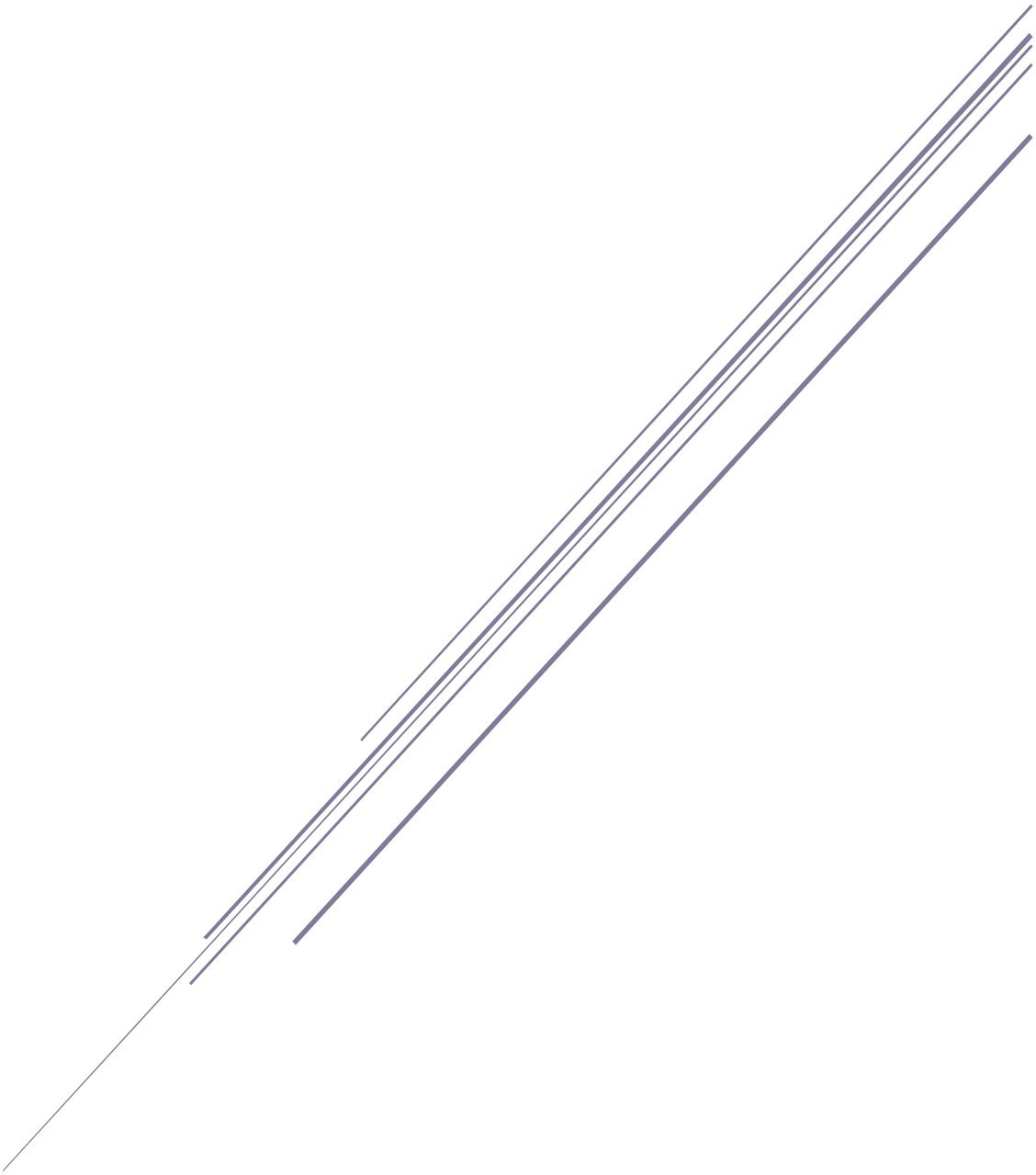
Generation Connect: A Collaborative Family History Platform

Technical Report for Dissertation



Author: Purnima Maheshwari

Word Count: 19,590 (without code snippet)

(excluding Titles of Contents, Chapter Overview Page, Titles, Heading, Program code, References, Images Caption and Appendix

Declaration

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do these, amounts to plagiarism and will be considered grounds for failure in this module and the degree examination.

Name: Purnima Maheshwari

Date: May 16th, 2025

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Newman Lau, for his exceptional guidance, unwavering support, and thoughtful mentorship throughout the course of this project. His contributions to report writing at various stages, along with his innovative ideas for exploring new design features, were invaluable. His expertise and critical insights played a vital role in shaping both the direction and quality of this work.

I am also sincerely thankful to the course staff, particularly Professor John Drake, whose sessions provided clear and structured guidance, enabling me to navigate each task and deadline effectively.

I would like to acknowledge one of my dearest friend for his consistent encouragement and intellectual companionship throughout this journey. His perspective and experience greatly supported my learning of the technologies used in this project and provided both clarity and motivation during key moments.

I am also deeply grateful to Tanvi Ambre and Pratik Khalane for their insightful suggestions—especially encouraging me to explore UI design—and for consistently pushing me to do my best. Their support, both technically and personally, enriched my experience and helped me stay focused and inspired throughout.

Thank you, everyone.

Table of Contents

Glossary of Terms

Term	Definition
Generations Connect	A collaborative, web-based genealogy platform to preserve family history through trees and multimedia.
Family Tree	A hierarchical model showing individuals and their family relationships across generations.
Owner	Has full control over a family tree, including user management, editing, and deletion.
Contributor	Can add stories, media, and tag records but cannot manage users or delete the tree.
Viewer	Has read-only access to the tree; cannot modify or add any content.
Public Tree	Accessible to all platform users, but editable only by Contributors and Owners.
Private Tree	Visible only to invited users with specific roles assigned on joining the Family Tree.
Tagged Record	An archival document linked to a family tree profile for historical reference.
Media Integration	Feature allowing users to upload and view photos, videos, and documents.
Access Management	Admin capability to assign roles, remove users, and manage permissions.
Profile Post	A story or media entry linked to an individual in the family tree.
Comment Thread	A series of user comments attached to a post for discussion and storytelling.
Invite Token	A unique email token used to invite users to private trees securely.
Neo4j	A graph database used for storing and navigating family relationships.
PostgreSQL	A relational database used for user accounts, roles, posts, and structured data.
Spring Boot	Java framework used for building backend REST APIs and services.
React	JavaScript library used to build the frontend UI of the application.
D3.js	JavaScript library for dynamic, interactive data visualizations such as family trees.
JWT (JSON Web Token)	Used for stateless authentication and user session validation.
Docker	A tool for containerizing application components to ensure consistent deployment.
Docker Compose	Defines and runs multi-container Docker applications using a YAML config.

NGINX	Acts as a reverse proxy, load balancer, and gateway for frontend/backend routing.
Lua Scripts	Used with NGINX to implement custom JWT-based access control.
Reverse Proxy	Server that forwards client requests to backend services while hiding internal architecture.
Load Balancer	Distributes incoming traffic evenly across multiple backend services.
OpenResty	An NGINX-based platform extended with Lua for dynamic routing and validation.
SSL/TLS	Encryption protocols used to secure HTTP connections and data transmissions.
Role-Based Access Control	Defines user permissions based on roles like Owner, Contributor, Viewer.
Stateless Session	Each request is authenticated independently without server-stored sessions.
Audit Logging	Tracks user activities events for monitoring.
Research API Integration	Links with public archives (e.g., National Archives UK) to search and tag records.
REST (Representational State Transfer)	API design style using standard HTTP methods for CRUD operations.
CORS (Cross-Origin Resource Sharing)	Controls access permissions between frontend and backend servers from different origins.
Cypher Query Language (CQL)	Neo4j's language used to define and traverse relationships between graph nodes.
API (Application Programming Interface)	A set of rules enabling communication between software components, such as frontend, backend, and external services.
HTTP (Hypertext Transfer Protocol)	A protocol used for transmitting data over the web, enabling communication between clients (like browsers) and servers.
HTTPS (Hypertext Transfer Protocol Secure)	A secure version of HTTP that encrypts data between the client and server to protect against interception and tampering.

01

Abstract

The Abstract provides a high-level overview of the project, outlining its purpose, motivation, and key objectives. Additionally, it offers a guide to the structure of the report, explaining how each section contributes to the overall understanding of the project.

1. Abstract

1.1 Project Overview

Family history is a rich tapestry of stories, memories, and relationships passed down through generations. However, these invaluable narratives often fade over time, leaving gaps in context and meaning [1][2].

The aim of “Generations Connect” is to preserve and share family heritage in an interactive and meaningful way. Designed as a collaborative platform, it enables users to explore, document, and relive their genealogy [1][3].

This web application transforms family history into a living archive by allowing users to create interactive family trees with zoom in-and-out functionality and visualized node connections. Users can record personal stories for everyone individually, create a shared family profile for all members of the tree, and incorporate multimedia elements such as photos, audio, and videos [1][3]. These features also support posting and commenting for both individuals and families.

A key distinction from existing applications is the inclusion of centralized collaborative research tools, enabling multiple family members to contribute to genealogical research. It allows them to rediscover ancestors and long-lost relatives using national archives, tag records, and revisit them through a unified interface [1][3][4][5].

Another key highlight is the integration of AI, allowing users to query the family tree for upcoming events, timelines, and other related questions.

To ensure privacy and security, the platform provides customizable access permissions, allowing families to collaborate while protecting personal information. Additional features, such as inviting users via email to join a family tree, make it easy to build and extend family connections in just a few simple steps.

Ultimately, Generations Connect is more than just a genealogy tool—it serves as a bridge between the past, present, and future, allowing family legacies to be preserved and cherished for generations to come.

Figure 1 Uncovering Ancestors and Growing the Family Story

source: https://www.freepik.com

1.2 Structure of the Report

This report follows a clear and structured flow, starting with an abstract to introduce the project's purpose. It then outlines the aim and objectives, setting the foundation for what the project set out to achieve.

A comparison with existing solutions helps highlight the need for the proposed system, which is then explored the technical side—detailing both functional using a User Story and technical requirements and providing a background on the tools and technologies that shaped the development process. through system architecture, data models, and key design decisions.

The implementation chapters cover Entity Relationship Diagram, Business logic, API Documentation, User Interface design, and real-world usage scenarios based on earlier user stories. They also detail functional testing, testing approaches, and Docker-based deployment to demonstrate how the Generations Connect application was validated.

The final sections reflect on the project's outcomes, challenges faced along the way, and areas for future improvement. Supporting materials such as references, appendices, and related work.

***Note**: The project's planning, development, and testing—covering background research, implementation, reporting, configuration, and architectural setup—is entirely original excerpt, except for the family-chart component's implementation, which is sourced from a GitHub repository[20] as referenced in Section 9.1.*

02

Project Aim & Objectives

This chapter outlines the overarching aim of the project—developing an interactive platform to preserve and share family heritage in a meaningful way.

2. Project Aim & Objectives

2.1 Project Aim

The aim of this project is to design and implement Generations Connect, a full-stack web application that enables users to collaboratively build and manage interactive family trees enriched with multimedia content. The platform integrates user-generated data with structured genealogy features, offering centralized research tools, AI integration to query family tree, customizable privacy controls, and dynamic visualization through zoomable node-based interfaces in family tree. The goal is to create an interactive, secure, and user-friendly system that preserves family history as a shared digital archive, supporting long-term access, contribution, and discovery across generations.

2.2 Specific Objectives

To meet the overarching aim of this project, the system must fulfil a range of functional and non-functional goals that ensure usability, security, and collaboration. The key objectives are as follows:

1.	**Secure User Access and Role-Based Authorization** The platform will support secure sign-up and login processes, ensuring that users are authenticated before accessing any personalized features. It will distinguish between general users and administrators, allowing the latter to perform audits, monitor activity, and manage platform-wide settings.
2.	**Interactive Family Tree Visualization** Users should be able to build and explore multi-generational family trees through an intuitive interface featuring zoom, panning, and dynamic node interactions. The system will present complex family relationships in a visually clear and user-friendly format.
3.	**Storytelling and Interaction** To enhance personal engagement, the system will allow users to post stories, memories, and comments associated with individuals in the family tree. This replaces traditional static formats with dynamic, collaborative storytelling.
4.	**Centralized Family Profiles** Each family tree will be accompanied by a shared family profile that hosts collective memories, events, and multimedia. This creates a centralized space for preserving family identity and shared heritage.

5.	**Public and Private Tree Access** Public family trees will be accessible to all users on the platform, though only Contributors and Owners can make edits. In contrast, private family trees will be accessible by invitation only, and visibility will depend on the assigned role.
6.	**Flexible Privacy Settings and Role Management** The platform will enable tree owners to manage access by assigning roles—Owner, Contributor, and Viewer. Owners will have full control, including inviting users, editing roles, and deleting the tree. Contributors can add content but not manage users, while Viewers will only have read-only access when explicitly invited.
7.	**Integrated Research and Tagging Tools** The system will support integration with open-access archives and directories, allowing users to search for historical records and tag them to family profiles. This functionality will be restricted to Contributors and Owners to ensure data accuracy and relevance.
8.	**Integration with AI** AI will enable users to ask natural-language questions about upcoming events, timelines, and other family tree insights, making exploration more interactive and personalized.

03

Analysis of Existing Solutions

This chapter explores various existing solutions in the market, comparing their capabilities and relevance. It provides insight into their advantages and limitations, forming the basis for defining a more effective and tailored approach for proposed design

3. Analysis of Existing Solutions

Several family tree and genealogy applications currently exist in the market, each catering to different user needs and offering varying levels of functionality. This section presents a structured analysis of relevant existing solutions, with a focus on both general-purpose applications and open-source alternatives.

3.1 Market Alternatives

The following are some of the most prominent applications in the genealogy space:

- Confinity - A platform focused on visualizing family trees with collaborative capabilities.
- iMeUsWe - A culturally rooted application tailored toward Indian heritage, emphasizing traditional lineage and community connections.
- Ancestry - One of the most widely used commercial platforms, offering access to a large historical record database and smart hints for lineage discovery.
- Gramps - A feature-rich, open-source genealogy tool that supports extensive data modelling and research tracking.

While Confinity and Ancestry provide visually rich and data-driven family trees, their functionality is often gated behind subscriptions, and they remain largely proprietary [14][16]. iMeUsWe offers a region-specific experience, delivering curated insights into Indian familial history [15].

Gramps, in contrast, stands out as a community-driven open-source solution aimed at power users and researchers. Gramps is packed with powerful features like detailed relationship tracking, event and source management, citation handling, and a variety of reports and charts. It's highly extensible and designed for serious genealogical work [17].

3.2 Strengths and Limitations of an Existing Solution

Gramps offers a strong benchmark for open-source genealogy systems. It includes capabilities such as:

- Complex relationship and family event modelling
- Source and citation management
- Advanced filtering and data grouping
- Comprehensive reporting and visual charts
- Multilingual support and plugin-based extensibility
- Support for common data exchange formats (XML, CSV)

Despite these strengths, Gramps exhibits several limitations that affect its accessibility and adaptability for modern users:

- **Desktop-Only Interface**: The software is confined to desktop environments, limiting collaborative and cross-platform usage.
- **Steep Learning Curve**: Its interface and dense features can be challenging for users with limited technical background.
- **Lack of Modern UX**: The user experience lacks contemporary design elements such as responsive UI, smooth navigation, or visual interactivity found in web applications [17].

These observations highlight the opportunity for improvement and innovation in the space.

Need for a Family-Research Centric Application

Currently, social media platforms like Facebook, WhatsApp, and Instagram are widely used for communication. However, these platforms are designed for broader social connections, including friends, acquaintances, and even strangers, making them less suitable for maintaining family privacy.

With this application, a structured family tree system will ensure relationship-based identification, limiting interactions to verified family members. By using relationship-based keywords such as 'father of,' 'mother of,' 'brother of,' 'sister of,' etc., the system can accurately build and validate a well-formed family tree.

3.3 Proposed Differentiation: Generation Connect

Generation Connect seeks to bridge the gap between robust genealogical features and modern usability. It is being designed as a web-based, open-source platform that leverages opensource backend dependencies and framework as well as modern frontend libraries such as React and D3.js to offer:

- Intuitive and seamless navigation
- Dynamic and interactive family tree visualizations
- Role-based access control and real-time collaboration
- Integrated document/media handling for enriched profile data
- Full data ownership through two types of trees: Public and Private tree. Contribution privileges can be modified with simple clicks.

Integration of Open AI model

A key differentiator is the integration of an AI-powered assistant, enabling users to explore events, timelines, and relationships within their family tree through natural-language queries—bringing smart, personalized interaction to genealogy. For more information, check out *Section 5.7* of the report.

Integration of External Research Tools

In addition, Generation Connect plans to integrate with external research data sources, enabling direct access to primary archival records:

1. National Archive – UK – *(currently integrated for prototype)*
    - **Website:** Discovery
    - **Developer Resources:** API Documentation
    - **API Access:** Sandbox API

2. National Archive CatLog – USA – *(Planned for future integration)*
    - Scalability for this integration has already been accounted for in both the backend and frontend, with a prototype structure in place to support upcoming features.
    - **Website:** National Archives Research Tools
    - **Application:** National Archives Catalog
    - **API Access:** API Documentation

These integrations allow users to conduct meaningful historical research from within the platform, tag relevant archival documents to individuals or events in their family tree and build a more authentic and documented family history.

With ongoing development, Generation Connect is well-positioned to match and surpass traditional tools, combining deep research capabilities with a beautifully designed, web-first experience—accessible to everyone from casual users to expert genealogists.

04

Product and Functional Requirement

This chapter presents a detailed overview of the platform's core functionality and technical foundation. It includes user stories to illustrate key interactions, outlines essential system features, highlights potential enhancements, and sets a vision for future development and scalability.

4. Product and Functional Requirement

4.1 Functional Requirements (User Story)

The following storyboard illustrates key functional requirements through the perspective of a family using the platform. It demonstrates how users create and manage trees, contribute stories and media, and collaborate on a shared family profile—highlighting core system interactions in a real-world scenario.

A Day in the Life of Alice and Her Family on Generations Connect

Alice, a passionate family historian and retired librarian, decided it was time to document her family's journey for future generations. She signed up for Generations Connect, a platform designed to preserve family heritage in a collaborative and interactive way. As the Owner of her family tree, Alice had full control—managing access, assigning roles, and configuring visibility for each tree.

Using the platform's interactive tree visualization, she began mapping out her lineage—starting with her grandparents, parents, and eventually herself. The system's zoomable node-based interface allowed her to easily explore complex family relationships across generations.

Alice invited her daughter, Carol, to join the project as a Contributor via email. As a new user, Carol was prompted to create an account, log in, and accept the invitation. Upon clicking the invite link, her Contributor access was granted, and she began contributing to the family tree. Carol enriched the profiles by adding personal stories, photos, and even an old, uploaded video of her grandfather. Her son, Dave, also joined as a Contributor, focusing on uploading historical documents and tagging significant records using the platform's centralized research tools.

Alice wondered—what if the platform had AI accessibility? With the newly added AI assistant, she could now ask questions like "Who are my grandfather's siblings?", "Whose birthday is coming up in next month?" or "How is Carol related to Dave?" and get instant, meaningful answers. This made exploring the tree more intuitive and brought a new level of engagement to the experience, especially for younger members like Dave.

Figure 2 Journey with Alice and her family on Generations Connect

source: https://chatgpt.com/share/681fc3b3-f1f4-8009-8a13-87ec898e6dbf

To make the tree more accessible, Alice sets up a new public tree—meaning any registered user on the platform could view it. However, only Contributors and Owners could modify or add content. This allowed distant relatives or curious researchers to explore her family's history while maintaining control over the data.

She also created a separate private family tree for her husband's lineage, which included sensitive documents and less-known connections. This tree was completely hidden from other users unless Alice explicitly invited them. Her granddaughter, Emily, was added as a Viewer, giving her read-only access, while Dave had Contributor rights to help populate and verify records.

Later, Alice's brother, Frank, joined the platform. Curious but not ready to contribute, he was granted Viewer access to the public tree—able to see everything but unable to change any content.

Through Generations Connect, Alice united three generations—her ancestors, her children, and her grandchildren—on a single platform using different family as per her choice. Each story, photo, and record helped weave together their family legacy, now preserved for future generations.

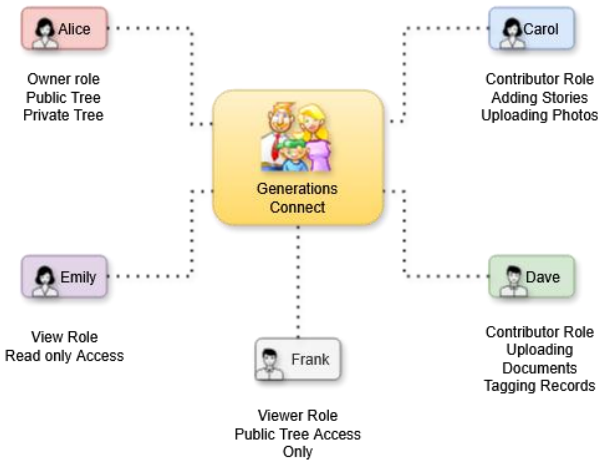


Figure 3 Meet Alice and Her Family:
Defining Their Roles Across Generations on Generations Connect.

4.2 Technical Requirements

4.2.1 Essential Features

- **User Account Creation and Signing in:**

    - **Feature Overview:** Users can create accounts and log in to access the platform. The system supports two roles:

        - Standard users can join family trees and contribute stories or media.
        - Admin users have access to platform metrics and audit logs.

        Authentication is handled via JWT tokens stored in cookies, ensuring secure sessions. A default admin is created on startup to manage the system from the beginning.

    - **Motivation:** Secure sign-up and login ensure that only verified users can access or modify sensitive family data. Role-based access gives admins control over platform activity, helping maintain privacy, track usage, and support future improvements.

- **Interactive Family Trees:**

    - **Feature Overview:** Allowing users to create and visualize multi-generational family trees, including ancestors, descendants, and extended family members. Provide interactive features such as clickable nodes, tooltips, and zooming capabilities for exploring family relationships and connections [1][2][4].
    - **Motivation**: The lives of those who came before—grandparents' parents and beyond—have always sparked curiosity. Stories shared by grandparents offer glimpses of how life used to be, from daily routines to modes of transportation and the joys experienced. However, these stories often feel incomplete. Memories have faded over time, and the few photographs available—just one or two—are not enough to fill the gaps in those recollections. To ensure that future generations do not face the same challenge, a platform is visioned where, with just a tap, descendants can trace the family's journey, uncover forgotten stories, and bring past generations to life.

- **Collaborative Storytelling:**

    - **Feature Overview:** Enable users to add personal stories and memories to individual profiles within the family tree. Facilitate collaboration among family members by allowing multiple contributors to contribute to shared family stories and narratives [1][3][4].
    - **Motivation:** Family stories fade over time unless they are recorded and passed down. Providing a collaborative space for storytelling ensures that personal anecdotes and experiences are preserved for future generations.

- **Media Integration:**

  - **Feature Overview:** This feature allows users to upload, view and retrieve files (e.g., images, videos, PDFs) associated with comments, posts, or tree related entities. Each medium must be linked via media type (such post, research, with future scalability), securely stored with a unique name, and retrieved by folder and filename. Users should also be able to delete the uploaded files or view all media associated with specific content (like a post or family tree) [1][3].
  - **Motivation**: Multimedia enriches family storytelling by adding visual and audio context to memories and profiles. This feature ensures that users can attach relevant files to enhance engagement and preserve family history. Centralized and structured storage makes media easy to access, share, and manage—strengthening the platform as a living archive.

- **Privacy Controls and Access Permissions:**

  - **Feature Overview**: Implement privacy controls to allow users to define the visibility and access permissions for different parts of their family trees and content. Offer options for private, shared(via adding or inviting user to the family tree), or public family trees, with granular control over who can view, edit, or contribute to each tree [6].
  - **Motivation**: Family history is deeply personal, and users should have control over who can access and edit their data. Privacy settings ensure a secure, family-exclusive experience.

- **Collaborative Research Tools:**

  - **Feature Overview:** This feature allows users to search real-time historical records from external sources like the UK National Archives using detailed filters (e.g. names, dates). Users can tag selected records to their family tree, storing them along with metadata and original record data. Duplicate tagging must be avoided, and only users with the appropriate roles (owner or contributor) must be allowed to tag records. Tagged records are retrievable and displayed per tree [1][4][5].
  - **Motivation:** Historical records offer key insights into a family's past but are often scattered across external archives. This feature centralizes relevant discoveries, making it easier for users to connect real data with family stories. Tagging ensures important findings are preserved and shared within the family tree, enriching its historical and cultural value.

4.2.2 Desirable Enhancements

- **Event Tracking and Timeline (Enhanced With AI):**
    - **Feature Overview:** Create a timeline feature that displays key events, milestones, and historical context relevant to the user's family history. Enable users to add and track significant life events such as births, marriages, deaths, immigrations, and migrations within their family timelines [1][3].
    - **Motivation:** Key events shape family history and tracking them helps family members understand the journey of their ancestors. With the use of AI, this feature becomes even more powerful—enabling users to query large family trees and receive personalized, natural-language answers within seconds.

- **Manage User with different roles for different Family trees:**
    - **Feature Overview:** Users can be added to family trees through direct access assignment or invitation links. Only the tree owner can manage access—adding, removing, or assigning roles like contributor or viewer. Users receive email invites with time-limited tokens to join trees securely. The system also tracks who added each user, ensures only authorized members can manage access, and prevents unauthorized modifications, such as changing the owner's role.
    - **Motivation:** Family trees often involve multiple contributors across generations. This feature ensures secure and controlled collaboration by allowing owners to manage access while preserving privacy. Email-based invites make onboarding easy for new family members, and role management ensures that sensitive data stays protected while still enabling rich, collaborative storytelling.

- **Audit Logging and Request Tracing at Administration end:**
    - **Feature Overview:** Maintain logs of user activities for security and compliance tracking [6].
    - **Motivation:** Ensuring data integrity and security is crucial, especially for collaborative platforms where multiple users contribute to shared records.

- **Recent Activity View (Family Tree Member View) - Additional Requirement for Enhancement:**
    - **Feature Overview:** Leverage audit logs to display the recent actions performed within the family tree. Includes collapsible and expandable sections for improved navigation and readability.

- **Motivation:** Provide family tree members with a clear overview of recent activity, enhancing transparency and trust in a collaborative environment where multiple users are involved in maintaining shared data.

- **Usage Metrics at Administration end:**

  - **Feature Overview:** Track and display metrics such as the number of family trees created and, the total number of users, and the frequency of research tool usage, providing insights into platform engagement and activity.
  - **Motivation:** Understanding user engagement helps improve the platform and encourages family members to contribute more actively.

Note: *Community Forums and Discussion has been moved to Vision for Future Development and replaced by Usage Metrics under Desirable Enhancements, in alignment with previous plans and reports.*

4.2.3 Vision for future Development

- **Community forums and Discussion:**
    - **Feature Overview:** Establish community forums and discussion boards where users can connect with other genealogy enthusiasts, share research tips, and seek advice from experienced researchers. Encourage collaboration and knowledge sharing among users by facilitating discussions on genealogical topics, family traditions, and cultural heritage [1].
    - **Motivation:** Sharing knowledge and experiences with like-minded individuals can enhance genealogical research and create a sense of community among family members and researchers.

- **Geographical Mapping of Family History**
    - **Feature Overview:** Provide an interactive map displaying migration patterns and historical locations of ancestors [5].
    - **Motivation:** Visualizing migration patterns can provide a clearer understanding of how families moved across regions over generations.

- **Celebration Features:**
    - **Feature Overview:** Introduce badges for users as they grow and celebrate them, such as their birthday, anniversary or any milestone as they share automatically to their connected family members [2].
    - **Motivation:** Recognizing family milestones fosters a sense of continuity and togetherness, encouraging active participation in preserving family history.

05

Technology and Research Background

This chapter explores the core technologies supporting the Generations Connect platform. It covers containerization using Docker, secure routing via NGINX with Lua scripts, and microservice communication with Spring Security. It also examines Neo4j for graph data management, React charting libraries for visualization, integration of research APIs, and the use of OpenAI for AI-driven interactions.

5. Technology and Research Background

5.1 Docker and Containerization

This background work and research is based on official documentation available for Docker.

“Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications [18].”

To understand Docker in layman’s term - Imagine you’re a chef preparing a meal. You need ingredients (your application code), kitchen tools (libraries and dependencies), and a recipe (your server configuration). Normally, if you want someone else to cook the exact same dish, you'd have to write out the recipe, ensure they have the same tools, and hope it turns out the same.

Docker solves that problem. It packages the ingredients, tools, and recipe into a sealed box—called a Docker container. Now, anyone can take your box, open it in their kitchen (server), and cook the exact same dish—no mess, no surprises. At the foundation of Docker there are several core concepts as defined below:

Dockerfile – Like a recipe card. It lists step-by-step instructions on how to build your container (install dependencies, copying application file, mounting volume, set up configuration such as environment variables and commands to run the application such `mvn clean install` or `yarn install`, etc.).

Docker Image – A snapshot (outcome of Dockerfile processing) of everything your app needs to run after processing Dockerfile. It’s like a blueprint or a frozen copy of your application which is ready to launch anywhere wherever is needed.

Docker Container – A running version of your Docker image. Think of it as a live instance of Docker image that runs the app. You can run as many containers/instances as you want from one image which make it easy to scale services or replicate environments.

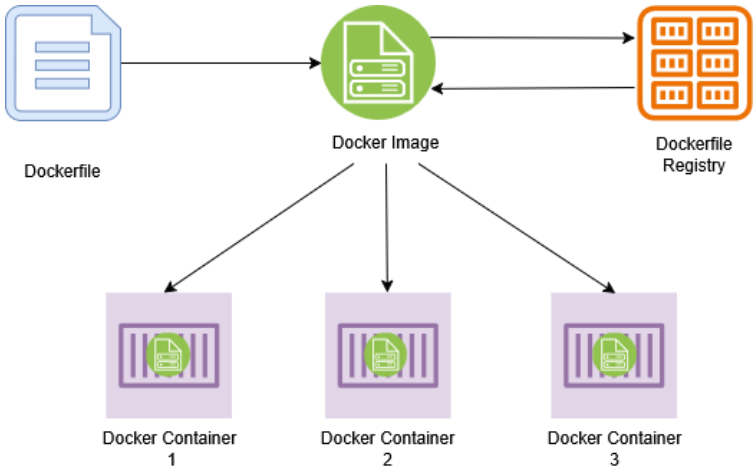


Figure 4 How Dockerfile, Image, Container, and Registry Connect

Docker Registry – A place to store your Docker images (public or private). Docker Hub is the most popular one—kind of like GitHub for Docker images.

Docker CLI / Docker Compose – Command line tools that let you build, run, and manage containers. Docker Compose is great when your app needs multiple containers (like a web server + database).

