoldsChildren] =
  useState<ITreeMemberDescendant | null>();

 const [newUserRoleInTheTreeForSelectedUser,
setNewUserRoleInTheTreeForSelectedUser] =

useState<TreeMemberActionSheetOptions>(TreeMemberActionSheetOptions.AddCh
ild);

 const [initialOffsetY, setInitialOffsetY] = useState(0);
 const [containerHeight, setContainerHeight] = useState(0);

 useEffect(() => {
  if (
   user.data?.id &&
   authUser.data?.id &&
   authUser.data?.myTreeIdByParent1 &&
```

```
    authUser.data?.isInitialFetchFinished
  ) {
    if (user?.recentlyDeletedUserId) {
      dispatch(
        getTreeInPartsByIdAsync({
          treeId: tree?.data?.id,
          userId:
            tree?.data?.bottomPartTree[0]?.descendant?.length > 0 &&
            user?.recentlyDeletedUserId !==
              tree?.data?.bottomPartTree[0]?.descendant[0]?.user?.identity
              ? tree?.data?.bottomPartTree[0]?.descendant[0]?.user?.identity
              : tree?.data?.bottomPartTree[0]?.user?.identity,
        })
      );
    } else {
      if (
        tree?.data?.id &&
        tree?.data?.id !== authUser?.data?.myTreeIdByParent1 &&
        tree?.treeViewer
      ) {
        dispatch(getTreeInPartsByIdAsync({treeId: tree?.data?.id, userId:
tree?.treeViewer}));
      } else {
        dispatch(
          getTreeInPartsByIdAsync({
            treeId: authUser?.data?.myTreeIdByParent1,
            userId: getGetTreeInPartsByIdAsyncUserIdOfAuthUserTreeParamHelper(
              tree.data?.bottomPartTree
                ? tree.data?.bottomPartTree[0]?.user?.identity
                : authUser?.data?.id,
              authUser?.data?.id,
              newUserRoleInTheTreeForSelectedUser,
              tree.data?.bottomPartTree &&
tree.data?.bottomPartTree[0]?.descendant?.length > 0,
              tree.data?.bottomPartTree &&
                tree.data?.bottomPartTree[0]?.descendant[0]?.user?.identity
            ),
          })
        );
      }
    }
  }, [
    dispatch,
    authUser?.data?.myTreeIdByParent1,
    tree?.newUserJoinedId,
    user?.recentlyDeletedUserId,
    authUser?.data?.isInitialFetchFinished,
  ]);

  useEffect(() => {
    if (user?.newUserData?.id && selectedUserId) {
      dispatch(
        updateTreeByIdAsync({
          treeId: tree?.data?.id,
          toUserId: selectedUserId,
          newUserId: user?.newUserData?.id,
          roleType: newUserRoleInTheTreeForSelectedUser,
          isRootUser: isRootMember(selectedUserId,
tree?.data?.bottomPartTree[0]?.user?.identity),
          isWifeOfRootUser: selectedUserId ===
tree?.data?.bottomPartTree[0]?.married[0]?.identity,
          treeMemberThatHoldsChildren: selectedTreeMemberThatHoldsChildren,        })
      );
      dispatch(resetNewUserData());
      setIsModalVisible(false);
      setSelectedTreeMemberThatHoldsChildren(null);
    }
  }, [dispatch, user?.newUserData]);

  const {width, height} = useWindowDimensions();
  const {showActionSheetWithOptions} = useActionSheet();

  const onOpenActionSheet = (
    userId: string,
    userSpouses: IRelative[],
    isUserHasChildren: boolean,
    isSpouse: boolean,
    memberItem: ITreeMemberDescendant | IRelative,
    descendantLineItem: ITreeMemberDescendant,
```