5.2 NGINX and Reverse Proxy Configuration using Lua scripts

This background work and research is based on official documentation while doing implementation available for Nginx. Traditionally, a web server is a software component that accepts client requests (typically via HTTP or HTTPS), processes them, and returns responses. These responses may consist of static content (such as HTML, JavaScript, and CSS files) or forwarded requests to backend services that generate dynamic content. In modern web development, most web servers—including NGINX—perform hybrid functions, such as proxying, load balancing, and request filtering.

NGINX follows a multi-process architecture that separates management tasks from actual request handling to maximize efficiency and scalability [21].

When NGINX starts, it launches a Master Process and multiple Worker Processes:

- **Master Process**:
  The master process is responsible for reading configuration files, maintaining worker processes, and handling administrative operations such as reloading configurations and restarting worker processes if needed. It does not handle any client traffic directly [21].
- **Worker Processes**:
  Worker processes handle the actual HTTP requests from clients. Each worker is single-threaded and operates asynchronously, using non-blocking I/O to serve thousands of connections concurrently. Each incoming connection is assigned to a worker, and the worker processes requests independently without communication between workers [21].

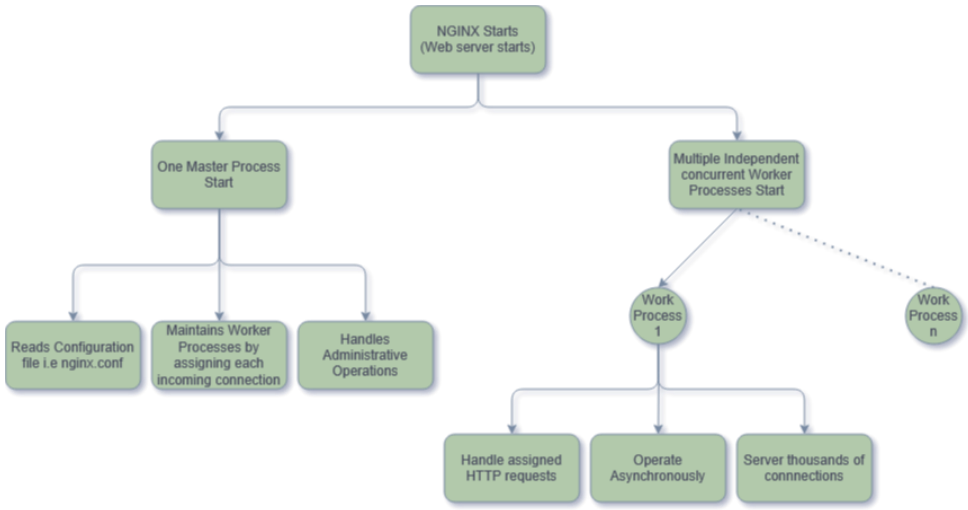


Figure 5 Inside NGINX: Multi-Processor Workflow at Startup

The `nginx.conf` file where all the configurations are done introduces several key concepts that are essential to the system's operation. Each is detailed below:

Reverse Proxy

A reverse proxy is a server that intercepts client requests and forwards them to one or more internal backend servers. Unlike a forward proxy, which acts on behalf of clients, a reverse proxy acts on behalf of servers, often hiding the architecture behind it for security and efficiency [22].

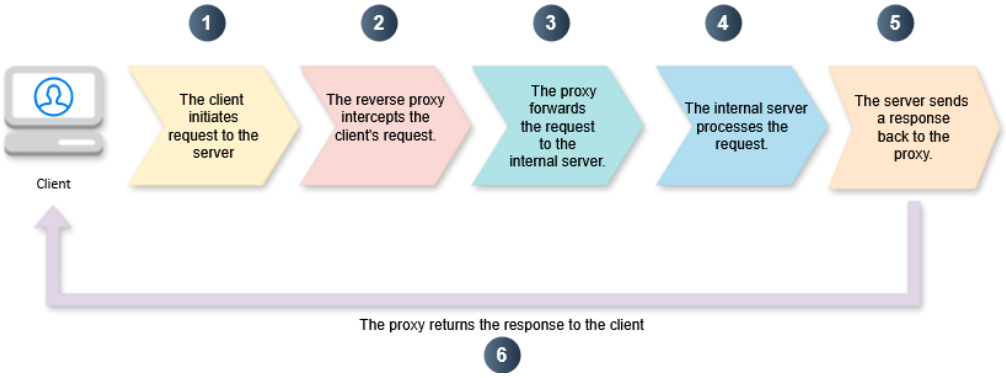


Figure 6 NGINX in Action: Serving as a Reverse Proxy

The configuration and implementation are covered in *Section 11.4*.

Load Balancing

Load balancing is the process of distributing client requests across multiple servers to improve availability and reliability. NGINX supports multiple loads balancing strategies, including round-robin, least-connections, and IP-hash based routing [23].

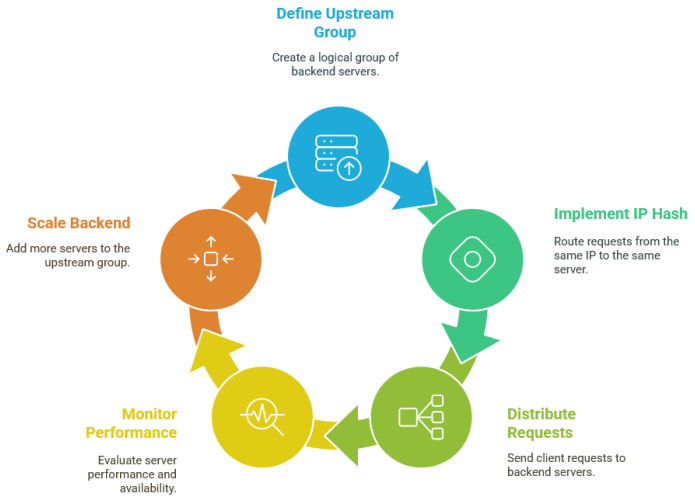


Figure 7 NGINX Load Balancing Cycle: Managing Traffic and Scaling Backends [33]

Currently, load balancing is configured for a single instance per service, but it can be scaled to multiple instances when needed, as outlined in *Section 11.1*.

API Gateway Functionality

An API Gateway centralizes request routing, authentication, and authorization for microservices-based architectures. In the any application, NGINX serves as a lightweight API gateway [24].

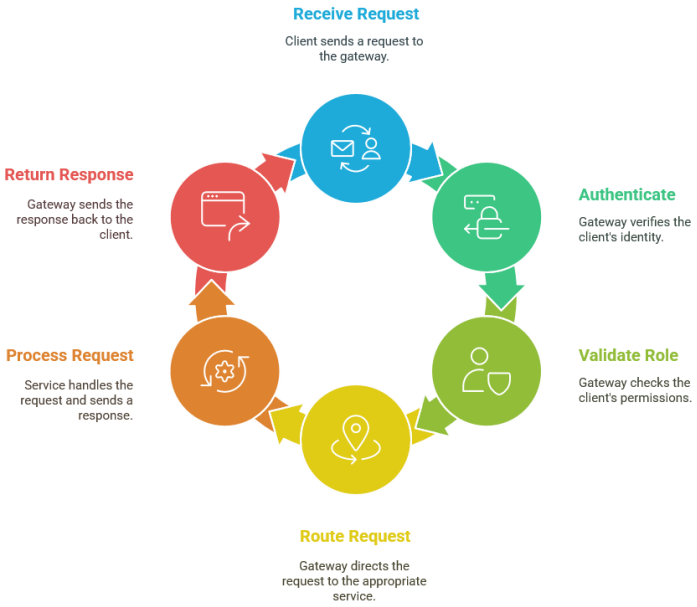


Figure 8 API Gateway Request Processing Flow: From Authentication to Response [33]

The configuration and implementation are covered in *Section 11.5*.

Server Block

A server block defines the virtual server within NGINX, specifying how requests should be handled based on the requested domain, IP address, or port.

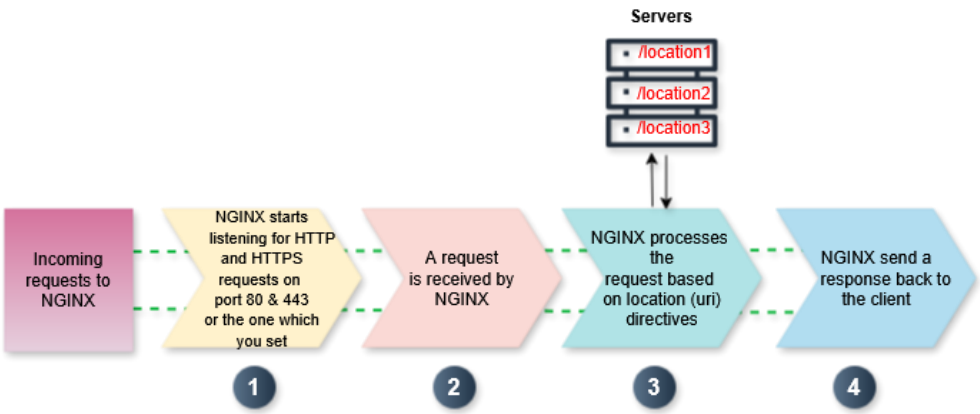


Figure 9 NGINX Workflow: Listening, Processing, and Responding to Requests

The configuration and implementation are covered in *Section 11*.

Location Block

A location block specifies how NGINX should process requests that match URI patterns. To understand the process, follow the flowchart provided below:

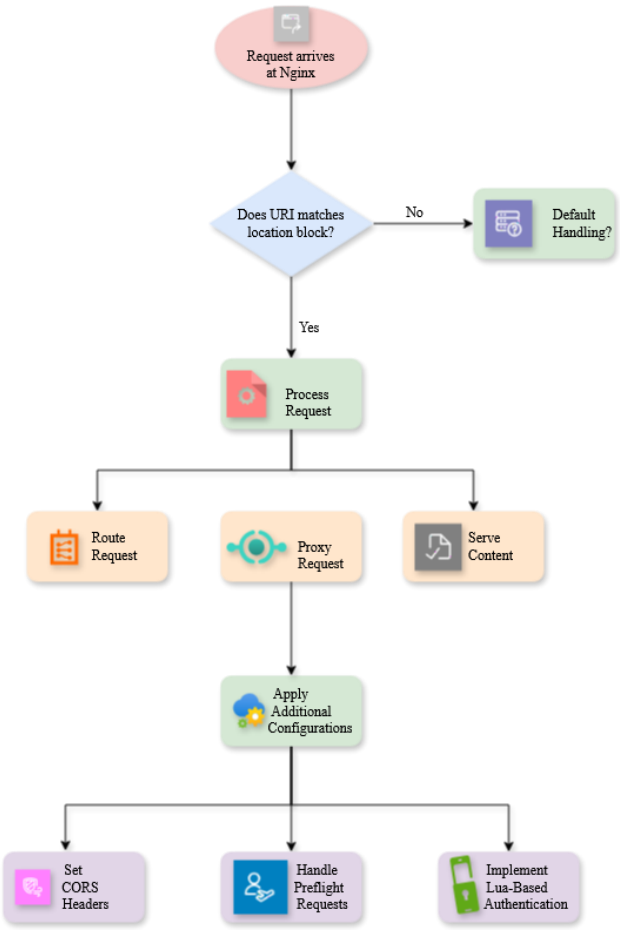


Figure 10 NGINX Location Block Processing Flow: Matching, Routing, and Handling Requests

The configuration and implementation are covered in *Section 11.4*.

Lua Access Control

A key enhancement in the NGINX configuration is the use of Lua scripts for implementing JWT-based authentication and role validation.

The Lua block retrieves the JWT from the user's cookies, verifies the token, checks the user's roles (such as ROLE_ADMIN or ROLE_USER, as per the Generations Connect's requirements), and determines whether to grant access, redirect to an appropriate dashboard, or deny the request.

By handling authentication at the NGINX level, backend services remain stateless and simpler, and unauthorized access is blocked at the edge of the system.

The system differentiates between public and private endpoints based on business rules and security needs:

- **Public Endpoints:**
  Certain URLs, like accepting an invite or registering an account, are open to users without requiring a login. These public paths are recognized in the Lua scripts and skip the usual authentication checks.
- **Private Endpoints:**
  All other parts of the application require users to have a valid JWT token. If a user isn't authenticated or doesn't have the right role, they are either redirected to the homepage or blocked with an error response like 401 Unauthorized or 403 Forbidden.

This way, important public actions remain easy for users, while the secure areas of the application stay properly protected.

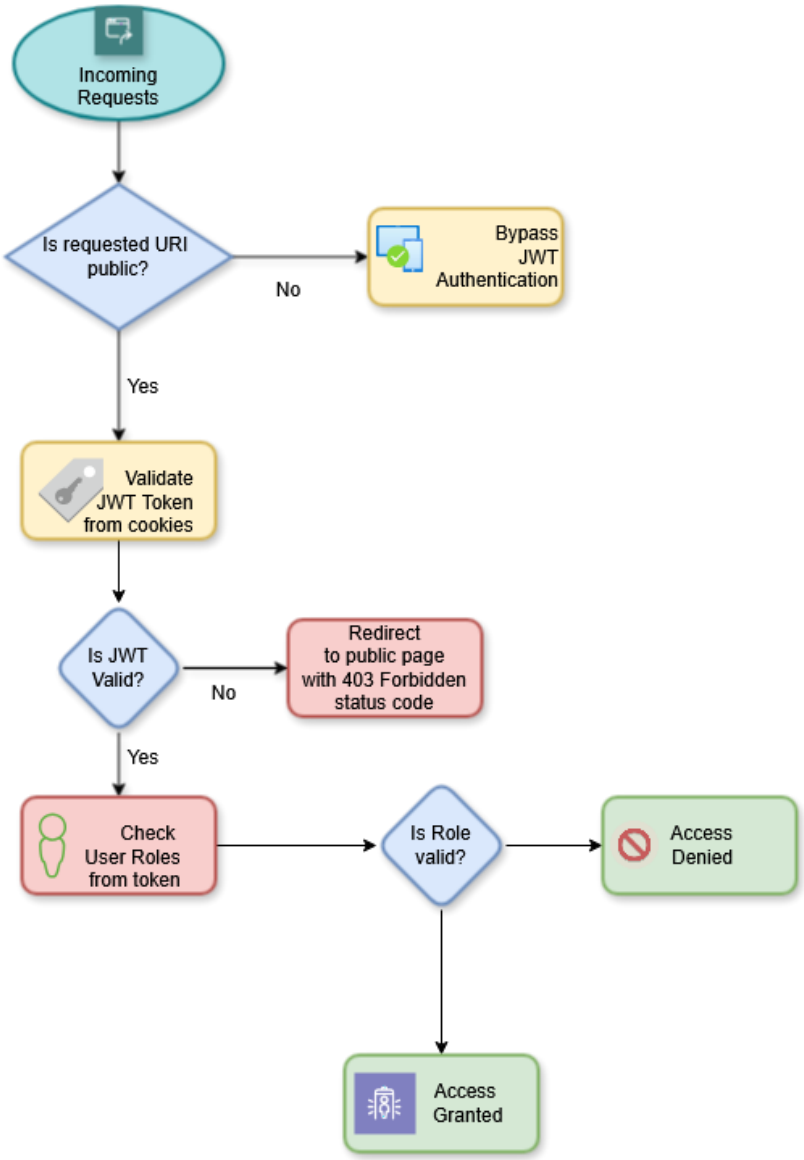


Figure 11 JWT Authentication and Authorization Flow for Incoming Requests

The configuration and implementation are covered in *Section 11.5*.

SSL Configuration to upgrade HTTP to HTTPS protocol

To support secure communication across the platform, HTTPS was enabled at the NGINX gateway using SSL/TLS. This enhancement ensures that all data exchanged between clients and backend services is encrypted, preventing unauthorized interception of sensitive information such as authentication tokens and personal user data.

The system was initially served over HTTP but was upgraded to HTTPS by introducing a redirect mechanism and secure server setup. Self-signed certificates and cryptographic parameters were generated to support encrypted communication during local development. The NGINX reverse proxy was updated to enforce HTTPS and leverage these certificates, aligning with modern browser security standards.

This approach not only improves data confidentiality and integrity but also ensures the platform adheres to industry best practices for handling secure sessions and cookie-based authentication. Technical implementation details, including certificate generation and server configuration, are covered in the implementation section of this report.

The configuration and implementation are covered in *Section 11.2*.

5.3 Secure Microservice Communication using Spring security framework

This background work and research is based on official documentation while doing implementation available for Spring Security.

In addition to the NGINX configuration using Lua scripts described in *Section 11.5* under Lua Access Control, securing backend APIs independently is essential.

In microservice architectures such as Generation Connect, backend services are recommended to be protected using security frameworks like Spring Security to ensure that unauthorized access is prevented, even if external gateway protection is compromised.

Based on the research and understanding of best practices, several key aspects are involved in securing microservice communication effectively.

1. JWT-Based Authentication

One of the foundational elements in securing backend communication is the use of JWT-based authentication [30].
Typically, a custom authentication filter, such as `JwtAuthFilter.java`, extends `OncePerRequestFilter.java` to intercept each incoming HTTP and HTTPS request.

The process generally involves:

- Extracting the `accessToken` from the client's cookies (has been used in Generation Connect).

Parsing the token using a `JwtService.java` to retrieve the embedded username.

Loading user details through a service that implements Spring Security's `UserDetailsService.java`.

- Validating the token's integrity and authenticity.

  Upon successful validation, setting a `UsernamePasswordAuthenticationToken.java` within the Spring Security `SecurityContext.java` to represent the authenticated user during the request lifecycle.

By applying token validation early in the filter chain, the system ensures that only authenticated users can access protected resources [30].

2. Role-Based Authorization

Following authentication, systems typically enforce role-based authorization to control access based on user privileges. Spring Security configurations usually define a clear separation between public and protected endpoints [30].

- Public endpoints (as per the requirements of Generation Connect Application), such as
  - `/auth/login` – To access sign-in page or hit Backend API for Users and Admin
  - `/auth/signup` – To access sign-in page or hit Backend API
  - `/family-tree/manage-users/accept` – To accept the invitation sent via Owner

Above endpoints are accessible without prior authentication to facilitate user onboarding processes.

- Protected endpoints (as per the requirements of Generation Connect Application), such as
  - `/person/**` - To create and update the person in the family tree
  - `/family-tree/**` - To create and update the Public or Private family tree
  - `/admin/**`, - To access the metrics and audits of the applications

Above endpoint require a valid JWT token for access and enforce authorization checks based on user roles (e.g., `ROLE_USER` or `ROLE_ADMIN`).

Unauthorized access attempts are generally handled by returning appropriate HTTPS status codes like 401 Unauthorized or 403 Forbidden, maintaining the security integrity of the application.

For, complete implementation of this research could be found in *Section 8.1*.

3. Cross-Origin Resource Sharing (CORS) Management

In addition to gateway-level enforcement, managing CORS policies within the backend enhances security and interoperability.
Research suggests that CORS configurations typically define:

- Permitted origins,
- Allowed HTTP and HTTPS methods,
- Accepted headers.

By internally enforcing CORS through Spring Security and `WebMvcConfigurer`, applications ensure consistent cross-origin access control, thus minimizing risks associated with unauthorized cross-domain requests as below:

```
package com.generation.connect.config;

@Configuration public class WebConfig {
    @Bean     public WebMvcConfigurer corsConfigurer() {
        return new WebMvcConfigurer() {
            @Override             public void addCorsMappings(CorsRegistry registry) {
                registry.addMapping("/**")
                        .allowedOrigins("http://localhost", "https://localhost")
                        .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
                        .allowedHeaders("*")
                        .allowCredentials(true);
            }
        };
}}
```

4. Stateless Session Management

Modern backend services, particularly in microservice architectures, adopt stateless session management principles. Statelessness ensures scalability and simplifies horizontal scaling.

This is often achieved by configuring the session policy as stateless, as in:

```
.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
```

In such a configuration, the server does not maintain any session information between requests. Each request must independently present valid authentication credentials (typically in the form of a JWT), aligning the system with RESTful principles and enhancing security against session hijacking [30].

5. Password Security and Encryption

An important aspect of backend security involves protecting user credentials.
Research highlights the use of strong hashing algorithms such as `BCrypt` for password storage. Passwords are securely hashed using `BCryptPasswordEncoder` before being saved to the database.

This approach ensures that:

- Passwords remain protected even if database breaches occur.
- Brute-force decryption attacks are significantly mitigated.

By adhering to these practices, the system aligns with modern security standards for credential protection [30]. For detailed implementation of configuring Security on Generations Connect application check *Section 8.1.*

5.4 Neo4j and Graph Data Management

This background work and research is based on creating Proof of concept via brute force method on Neo4j application and its advantages over relational database.

Initially, the plan for the Generation Connect application was to build the family tree using custom graph or tree data structures implemented manually within a traditional relational database like PostgreSQL. This approach would have involved setting up tables with primary keys, composite keys, foreign keys, and carefully structured groupings of data to model the relationships between individuals.

However, during the early design phase, it became evident that manually identifying and managing complex node relationships would introduce significant rigidity and complexity into the system. Handling deep, flexible, and dynamic relationships within a traditional relational schema proved to be highly cumbersome and difficult to maintain.

As a result, further research was conducted into alternative solutions, leading to the discovery of Neo4j, a purpose-built graph database designed to manage highly connected data more intelligently. Neo4j provides a more natural, scalable, and flexible way to model and manipulate relationships between nodes, making it an ideal choice for building and managing a dynamic family tree.

In Neo4j, nodes (representing entities such as people) and edges (representing relationships) are first-class citizens.

Unlike traditional SQL databases that rely on sequential tables and joins, Neo4j leverages the Cypher Query Language (CQL), which is specifically designed for navigating and querying graph-based data models [32].

There are notable differences between Neo4j repositories and traditional JPA repositories:

- A Neo4j repository is like a JPA repository in its arrangement of methods and data access patterns.
- However, Neo4j operates using Cypher Query Language (CQL), whereas JPA relies on Structured Query Language (SQL).
- In CQL, relationships are explicitly modelled through edges connecting two actors (nodes), supporting both unidirectional and bidirectional navigation.

For example, in the Generation Connect family tree:

- If Person A is the son of Person B, then B is also the father of Person A — a bidirectional relationship that can be traversed from either node.
- Similarly, if Person B and Person C are spouses, the relationship is modelled bidirectionally, so fetching either B or C would reveal the connection to the other.

Neo4j also supports multiple nodes being connected to a single node, allowing complex relationships (such as multiple children linked to one parent) to be handled naturally and efficiently.

To optimize the architecture, a hybrid approach is adopted where:

- Family tree metadata (such as creation details, tree ownership, and user permissions) is managed in PostgreSQL.

- Detailed family tree structure and relationships between individuals are stored and navigated using Neo4j.

This combination of PostgreSQL and Neo4j provides an impressive balance:

- PostgreSQL efficiently manages structured relational data, minimizing unnecessary load on the graph database.
- Neo4j handles complex, dynamic relationship data, enabling rapid querying, intelligent relationship navigation, and seamless family tree expansion.

By separating concerns between the two databases, the system architecture avoids overloading either system, ensuring performance optimization, scalability, and maintainability.

With the backend data management architecture established through the combination of PostgreSQL and Neo4j, attention must also be given to the effective presentation of this data on the client side.

Visualizing complex relationships, such as those within a family tree, requires robust and dynamic frontend solutions. The next section discusses the selection and integration of appropriate React charting libraries to support interactive, user-friendly data visualization within the Generation Connect application.

For more detail with implementation and other attributes check *Section 7.3*, *Section 8.2* and *Appendix 53*.

5.5 React Charting Libraries

This background work and research is based on creating Proof of concept via step-by-step implementation in react for frontend application while keeping backend API's in mind.

As part of the Generation Connect application development, a significant research effort was dedicated to finding an appropriate frontend solution for visualizing complex family tree relationships.

Given the dynamic and hierarchical nature of genealogical data, a robust and flexible charting library was essential to accurately represent person nodes and their interconnected relationships.

The research began with the exploration of the D3.js library integrated into React.

Since React was relatively new to the project at this stage, the initial focus was placed on implementing backend functionality in small, manageable components before attempting a fully integrated visualization.

The backend proof of concept (POC) involved:

- Creating a basic person node in Neo4j.
- Creating additional person nodes.
- Using Cypher Query Language (CQL) to establish relationships such as "mother of", "father of", "son of", and "daughter of" between these nodes.

The initial objective was to create a simple three-generation family graph on the backend, validating the underlying data model.

While basic graph construction worked as expected, it required a two-step process: first creating nodes, and then separately establishing relationships - rather than achieving this dynamically in a single operation.

Despite being incremental, this approach provided critical insight into how relationship management needed to be structured.

Following backend validation, frontend development commenced using React D3 libraries.

The first attempts involved:

- Hitting backend APIs to create person nodes dynamically from the React application.
- Programmatically creating new nodes in the frontend visualization.
- Writing custom logic to create edges between nodes, attaching properties such as gender and relationship type.

Although this was a step-by-step manual process, it served the purpose of demonstrating that D3 with React could support the dynamic and hierarchical structure required for a live family tree system.

To further evaluate options, additional research was conducted across public repositories on GitHub to identify existing open-source frontend libraries capable of handling complex graph visualizations.

Several libraries were explored, with a focus on suitability for Neo4j-based data. Among these, one is React D3 based static page project — designed for rendering family charts — appeared well-suited

for integration with the implemented backend API's as given here in *20 (consider this as the citation in code as well under /pm455/frontend/src/component/family-chart in Gitlab)*

However, this identified library was originally designed as a static frontend module, without any backend integration or persistence layer.

It lost all its data upon a browser refresh, as it was not connected to a database or server.

Recognizing its potential, modifications were made to bridge the gap:

- Backend services were enhanced to expose APIs compatible with the module's structure.
- Callback functions were injected into the frontend module to trigger backend API calls developed specifically for the Generation Connect application.

By adapting both backend and frontend components, a working prototype was created wherein real-time family tree updates could be persisted and retrieved dynamically — fulfilling the intended goal of linking Neo4j's dynamic graph data with a reactive and visually intuitive presentation layer.

This research with Proof of Concept validated the feasibility of using a combination of React D3 libraries and custom backend integration to build a scalable and dynamic family tree visualization system for the Generation Connect platform.

Checkout the Gitlab for its integration with the implementation done in *Section 9.1*.

5.6 Research API Integration

In the early stages of designing the Generation Connect application, research on existing genealogy and archival platforms, as discussed in *Section 3* played an important role in shaping the Research API integration approach.

The initial plan involved analysing several well-known genealogy platforms to understand how they allow users to access historical records.

However, it was soon discovered that many of these platforms either provided limited API access or required paid licenses and were often built as desktop-based applications rather than web services.

Because of these limitations, direct integration with such APIs was considered neither feasible nor appropriate for an open web-based platform like Generation Connect.
Therefore, the focus shifted towards identifying open-source or government-maintained directories that allow lawful, free access to archival data.

During this phase of research, it became clear that while many government archives offer open access to their records, they often restrict the availability of sensitive personal information such as full birthdates, death dates, or private family details — primarily for privacy and legal reasons. Nevertheless, even with these restrictions, integrating external research tools into the Generation Connect platform was seen as an interesting opportunity to enhance the user experience, without violating any data usage policies.

The research led to the selection of the following external sources:

1. National Archive – UK – *(currently integrated for prototype)*

Integrated into the prototype to allow basic archival record searching using their publicly available sandbox API.

- **Website:** Discovery
- **Developer Resources:** API Documentation
- **API Access:** Sandbox API

2. National Archive CatLog – USA – *(Planned for future integration)*

Identified as a future integration candidate, with scalability already accounted for in the backend and frontend architecture.

- **Website:** National Archives Research Tools
- **Application:** National Archives Catalog
- **API Access:** API Documentation

At this stage, the focus was on developing a working prototype for the Discovery platform in the UK. The backend was structured to accept requests and filter responses in a JSON format, ensuring that non-null fields were passed between frontend and backend systems consistently.
This structure made it easier to work with external archival records while keeping the application design clean and extendable.

To connect the research integration with the family tree, a tagging system was conceptualized. When a user finds a relevant record during an external search, they have the option to tag that record to a specific family tree.
This tagging process involves selecting the relevant information from the search results and submitting it to the backend, where it is saved and linked to the appropriate family tree.

Because the platform defines three distinct user roles — Owner, Contributor, and Viewer — access to tagging is carefully controlled:

- All users (Owner, Contributor, Viewer) can perform searches and view external records.
- Only Owners and Contributors are permitted to tag records and save them to the family tree.
- Viewers are restricted to read-only access and cannot modify or tag external research data.

This granular access control ensures that sensitive archival data is only associated with family trees by trusted users, preserving the privacy, authenticity, and historical integrity of the information.

Through this research and prototyping effort, it was validated that integrating external research data into Generation Connect is not only technically feasible but also enhances the platform's usability, encouraging deeper exploration and richer family histories.

5.7 Open AI API Integration

In the last two weeks of the project, a new feature was added that allows users to ask questions about their family tree and get helpful responses. This was not part of the original plan, but the idea came up during a team discussion. To support this, different online services were tested while implementation using Brute-force technique to find the one that works best with the project.

1. **RapidAPI Platform**

   Link explored: OpenAI via RapidAPI, Pipedream Integrations and Medium: Guide on Using RapidAPI

   - Recommended by a peer for quick API testing and integration.
   - Easy to set up via a shared platform.
   - However, the feature set was limited for project needs.
   - Responses were sometimes incomplete or failed entirely.

   Decision: *Not suitable for the final implementation.*

2. **Gemini by Google**

   Explored Using: Gemini API QuickStart and online Java Examples

   - This service was tested using online code examples and Java tools.
   - It had good features and looked promising.

   Decision: *Not used in the project***.**

3. **OpenAI Platform**

   Documentation: OpenAI API Reference

   Smooth integration with the Spring Boot backend using available Java dependencies.

   - Allowed backend services to relay user queries and fetch meaningful, context-aware responses related to the family tree structure.
   - Easy to configure with clear documentation.
   - Reliable in delivering consistent and accurate results.
   - A paid trial was used, with $5 spent for testing.

   Decision: *Selected for final implementation due to ease of integration, reliability, and availability of Spring-compatible libraries.*

06

Proposed Solution and Design

This chapter outlines the proposed system architecture, including selected technologies, integration strategies, and key dependencies.

It also presents a high-level design and evaluates how the solution addresses the identified challenges effectively.

6. Proposed Solution and Design

6.1 Core Technologies and Frameworks Selections

To build the Generation Connect platform, the objective was to utilize modern and emerging technologies that not only meet current requirements but also provide a foundation for future innovation.

The platform aims to offer users a simple, lightweight, secure, and scalable solution for connecting with their families across generations, while embracing open-source principles wherever possible.

After careful analysis and research, the following core technologies and frameworks were selected to achieve various objectives:

Proposed Technical Stack -

- **Backend:** Java with Spring Boot, following MVC architecture principles.
- **Database:** PostgreSQL for structured data and Neo4j for managing family tree relationships.
- **Frontend:** React with integrated D3.js for interactive tree visualization.
- **Authentication & Authorization:** Handled using Spring Security with role-based access (Owner, Contributor, Viewer).
- **Media Storage:** Server-side storage using Docker volumes for persistent media handling.
- **Load Balancing:** NGINX for efficient traffic routing and high availability.
- **Deployment:** Docker and Docker Compose for containerized multi-service setup.
- **Testing:** API testing with Postman, unit testing with Mockito, and integration testing with Postman Collections.
- **Documentation:** API documentation generated using Swagger (OpenAPI).
- **Version Control & CI/CD:** GitLab for source code management and project automation.

Through this carefully selected technology stack, Generation Connect is designed to achieve a balance between robustness, performance, security, and future extensibility.

The choices made are based on industry best practices, supporting the platform's goal of providing a modern, user-friendly, and reliable genealogy application.

6.2 System Dependencies

The Generation Connect platform relies on several system dependencies and components that work together to ensure smooth, secure, and scalable functionality.

This section outlines the core technologies involved and describes the integration strategy adopted to connect the different layers of the application. Although containerization technologies like Docker and orchestration tools like Docker Compose simplify deployment and environment consistency, it is essential to identify and explain the individual dependencies that contribute to the overall system architecture.

System Dependencies

Based on Core Technologies and Framework selection in *Section 6.1*, the key system dependencies for Generation Connect include:

- Java Development Kit (JDK): Version 17 or higher
- Spring Boot Framework: Version 3.4.1 or higher
- Tomcat: Version selected from Docker registry for Tomcat as below:

```
FROM tomcat:10.0-jdk17-openjdk-slim
COPY ./gc-api.jar /app/gc-api.jar
RUN chmod -R 777 /app/gc-api.jar
WORKDIR /java/jdk/bin
CMD echo Starting Docker Container
EXPOSE 8080/tcp
VOLUME /data
ENTRYPOINT ["java","-jar","/app/gc-api.jar"]
```

- PostgreSQL Database: Version selected from Docker registry for PostgreSQL

```
FROM postgres:latest
ENV POSTGRES_PASSWORD gc-password
ENV POSTGRES_DB gc-db
ENV POSTGRES_USER gc-user
EXPOSE 5432
```

- Neo4j Graph Database: Version selected from Docker registry for Neo4j as below:

```
FROM neo4j:latest
ENV NEO4J_AUTH=neo4j/gc-family-password
ENV NEO4J_dbms.default_database=gc-family-db
EXPOSE 7687
```

- React and React Libraries (D3.js): Version `^18.3.1` and `^6` in `package.json`
- NodeJS: Version selected from Docker registry for Node.js as below:

```
FROM node:20.16.0 AS build
WORKDIR /app
COPY package*.json .
COPY . .
RUN yarn install
ENV NODE_OPTIONS=--openssl-legacy-provider
EXPOSE 3000
CMD ["/bin/sh", "-c", "yarn start"]
```

- NGINX Server as described in length here in *Section 11*

```
FROM openresty/openresty:alpine-fat
RUN /usr/local/openresty/luajit/bin/luarocks install lua-resty-jwt
EXPOSE 80
CMD ["openresty", "-g", "daemon off;"]
```

- Docker and Docker Compose as described in length here in *Section 12.2.2*.
- Postman for REST API testing validation during the development and debugging.
- Mockito Framework for Unit testing

6.3 High-Level Architecture Diagram

The below diagram represents the high-level architecture of the Generation Connect website based on *Section 5*. It is designed around a modern, scalable, and containerized architecture using Docker Compose, separating the frontend and backend services while maintaining flexibility, efficiency, and future scalability.

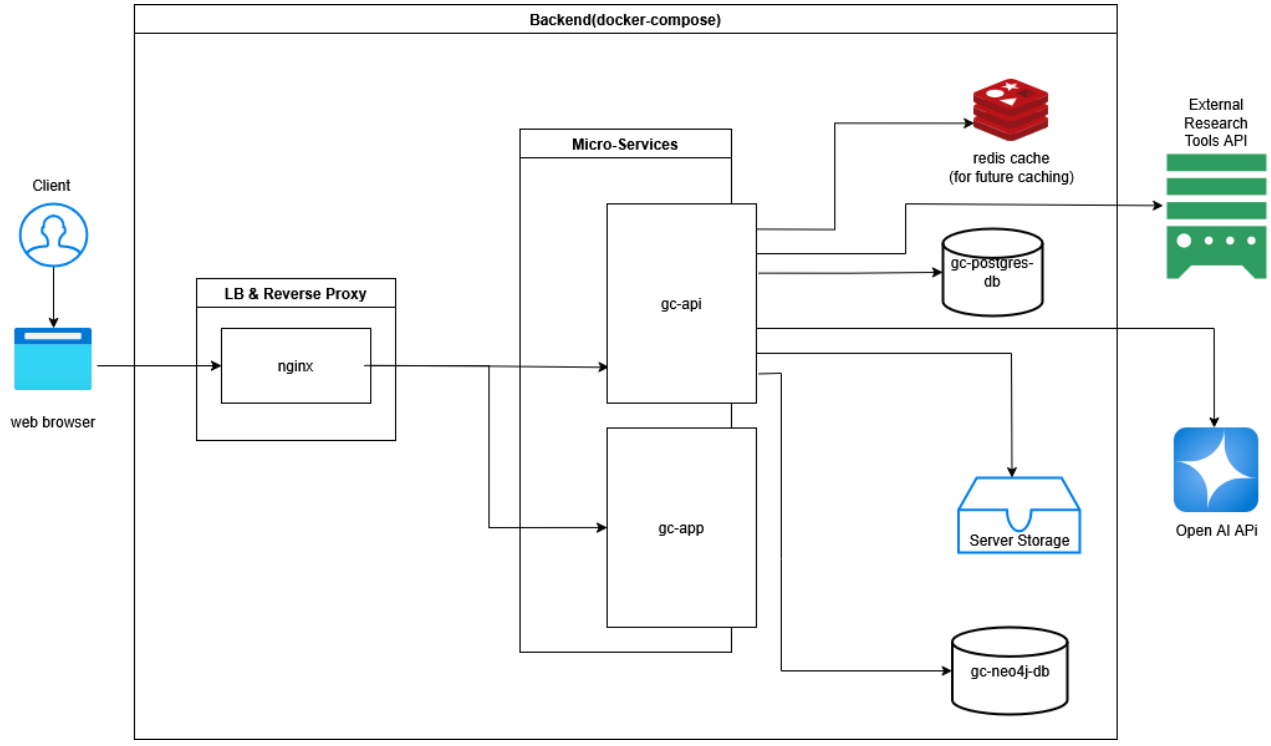


Figure 12 Generations Connect High-Level Architecture Diagram

Architecture diagram's Components Overview

1. **Client (Web Browser)**
   - End-users interact with Generation Connect via a web browser.
   - All requests from the client are routed through a centralized entry point before reaching the appropriate service.

2. **Load Balancer & Reverse Proxy (Nginx)**
   - Incoming requests first hit the Nginx service.
   - Nginx acts as:
     - Load Balancer: Distributes traffic efficiently between services.
     - Reverse Proxy: Directs requests either to the frontend `(gc-app)` or backend `(gc-api)` based on routing rules.

3. **Microservices**
   - **`gc-app` (Frontend Application):**
     - This is the user-facing frontend application (likely built with a framework like React).
     - It provides the interface for browsing family trees, managing profiles, posting updates, and more.
   - **`gc-api` (Backend API Service):** This is the backend server that handles business logic, authentication, data processing, and integration with databases and external APIs.

4. **Databases and Server Storage**
   - **`gc-postgres-db` (PostgreSQL):** A relational database for storing structured data such as user information, posts, comments, and application settings.
   - **`gc-neo4j-db` (Neo4j):** A graph database ideal for storing and querying complex relationship-based data like family trees and genealogical connections.
   - **Server Storage:** Dedicated file storage to manage uploaded media files (images, videos, documents).

5. **Caching Layer:** A Redis service is included in the architecture to show the system could be integrated for future caching needs (e.g., session management, frequently accessed data) to enhance performance and scalability. However, it has not been implemented yet.

6. **External Research Tools API**

   The `gc-api` service communicates with external research tools and APIs to fetch enriched genealogical or historical data, enhancing user research capabilities.

7. **AI Integration - AI Chat Module (via OpenAI)**
   Integrated within the `gc-api` service, this module enables intelligent interactions via answering genealogy-tree related questions.

   Uses the OpenAI GPT model (e.g., `gpt-3.5-turbo`) via API calls to generate human-like responses. Natural language queries for historical records.

6.4 Evaluation of the Proposed Design

The architecture designed — and successfully implemented — for the Generation Connect platform shows a thoughtful, modern, and practical approach. It's built to manage genealogical data, support research integrations, and grow as user demand increases.

Right now, the system is deployed locally using Docker Compose, but the structure and setup clearly point toward future scalability. When needed, the platform can easily move to a live environment with more advanced infrastructure in place.

Here's a breakdown of how the design holds up across key areas:

1. Scalability with future scope could be configured very easily with the system

The way the system is structured makes scaling straightforward:

- **Separation of Frontend and Backend:**
  The frontend `(gc-app)` and backend `(gc-api)` are kept separate, meaning they can be scaled independently based on usage. For example, if there's a spike in users, just the frontend can be scaled up (after doing configurational changes which is not done yet) without touching backend services.
- **Load Balancing with Nginx:**
  Using Nginx ensures incoming traffic is evenly distributed, helping the system stay responsive even when demand grows.
- **Docker Setup:**
  Running everything through Docker Compose on a local machine keeps it simple for now, but it's been designed in a way that it can be easily scaled to cloud environments like AWS or Kubernetes when the time comes.

Conclusion: The system is ready to grow when needed after doing Scalability configuration, without requiring a full re-architecture.

2. Maintainability and Flexibility

Maintaining and updating the platform will be easy going forward:

- **Microservice Structure:**
  Because services are isolated, updates to either the frontend or backend can happen without risking the stability of the whole system.
- **Smart Database Choices:**
  PostgreSQL handles structured data, and Neo4j handles family tree nodes and their relationships — keeping things simple, clean, and avoiding unnecessary complexity.
- **Future-Friendly Integration:**
  Adding new features, like external research APIs, won't cause much of complexity. The system is designed to accept those integrations without major changes.

Conclusion: It's built to last and easy to work with as the platform evolves.

3. Security

Security is baked into multiple layers of the system:

- **Authentication at the Edge:**
  Nginx uses Lua scripts to check JWT tokens before traffic even reaches the backend.
- **Spring Security on the Backend:**
  Backend services verify authentication and user roles separately, adding an extra layer of protection.
- **Stateless Sessions:**
  Using JWT tokens means there's no need for traditional sessions — making the system more secure and scalable.
- **Role-Based Access:**
  Different actions, like tagging research records, are protected by fine-tuned user roles enumerations (Owner, Contributor, Viewer).

Conclusion: Security is strong and smartly layered, matching modern best practices.

4. Performance and Optimization for future could be configured very easily

Even though the system is still local, it's already tuned for good performance:

- **Efficient Data Handling:**
  Neo4j handles complex family relationships far better than a traditional database would, making queries faster and cleaner.
- **Lightweight Frontend Delivery:**
  By serving the frontend separately through Nginx, users get faster load times and a smoother experience.

Conclusion: The system runs efficiently now and can be optimized even further when user numbers grow.

5. Extensibility and Future Proofing

The platform isn't built for now — it's built for what's coming and to showcase the prototype of idea:

- **External APIs:**
  More research APIs (like from the U.S. National Archives) can be added easily without breaking anything via following up the pattern used to integrate Research API's of UK National Archives.
- **Interactive Family Trees:**
  React combined with D3.js creates a dynamic family tree view that will scale well as user data grows.
- **Conclusion:** Future growth has already been considered into the design with future evolution in design.

07

Entity-Relationship Model

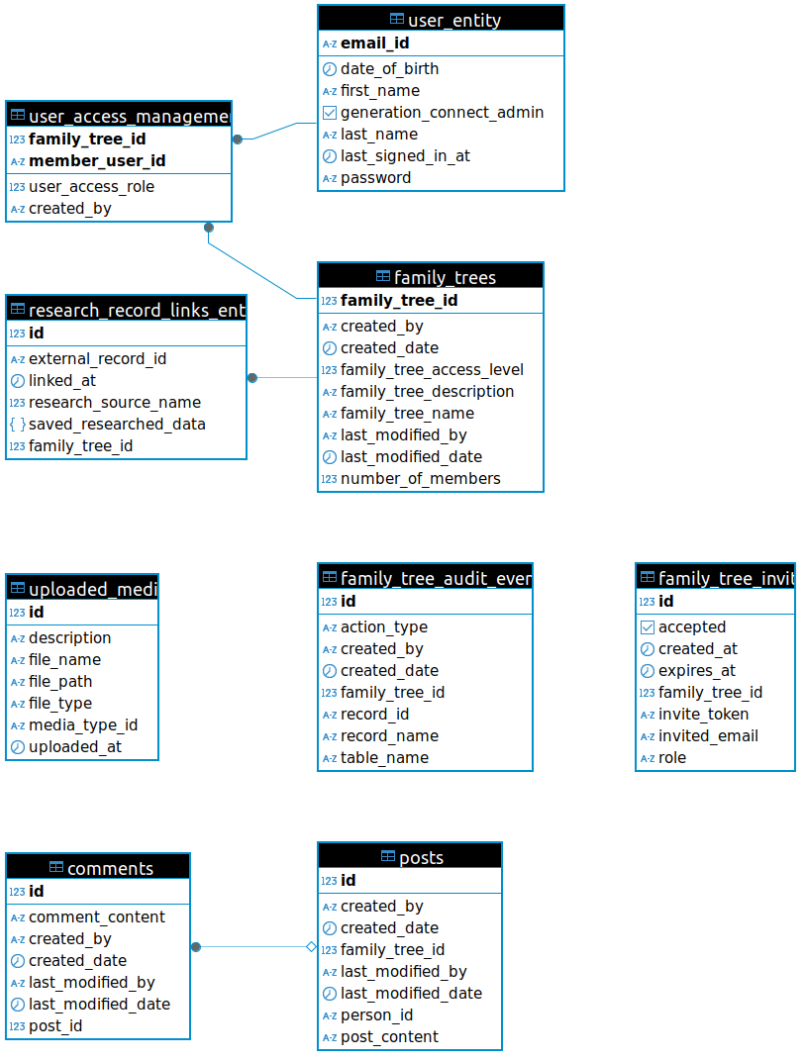
Outlines the core entities and their relationships, forming the foundation of the database structure. This model supports robust data integrity and aligns with the application's functional requirements.

7. Entity-Relationship Model

The database schema has been designed to support the core functionality of a genealogy platform, which includes managing family trees, user collaboration, posts, comments, media uploads, external research references, and access control. The system is inherently relational, with strong dependencies between users, trees, and associated content. As such, relational database design principles were followed to ensure data consistency, enforce referential integrity, and optimize query performance.

7.1 Entity-Relationship Diagram

The relational structure described above is visually represented in the following entity-relationship diagram. This diagram illustrates the primary keys, foreign keys, and table relationships that underpin the system's data model.



8

Figure 13 Entity-Relationship diagram showing core tables and their relationships in the genealogy database.

7.2 Entity Descriptions and Relationships

1. `user_entity` table

The `user_entity` table stores all user registration and profile information. Each user is uniquely identified using the `email_id` field, which acts as the Primary Key due to its guaranteed uniqueness and role in authentication. The table also includes fields such as `generation_connect_admin`, a `boolean` flag that identifies admin users, and `last_signed_in_at`, a timestamp used to track user activity and engagement patterns.

2. `family_trees` table

The family_trees table represents each individual family tree created by users. It uses `family_tree_id` as the Primary Key, defined as an auto-incrementing BIGINT, allowing the system to support a large number of entries. A notable field in this table is `family_tree_access_level`, which is regulated using a CHECK Constraint to restrict its value to valid options—typically 0 for public and 1 for private trees. This constraint ensures data consistency and access control across the system.

3. `user_access_management` table

To manage user permissions within family trees, the `user_access_management` table was implemented. This table defines which users have access to which trees and their respective roles (e.g., owner, contributor, or viewer). It uses a Composite Primary Key consisting of `family_tree_id` and `member_user_id`, ensuring a user cannot have more than one role in the same tree. The table also includes two Foreign Keys:

- `family_tree_id` referencing the `family_trees` table, and
- `member_user_id` referencing the `user_entity` table.

These keys maintain data integrity and ensure only valid relationships are stored.

4. `family_tree_invite` table

The `family_tree_invite` table is used to handle invitations sent to users to join a family tree. Each record includes an auto-generated id as the Primary Key. To maintain invitation integrity, the `invite_token` field is enforced with a UNIQUE Constraint, ensuring that each token is distinct and valid for one-time use. Additionally, the role field is controlled using a CHECK Constraint that only allows permitted values: OWNER, CONTRIBUTOR, or VIEWER.

5. `posts` table

The posts table stores user-generated posts linked to family trees and, optionally, to specific individuals. Each post is uniquely identified by an id, which serves as the Primary Key. A Foreign Key on `family_tree_id` ensures that each post is correctly associated with a valid family tree. To

enhance query performance, Indexes are created on both the `family_tree_id` and `person_id` fields, supporting efficient filtering and retrieval of posts by tree or person.

6. `comments` table

The comments table captures user comments made on posts. Each comment is identified by a unique id, used as the Primary Key, and is linked to its parent post using the `post_id` field. This field is a Foreign Key referencing the posts table, maintaining referential integrity and allowing structured comment threads.

7. `uploaded_media` table

The `uploaded_media` table is designed to store metadata about media files uploaded by users. This includes files such as images, videos, and documents. The id field serves as the Primary Key, and an Index is defined on the `media_type_id` field to allow quick filtering based on media category. This supports efficient access to media based on type or usage context.

8. `research_record_links_entity` table

The `research_record_links_entity` table manages external research records—such as archival documents—that users link to their family trees. Each record is identified using an auto-generated id as the Primary Key. A Foreign Key on `family_tree_id` ensures that the research record is properly linked to an existing tree. A UNIQUE Constraint is placed on the combination of `external_record_id` and `family_tree_id`, which prevents the same external record from being tagged multiple times in the same tree. The `saved_researched_data` field uses the `jsonb` data type to flexibly store structured or semi-structured data from different research sources.

9. `family_tree_audit_events` table

Lastly, the `family_tree_audit_events` table was created to maintain a log of key operations—such as additions, updates, or deletions—performed on family tree data. Each entry includes a unique id as the Primary Key, and the `action_type` field is controlled using a CHECK Constraint that only accepts valid operations: ADD, UPDATE, or DELETE. This logging mechanism helps maintain a clear audit trail and supports traceability and accountability within the system.

7.3 Graph Database Design: Neo4j Person Node

In addition to the relational PostgreSQL schema, a graph-based representation is implemented using Neo4j to model family relationships. Graph databases are well-suited for representing genealogical data due to their ability to efficiently store and query complex, hierarchical relationships such as parent-child, spouses, and extended family.

The central node in this structure is the Person entity, which represents an individual in the family tree and serves as the foundation for building relational paths between people in the graph.

7.3.1 Node Definition and Attributes

The Person class is defined as a Neo4j `@Node` annotated entity. It contains a variety of attributes that describe an individual, both structurally for the family tree and contextually for richer historical and biographical details.

Key Attributes:

- `personId (String)`: Unique identifier for each node, generated as a UUID string. Defined using @Id and @GeneratedValue.
- `familyTreeId (Long)`: Links this person node to a specific family tree in the relational schema. Enables cross-database consistency.
- `firstName, lastName (String)`: Standard name fields, out of them, `firstName` is mandatory.
- `birthdate (LocalDate)`: Date of birth.
- `gender (Enum: GenderEnum)`: Gender classification using a type-safe enumeration.
- `firstNode (Boolean)`: Marks the root or first ancestor of the tree, useful for traversal starting points.
- `userId (String)`: Maps the person node to the user who created or owns this person, if applicable.

Descriptive Biographical Fields:

Includes a wide range of optional attributes that enrich the person node with historical context, such as:

- `placeOfBirth`, `residence`, `occupation`, `employer`, `positionHeld`
- `languagesSpokenWritten`, `significantEvent`, `awardReceived`, etc.

These fields can support advanced querying and filtering for research use cases and enable detailed person profiles as part of future enhancements.

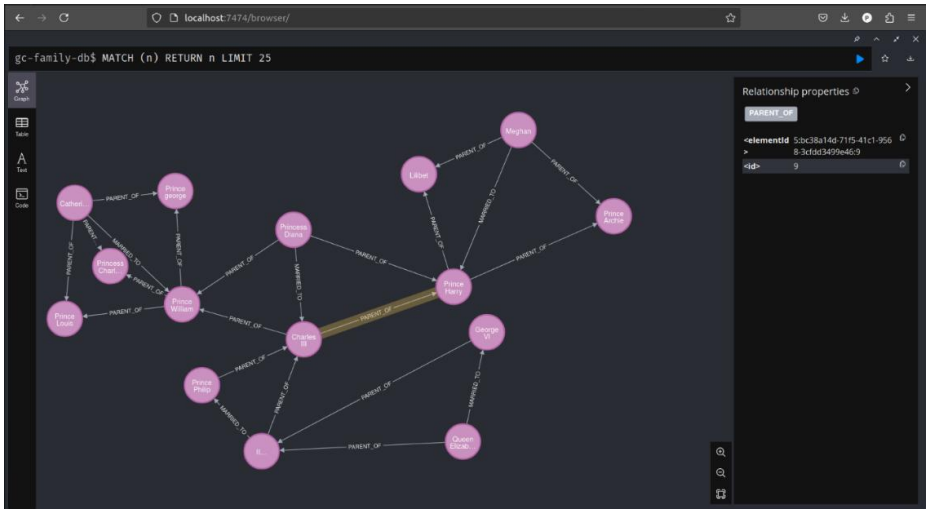


Figure 14 Neo4j Browser Dashboard

7.3.2 Audit Metadata

The node uses Spring Data annotations to track audit information:

- `@CreatedBy, @CreatedDate`: Automatically stores the user and timestamp when the node was created.
- `@LastModifiedBy, @LastModifiedDate`: Automatically updated on node modification.
- `@Version`: Enables optimistic locking to manage concurrent updates safely.

7.3.3 Design Considerations

- **UUID-Based Identifiers**: Using UUIDs for `personId` avoids collisions and allows easier merging of trees across distributed systems.
- **Loose Coupling with Relational DB**: While Neo4j handles the graph-specific data (relationships and traversal), linkage to the PostgreSQL `family_tree_id` maintains coherence between systems as shown in *Appendix 50*.
- **Extendibility**: The person node includes many optional biographical fields by design. These do not impact graph performance significantly but allow the application to support a wide variety of research and historical use cases.
- **Audit Integration**: Audit fields ensure all data changes are traceable, supporting collaborative genealogy research with accountability.

08

System Logic and API Design

This chapter presents the system's functional logic using flows using code-snippets, and sequence diagrams. It covers role-based authentication, privacy controls, family tree management, user interaction features, invitation handling and user management within the family tree, AI ontegration, research API integration, and auditing of the system along with usage metrics on the admin side.

8. System Logic and API Design

As an architecture, all the endpoints serving the frontend are developed in Spring Boot Framework using the Model-View-Controller (MVC) pattern, which is an architectural design approach that separates the application based on separation of concerns to handle and manipulate data effectively.

The three key components—Model, View, and Controller—are interconnected in such a way that, when an endpoint is triggered through the Controller layer, it utilizes Data Transfer Objects (DTOs) (which are distinct from database columns) for both request and response, as used and received by the user [34].

The responsibility of the controller is simply to forward the request to the corresponding service interface and return the response received from it. When a request reaches the service interface, all classes that implement this interface become potential candidates to handle the request. This approach proves advantageous in multi-tenant (multi-platform) systems, where each client platform can have its own specific implementation without affecting others.

Once the request reaches the service implementation, the actual business logic is executed—this includes fetching, manipulating data, and interacting with the database by calling the repository layer. This layer communicates with the database using its implemented methods, which execute SQL queries to fulfil the request.

In the Generation Connect application, both JPA and Neo4j repositories are configured at the repository layer, enabling a hybrid and flexible architecture that supports transactional changes involving both relational and graph-based data.

After the response is generated in the service implementation layer, it is passed back through the service interface to the controller, which then returns the response (again in the form of a DTO) to the user. Exception handling is integrated throughout the process to manage scenarios such as unauthorized access, resource not found, internal server errors, and many others.

This architectural flow is consistently followed across all endpoints designed in the application.

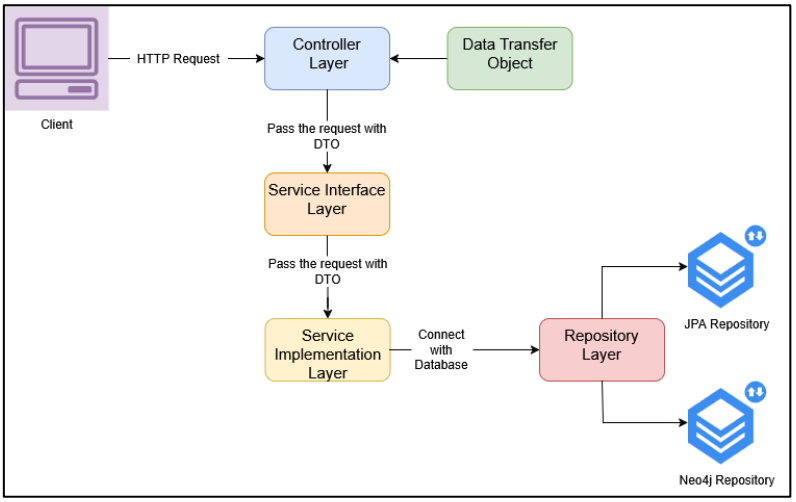


Figure 15 Used MVC architecture flow

8.1 Role-Based Authentication Flow

To keep the application secure, user login and access control are managed using Spring Security along with JWT (JSON Web Token). This setup ensures that only authenticated users can access protected parts of the system, and their access is based on their assigned roles.

The main flow includes three key actions: signing up, logging in, and logging out. All of this is handled through the `UserSigningController.java`. Once a user logs in, their token is processed by a custom `JwtAuthFilter`, which checks if the token is valid. To access user-specific data or perform actions, other parts of the system use `AuthUtils` to reliably identify the currently logged-in user.

After following the sequence described in *Appendix 9* to register a user, the system saves the user in the database with encrypted credentials and role information.

1. Sign-In and Token Generation

During login (as outlined in *Appendix 10*), authentication is performed using the `AuthenticationManager`:

```
Authentication authentication = authenticationManager.authenticate(
    new UsernamePasswordAuthenticationToken(signInRequestDTO.getUsername(),
signInRequestDTO.getPassword())
);
```

If the user is authenticated, a JWT token is generated and stored in a cookie:

```
String accessToken = jwtService.GenerateToken(username, roles);
```

The token includes claims such as roles and username, and is created using:

```
Jwts.builder()
    .setClaims(claims)
    .setSubject(username)
    .setIssuedAt(new Date(System.currentTimeMillis()))
    .setExpiration(new Date(System.currentTimeMillis() + 1000000 * 60))
    .signWith(getSignKey(), SignatureAlgorithm.HS256)
    .compact();
```

The token is added to the response as a cookie:

```
ResponseCookie responseCookie = ResponseCookie.from("accessToken", accessToken)
        .httpOnly(false)
        .secure(false)
        .path("/")
        .domain("localhost")
        .maxAge(86400) //1 day in seconds
        .build();
httpServletResponse.addHeader(HttpHeaders.SET_COOKIE, responseCookie.toString());
```

2. Token Validation in JwtAuthFilter

Every incoming request is filtered by JwtAuthFilter, which:

- Extracts the token from the cookie
- Parses the username
- Validates the token signature and expiration
- Sets the user in SecurityContextHolder

```
String username = jwtService.extractUsername(token);
if (jwtService.validateToken(token, userDetails)) {
    UsernamePasswordAuthenticationToken authenticationToken =
        new UsernamePasswordAuthenticationToken(userDetails, null,
userDetails.getAuthorities());
    SecurityContextHolder.getContext().setAuthentication(authenticationToken);
}
```

This is what makes the user appear "authenticated" in the rest of the application.

3. Accessing Current User Info

Anywhere in the backend, the current logged-in user can be fetched using:

```
String userId = authUtils.getCurrentUserId();
```

This internally uses Spring Security's context:

```
Authentication auth = SecurityContextHolder.getContext().getAuthentication();
if (auth.getPrincipal() instanceof GCAuthUserDetails userDetails) {
    return userDetails.getUsername(); // returns emailId
}
```

4. Logging Out

As described in *Appendix 11*, logging out clears the authentication cookie.

5. Role-Based access control

In `SecurityConfig`, protected routes are restricted based on roles using `.hasRole(...)`:

```
.requestMatchers("/person/**").hasRole("USER")
.requestMatchers("/admin/**").hasRole("ADMIN")
```

The user's role is stored as part of the JWT and interpreted by Spring Security in:

```
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
    SimpleGrantedAuthority roleUser = new SimpleGrantedAuthority(
        user.isGenerationConnectAdmin() ? "ROLE_ADMIN" : "ROLE_USER");
    return List.of(roleUser);
}
```

With this centralized setup for authentication and authorization:

- Only authenticated users can access sensitive endpoints
- Roles (USER/ADMIN) define what each user can do
- JWT tokens are securely issued and validated on each request
- User identity is consistently managed through Spring Security's context

8.2 Family Tree Creation and Maintenance

1. User Triggers a Create Request

A user who is logged in on the frontend fills in a form to create a new family tree. They enter a name, an optional description, and select an access level (such as private or public).
When they click the "Create" button, the frontend sends a POST request to:

```
POST /family-tree/create
```

The request structure is provided in the *3.A Create family Tree.*

Although memberUserId is sent from the frontend, the backend also fetches the logged-in user's ID from the security context. As discussed below in this section, this value overrides the one sent from the frontend to ensure security and prevent misuse.

2. Controller Picks Up the Request

The request is received by the `FamilyTreeController` at the Controller layer. The controller performs the following:

- Retrieves the logged-in user's ID from Spring Security and sets the memberUserid field in the familyTreeDTO object:

```
String userId = authUtils.getCurrentUserId();
familyTreeDTO.setMemberUserId(userId);
```

- userId received is added to the `familyTreeDTO` object as `memberUserId`. This means the backend now knows who is creating the tree, even if the frontend didn't say it.

And then request is forwarded to `FamilyTreeService`, i.e Service Interface as below:

```
FamilyTreeDTO savedTree = familyTreeService.createFamilyTree(familyTreeDTO);
```

At this stage, the controller waits for a response. If successful, it returns a 201 CREATED status along with the saved tree data.