```
  treeType: TreeTypes,
  spouseDataResponse?: IUser
) => {
  const userInterfaceStyle: 'light' | 'dark' | undefined = 'light';

  const basicOptions: AddRelativesActionSheetOptions = isRootMember(
    userId,
    tree?.data?.[treeType][0]?.user?.identity
  )
    ? getRootTreeMemberActionSheetOptions(treeType, userSpouses,
tree?.data?.rootPartTree)
    : getDefaultTreeMemberOptions(treeType, userSpouses);
  const options: AddRelativesActionSheetOptions = isSpouse
    ? getTreeMemberSpouseOptions(
        isUserHasChildren,
        treeType,
        spouseDataResponse?.parents ? spouseDataResponse?.parents?.length > 0 :
false,
        authUser.data?.id === userId
      )
    : getFinalTreeMemberOptions(
        isUserHasChildren,
        basicOptions,
        userId === authUser.data?.id,
        isRootMember(userId, tree?.data?.[treeType][0]?.user?.identity),
        userSpouses
      );

  const cancelButtonIndex: number = options.length - 1;
  const destructiveButtonIndex: number | undefined = options.includes(
    TreeMemberActionSheetOptions.DeleteTreeMember
  )
    ? options.length - 2
    : undefined;
  const gotToUserTreeButtonIndex: 0 | undefined = options.includes(
    TreeMemberActionSheetOptions.ViewMembersTree
  )
    ? 0
    : undefined;

  showActionSheetWithOptions(
    {
      options,
      cancelButtonIndex,
      destructiveButtonIndex,
      userInterfaceStyle,
      useModal: true,
    },
    (buttonIndex: number | undefined): void | Promise<void> => {
      if (isSpouse && !isSpouseOfAuthUser(descendantLineItem.user.identity,
authUser?.data?.id)) {
        dispatch(setTreeViewer(descendantLineItem.user.identity));
      } else {
        dispatch(setTreeViewer(userId));
      }
      if (
        buttonIndex !== gotToUserTreeButtonIndex &&
        buttonIndex !== destructiveButtonIndex &&
        buttonIndex !== cancelButtonIndex
      ) {
        setNewUserRoleInTheTreeForSelectedUser(options[buttonIndex as number]);
        setIsModalVisible(true);
        if (isSpouse) {
          setSelectedTreeMemberThatHoldsChildren(descendantLineItem);
        }
      } else {
        if (
          buttonIndex === destructiveButtonIndex &&
          buttonIndex !== gotToUserTreeButtonIndex &&
          buttonIndex !== cancelButtonIndex
        ) {
          dispatch(deleteUserById(userId));
        }
        if (
          buttonIndex === gotToUserTreeButtonIndex &&
          buttonIndex !== destructiveButtonIndex &&
          buttonIndex !== cancelButtonIndex
        ) {
          if (treeType === TreeTypes.MotherAncestryTreeLine) {
            const subTreeOwner = tree?.data?.bottomPartTree[0]?.married[0];
```

```
            dispatch(
              getTreeInPartsByIdAsync({
                treeId:
                  memberItem?.properties?.myTreeIdByParent1 ||
                  subTreeOwner?.properties?.myTreeIdByParent1 ||
                  authUser?.data?.myTreeIdByParent1,
                userId,
              })
            );
          } else {
            dispatch(
              getTreeInPartsByIdAsync({
                treeId:
                  memberItem?.properties?.myTreeIdByParent1 ||
authUser?.data?.myTreeIdByParent1,
                userId,
              })
            );
          }
          setSelectedUserId('');
          dispatch(resetSpouseData());
          dispatch(hideSpouseActionSheet());
        }
        if (buttonIndex === cancelButtonIndex) {
          setIsModalVisible(false);
        }
      }
    }
  );
};


const onTreeMemberPress =
  (isDescendant: boolean, treeMemberIdentity: string, spouseIdentity: string): (() =>
void) =>
  (): void => {
    if (isDescendant) {
      navigateToUserProfile(treeMemberIdentity, navigation);
    } else {
      navigateToUserProfile(spouseIdentity, navigation);
    }
  };