3. Service Validates the User

Once the service receives the request, the implementation class `FamilyTreeServiceImpl` takes over. Inside the `createFamilyTree`(...) method, the following check is made:

```
UserEntity userEntity = userRepository.findUserByEmailId(familyTreeDTO.getMemberUserId());
```

This verifies whether the user exists in the application's user table. Even though the user is authenticated, this ensures the user is properly registered in the system.

If the user is not found, an error is thrown:

```
throw new UnauthorizedAccessException("Unauthenticated member trying to create
familyTree..");
```

This acts as a safeguard to prevent unauthorized users from proceeding further.

4. Tree Data Is Saved

If the user is valid, the family tree is saved through the JPA repository:

```
FamilyTreeEntity saved =
familyTreeRepository.save(FamilyTreeConverter.getEntity(familyTreeDTO));
```

At this point:

- The DTO is converted into an entity.
- The entity is saved into the `family_tree` database table.
- A unique `familyTreeId` is generated automatically.

The family tree is now successfully stored in the relational database.

5. Ownership Record Is Created

After the tree is saved, the system creates a separate record assigning the logged-in user as the OWNER of the tree. This record is saved in the `user_access_management` table.

First, it builds a key to uniquely identify the relationship between the tree and user via creating composite key:

```
UserAccessManagementEmbeddedId id = new UserAccessManagementEmbeddedId();
id.setFamilyTreeId(saved.getFamilyTreeId());
id.setMemberUserId(saved.getCreatedBy());
```

Then, it builds the ownership entity:

```
UserAccessManagementEntity entity = new UserAccessManagementEntity();
entity.setId(id);
entity.setFamilyTree(saved);
entity.setUser(userEntity);
entity.setUserAccessRole(UserAccessRoleEnum.OWNER);
entity.setUserAddedBy(saved.getCreatedBy());
```

Finally, it is saved:

```
manageUserAccessRepository.save(entity);
```

6. Audit Log Is Written

To keep track of system activity, an audit log entry is created:

```
familyTreeAuditEventService.addEvent(
    saved.getFamilyTreeId(),
    FAMILY_TREE,
    isCreate ? ActionType.ADD : ActionType.UPDATE,
    saved.getFamilyTreeId().toString(),
    saved.getFamilyTreeName()
);
```

This stores metadata about the action — such as what was done, who did it, and when — which can be used later for tracking and reporting. For a visual representation of this flow, refer to the sequence diagram in *Appendix 15*.

7. Final Response Is Returned

Once all actions are completed, the service sends the saved tree data back to the controller:

```
return ResponseEntity.status(HttpStatus.CREATED).body(savedTree);
```

The frontend receives the tree details and render the same.

This marks the initial step in the family tree feature. The next major functionality is adding persons to the tree starting with the root person and building relationships. As discussed in *Section 5.4*, after exploring various options, the system was designed to use a Neo4j graph database to manage family members and relationships, instead of extending the relational JPA structure. This decision allowed more flexibility in representing complex family structures.

The following section explains how the system handles person creation and updates using Neo4j.

8. User Triggers the Add/Update Endpoint as a new request

Once the family tree is created, the user can start adding people to it. From the frontend, the user fills in a form with details like name, birthdate, gender, etc., and optionally selects relationships such as parents, children, or spouse.
When the form is submitted, the frontend sends a POST request to:

```
POST /person/update2/{userId}/{familyTreeId}/person
```

This request contains the person's information as well as relationship mappings (like fatherId, motherId, spouses, children) in the body. The structure is based on PersonUpdateRequest.

9. Controller handles the Request

The request is received by the `PersonController` class. The method checks whether the user has access to the given family tree:

```
if (familyTreeService.getAllFamilyTreeByUserIdIncludingPublic(userId) == null) {
    throw new FamilyTreeNotFoundException("No family tree found connected to userId " +
userId + " and " + familyTreeId + " familyTreeId.");
}
```

If access is valid, the request is passed to the service layer:

```
personUpdateRequest.setFamilyTreeId(familyTreeId);
Person updatedPerson = personService.updatePerson(personUpdateRequest, userId);
```

This passes both the form data and user information to the service that handles business logic via service interface.

10. Service Validates Access

In the `PersonServiceImpl`, the first check ensures the user has either OWNER or CONTRIBUTOR role for the family tree:

```
validateContributorOrOwnerAccess(personUpdateRequest.getFamilyTreeId(), userId);
```

This uses the `user_access_management` table to verify that the logged-in user is allowed to modify the tree. If the user doesn't have the right role, an error is thrown.

11. Person Is Created or Updated

Next, the service looks for an existing person in the Neo4j graph database:

```
Optional<Person> personForUpdate =
personRepository.findPersonByPersonId(personUpdateRequest.getPersonId());
Person person = personForUpdate.orElseGet(Person::new);
```

If the person exists, it updates the record. If not, it creates a new node. The service sets all the values sent from the frontend:

```
person.setFirstName(...);
person.setLastName(...);
person.setBirthdate(...);
// ... sets other fields like occupation, languages, avatar, etc.
```

Then the person node is saved under the correct `familyTreeId`:

```
Person saved = personRepository.save(person);
```

12. Relationships Are Created

After saving the person, the system checks if any relationships are defined in the request. These include:

- father
- mother
- spouses
- children

For each relationship type, the system looks up the related person node in the Neo4j graph database. If found, it creates a relationship using Cypher queries defined in the repository interface. For example,

Parent-Child Relationships: If a father or mother ID is provided, the system creates a PARENT_OF relationship from the parent to the current person.

Spouse Relationships: If spouse IDs are provided, the system creates a MARRIED_TO relationship between the current person and each spouse.

Child Relationships: If child IDs are provided, the system creates a PARENT_OF relationship from the current person to each child.

These queries are executed through annotated methods in `PersonRepository`, such as:

```
personRepository.createParentChildRelationship(fatherId, saved.getPersonId());
personRepository.createMarriageRelationship(saved.getPersonId(), spouseId);
personRepository.createParentChildRelationship(saved.getPersonId(), childId);
```

Each method internally runs a Cypher query, Neo4j's graph-based query language. For example, the parent-child relationship uses the following Cypher:

```
MATCH (p1:Person), (p2:Person)
WHERE p1.personId = $parentId AND p2.personId = $childId
MERGE (p1)-[:PARENT_OF]->(p2)
```

Similarly, the marriage relationship uses:

```
MATCH (p1:Person), (p2:Person)
WHERE p1.personId = $person1Id AND p2.personId = $person2Id
MERGE (p1)-[:MARRIED_TO]-(p2)
```

These queries are automatically executed by Spring Data Neo4j when the corresponding repository methods are called.

If a related person is not found in the tree, the system logs the issue (e.g., missing parent or child) but continues processing without failing. This allows flexibility while building relationships step by step.

13. Action Is Logged

```
familyTreeAuditEventService.addEvent(
    person.getFamilyTreeId(),
    "Person",
    ActionType.UPDATE,
    person.getPersonId(),
    person.getFirstName() + " " + person.getLastName()
);
```

This makes sure all changes are traceable.

14. Final Response Is Sent

Finally, the updated person object is returned back to the frontend:

```
return ResponseEntity.ok(updatedPerson);
```

The frontend receives confirmation and the complete person data and can now reflect the changes visually in the family tree.

A complete code snippet to add/update Person to the family tree in *Appendix 53*.

A complete view of this workflow - is illustrated in the *Appendix 15* and *Appendix 16***.** Additionally, the resulting person-to-person connections such as PARENT_OF and MARRIED_TO are visualized in the relationship graph diagram in *Appendix 42* for better understanding of how data is structured within the family tree.

After persons are added to a family tree, the system provides several supporting operations to help users manage and interact with the tree effectively. These features include fetching people, managing collaborators, and handling user invitations.

Each operation follows a well-defined backend interaction pattern and is covered with detailed sequence diagrams in the Appendix as below:

Get All Persons of the Tree

Users can view all individuals within a specific family tree. This is achieved by performing a recursive traversal (depth-first) from the root person, collecting their relationships and constructing a hierarchical view.
See sequence diagram in Appendix 17

Get Users of the Tree

Authorized users can retrieve a list of all people who currently have access to the family tree, including their roles. This helps the owner manage permissions.
Refer to Appendix 18 for the detailed flow.

Add User to the Tree

A user with OWNER rights can add another user to the tree by email. The backend validates the ownership and avoids adding duplicates or reassigning the same user.
Flow illustrated in Appendix 32.

Update User Role

The OWNER can update another user's access role (e.g., from VIEWER to CONTRIBUTOR). The system ensures role-based security by preventing demotion of the owner or unauthorized changes.
Covered using detailed Sequence Diagram in Appendix 32.

Delete User from the Tree

Owners may remove users from the tree, unless the user being removed is the only OWNER. Appropriate error handling and audit logging are included.
Detailed sequence diagram in Appendix 34.

Send Invite

Users can send an email invite to someone to join a specific family tree. The system checks for existing invites and ownership before generating a secure token and sending the email.
Sequence diagram in Appendix 33.

Accept Invite

Invitees can accept invitations using a tokenized link. The system verifies the token, prevents duplicate access, and then grants the specified access role.
See full flow using Sequence Diagram in Appendix 12.

8.3 Interaction Through Posts and Comments

Once the family tree is in place and people have been added, users can start make stories and memories related to individuals or the family. These are added as posts — either linked to a specific person or to the entire tree — and help bring the family history to life beyond just names and dates.

1. User Triggers a Post Creation Request

The user fills in a post form on the frontend, either while viewing a person's profile or a general family tree timeline. The form may include:

- A textual description (`postContent`)
- Target type: `personId` (optional) or `familyTreeId` (required)

When the form is submitted, a POST request is sent to:

```
POST /posts/
```

This request sends a `PostDTO` object that includes post content, the linked `familyTreeId`, and optionally a `personId`.

2. Controller Handles the Request

The request is received by the `PostController` class. First, the controller retrieves the current user's ID from the authentication context:

```
String userId = authUtils.getCurrentUserId();
postDto.setCreatedBy(userId);
```

Then the controller forwards the request to the `PostService`:

```
PostDTO createdPost = postService.createPost(postDto, userId);
```

If successful, the controller returns a 201 CREATED response:

```
return new ResponseEntity<>(createdPost, HttpStatus.CREATED);
```

3. Service Validates Access

Inside the `PostServiceImpl`, the `createPost`(...) method begins by validating that the current user has either OWNER or CONTRIBUTOR access to the specified family tree:

```
manageUserAccessService.validateUserRole(postDTO.getFamilyTreeId(), userId, List.of(OWNER,
CONTRIBUTOR));
```

This ensures that only authorized users can post content linked to a family tree.

4. **Post Entity Is Saved**

   If access is valid, the DTO is converted into a JPA entity using a converter:

```
PostEntity postEntity = postConverter.toEntity(postDTO);
PostEntity savedPostEntity = postRepository.save(postEntity);
```

   The post is saved to the post table in the relational database. Each post is linked to a family tree, and optionally a person if it's a user profile post.

5. **Action Is Logged**

   Once the post is successfully saved, the system creates an audit log entry to record the action. This logging mechanism helps track key user activities within the application and supports future features like history view, moderation, and admin reporting.

   The logging is handled by the `familyTreeAuditEventService`, which accepts details about the event such as the tree it belongs to, the type of post, the action performed, and a reference ID.

   The method call looks like this:

```
familyTreeAuditEventService.addEvent(
    savedPostEntity.getFamilyTreeId(),
    Objects.nonNull(postDTO.getPersonId()) ? USER_PROFILE_POST : FAMILY_PROFILE_POST,
    ActionType.ADD,
    postEntity.getId().toString(),
    null
);
```

   The second parameter uses a condition to decide the post type:

   - If the post is linked to a person (`personId` is not null), it is classified as a `USER_PROFILE_POST`. Otherwise, it is treated as a `FAMILY_PROFILE_POST`.
   - `ActionType.ADD`: Specifies that this is a new post being created.

   This audit entry is stored in a dedicated table for tracking family tree-related actions. It ensures that every content addition is traceable and can be reviewed or reported later if needed.

6. **Final Response Is Returned**

   After the post is saved and logged, the service converts it back to a DTO and returns it to the controller. The controller then responds with the newly created post, which is rendered in the frontend UI.

   As posts can be created not only for individual profiles but also for the shared family wall, it is important to ensure that person-specific posts do not appear under the general family profile

post section. To support this distinction, the system provides a dedicated flow for retrieving posts linked to a specific person.
These are fetched in descending order of creation time,

- as outlined in the sequence diagram in *Appendix 28* for individual person posts, and
- are associated with the relevant family tree context, as shown in *Appendix 26*.

Posts are retrieved in descending order of creation time:

```
findByPersonIdOrderByCreatedDateDesc(personId)
```

If no posts are found, the system returns a 204 No Content response.
Again, results to fetch posts are ordered by most recent:

```
findByFamilyTreeIdOrderByCreatedDateDesc(familyTreeId)
```

If no posts exist, the controller returns a 204 No Content.

A complete view of the workflow for adding posts to both individual profiles and the shared family tree is illustrated in the sequence diagram provided in *Appendix 21*.

This feature becomes even more collaborative when comments are added to a post, turning it into an ongoing thread of shared memories and storytelling. The process of adding comments follows a clearly defined flow, as shown in *Appendix 22,* and retrieving them is handled according to the sequence described in the *Appendix 23*.

8.4 Media Integration

To further enhance storytelling and historical context, the system allows users to upload multimedia content such as photos, documents, and videos. These media files can be attached to posts — whether they are linked to an individual or a family tree — or uploaded as standalone files to support genealogical research.

This feature helps bring personal stories to life visually and provides a way to preserve important artifacts like certificates, photographs, scanned letters, or archival material.

1. User Uploads Media Files

Users can upload one or more files along with a short description. The form captures:

- The list of files (`MultipartFile[]`)
- An optional description
- A required `mediaTypeId`, which determines where the uploaded files will be linked

The `mediaTypeId` is created on the frontend depending on what the files are being uploaded for. This helps the backend know whether the files are related to a post, the family tree, or a research document. The format follows a clear pattern:

- For individual posts: `"post_" + post.id`
- For shared family tree posts: `"family_profile_post_" + post.id`
- For research documents: `"family_tree_document_" + familyTreeId`

Using this pattern makes it easy for the current system to organize and retrieve files correctly across different features. It also makes the media upload feature scalable for future use cases that may need to distinguish between different upload types in a similar way.

When the user submits the form, a multipart POST request is sent to:

```
POST /media/upload
```

This request is handled by the `CommentMediaUploadController`, which forwards the files and metadata to the media storage service:

```
mediaStorageService.storeFiles(mediaTypeId, description, files);
```

2. Retrieve Uploaded Files

Files can be fetched by their associated mediaTypeId using the following endpoint:

```
GET /media/by-media-type/{mediaTypeId}
```

The system queries all media entries linked to that ID, loads their metadata, calculates the file size, and formats the upload date for display:

```
return mediaRepository.findByMediaTypeId(mediaTypeId).stream()
    .map(media -> new UploadedMediaDTO(...))
    .collect(Collectors.toList());
```

The frontend then shows media previews or downloadable file links within posts, comments, or research sections.
A complete view of the workflow for uploading posts to both individual profiles, shared family tree and historical research document is illustrated in the sequence diagram provided in *Appendix 24* and *Appendix 25*.

8.5 Research – Search and Tag Historical Record

1. User Triggers a Search Request

The user initiates a search by entering name, date of birth, or other filters into the research form on the frontend. They also select a supported research source such as The National Archives UK.

When the "Search" button is clicked, the frontend sends a GET request to:

```
GET /research/records/live/{researchSourceName}
```

The path variable `researchSourceName` is mapped to an `enum` called `ResearchSourceNameEnum`, and query parameters such as firstName, lastName, `dateOfBirthFrom`, and `searchQuery` are passed along.

The request is captured and mapped to a `ResearchRequestDTO`, which contains request body as given in *Appendix 7.1*

2. Controller Picks Up the Request

The `ResearchController` receives the request at:

```
@GetMapping("/records/live/{researchSourceName}")
```

It prepares the DTO and forwards it to the service layer:

```
Object result = researchService.fetchLiveRecords(researchRequestDTO, researchSourceName);
```

The controller then returns the response to the frontend.

3. Service Determines Source and Calls External API

Inside the service layer (`ResearchServiceImpl`), the method `fetchLiveRecords`(...) decides which external source to query:

```
if (researchSourceNameEnum.equals(ResearchSourceNameEnum.THE_NATIONAL_ARCHIVE_UK)) {
    return searchLiveRecordsOfUK(researchRequestDTO);
}
```

Currently, only UK records are supported. If the user selects an unsupported source, the system returns a message:

```
response.put("status", "error");
response.put("message", "Record searches for 'National Archive US' are not yet supported.");
```

4. Dynamic URL Is Constructed and Request Sent

If the source is supported, the system builds the full API URL using `UriComponentsBuilder`, dynamically adding filters only if they are non-null:

```
UriComponentsBuilder uriBuilder = UriComponentsBuilder.fromPath("/API/search/v1/records")
    .queryParam("sps.recordCollections", "All")
    .queryParam("sps.searchQuery", researchRequestDTO.getSearchQuery());
```

Other fields like first name, last name, and birthdate ranges are appended conditionally.
The URL is passed to a configured `WebClient`:

```
String json = researchWebClientConfig.nationalArchivesClient().get()
    .uri(uri)
    .retrieve()
    .bodyToMono(String.class)
    .block();
```

The JSON response is parsed using Jackson:

```
return objectMapper.readValue(json, Object.class);
```

The parsed object is returned to the controller and then to the frontend.
A complete view of this workflow is available as Sequence Diagram in Appendix 29.

5. User Tags Record to a Family Tree

Once a record is selected on the frontend, the user may tag it to a family tree by submitting a POST request and fetch Record using Get request by following the sequence diagram as in *Appendix 30* and *Appendix 31* respectively.

8.6 OpenAI Integration for Family Tree Queries and Events

To offer users intelligent assistance and deeper engagement with their family tree, an AI-powered chat feature is implemented during the final stages of development. This feature allows users to ask questions related to their family tree graph and mainly it is configured to support coming up events tracking and timelines of the family tree.

The AI assistant acts as a conversational support layer, enhancing both events tracking and timelines.

1. **User Sends a Chat Message**
   On the frontend, the user types a query and clicks the "Send" button. This sends a plain text string as the body of a POST request to the endpoint:

```
POST /api/ai/chat
```

   The backend does not require additional metadata — just the raw user message in the request body.

2. Controller Picks Up the Request

The request is received by the `AiController` class at the controller layer. The controller performs the following:

- Wraps the plain message into a `ChatMessage` object.
- Sets the role to "user" to match OpenAI's expected structure.
- Forwards the constructed message list to the service layer:

```
ChatMessage message = new ChatMessage();
message.setRole("user");
message.setContent(userMessage);

String reply = openAiService.getResponse(List.of(message));
```

At this point, the controller waits for a response. If successful, it returns a 200 OK status and the AI-generated reply to as plain text.

3. Service Prepares the OpenAI Request

The request is handled by the `OpenAiService` class. Inside the `getResponse(...)` method, the following operations are performed:

Headers are set to include the API key and indicate JSON content:

```
headers.setBearerAuth(apiKey);
headers.setContentType(MediaType.APPLICATION_JSON);
```

A request body is constructed using the `OpenAiRequest` DTO:

```
OpenAiRequest request = new OpenAiRequest();
request.setMessages(messages);
```

The default model **gpt-3.5-turbo** is automatically set in the DTO.

3. OpenAI API Is Invoked

The backend uses Spring's `RestTemplate` to send the POST request to the configured OpenAI endpoint:

```
ResponseEntity<OpenAiResponse> response = restTemplate.postForEntity(apiUrl, entity,
OpenAiResponse.class);
```

The response contains a list of choices. The content of the first choice is extracted and returned:

```
return response.getBody().getChoices().get(0).getMessage().getContent();
```

5. Final Response Is Returned

The service sends the extracted AI-generated response back to the controller, which then returns it as a simple string in the response body. Refer *Appendix 36* for Sequence Diagram.

8.7 REST API Endpoint Documentation

REST API endpoint documentation is showcased using the Swagger page, which includes all endpoints consumed by the current frontend along with a few additional endpoints. Refer to *Appendix 41*.

This section outlines all available RESTful endpoints in the Generation Connect platform, categorized into Public and Protected APIs.

Public Endpoint Available without Authentication

These endpoints are accessible without logging in and are primarily used for account onboarding and invitation acceptance.

API Title	Endpoint	Responsibility	References in Appendix
User Signup	POST /auth/signup	• Register new user • Check duplicate email • Encode password • Save to DB	Request & Response Body Sequence Diagram
User Login	POST /auth/login	• Authenticate user • Verify role • Issue JWT Set auth cookie	Request & Response Body Sequence Diagram
User Logout	POST /auth/logout?user=`user_email_id`	• Clear access token cookie • Logout specified user	Request & Response Body Sequence Diagram
Accept Sent Invite	POST /family-tree/manage-users/accept	• Validate invite token • Verify identity • Assign access role • Mark invite as accepted	Request & Response Body Sequence Diagram

Protected Endpoint accessible only via generating cookie via Signing Using above API's

Accessible only after logging in, these endpoints require a valid session token.

Administrative APIs

API Title	Endpoints	Responsibility	References in Appendix
User Metrics	GET /admin/metrics	• Return overall app statistics • Include user count, post/comments totals • Show number of active and averages	Request & Response Body Sequence Diagram
Audit Tree List	GET /admin/family-tree-list/all	• Fetch metadata for all family trees • Include ID, name, creator, description, access level • Sort by ID	Request & Response Body Sequence Diagram

Family member or User APIs

Family Tree Generation			
API Title	Endpoint	Responsibility	References in Appendix
Create Family tree	POST /family-tree/create	• Create a new family tree • Assign OWNER role to the creator • Persist tree and access record • Log creation in audit events	Request & Response Body Sequence Diagram
Add/Update Person	POST /person/update2/johnDoe@gc.ac.in/{familyTreeId}/person	• Add a new person or update existing • Manage relationships (parent, child, spouse) • Enforce access roles • Log update in audit events	Request & Response Body Sequence Diagram
Get Family Tree member	GET /person/get-family/{familyTreeId}	• Traverse tree from root person • Return all connected person nodes • Provide person details and relationships	Request & Response Body Sequence Diagram

List all User's trees	GET /family-tree/getAllTrees	• Fetch trees created by user • Include shared (private) trees • Include public trees	Request & Response Body Sequence Diagram
Delete family tree	DELETE /family-tree/delete/{familyTreeId}	• Verify requester is the creator • Delete Neo4j and JPA-linked resources • Log deletion in audit events	Request & Response Body Sequence Diagram
Delete person	DELETE /person/delete/{userId}/{familyTreeId}/{personId}	• Verify OWNER or CONTRIBUTOR access • Should not delete Owner of Family tree • Delete and log action.	Request & Response Body Sequence Diagram
Ask AI	POST /api/ai/chat	• Create temporary chat request with messages • Authenticate request using bearer token • Send to OpenAI API and retrieve response content	Request & Response Body Sequence Diagram

Post and Comments (Shared across family & Individual profiles)			
API Titles	Endpoints	Responsibility	References in Appendix
Add Post	POST /posts/	• Create a new post for family or individual profile • Validate OWNER or CONTRIBUTOR access • Log post creation in audit events	Request & Response Body Sequence Diagram
Add Comment	/comments/{familyTreeId}	• Create a comment on a post • Enforce access control • Save comment metadata	Request & Response Body Sequence Diagram
Get Comments	/comments/post/{postId}	• Fetch all comments for the given post ID • Return as a list ordered by creation date	Request & Response Body Sequence Diagram
Upload Media	POST /media/upload	• Upload one or more media files (images, PDFs, etc.) • Store in structured directory	Request & Response Body Sequence Diagram

		• Persist metadata	
Get Media by type	GET /media/by-media-type/{mediaTypeId	• Retrieve all media files linked to a specific media type (e.g., post ID) • Return metadata including size and file type	Request & Response Body Sequence Diagram

Family Tree Profile Wall			
API Titles	Endpoints	Responsibility	References in Appendix
Get Family Posts	GET /posts/family/{familyTreeId}	• Retrieve all posts created under the specified family tree • Ordered by creation date	Request & Response Body Sequence Diagram
Get Family Tree Recent Activities	GET /family-tree-events/{familyTreeId}	• Fetch recent activities (e.g., post, comment, person updates) on the family tree • Supports audit log view for admins or tree owners	Request & Response Body Sequence Diagram

Individual Profile of Person in Family Tree			
API Title	Endpoint	Responsibility	References in Appendix
Get Person Posts	GET /posts/person/{personId}	• Fetch all posts associated with a specific person in a family tree • Ordered by creation date	Request & Response Body Sequence Diagram

Family Tree Historical Research			
API Title	Endpoints	Responsibility	References in Appendix
Search Historical Records	GET /research/records/live/`researchSourceName` searchQuery=`searchQuery`	• Build request based on search fields (name, DOB, etc.) • Call external archive API (e.g., UK National Archive) • Return parsed results	Request & Response Body Sequence Diagram

Tag Record to Family Tree	POST /research/records/tag	• Validate user access (OWNER/CONTRIBUTOR) • Check for duplicates • Save research record with metadata and source	Request & Response Body Sequence Diagram
Get Tagged Research Records	GET /generation-connect-api/research/records/tagged/{familyTreeId}	• Fetch all tagged research records for a family tree • Return structured list of tagged records with metadata	Request & Response Body Sequence Diagram

Manage Users in the Family Tree			
API Title	Endpoint	Responsibility	References in Appendix
Add existing user to the family tree	POST /family-tree/manage-users/{familyTreeId}	Grants access to an existing user for a specified family tree based on their email and role.	Request & Response Body Sequence Diagram
Send invite	POST /family-tree/manage-users/send	Generates a time-limited invite token and emails the invite link to the specified user.	Request & Response Body Sequence Diagram
Update Role	POST /family-tree/manage-users/{familyTreeId}	Changes the access role of a user already added to the family tree (except OWNER role).	Request & Response Body Sequence Diagram
Remove user	DELETE /family-tree/manage-users/{familyTreeid}	Removes a user from the specified family tree. Only the tree owner is authorized.	Request & Response Body Sequence Diagram
Get users	GET /family-tree/manage-users/{familyTreeId}/users	Returns the list of users with access to the specified family tree and their roles.	Request & Response Body Sequence Diagram

09

User Interface and Presentation Layer

This section focuses on the user interface and presentation layer of the application. It outlines the structure of frontend components, organized based on functionality. Additionally, the visual style guide defines the used color palette and layout at high level.

9. User Interface and Presentation Layer

The frontend application is developed using React with a component-based architecture. While the project does not strictly follow advanced modular or scalable component styling patterns, the structure is organized in a way that aligns with practical ease of development and role-based usage separation.

9.1 Component Directory Structure

The `/src` directory under `/frontend` is divided into two main segments: component and pages.

/components

The component folder contains reusable UI building blocks and visual elements used across multiple pages. These include:

1. **family-chart**:

   This component uses an open-source chart library (f3) [20] to visualize family tree data stored in Neo4j. While the library itself is included as an external module, the integration with the application—particularly the dynamic rendering of the tree through callback mechanisms—is custom-implemented as part of this project.
   The family-chart component is not only used to display data—it also supports interactive modifications by users with the appropriate permissions.
   As part of this project, a custom integration was implemented using the `setDatumCallback` method, which enables real-time updates or deletions of individuals within the family tree. This callback functionality was developed and integrated specifically to allow user-initiated changes to be reflected in both the UI and the underlying Neo4j database.

```
f3EditTree.setDatumCallback(async (d, props, isAddRelative) => {
  if (props && props.delete) {
    await FamilyTreeService.deletePerson(userId, familyTreeId, d.id);
    return;
  }
  await FamilyTreeService.updatePerson(userId, familyTreeId, d);
});
```

   The function acts as a centralized handler for both updating and deleting individuals (nodes) in the family tree. This callback is triggered by the f3 library whenever a user performs an edit action on a node—such as modifying personal details or removing a node entirely.

   The function receives three arguments:

   - `d`: the data object representing the person (node) being edited or deleted.
   - `props`: an object containing metadata about the action. If the delete flag is present, it indicates that the user has initiated a deletion.

- `isAddRelative`: a boolean indicating whether the action involves adding a new relative node.

When a deletion is detected (`props.delete` is `true`), the system calls `FamilyTreeService.deletePerson(...)` to remove the person from the database using the `userId`, `familyTreeId`, and the person's unique ID `(d.id)`.

For all other cases, the assumption is that the node is being added or updated. The `FamilyTreeService.updatePerson(...)` method is called to synchronize the update node `data (d)` with the backend.

This implementation abstracts backend interaction behind a single callback, allowing the tree to reactively update in the frontend once the operation is complete.
The system determines whether a user has editing privileges based on their role—only users with the role Owner or Contributor are permitted to modify the family tree. If the user is allowed to edit (`allowEdit` is true), the tree initializes in edit mode, enabling interactions like opening a node for editing or adding relatives.

For users without edit rights (e.g., Viewers), the tree remains in view-only mode, allowing them to explore but not modify the structure.

```
const allowEdit = userRole === "Owner" || userRole === "Contributor";

if(allowEdit) {
      f3EditTree.setEdit();
      f3Card.setOnCardClick((e, d) => {
          console.log("onCardClick", d);
        f3EditTree.open(d);
        if (f3EditTree.isAddingRelative()) return;
        f3Card.onCardClickDefault(e, d);
      });
    } else {
      // View-only interaction
      f3Card.setOnCardClick((e, d) => {
        console.log("onCardClick (view-only)", d);
        f3Card.onCardClickDefault(e, d);
      });
    }
```

Additionally, every tree node includes a profile button. When clicked, the following interaction is triggered:

```
f3Card.setOnProfileButtonClick((e, d) => {
  console.log('Profile clicked:', d.data);
  setProfileData(d.data);
  setIsProfileOpen(true);
});
```

This opens a profile overlay for the selected individual in the tree, enabling users to view detailed information without altering the tree layout. This feature supports both editable and view-only modes, enhancing the user experience by providing seamless access to profile details directly from the tree.

Additionally, date validations have been implemented to ensure that the birthdate cannot be set in the future, and the demise date must not be in the future or earlier than the birthdate.

The first name field has also been made mandatory, with an overlay error message shown on invalid submission, as demonstrated below and styling could be found in *Appendix 52*:

```
function postSubmit(props) {
    if(!datum?.data?.firstName){
      console.log("form submit :" + JSON.stringify(datum?.data?.firstName));
      this.showOverlayMessage("First name is mandatory");
      return;
    }
    const todayDate = new Date().toISOString().split("T")[0];
    const birthdate = datum?.data?.birthdate;
    if(birthdate && birthdate > todayDate) {
      this.showOverlayMessage("Birthdate cannot be in the future.");
      return;
    }

    const demiseDate = datum?.data?.demiseDate;
    if (birthdate && demiseDate) {
      if (demiseDate < birthdate) {
        this.showOverlayMessage("Demise date cannot be before birthdate.");
        return;
      }
      if (demiseDate > todayDate) {
        this.showOverlayMessage("Demise date cannot be in the future.");
        return;
      }
    }
//..
```

2. **/post**:

   This component handles posts created by users. Posts can include text and media like images or videos. It is connected to a media service that handles uploading and saving files. This is used in family profiles to share memories or updates.

3. **/recent-actions**:

   This shows a list of recent actions by the user. For example, if a user adds a person to the tree or uploads a document, that action is shown here. It helps users keep track of what they've done recently.

4. **ResponseMessage.js**:

   This is a small but important component used to show success, error, or info messages in a standard way. It is used in many places across the app to give users feedback after an action, like saving a profile or deleting a post.

5. **/AiChat/AiChatBox.js:**

   This component enables users to interact with an AI assistant by sending questions or prompts related to genealogy, historical information, or platform guidance. The AI assistant responds with helpful suggestions or narrative support, enhancing the overall user experience.
   The component uses a floating chat interface that can be toggled open or closed. Once open, users can type a message and receive a dynamic response from the backend-integrated OpenAI API.
   When the user clicks "Send", the following occurs:

   - The current date is appended to the request.
   - The full family tree data (passed via `familyTreeData` prop) is included to give context.
   - The user's input is also wrapped with a special instruction asking the AI to "respond in HTML markup", allowing the frontend to display formatted responses.

```
body: JSON.stringify("current date : " + new Date().toISOString())
  + "\n" + JSON.stringify(familyTreeData)
  + "\n" + JSON.stringify(userMessage)
  + "\n" + JSON.stringify("respond in html markup")
```

This entire payload is sent to the backend via a POST request method. The backend processes this and returns a plain text response containing the AI's reply.

Once received, the reply is displayed inside the chat box UI. The response supports basic HTML, making it possible to show structured content like paragraphs, bullet points, or emphasized text.

In case of failure or connection issues, a fallback message such as 'Error connecting to AI service.' is shown to the user.

/pages

The pages directory is organized based on high-level views of the application, structured as follows:

1. **/auth**:

   Manages user authentication and is subdivided into admin and user folders to reflect different access roles. This separation allows each role to maintain specific login, signup, or logout interfaces.

2. **/dashboard**:

   Contains the user and admin dashboards, further divided into:

   - /family_tree – For managing and viewing family trees.
   - /genealogy_research – For research integration with external record sources.
   - /profiles – Displays user profiles linked to family trees.

3. **/home**:

   Includes landing and static content pages such as `ContactPage.js`, `FaqsPage.js`, and `ResourcesPage.js`.

Each page-level component includes its own dedicated `*.styles.js` file to encapsulate styling logic using styled-components, and where applicable, a separate service file to handle API calls. This localized approach helps maintain readability and minimizes impact on other components during development.

***Note**: Currently, the family tree node supports profile pictures through image URLs only. Due to time constraints, the functionality for direct image upload was not implemented, as it required additional handling for integration with a very complicated library i.e family-chart [20]. Given more time, this feature could have been incorporated to enhance user convenience and flexibility.*

9.2 User Interface Design and Visual Style Buide

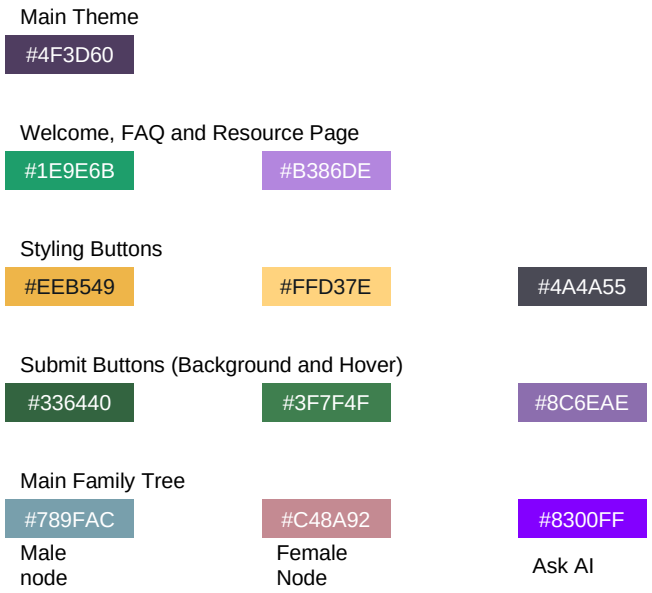
The interface design follows a practical layout pattern:

- Component segregation is based on functional grouping rather than strict atomic design.
- Styling is implemented through component-scoped styled-components.
- Services are split per feature to maintain isolation and avoid unintended coupling.

Colour Palette

This palette was chosen to:

- Ensure readability and contrast compliance for accessibility.
- Maintain a calm and professional appearance.
- Support visual feedback (e.g., success and error Response Message Type).



10

Functional Features and Usage Scenarios

This chapter highlights the core features of the system and demonstrates how users interact with them in real-world scenarios using Alice Story used previously. It showcases the practical application of functionalities to meet user needs effectively.

10. Functional Features and Usage Scenarios using UI

As in *Section 4.1*, the story of Alice and her family showed how the platform could be used in real life. It helped explain the purpose of the system by showing what different users can do—like creating a tree, inviting family members, and adding memories.

In this section, the same story is looked at in a different way. Instead of telling it like a journey, the focus here is on the real pages and parts of the app that were built to make those actions possible.

Each feature in the story is now matched with the actual screen or component that was developed. For example, when Alice signs up, she uses the Sign-Up Page. When she creates a tree, she uses the Create Family Tree Page. When her daughter uploads photos, she uses the Profile Page with the Post component.

This helps show how the system works from the user's point of view and connects the story to the working parts of the app. It also shows how different people use the system based on their role—like Owner, Contributor, or Viewer.

1. Discovering the Platform

Alice, a passionate family historian and retired librarian, wanted to document her family's history in a way that could be shared with future generations. She searched online for web applications that allowed people to build and preserve family trees.

After exploring several platforms, she discovered Generations Connect. On the Homepage, she explored its core features—collaborative tree building, profile posts, historical document tagging, and support for both public and private trees. It was free to use and aligned perfectly with her goals, so Alice decided to get started. Other pages such as FAQ's and Resource page is also available in *Appendix 37* and *Appendix 38*

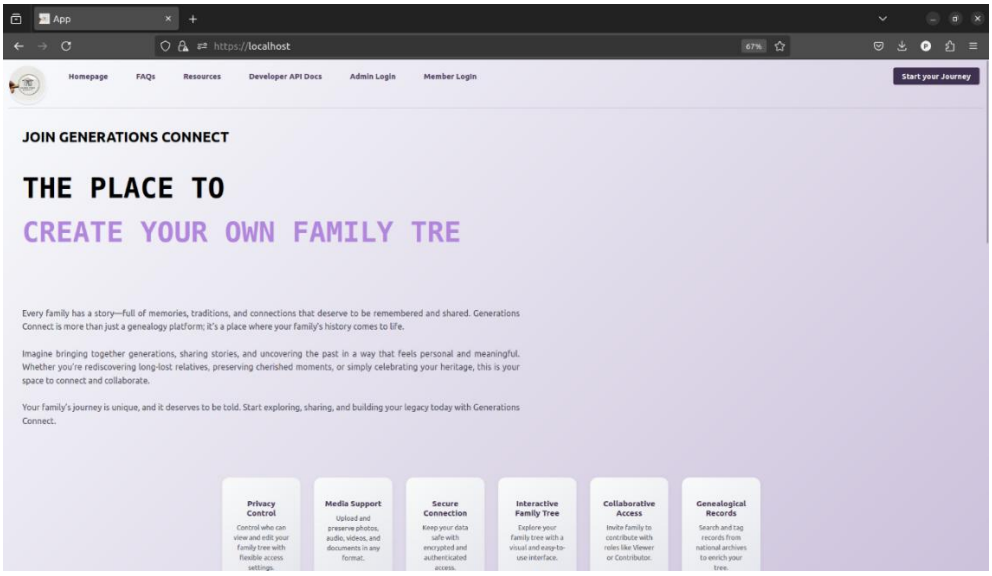


Figure 16 Generations Connect Homepage

2. Signing Up and Logging In

Alice opened the Sign-Up Page and filled out the registration form. Initially, she used a weak password, which triggered a validation message explaining the required format.

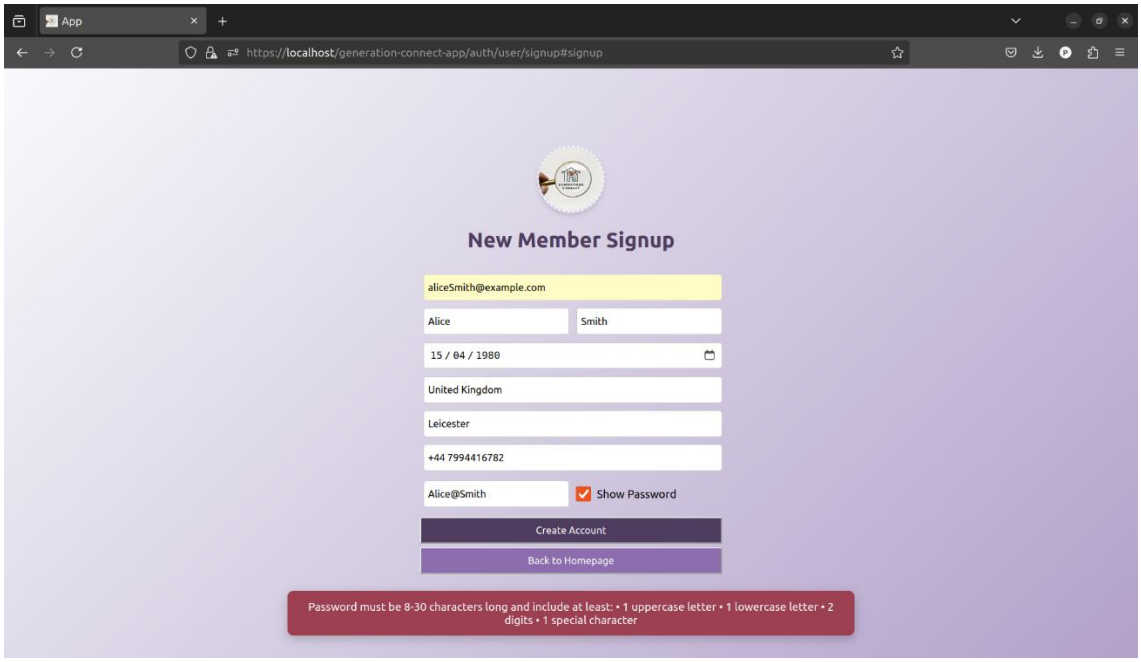


Figure 17 Signup Validation failed with required Password format error display

Alice updates her password to meet the guidelines, completed the form, and clicks "Create Account" A success message appears, and she is redirected to the Login Page.

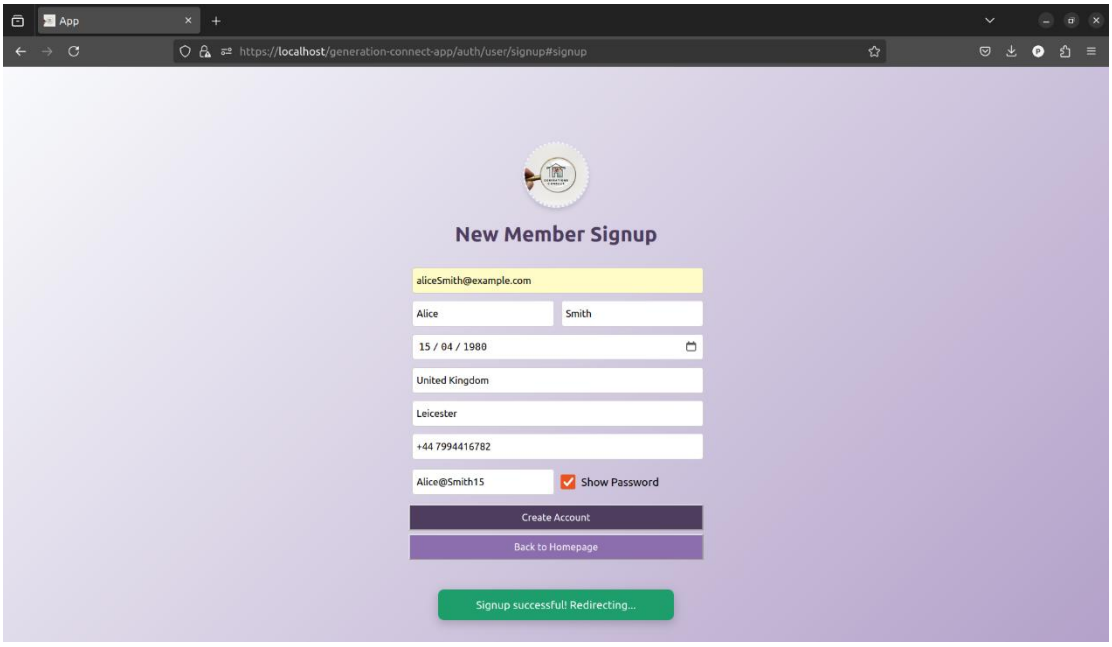


Figure 18 Successful Sign-Up Screen

Alice uses her credentials and Logs in her account at Generation Connect platform successfully. Otherwise, she gets the similar error message display as in **Figure 17**.

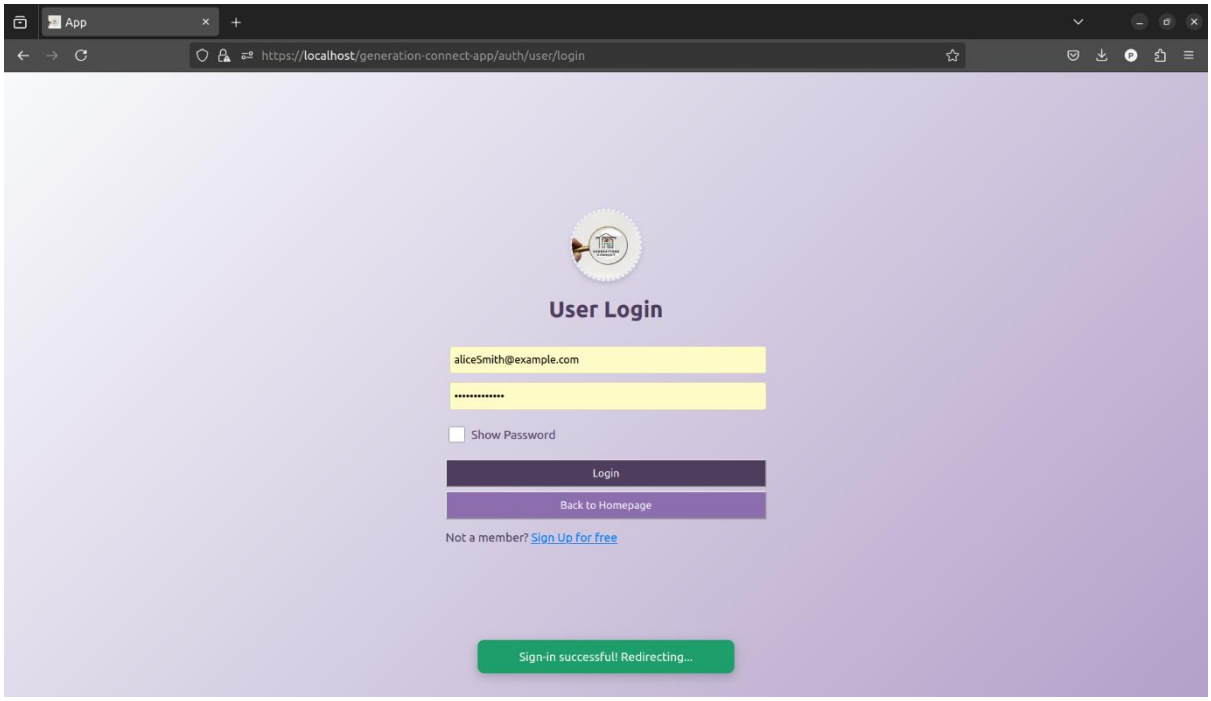


Figure 19 Login Page With Success

After logging in successfully, she lands on her Dashboard, where she see two tabs as Public Family Tree and Private Family Tree as shown below:

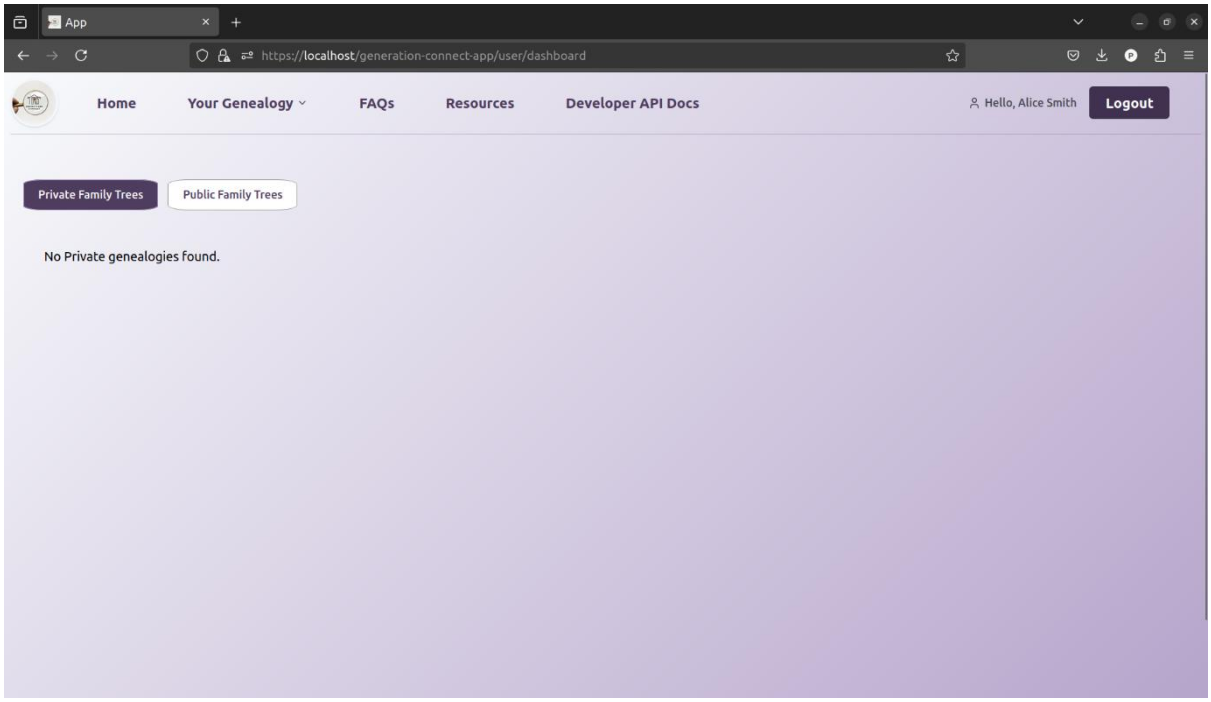


Figure 20 User dashboard after Successful Login

3. Exploring a Public Family Tree

Alice clicks the tab titled Public Family Trees. Curiously, she clicked on the House of Windsor Public Family Tree, created by johnDoe@example.com.

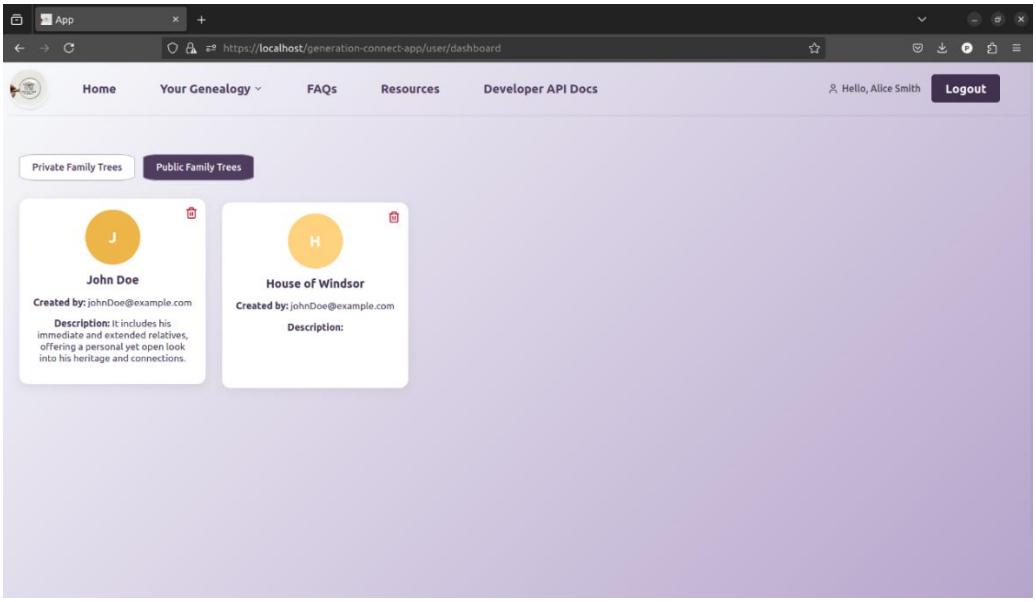


Figure 21 Active public Genealogy created by other Members of the Platform

The tree opened in the Genealogy Tree Page, visualizing Queen Elizabeth's family in a zoomable chart. Alice clicked on Elizabeth II's node to open the Profile View, where she saw a post created by John Doe containing a photo and short story.

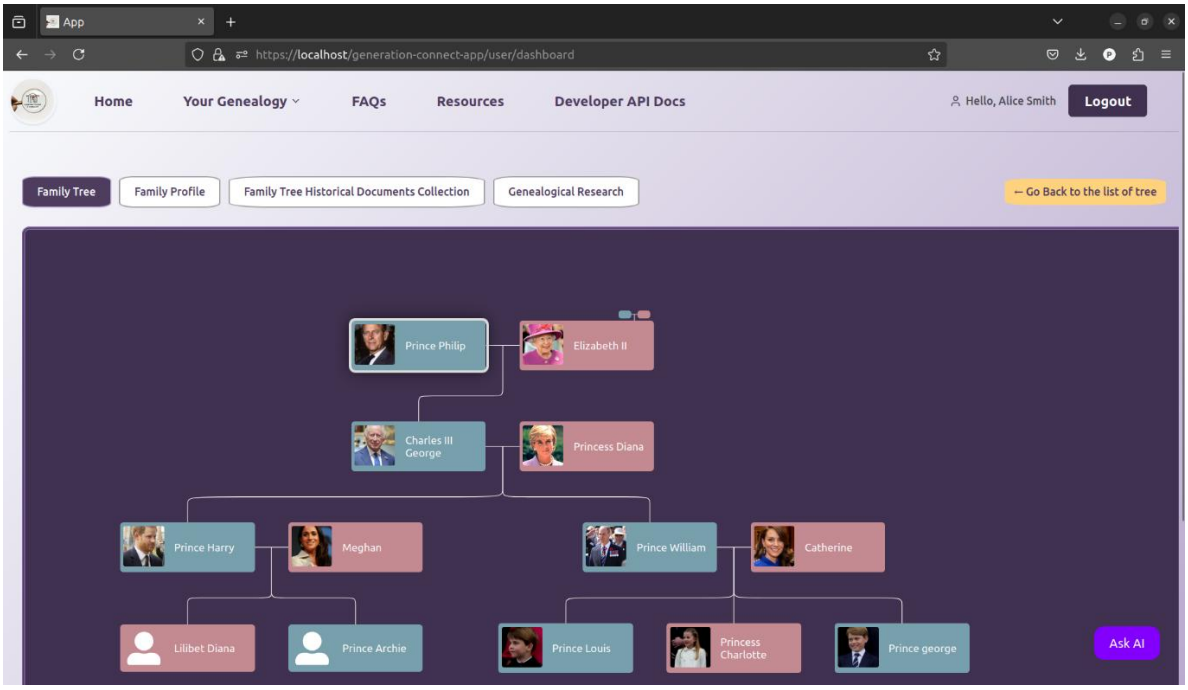


Figure 22 Public Family Tree Visualization

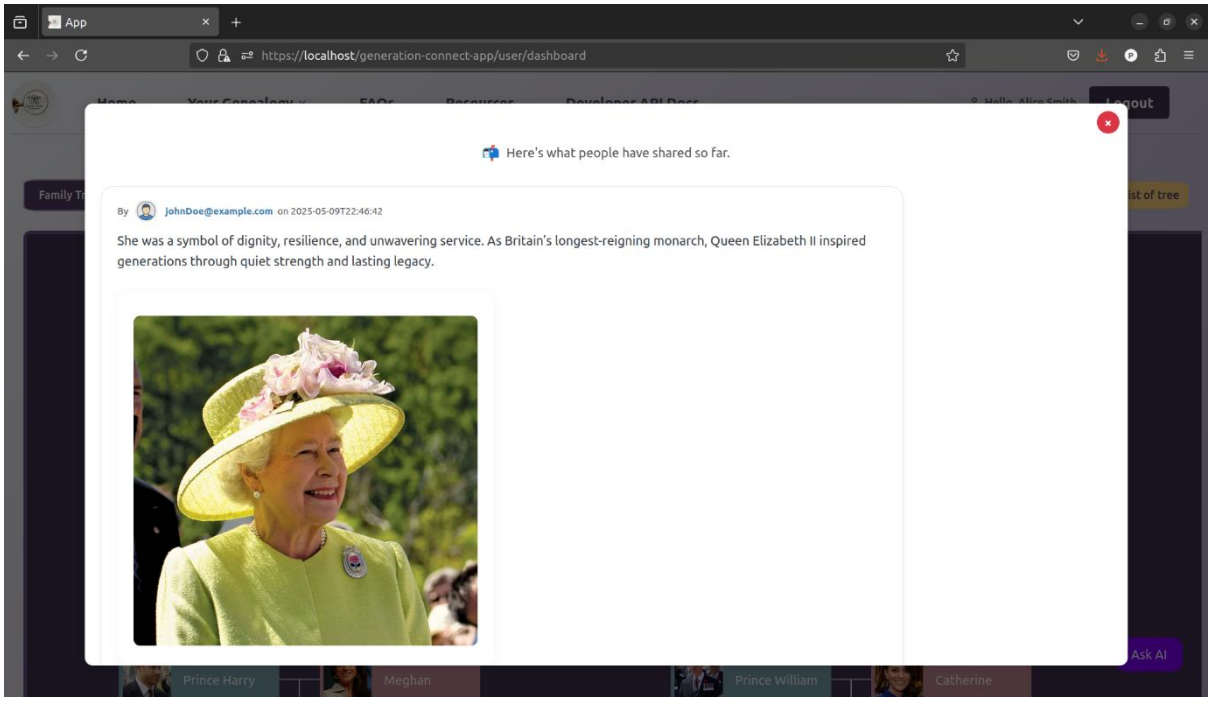


Figure 23 View Post for Queen Elizabeth II's Profile in the Genealogy Tree

She looks for a way to comment or contribute but found no editing options. That's when she realizes she has Viewer access only, which allows exploration but not contribution.

Inspired by this, Alice decided to create a tree for her own family.

4. Creating a Private Tree

Back in her dashboard, Alice clicks "Your Genealogy" -> "Create Family Tree." She selects the Private Tree option to explore the platform without sharing it publicly.

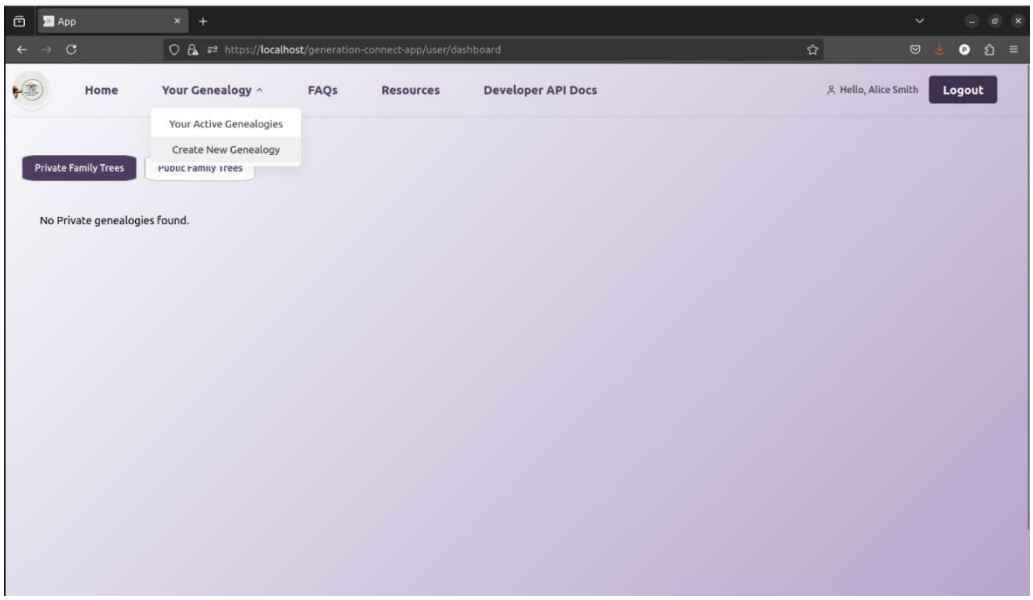


Figure 24 Click to Create New Genealogy

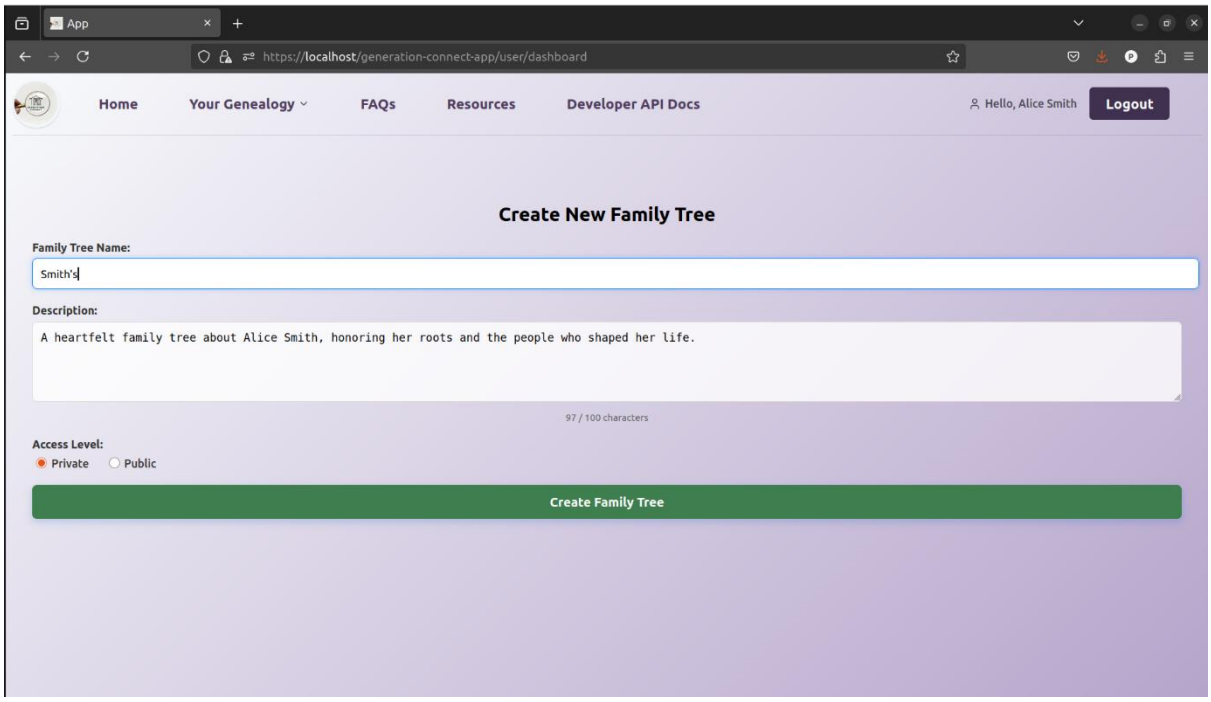


Figure 25 Alice creates a family tree and selects the "Private" option

The tree appears in her Tree List, and she opens it. Since it is newly created, it is empty. She begins by adding the first node - herself.