const onDescendantLongPress =
  (isDescendant: boolean, item: ITreeMemberDescendant, treeType: TreeTypes) =>
async () => {
    if (isDescendant) {
      setSelectedUserId(item.user.identity);
      onOpenActionSheet(
        item.user.identity,
        item.married,
        hasChildren(item),
        false,
        item,
        item,
        treeType
      );
    } else {
      try {
        const spouseResponse = await
dispatch(getSpouseByIdAsync(item?.married[0]?.identity));
        if (spouseResponse) {
          setSelectedUserId(item?.married[0]?.identity);
          onOpenActionSheet(
            item?.married[0]?.identity,
            item?.married,
            hasChildren(item),
            true,
            item?.married[0],
            item,
            treeType,
            spouseResponse.payload
          );
        }
      } catch (error) {
        console.log('catch 1 error :>> ', error);
      }
    }
  };


const onSpouseLongPress =
```

```
(
  isDescendant: boolean,
  descendantLineItem: ITreeMemberDescendant,
  spouse: IRelative,
  treeType: TreeTypes
): (() => void) =>
async (): Promise<void> => {
  if (isDescendant) {
    try {
      const spouseResponse = await dispatch(
        getSpouseByIdAsync(descendantLineItem?.married[0]?.identity)
      );
      if (spouseResponse) {
        setSelectedUserId(spouse.identity);
        onOpenActionSheet(
          spouse.identity,
          descendantLineItem.married,
          hasChildren(descendantLineItem),
          true,
          spouse,
          descendantLineItem,
          treeType,
          spouseResponse.payload
        );
      }
    } catch (error) {
      console.log('catch 2 error :>> ', error);
    }
  } else {
    setSelectedUserId(descendantLineItem.user.identity);
    onOpenActionSheet(
      descendantLineItem.user.identity,
      descendantLineItem.married,
      hasChildren(descendantLineItem),
      false,
      descendantLineItem,
      descendantLineItem,
      treeType
    );
  }
};

const onBottomTreeLayout = (event: LayoutChangeEvent): void => {
  setInitialOffsetY(event.nativeEvent?.layout?.y);
  setContainerHeight(event.nativeEvent?.layout?.height);
};

const renderTree = (
  data: TreePart,
  treeType: TreeTypes = TreeTypes.MainTree,
  level: number,
  parentIdentity?: string
) => {
  let rootGrandParent: ITreeMemberDescendant;
  let subGrandParent: ITreeMemberDescendant;

  if (treeType === TreeTypes.FatherAncestryTreeLine && level === 1) {
    const retrieveRootGrandParent = (treeMember: ITreeMemberDescendant): void
=> {
      if (rootGrandParent !== undefined) return;

      treeMember.descendant?.forEach(child => {
        if (
          (child as ITreeMemberDescendant)?.user?.identity === parentIdentity ||
          (child as IRelative)?.identity === parentIdentity
        )
          rootGrandParent = treeMember;
        else retrieveRootGrandParent(child as ITreeMemberDescendant);
      });
    };

    if (data?.length > 0) retrieveRootGrandParent(data[0]);
    data = [rootGrandParent];
    level = -2;
  }

  if (treeType === TreeTypes.MotherAncestryTreeLine && level === 1) {
    const retrieveRootGrandParent = (treeMember: ITreeMemberDescendant): void
=> {
```

```
    if (subGrandParent !== undefined) return;

    treeMember.descendant?.forEach(child => {
      if (
        (child as ITreeMemberDescendant)?.user?.identity === parentIdentity ||
        (child as IRelative)?.identity === parentIdentity
      )
        subGrandParent = treeMember;
      else retrieveRootGrandParent(child as ITreeMemberDescendant);
    });
  };

  if (data?.length > 0) retrieveRootGrandParent(data[0]);
  data = [subGrandParent];
  level = -2;
}

return data?.map((item: ITreeMemberDescendant, index: number) => {
  if (
    item?.user &&
    !isMotherOfRootTreeMember(
      tree?.data?.bottomPartTree[0]?.married[0]?.identity,
      item?.user?.identity
    )
  ) {
    return (
      <HStack
        key={`${item?.user?.properties?.firstName} +
${item?.user?.properties?.lastName} + ${item?.user?.identity} + ${index} +
${level}`}
        style={treeStyles.mainContainer}
      >
        <Box alignItems="center" paddingLeft={siblingGap / 2}
paddingRight={siblingGap / 2}>
          <HStack alignItems="flex-start">
            <VStack alignItems="center">
              {
                <Svg height="25" width="100%">
                  {!rootGrandParent && (
                    <Line
                      x1="50%"
                      y1="0"
                      x2="50%"
                      y2="100%"
                      stroke={getVerticalTopLineColorOfTreeMember(
                        isRootMember(
                          item?.user?.identity,
                          tree?.data?.[treeType][0]?.user?.identity
                        ),
                        shouldSwapSubTreeAndRootTree(tree?.data?.bottomPartTree[0]),
                        shouldRenderRootTree(tree?.data?.rootPartTree),
                        shouldRenderSubTree(
                          tree?.data?.bottomPartTree[0]?.married[0]?.identity,
                          tree?.data?.subTree?.length > 0
                            ? tree?.data?.subTree[0]?.enterPointToSubTree[0]?.identity
                            : null
                        ),
                        isDescendantOrSpouse(item,
getDescendantIDDependingOnGender(item)),
                        false,
                        treeType,
                        isSpouseOfAuthUser(item?.user?.identity, authUser.data?.id),
                        pathColor
                      )}
                      strokeWidth={strokeWidth}
                    />
                  )}
                  <Svg>
                    {getTopHorizontalLineOfDescendant(
                      isRootMember(
                        item?.user?.identity,
                        tree?.data?.[treeType][0]?.user?.identity
                      ),
                      isMarried(item),
                      isDescendantOrSpouse(item,
getDescendantIDDependingOnGender(item)),
                      treeType,
                      strokeWidth,
                      pathColor,
                      shouldSwapSubTreeAndRootTree(tree?.data?.bottomPartTree[0]),
                      shouldRenderRootTree(tree?.data?.rootPartTree),
```

```
                shouldRenderSubTree(
                  tree?.data?.bottomPartTree[0]?.married[0]?.identity,
                  tree?.data?.subTree?.length > 0
                    ? tree?.data?.subTree[0]?.enterPointToSubTree[0]?.identity
                    : null
                ),
                isSpouseOfAuthUser(item?.user?.identity, authUser.data?.id)
              )}
            </Svg>
          </Svg>
        }

        <TouchableOpacity
          onPress={onTreeMemberPress(
            isDescendantOrSpouse(item,
getDescendantIDDependingOnGender(item)),
            item.user.identity,
            item?.married[0]?.identity
          )}
          onLongPress={onDescendantLongPress(
            isDescendantOrSpouse(item,
getDescendantIDDependingOnGender(item)),
            item,
            treeType
          )}
          style={nodeStyle}
        >
          <FastImage
            source={
              renderDescendantAvatarDependingOnGender(item)
                ? {
                    uri: renderDescendantAvatarDependingOnGender(item),
                  }
                : noGenderAvatarImage
            }
            resizeMode={FastImage.resizeMode.cover}
            style={imageStyle}
          />
          <Text
            numberOfLines={1}
            fontFamily="Roboto-Regular"
            color={nodeTitleColor}
            style={nodeTitleStyle}
          >
            {renderDescendantNameDependingOnGender(item)}
          </Text>
        </TouchableOpacity>
      </VStack>