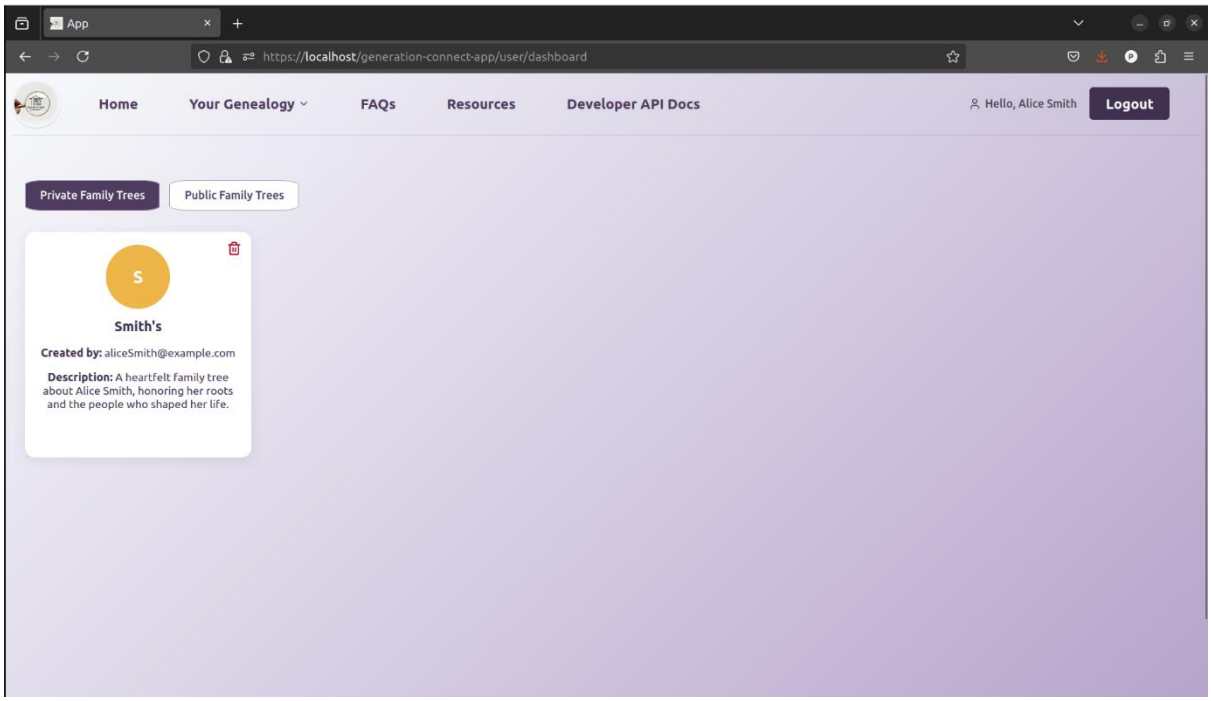


Figure 26 Alice successfully creates her first private family tree.

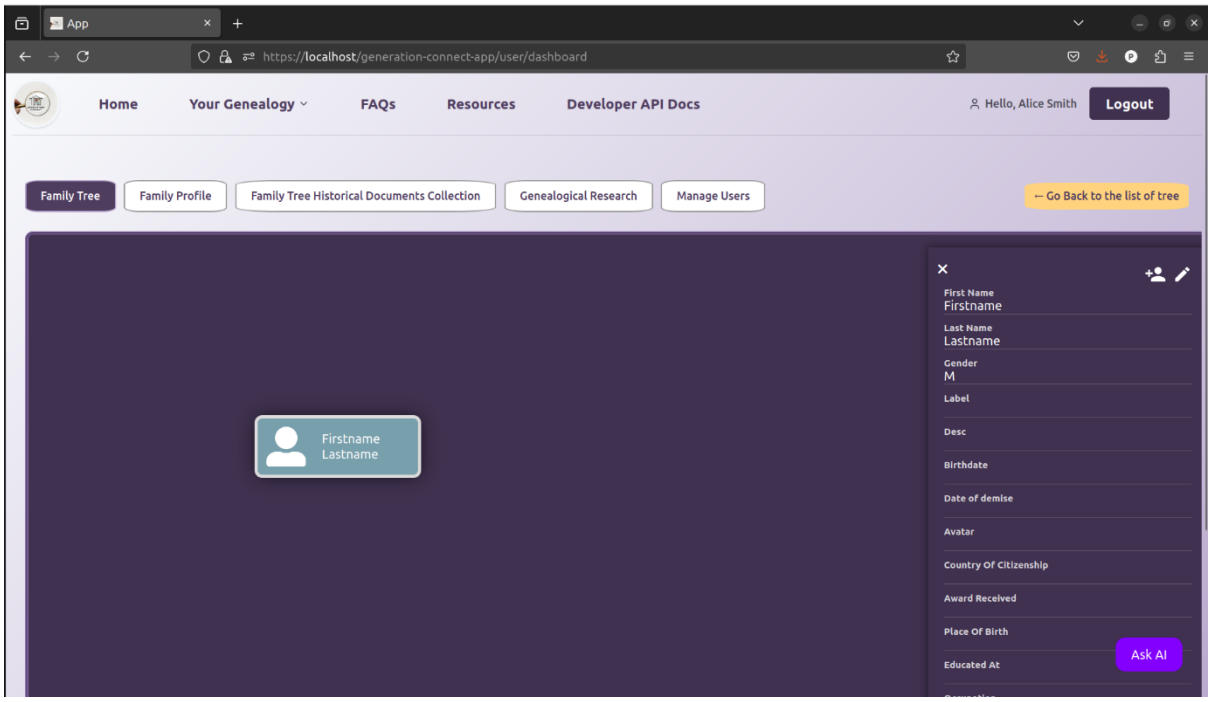


Figure 27 Empty Smith's Tree

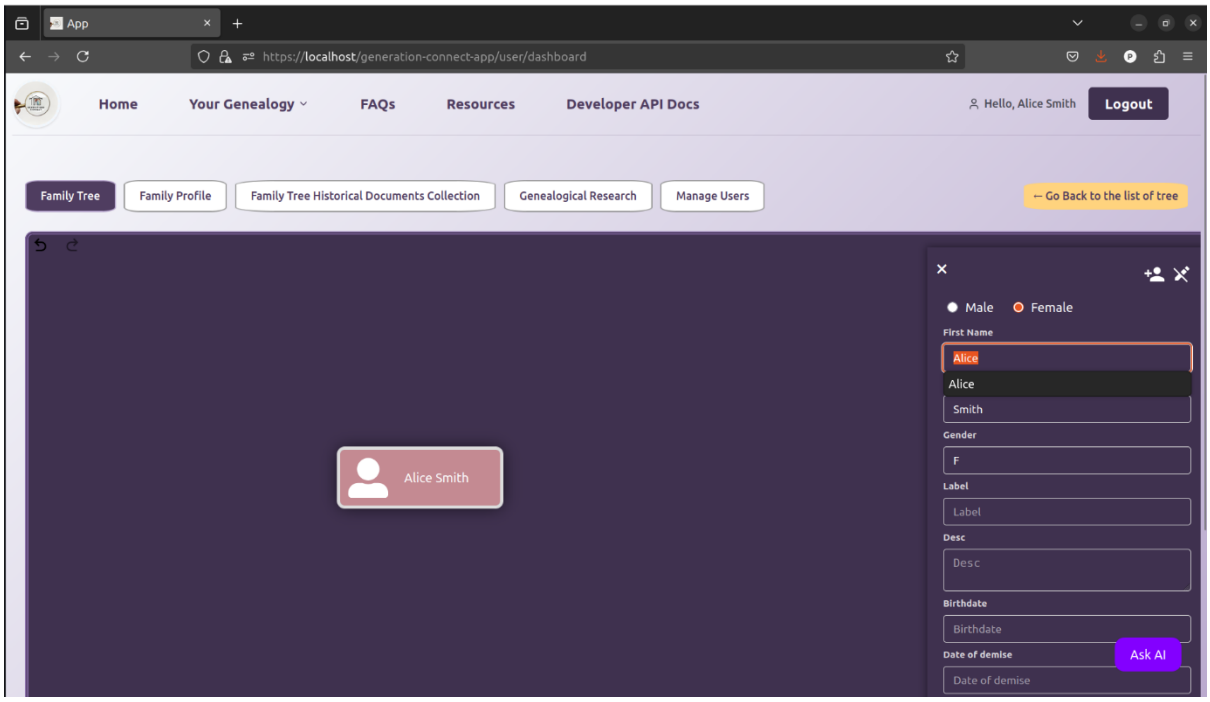


Figure 28 Alice Creates first node in Smith's Tree

5. Inviting a Family Member to Collaborate

Wanting help, Alice visits the Manage Users section. She sees her own email listed as the Owner, and below that, an option to add other users. She enters her daughter Carol's email address, selects the Contributor role, and clicks "Send Invite."

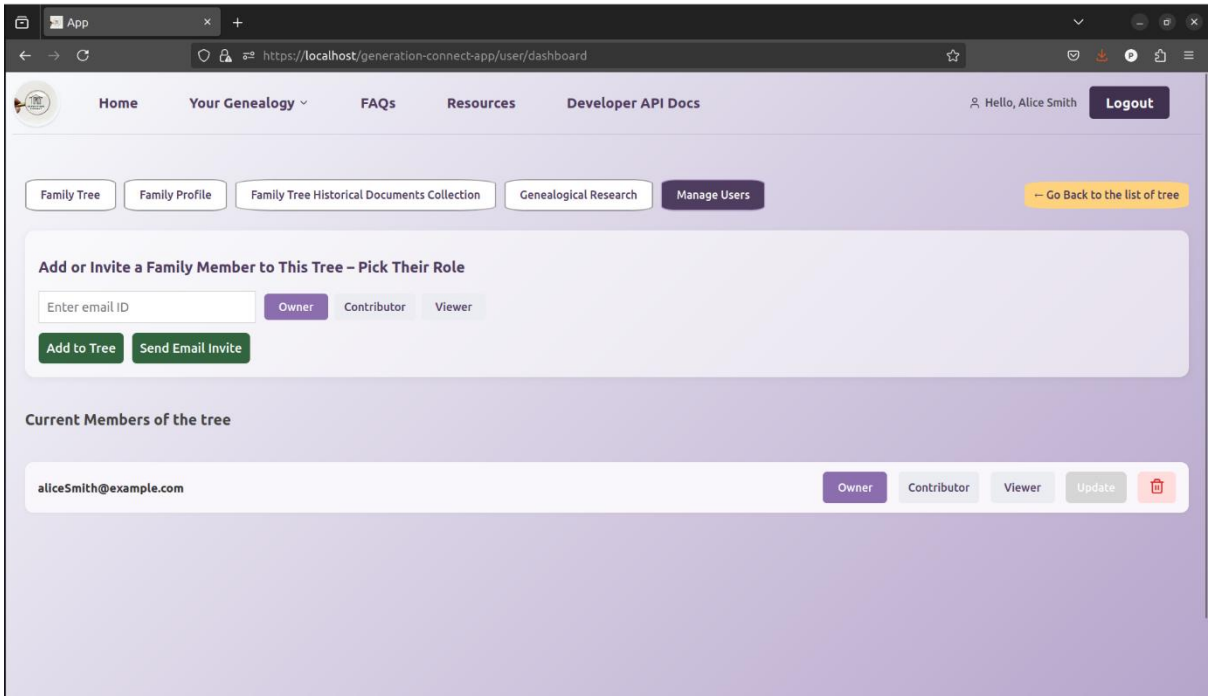


Figure 29 Manage Users tab for Smith's Tree

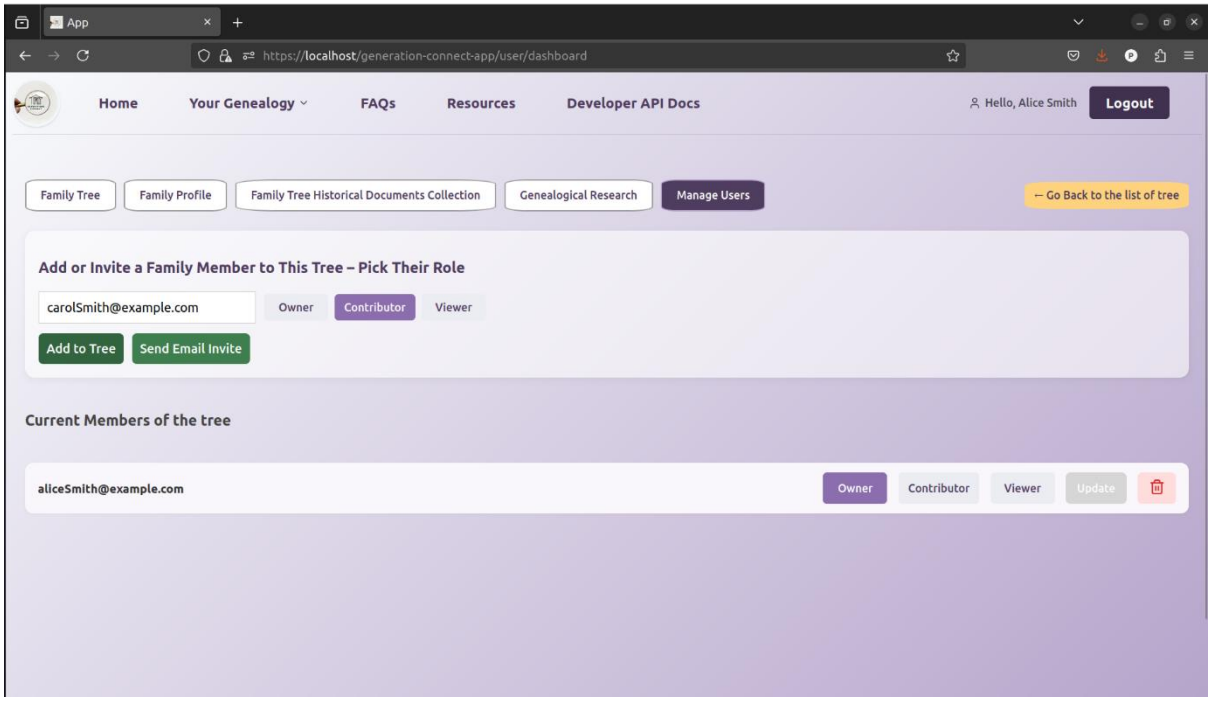


Figure 30 Alice invites Carol to Join the Smith's Tree

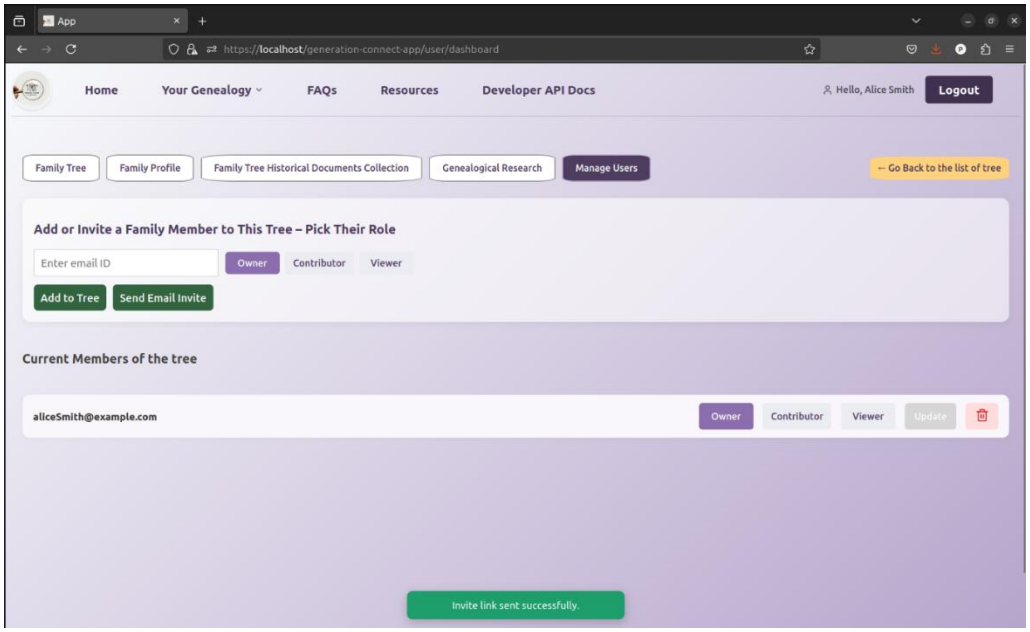
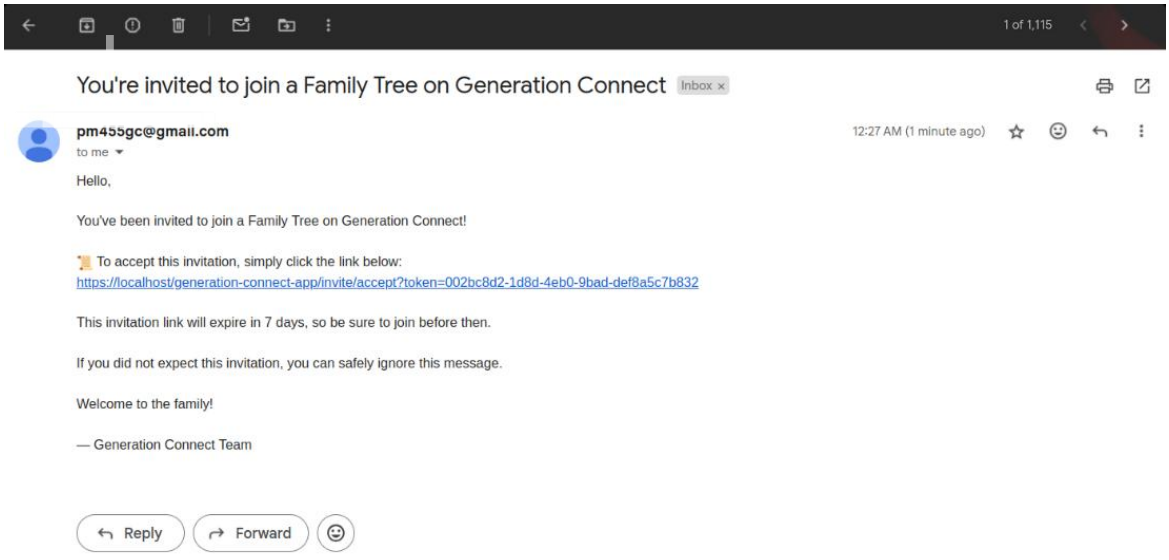
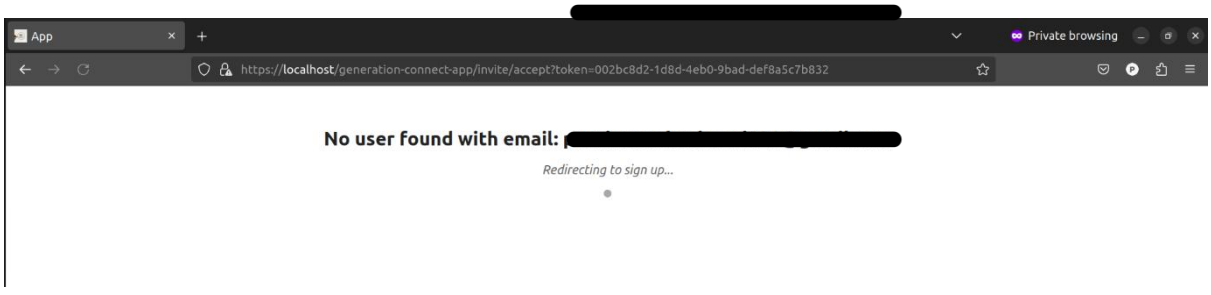


Figure 31 Invite Sent to Carol to join the Smith's Family tree

6. Accepts the Invitation and Joins the Tree

Carol receives the invitation via email. When she clicks on the invite link, the system detects that she has never used the application before and redirects her to the Sign-Up Page.





Note: *The application uses Google SMTP, so email verification is required. Placeholder addresses like @example.com are used for demonstration. For consistency in the story, I sent the invite to my personal Gmail and manually updated the database. During testing, invites from pm455gc@gmail.com to @student.le.ac.uk addresses were often marked as spam or not delivered. It's recommended to use a personal Gmail account for reliable invite testing.*

She completes the registration and logs in. After logging in, she re-clicks the invite link, which takes her to the Invite Accept Page.

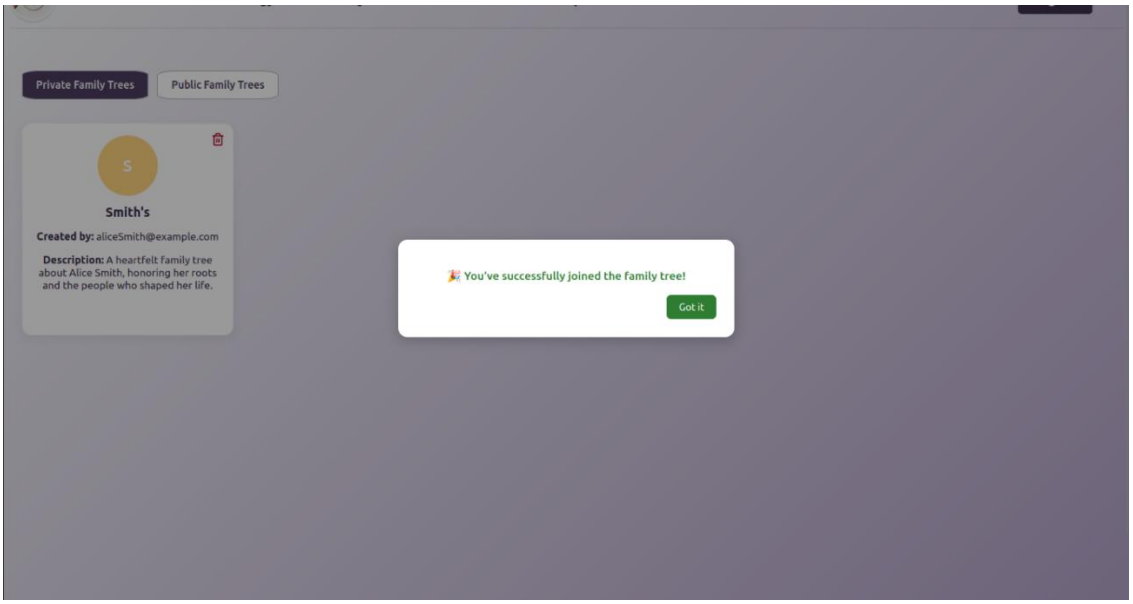


Figure 32 Accept Invite to Join the Tree Successfully

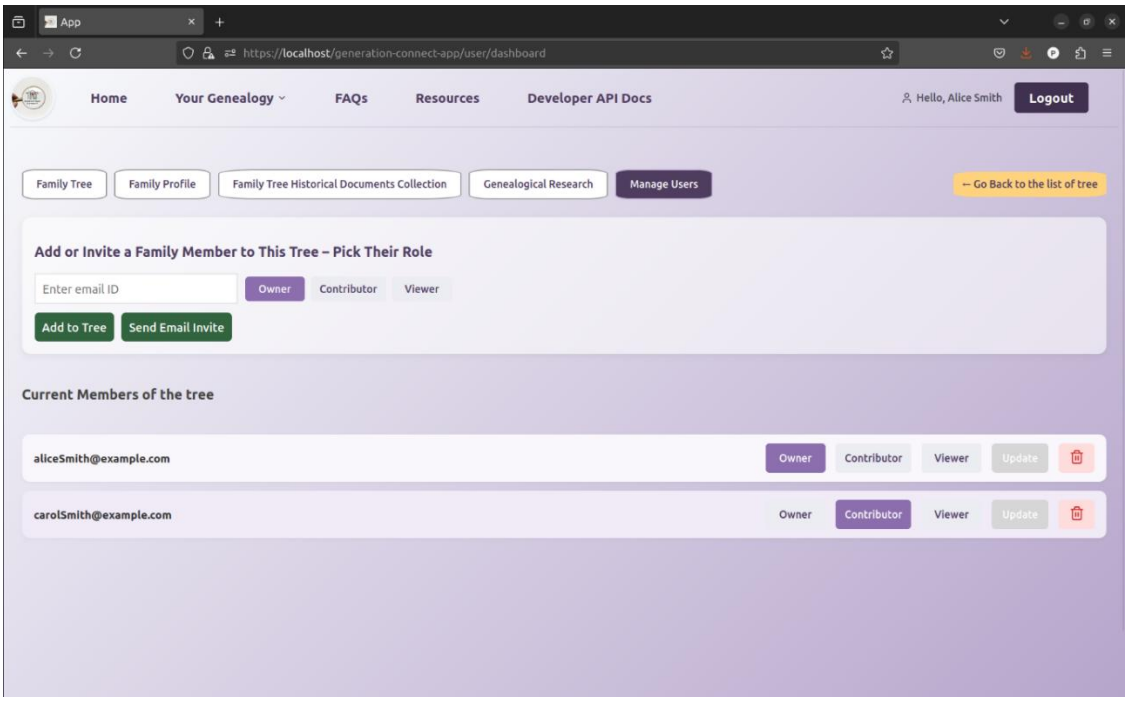


Figure 33 Carol appears as the current user on Alice's screen.

Once accepted, Carol sees Alice's family tree listed on her dashboard. She opens it to view the genealogy chart and confirms she now has Contributor access.