      <VStack>
        {isMarried(item) && (
          <Box
            alignSelf={
              isDescendantOrSpouse(item,
getDescendantIDDependingOnGender(item))
                ? 'flex-start'
                : 'flex-end'
            }
          >
            <Svg height="53" width="14">
              <Line
                x1="100%"
                y1={strokeWidth / 2}
                x2="0"
                y2={strokeWidth / 2}
                stroke={
                  !rootGrandParent &&
                  !isRootMember(
                    item?.user?.identity,
                    tree?.data?.[treeType][0]?.user?.identity
                  )
                    ? pathColor
                    : 'white'
                }
                strokeWidth={strokeWidth}
              />
            </Svg>
          </Box>
        )}
```

```
{isMarried(item) && (
 <Box marginTop={3} alignItems="flex-end">
  <Svg height="85" width="25">
   <Line
    x1="100%"
    y1={strokeWidth / 2}
    x2="0"
    y2={strokeWidth / 2}
    stroke={pathColor}
    strokeWidth={strokeWidth}
   />
   {hasChildren(item) && (
    <Line
     x1="50%"
     y1="0"
     x2="50%"
     y2="100%"
     stroke={pathColor}
     strokeWidth={strokeWidth}
    />
   )}
  </Svg>
 </Box>
)}
</VStack>


{
 isMarried(item) &&
  item.married.map((spouse, spouseIndex) => {
   return (
    <VStack
     key={`
      ${spouse?.properties?.firstName} +
      ${spouse?.properties?.lastName} +
      ${spouse?.identity} +
      ${spouseIndex} +
      ${level}
      `}
    >
```

```
{
 <Svg height="25" width="100%">
  <Line
   x1="50%"
   y1="0"
   x2="50%"
   y2="100%"
   stroke={getVerticalTopLineColorOfTreeMember(
    isRootMember(
     item?.user?.identity,
     tree?.data?.[treeType][0]?.user?.identity
    ),
shouldSwapSubTreeAndRootTree(tree?.data?.bottomPartTree[0]),
    shouldRenderRootTree(tree?.data?.rootPartTree),
    shouldRenderSubTree(
     tree?.data?.bottomPartTree[0]?.married[0]?.identity,
     tree?.data?.subTree?.length > 0
      ? tree?.data?.subTree[0]?.enterPointToSubTree[0]?.identity
      : null
    ),
    isDescendantOrSpouse(
     item,
     getSpouseIDDependingOnGender(spouse, item)
    ),
    true,
    treeType,
    isSpouseOfAuthUser(item?.user?.identity, authUser.data?.id),
    pathColor
   )}
   strokeWidth={strokeWidth}
  />
  {getTopHorizontalLineOfSpouse(
   isRootMember(
    item?.user?.identity,
    tree?.data?.[treeType][0]?.user?.identity
   ),
   isDescendantOrSpouse(
    item,
    getSpouseIDDependingOnGender(spouse, item)
```

```
                ),
                treeType,
                strokeWidth,
                pathColor,
                shouldSwapSubTreeAndRootTree(tree?.data?.bottomPartTree[0]),
                shouldRenderRootTree(tree?.data?.rootPartTree),
                shouldRenderSubTree(
                  tree?.data?.bottomPartTree[0]?.married[0]?.identity,
                  tree?.data?.subTree?.length > 0
                    ? tree?.data?.subTree[0]?.enterPointToSubTree[0]?.identity
                    : null
                ),
                isSpouseOfAuthUser(item?.user?.identity, authUser.data?.id)
              )}
            </Svg>
          }
          <TouchableOpacity
            onPress={onTreeMemberPress(
              isDescendantOrSpouse(item,
getDescendantIDDependingOnGender(item)),
              spouse.identity,
              item.user.identity
            )}
            onLongPress={onSpouseLongPress(
              isDescendantOrSpouse(item,
getDescendantIDDependingOnGender(item)),
              item,
              spouse,
              treeType
            )}
            style={nodeStyle}
          >
            <FastImage
              source={
                renderSpouseAvatarDependingOnGender(spouse, item)
                  ? {
                      uri: renderSpouseAvatarDependingOnGender(spouse, item),
                    }
                  : noGenderAvatarImage
              }
              resizeMode={FastImage.resizeMode.cover}
              style={imageStyle}
            />
            <Text
              numberOfLines={1}
              fontFamily="Roboto-Regular"
              color={nodeTitleColor}
              style={nodeTitleStyle}
            >
              {renderSpouseNameDependingOnGender(spouse, item)}
            </Text>
          </TouchableOpacity>
        </VStack>
      );
    })
  }
</HStack>