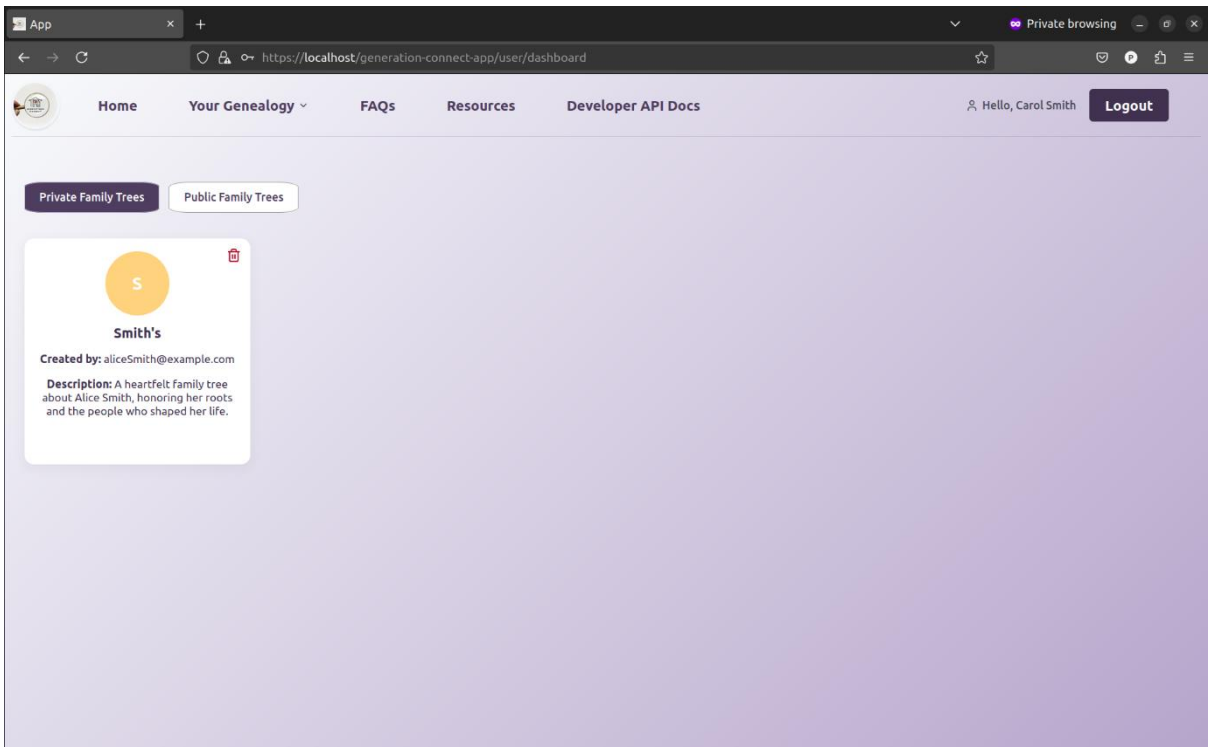


Figure 34 Carol Dashboard has Smith's Family Tree

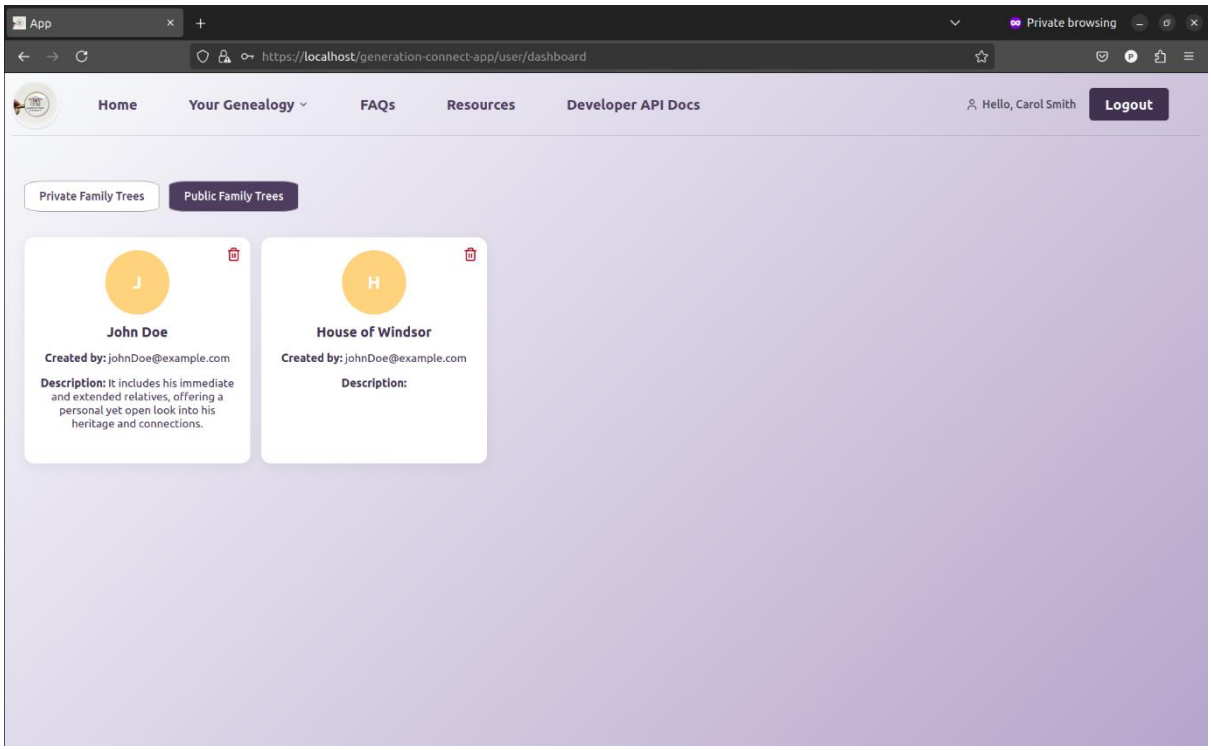


Figure 35 Carol could see available Public Family Tree

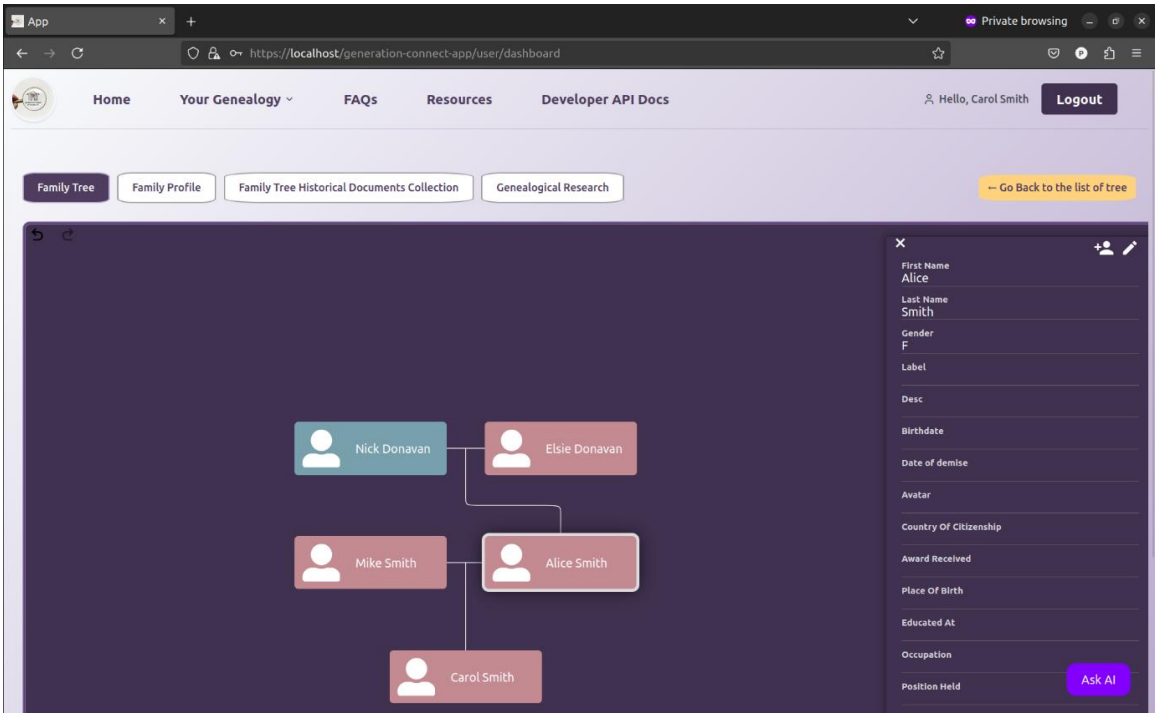


Figure 36 Carol's screen displays the Smith's Family Tree.

7. Adds Posts and Uploads Media

Carol clicks on her grandfather's profile picture in the tree. This opens his Profile View, where she creates a new post—adding a short story and uploading a photo using the Post Component. The Media Service saves the file and displays it in the profile's timeline.

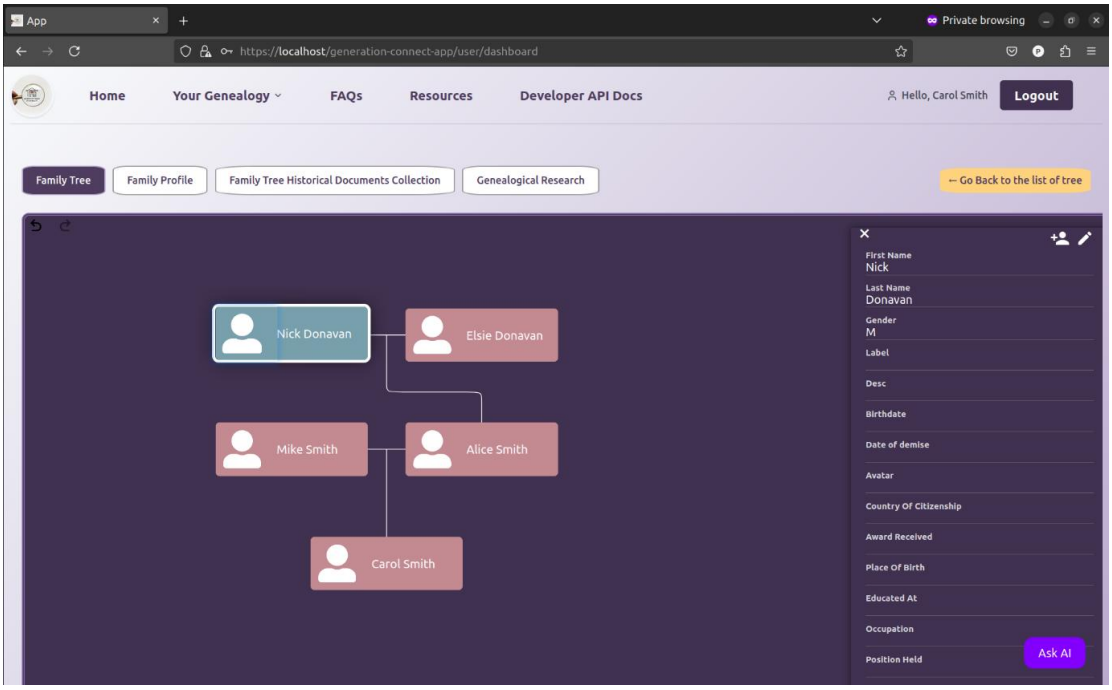


Figure 37 Carol clicks on Nick Donavan's profile to add a post for her grandfather

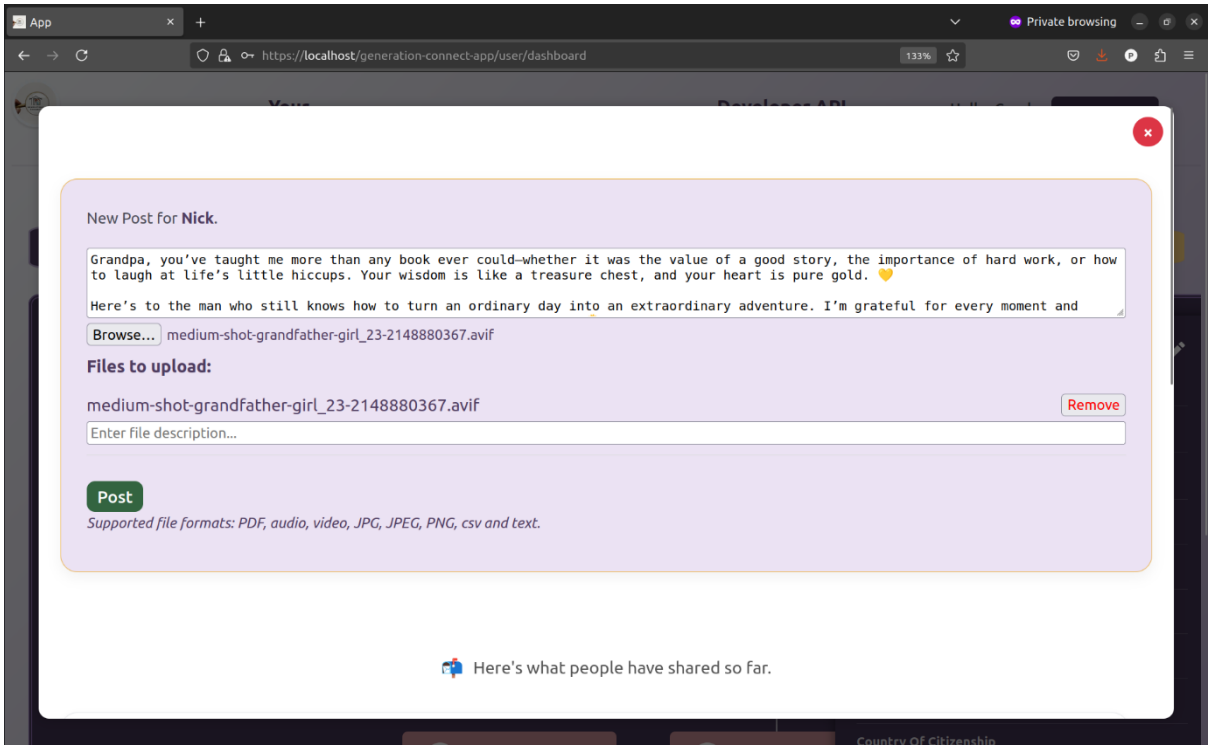


Figure 38 Carol adds text and a photo with her grandfather to the post.

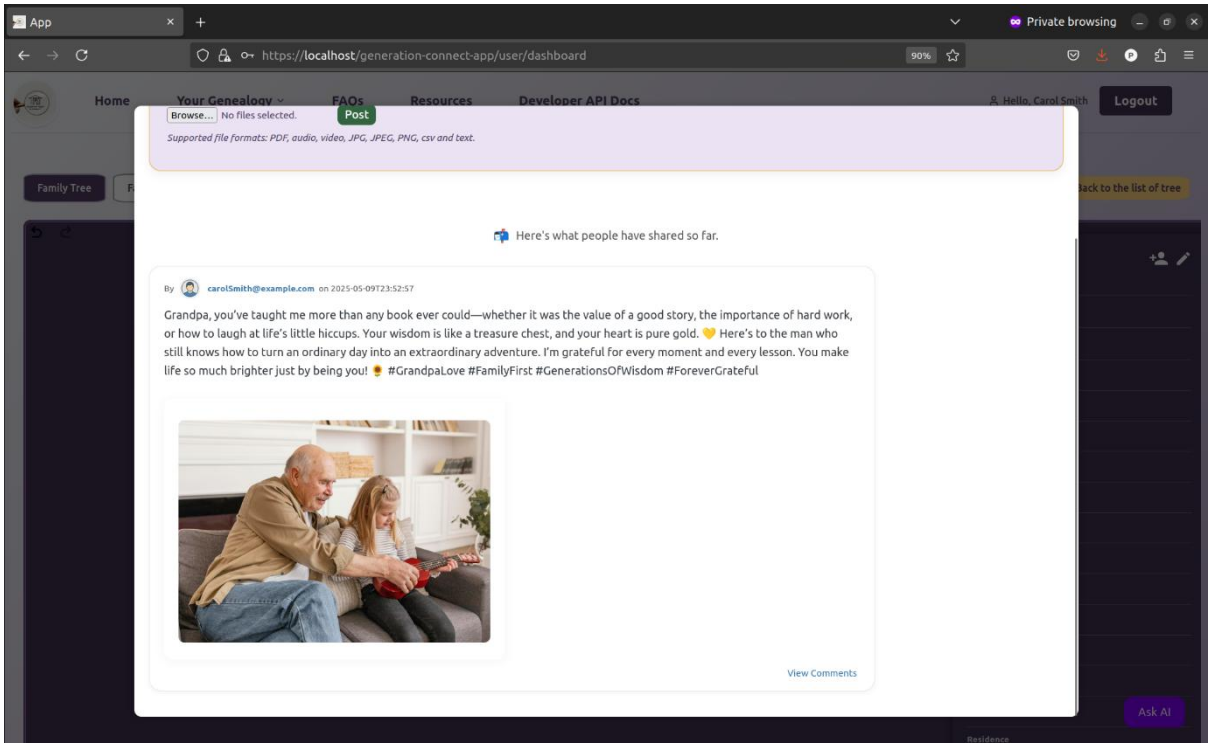


Figure 39 Carol's post is successfully published.

8. Joins and Tags Historical Records

Alice also adds her grandson, Dave, a history enthusiast, as a Contributor.

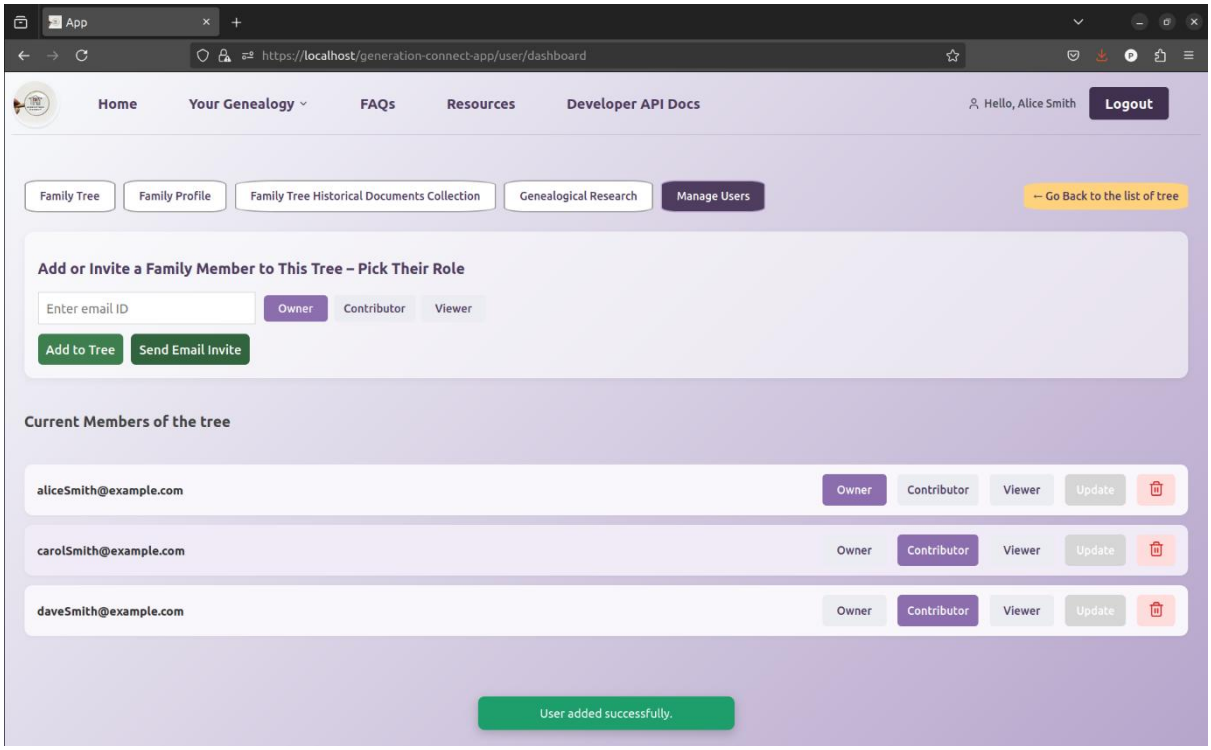


Figure 40 Alice adds an existing platform member to the tree.

Dave accesses the Genealogy Research Page. He searches for a family member's name, finds a birth record, and tags it to the family tree using the Tag Record Component. The document now appears under that person's Tagged Records section.

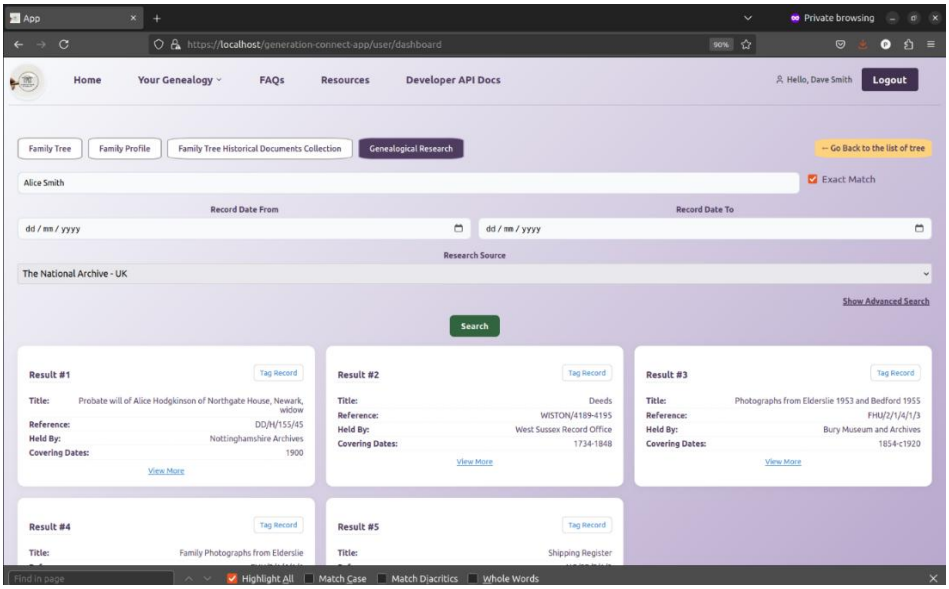


Figure 41 Dave navigates to the Genealogical Research tab to search for 'Alice Smith'

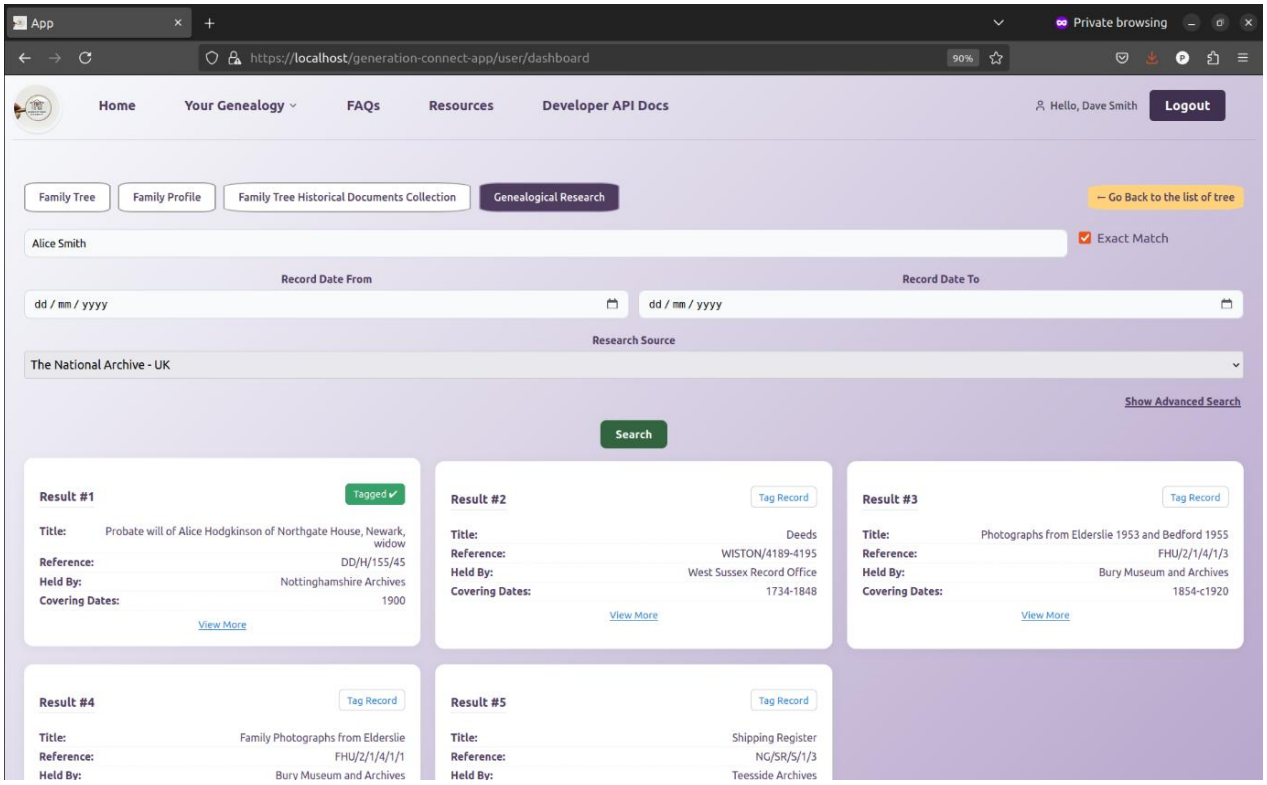


Figure 42 Dave tags the record to Family Historical Document Collection

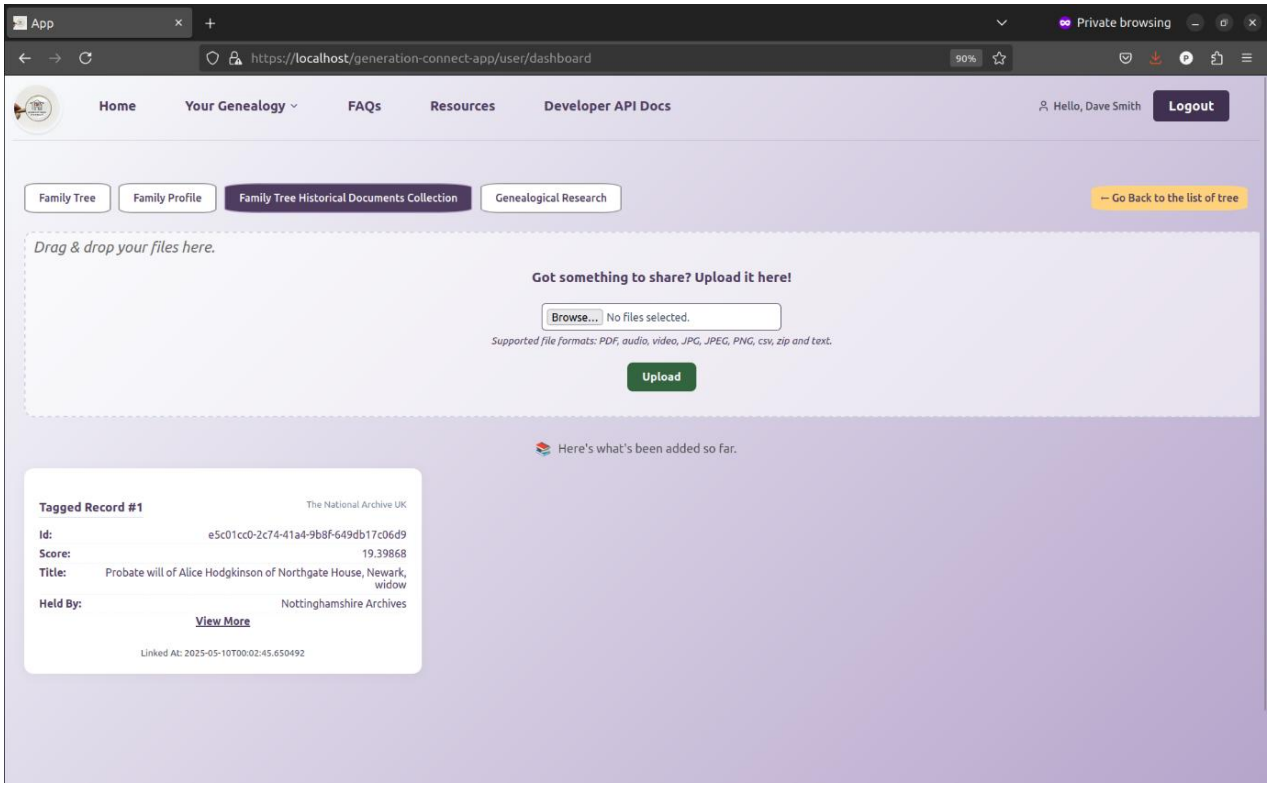


Figure 43 Dave finds the tagged record listed on the Family Tree Historical Document Collection screen and can also upload other media files here.

9. Tracking Contributions with Recent Actions

To stay updated on activity, Alice visits the Recent Actions view. This feature shows who adds posts, tags documents, or modifies profiles - helping her monitor contributions without navigating the entire tree.

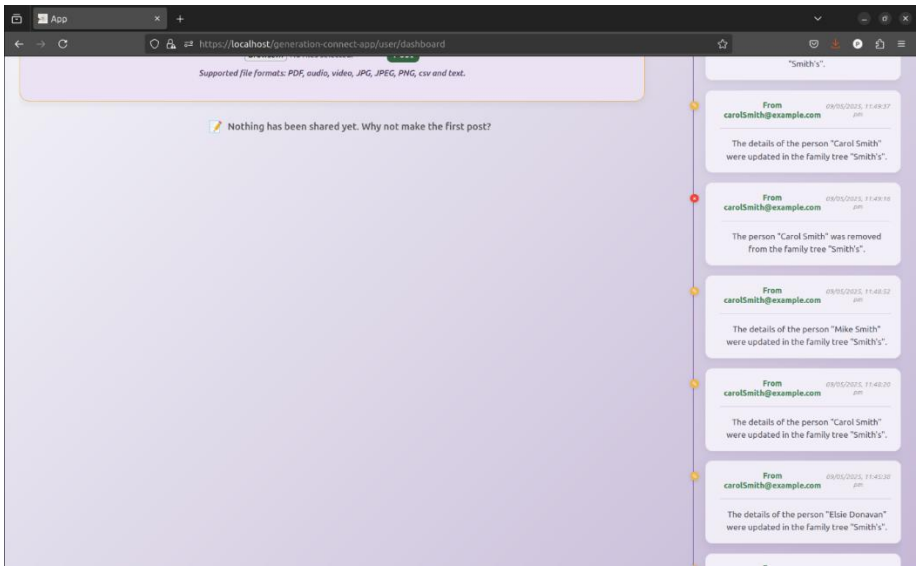


Figure 44 Smith's Family tree Recent Actions at Alice Screen

10. Managing Access and Creating a Public Tree

Confident in using the platform, Alice adds her friend Grace Williams to her private tree through the Manage Users tab with Viewer access. Grace can now explore the tree, while only Contributors and the Owner (Alice, Carol, and Dave) have editing rights.

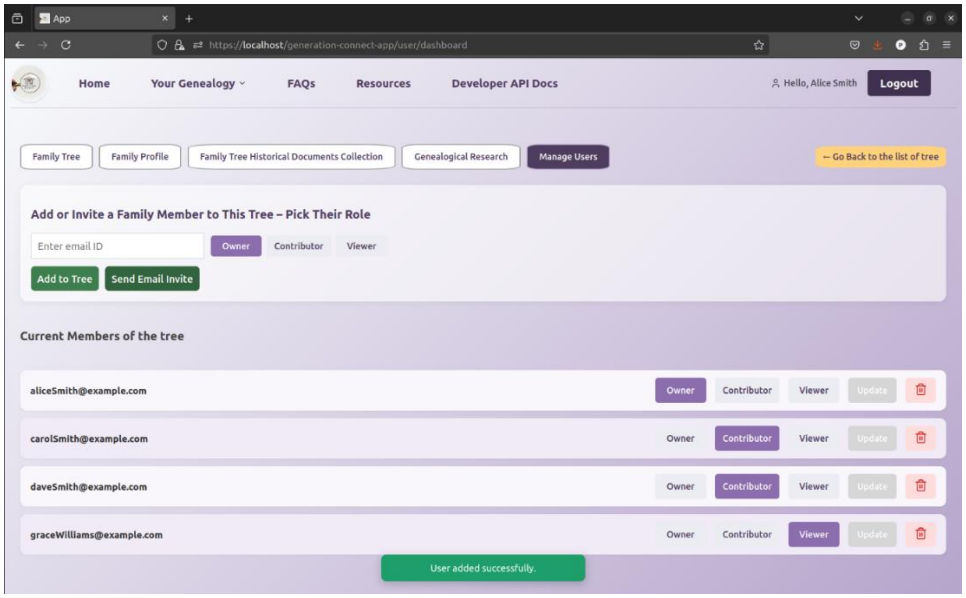


Figure 45 Alice adds Grace, her friend as Viewer to Smith's Tree

Alice also creates a second Private Tree for her husband's family. She adds her granddaughter Emily as a Viewer, and Dave with Contributor access.

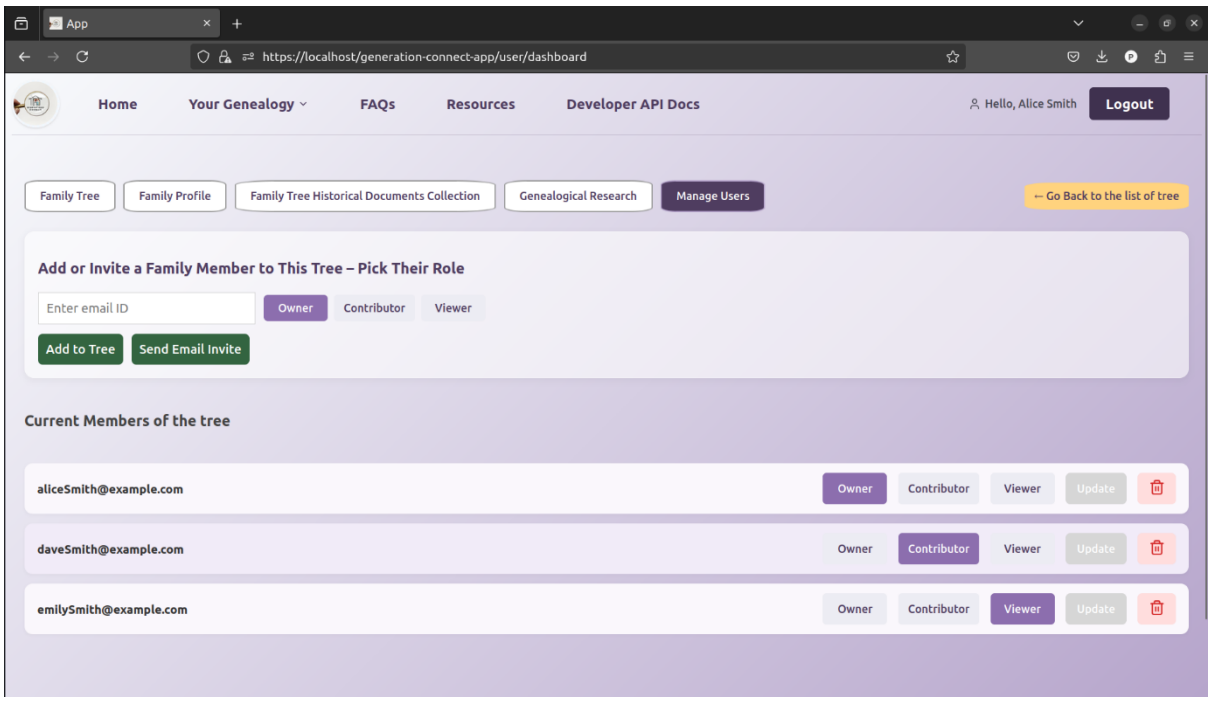


Figure 46 Alice's husband's Family tree Manage Users

While exploring more public trees, Alice finds an "Ask AI" button on the Family Tree page. Curious, she types a question related to that tree and receives a helpful response instantly.