{treeType === TreeTypes.FatherAncestryTreeLine ||
treeType === TreeTypes.MotherAncestryTreeLine ? (
  isMarried(item) ? (
    <Box
      justifyContent="center"
      alignItems="center"
      width={!isMarried(tree?.data?.bottomPartTree[0]) ? '100%' : null}
    >
      <Svg height={scale(80)} width="100">
        <Line
          x1="50%"
          y1="0"
          x2="50%"
          y2="150"
          stroke={pathColor}
          strokeWidth={strokeWidth}
        />
      </Svg>
      <Svg
        height="3"
        width={!isMarried(tree?.data?.bottomPartTree[0]) ? '100%' : '100'}
```

```
    >
      {getHorizontalLinesFromTopTreesToMainTree(
        shouldSwapSubTreeAndRootTree(tree?.data?.bottomPartTree[0]),
        treeType,
        strokeWidth,
        pathColor,
        !isMarried(tree?.data?.bottomPartTree[0])
      )}
    </Svg>
  </Box>
) : (
  <Box
    borderWidth={0}
    borderColor={'green.300'}
    width={
      treeType === TreeTypes.FatherAncestryTreeLine &&
      !isMarried(item) &&
      !isMarried(tree?.data?.bottomPartTree[0])
        ? scale(200)
        : scale(150)
    }
  >
    <Svg height="70" width="20" style={treeStyles.alignSelfCenter}>
      <Line
        x1="50%"
        y1="0"
        x2="50%"
        y2="150"
        stroke={pathColor}
        strokeWidth={strokeWidth}
      />
    </Svg>
    <Svg height="34" width="20" style={treeStyles.alignSelfCenter}>
      <Line
        x1="50%"
        y1="0"
        x2="50%"
        y2="150"
        stroke={pathColor}
        strokeWidth={strokeWidth}
      />
    </Svg>
    <Svg height="3" width="100%" style={treeStyles.alignSelfCenter}>
      {getHorizontalLinesFromTopTreesToMainTree(
        shouldSwapSubTreeAndRootTree(tree?.data?.bottomPartTree[0]),
        treeType,
        strokeWidth,
        pathColor,
        !isMarried(tree?.data?.bottomPartTree[0])
      )}
    </Svg>
  </Box>
)
) : (
  hasChildren(item) && (
    <Svg height="35" width="100%">
      <Line
        x1="50%"
        y1="0"
        x2="50%"
        y2="150"
        stroke={pathColor}
        strokeWidth={strokeWidth}
      />
    </Svg>
  )
)}
<HStack>
  {hasChildren(item) &&
    !isParentLineDuplication(item, treeType, parentIdentity) &&
    renderChildren(
      (item as ITreeMemberDescendant)?.descendant as TreePart,
      hasChildren(item),
      renderTree,
      treeType,
      level,
      strokeWidth,
      pathColor,
```

```
                parentIdentity
              )}
            </HStack>
          </Box>
        </HStack>
      );
    }
  });
};

if (treeLoading) {
  return <Spinner visible={treeLoading} color="#E8AD63" />;
} else {
  return (
    <>
      {spouseLoading && !treeLoading && (
        <ActivityIndicator size="large" color="#E8AD63"
style={treeStyles.spinnerStyle} />
      )}
      <ZoomableContainer
        authUserId={authUser.data?.id}
        authUserTreeId={authUser.data?.myTreeIdByParent1}
        initialOffsetY={
          containerHeight < 200
            ? 0
            : initialOffsetY < 100
            ? -300
            : containerHeight > 850
            ? -100
            : containerHeight < 500
            ? 300
            : 0
        }
      >
        <Box>
          <VStack justifyContent="center" alignItems="center">
            <HStack alignItems="flex-end">
              <VStack style={treeStyles.topTreesContainersStyle}>
                <Box alignItems="center" width={swapWidthAndHeightInPlaces(width,
height, 0.5)}>
                  {shouldSwapSubTreeAndRootTree(
                    tree?.data?.bottomPartTree && tree?.data?.bottomPartTree[0]
                  )
                    ? renderTree(
                        shouldRenderSubTree(
                          tree?.data?.bottomPartTree[0]?.married[0]?.identity,
                          tree?.data?.subTree?.length > 0
                            ? tree?.data?.subTree[0]?.enterPointToSubTree[0]?.identity
                            : null
                        ) &&
                          !!authUser?.data?.id &&
                          !isRootMember(
                            authUser.data?.id,
                            tree?.data?.bottomPartTree[0]?.user?.identity
                          )
                          ? tree?.data?.subTree
                          : null,
                        TreeTypes.MotherAncestryTreeLine,
                        1,
                        tree?.data?.bottomPartTree[0]?.married[0]?.identity
                      )
                    : renderTree(
                        tree?.data?.rootPartTree,
                        TreeTypes.FatherAncestryTreeLine,
                        1,
                        tree?.data?.bottomPartTree &&
                          tree?.data?.bottomPartTree[0]?.user?.identity
                      )}
                </Box>
              </VStack>
              <VStack style={treeStyles.topTreesContainersStyle}>
                <Box alignItems="center" width={swapWidthAndHeightInPlaces(width,
height, 0.5)}>
                  {shouldSwapSubTreeAndRootTree(
                    tree?.data?.bottomPartTree && tree?.data?.bottomPartTree[0]
                  )
                    ? renderTree(
                        tree?.data?.rootPartTree,
                        TreeTypes.FatherAncestryTreeLine,
                        1,
```

```
                tree?.data?.bottomPartTree &&
                  tree?.data?.bottomPartTree[0]?.user?.identity
                )
              : renderTree(
                shouldRenderSubTree(
                  tree?.data?.bottomPartTree &&
                    tree?.data?.bottomPartTree[0]?.married[0]?.identity,
                  tree?.data?.subTree?.length > 0
                    ? tree?.data?.subTree[0]?.enterPointToSubTree[0]?.identity
                    : null
                ) &&
                  !!authUser?.data?.id &&
                  !isRootMember(
                    authUser.data?.id,
                    tree?.data?.bottomPartTree &&
                      tree?.data?.bottomPartTree[0]?.user?.identity
                  )
                  ? tree?.data?.subTree
                  : null,
                TreeTypes.MotherAncestryTreeLine,
                1,
                tree?.data?.bottomPartTree &&
                  tree?.data?.bottomPartTree[0]?.married[0]?.identity
              )}
          </Box>
        </VStack>
      </HStack>
    </VStack>
    <VStack alignItems="center" onLayout={onBottomTreeLayout}>
      {renderTree(
        tree?.data?.bottomPartTree,
        TreeTypes.MainTree,
        tree?.data?.bottomPartTree &&
tree?.data?.bottomPartTree[0]?.descendant?.length > 0
          ? -1
          : 0
      )}
    </VStack>
    <AddUserToTreeModal
      isModalVisible={isModalVisible}
      setIsModalVisible={setIsModalVisible}
    />
    </Box>
  </ZoomableContainer>
 </>
 );
}
};
export default TreeScreen;
```