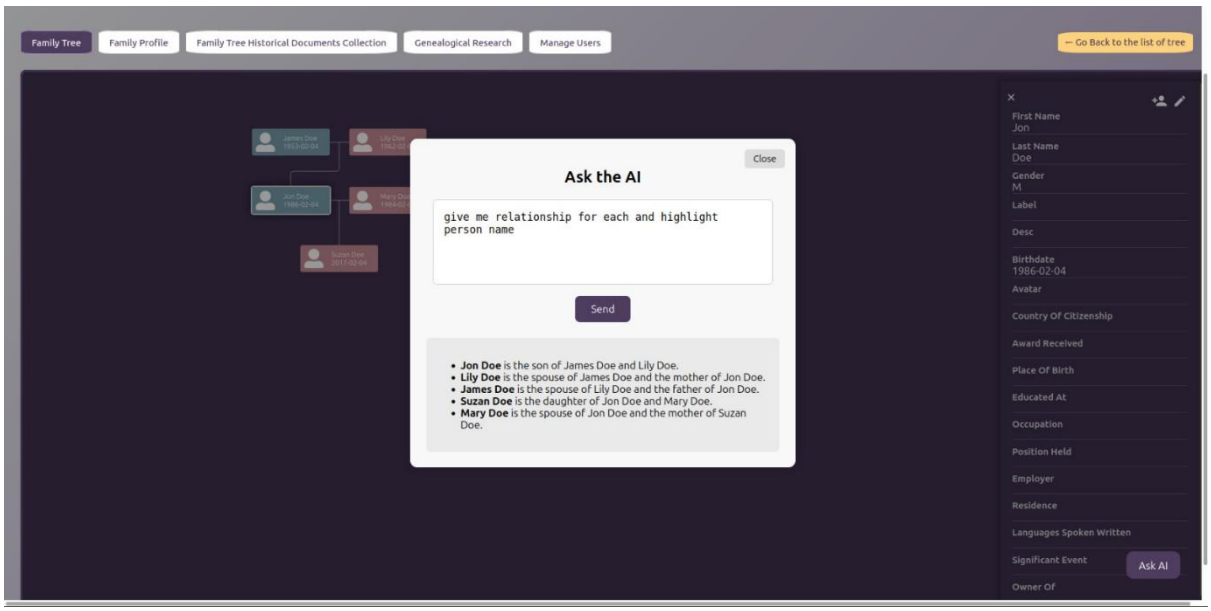


Figure 47 Alice queries uses Ask AI button to an unknown available public tree

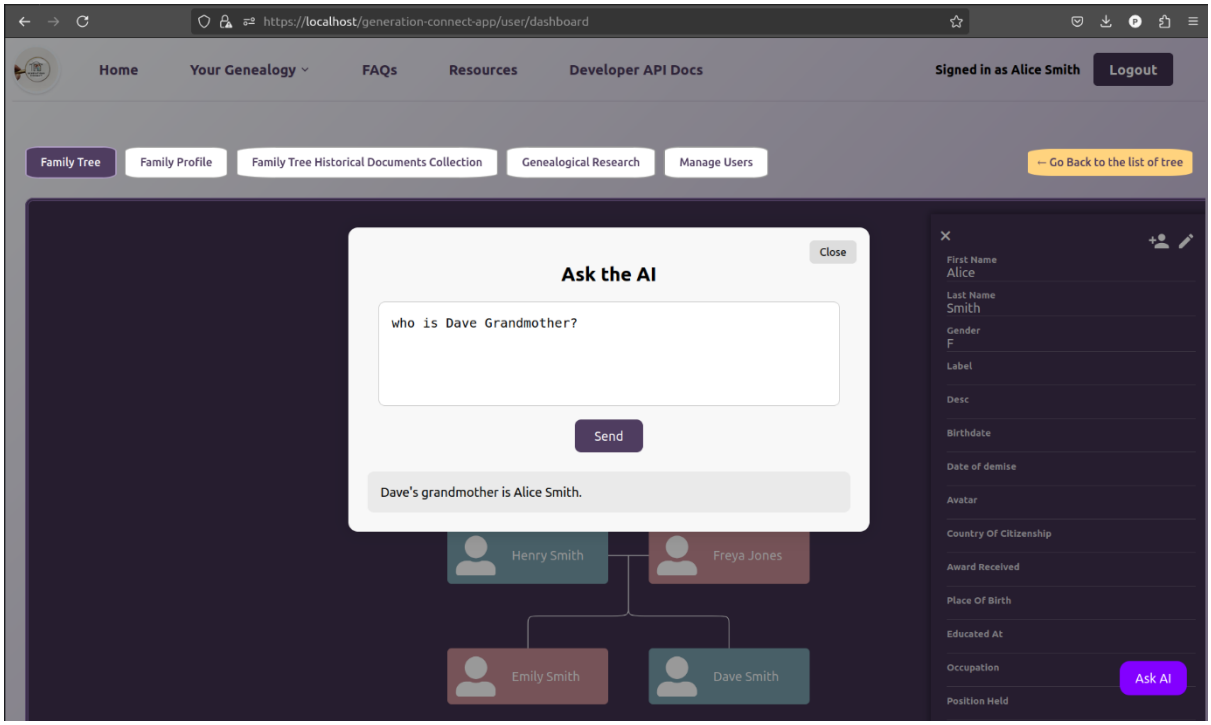


Figure 48 Alice use Ask AI button for Smith's Family tree

11.Admin Monitoring via the Dashboard

While Alice and her family collaborate, the platform administrator logs into the Admin Dashboard.

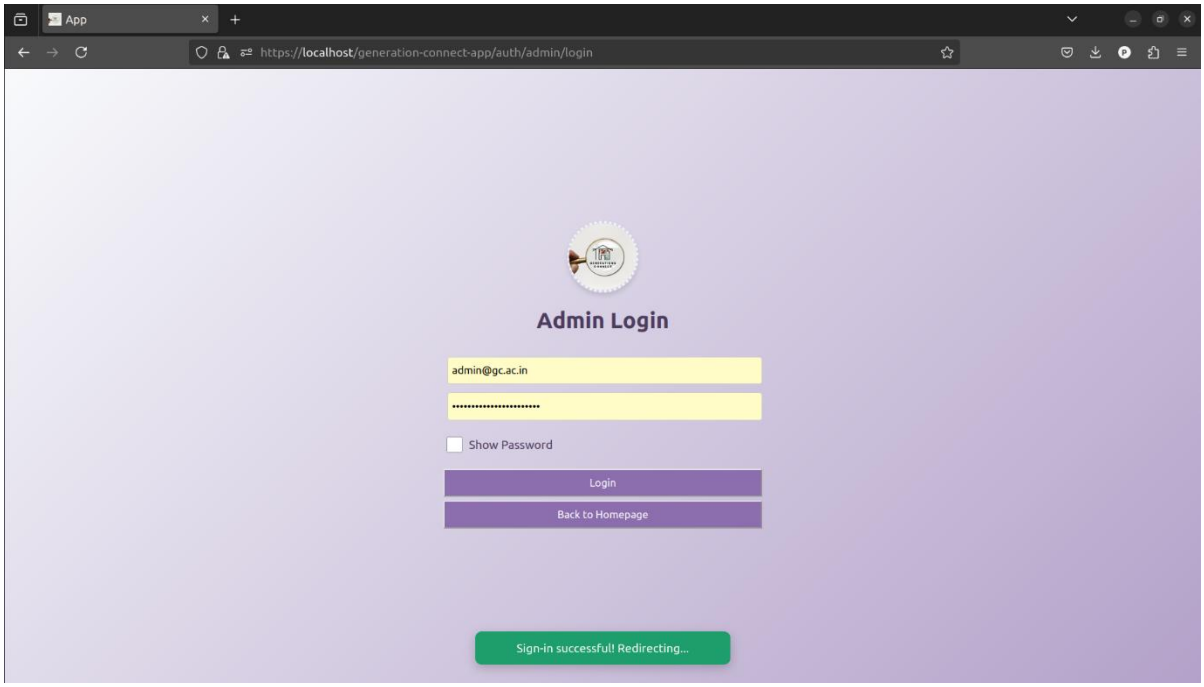


Figure 49 Admin's Successful Login

From here, the admin can monitor key metrics such as:

- Total number of family trees
- User activity logs
- Recent tree creation timestamps
- Lists of tagged records and profile posts

These tools ensure platform integrity, transparency, and smooth functionality.

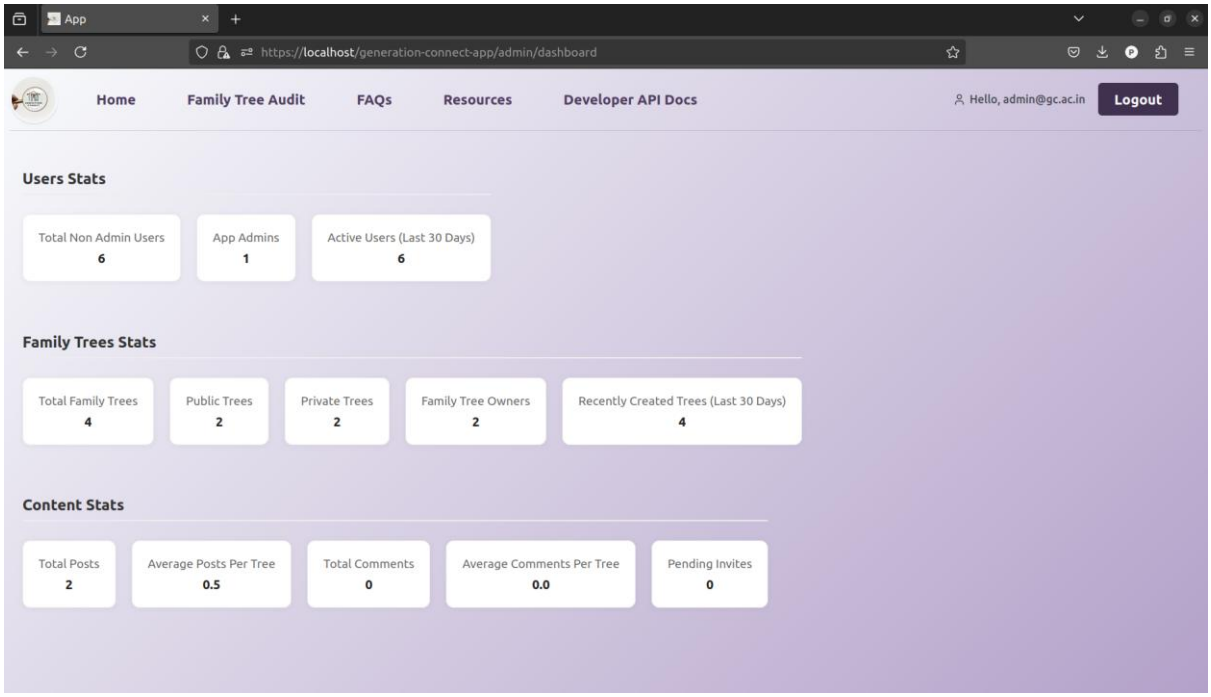


Figure 50 User Metrics Screen at Admin Portal

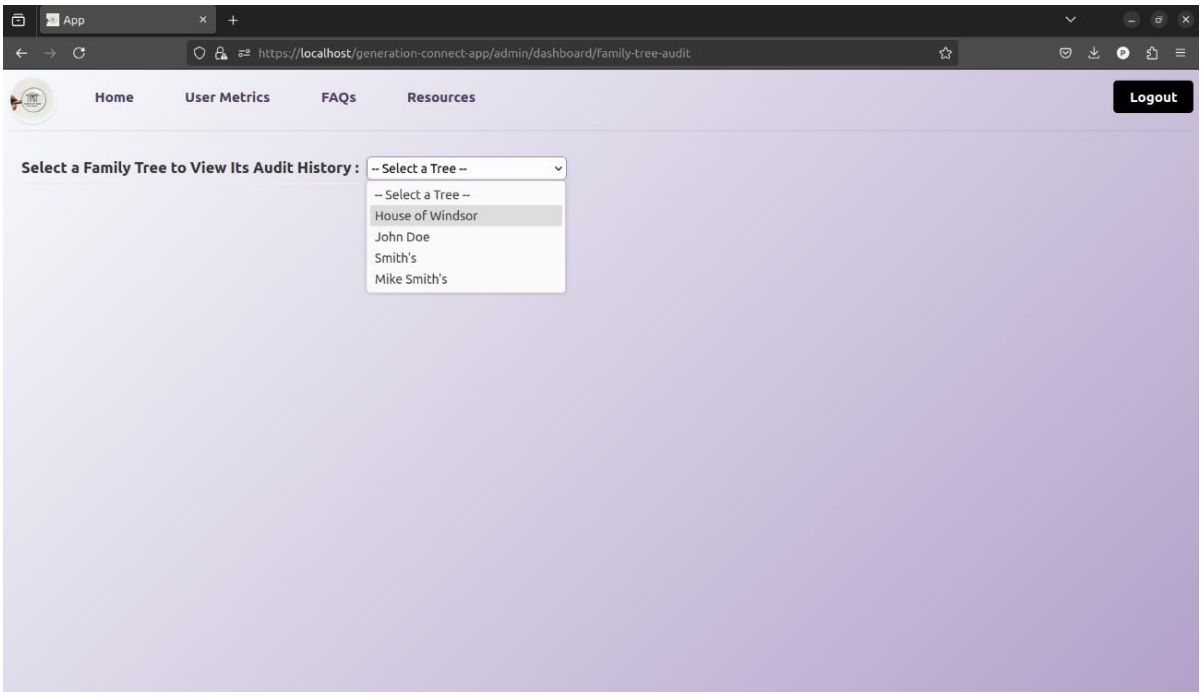


Figure 51 Admin to choose Family tree to see Audit

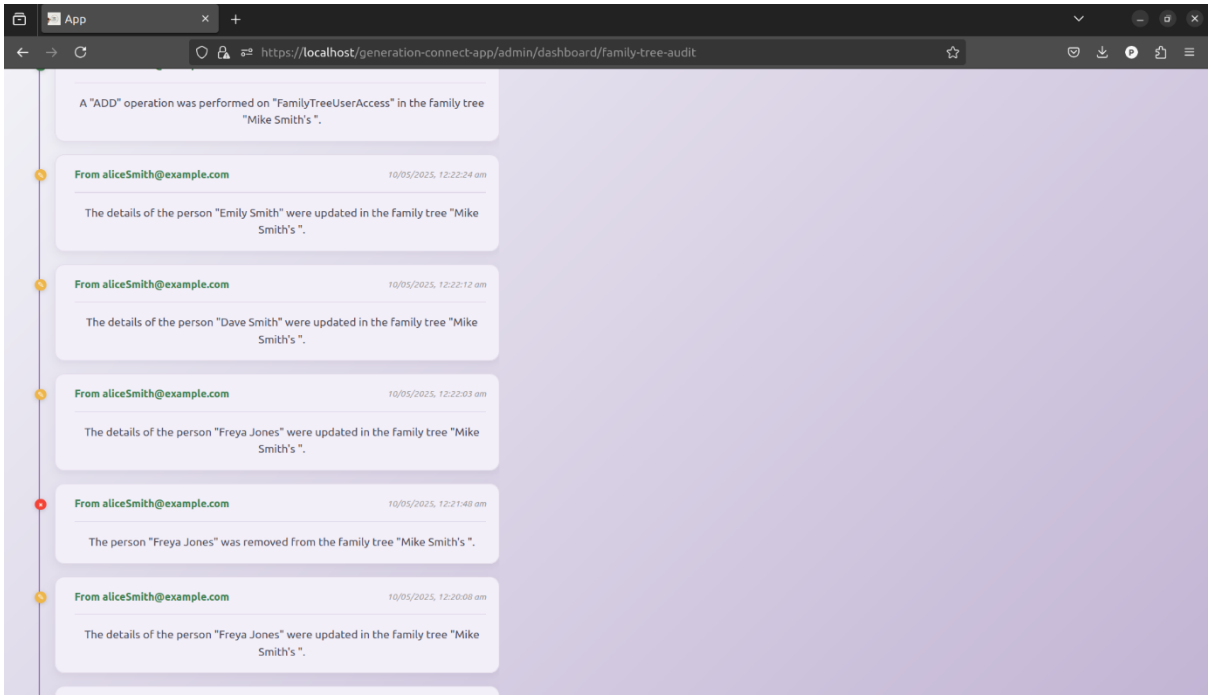


Figure 52 An audit list displays all user actions within the Smith's Family Tree

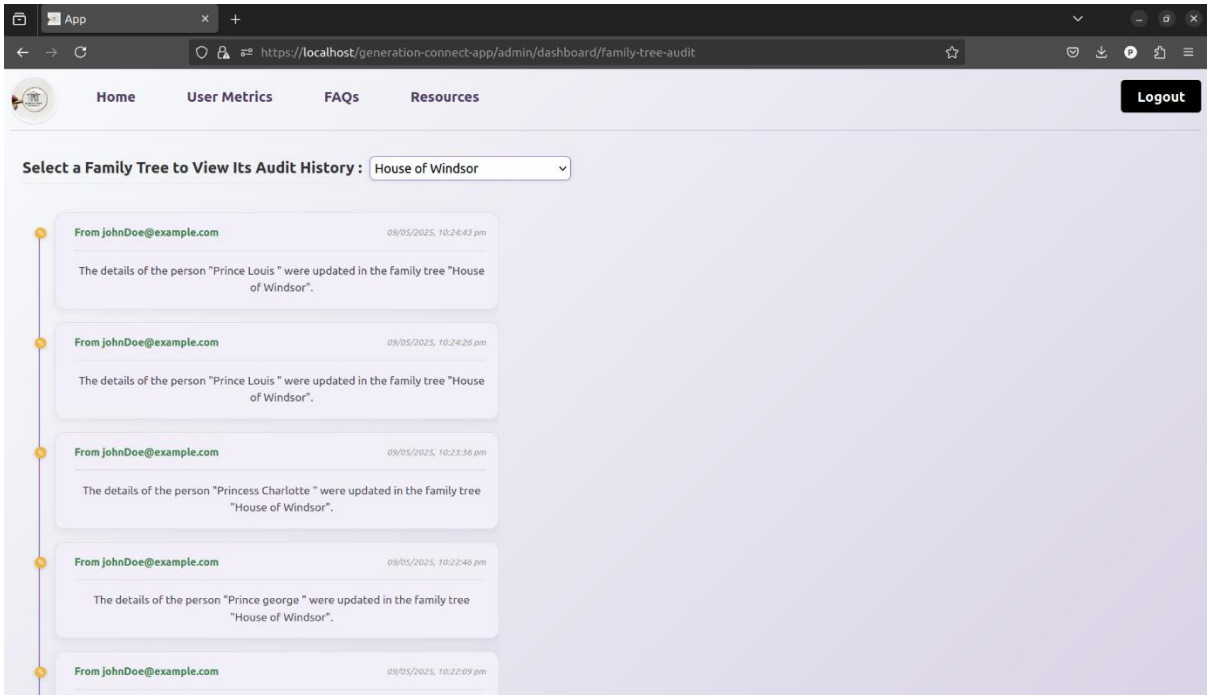


Figure 53 An audit list displays all user actions in House of Windsor Public Tree

This ensures the platform is functioning smoothly and all data is traceable.

11

NGINX Gateway Configuration and Secure Routing

This chapter explains the configuration of NGINX as a secure gateway for the application. It includes setting up upstream routing for frontend and backend services, enabling HTTPS using SSL certificates and Diffie-Hellman parameters, handling CORS and preflight requests, and securing routes with JWT extraction and role-based access validation using Lua scripts.

11. NGINX Gateway Configuration and Secure Routing

Based on the background study in *Section 5.2* usage and the proposed architecture in *Section 6.3* to route frontend and backend traffic, this section outlines the actual implementation used to enforce authentication, manage role-based access via Lua scripting with JWT validation, apply CORS policies, and secure communication using HTTPS and SSL certificates.

11.1 Upstream Server Configuration

To route traffic between the frontend and backend services running in Docker containers, two upstream blocks are defined - app for the frontend and api for the backend. These upstreams serve as named groups of backend servers that NGINX can proxy requests to.

The upstream configuration is placed inside the http context, which is required for handling HTTP traffic. This context allows upstream to be referenced from any server block within the configuration, supporting a modular and organized setup.

The ip_hash directive is used to ensure session stickiness by routing requests from the same client IP to the same backend container. This helps maintain consistency for users interacting with stateful services.

```
http {
    client_max_body_size 50M;
    upstream app {
        ip_hash;
        server app:3000;
    }

    upstream api {
        ip_hash;
        server api:8080;
    }

    server {
          listen 80;//before enabling https
    }

    # server blocks follow

}
```

This configuration enables reliable, consistent routing of frontend and backend traffic while supporting scalability and session-aware behaviour.

11.2 Enabling SSL/TLS (HTTPS)

Initially, the services serve over plain HTTP. This setup is not secure for production or even local development when dealing with authentication tokens or personal data. HTTPS encrypts all data exchanged between client and server, ensures cookie and token confidentiality, and aligns the platform with modern browser security standards.

To enable HTTPS, the upstream server configuration was updated by introducing an additional server block that redirects all HTTP requests (http://localhost) to the secure HTTPS version (https://localhost):

```
http {
    client_max_body_size 50M;
    upstream app {
        ip_hash;
        server app:3000;
    }

    upstream api {
        ip_hash;
        server api:8080;
    }

    server {
          listen 80;
          server_name localhost;
          return 301 https://localhost; #Redirect HTTP to HTTPS
    }

    #server blocks follow
}
```

11.2.1 Self-Signed Certificate Generation

In a production environment, HTTPS (SSL/TLS) is usually secured using certificates issued by trusted Certificate Authorities (CAs). However, during local development, obtaining and renewing CA-signed certificates is impractical. Instead, a self-signed certificate provides a convenient way to enable HTTPS locally. It allows the to:

1. Encrypt traffic between browser and server to simulate real-world HTTPS conditions.
2. Prevent sensitive data exposure, such as JWT tokens, credentials, or cookies, over plain HTTP.
3. Test secure browser features like Secure cookies or CORS preflights, which only function over HTTPS.
4. Prepare for a seamless transition to production, where a real certificate from a CA will be used.

The command used with openssl creates:

- A private key that is used to encrypt/decrypt traffic.
- A certificate that browsers can use to identify the server (though browsers will warn it's not trusted since it's self-signed).

```
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
  -keyout ./ssl/private/selfsigned.key \
  -out ./ssl/certs/selfsigned.crt
```

11.2.2 Generating Diffie-Hellman Parameters

Diffie-Hellman parameters are used to enable Forward Secrecy (FS) — a critical security feature in modern HTTPS configurations.

With forward secrecy:

- Even if the server's private key is compromised in the future, past encrypted sessions remain secure.
- A unique encryption key is generated per session, preventing retrospective decryption by attackers.

By generating own DH parameters with:

```
sudo openssl dhparam -out ./ssl/certs/dhparam.pem 2048
```

the system is:

- Enabling support for DHE cipher suites in system's TLS configuration.
- Ensuring that session keys are generated securely.
- Strengthening SSL configuration against known cryptographic attacks.

11.2.3 NGINX HTTPS Setup with SSL, Key and DH Parameters

The NGINX server block was updated to enable HTTPS on port 443 and to reference the previously generated SSL certificates and DH parameters:

```
server {
    listen 443 ssl; //Open secured port from :80 to :443
    server_name localhost;

    ssl_certificate     /etc/ssl/certs/selfsigned.crt;
    ssl_certificate_key /etc/ssl/private/selfsigned.key;
    ssl_dhparam         /etc/ssl/certs/dhparam.pem;

    # Other configuration like location blocks, CORS, proxy_pass, etc.
}
```

This setup ensures that all communication is securely encrypted using SSL/TLS and that the system can be safely accessed during local development without exposing sensitive data over plain HTTP.

11.3 CORS Headers and Preflight Handling

To support secure and seamless communication between the frontend and backend services across different origins, CORS headers are included across all relevant location blocks. This ensures that browser-based requests are permitted to access backend resources, even when served from a different domain or port.

The following headers are configured:

```
add_header 'Access-Control-Allow-Origin' '*';
add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS, PUT, DELETE';
add_header 'Access-Control-Allow-Headers' 'Origin, Content-Type, Accept, Authorization';
add_header 'Access-Control-Allow-Credentials' 'true';
```

These headers allow various HTTP methods, common request headers, and credentials like cookies or authorization tokens. They are applied to endpoints under `/generation-connect-app`, `/generation-connect-api`, `/auth`, and even the root / path.

In addition, all relevant paths handle OPTIONS preflight requests, which browsers send before executing certain cross-origin requests. Each location block contains a conditional check for the OPTIONS method and responds with a 204 No Content status, indicating that the request is allowed without returning a full response.

This CORS handling approach ensures that the application supports modern browser security standards while maintaining a smooth user experience across frontend and backend services.

11.4 Proxy Pass Directives

To route incoming client requests to the appropriate internal services, `proxy_pass` directives are used within the corresponding location blocks. These directives ensure that NGINX acts as a reverse proxy, forwarding HTTP requests to the correct upstream service based on the request path.

Two upstream services have been defined:

`http://app` for handling frontend traffic

`http://api` for backend API requests

Requests to `/generation-connect-app` paths are forwarded to the frontend container (`app:3000`), while requests to `/generation-connect-api` paths are forwarded to the backend container (`api:8080`). This routing is handled transparently by NGINX within Docker.

```
location ^~ /generation-connect-app {
    proxy_pass http://app;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
}
location ^~ /generation-connect-api {
    proxy_pass http://api;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;}
```

11.5 JWT Extraction and Role Access Validation via Lua

As part of the application gateway configuration, Lua scripting has been integrated within NGINX to perform centralized authentication and role-based access control for both frontend and backend services.

The objective of this configuration is to:

- Authenticate users using JWT tokens stored in cookies.
- Enforce role-based authorization policies (`ROLE_ADMIN`, `ROLE_USER`).
- Redirect frontend users to appropriate dashboard routes.
- Allow unauthenticated access to selected public endpoints (e.g., invite acceptance).
- Handle authentication at the reverse proxy level, offloading this responsibility from internal services.

The `accessToken` is extracted from the Cookie header of incoming requests. Using the lua-resty-jwt module, the token is verified. If the token is missing or invalid, the request is either denied with a 401 Unauthorized or redirected to the login page.

After successful verification, the JWT payload is inspected to determine the user's roles. Access is granted only to users with either `ROLE_ADMIN` or `ROLE_USER`. If no valid role is found, a 403 Forbidden response is returned.

In the case of frontend routes under `/generation-connect-app/auth/`, valid users are redirected to role-specific dashboard paths. Admin users are sent to `/admin/dashboard`, while regular users are redirected to `/user/dashboard`.

To support unauthenticated workflows, certain endpoints are explicitly excluded from authentication. For example, `/generation-connect-app/invite/accept and /generation-connect-api/family-tree/manage-users/accept` bypass token validation to enable public access.

```
access_by_lua_block {
    ngx.log(ngx.INFO,"Starting JWT validation...");

    local uri = ngx.var.request_uri

    -- Allow specific unauthenticated paths
    if uri:match("^/generation%-connect%-app/invite/accept") or
       uri:match("^/generation%-connect%-api/family%-tree/manage%-users/accept") then
        return
    end

    local jwt = require "resty.jwt"
    local cookie = ngx.var.http_cookie
    local token = nil

    if cookie then
        local jwt_cookie = cookie:match("accessToken=([^;]+)")
        if jwt_cookie then
            token = jwt_cookie
        end
    end
```

```
22.
23.     if not token then
24.         ngx.status = ngx.HTTP_UNAUTHORIZED
25.         ngx.redirect("/")
26.         return
27.     end
28.
29.     local jwt_obj = jwt:verify("lua-resty-jwt", token)
30.     if not jwt_obj then
31.         ngx.status = ngx.HTTP_UNAUTHORIZED
32.         ngx.redirect("/")
33.         return
34.     end
35.
36.     local roles = jwt_obj.payload.roles or {}
37.     local is_admin, is_user = false, false
38.
39.     for _, role in ipairs(roles) do
40.         if role == "ROLE_ADMIN" then
41.             is_admin = true
42.         elseif role == "ROLE_USER" then
43.             is_user = true
44.         end
45.     end
46.
47.     if not is_admin and not is_user then
48.         ngx.status = ngx.HTTP_FORBIDDEN
49.         ngx.redirect("/")
50.         ngx.exit(ngx.HTTP_FORBIDDEN)
51.     end
52.
53.     -- Dashboard redirection logic (for frontend auth paths)
54.     if uri:match("^/generation%-connect%-app/auth/") then
55.         if is_admin then
56.             ngx.redirect("/generation-connect-app/admin/dashboard")
57.         elseif is_user then
58.             ngx.redirect("/generation-connect-app/user/dashboard")
59.         end
60.     end
61. }
```

Above Lua-based validation block is applied to both the frontend and backend location blocks to ensure consistent enforcement of authentication policies as followed:

```
location ^~ /generation-connect-app {
    #CORS headers and preflight handling
    ...
    access_by_lua_block { <shared JWT logic above> }
    proxy_pass http://app.
    ...
}

location ^~ /generation-connect-api {
    #CORS headers and preflight handling
    ...
    access_by_lua_block { <shared JWT logic above> }
    proxy_pass http://api;
    ...
}
```

Each `proxy_pass` directive is accompanied by standard `proxy_set_header` lines to preserve the original client request information. This allows backend services to receive accurate headers such as the real IP address and host, which are often necessary for logging, authentication, or application logic.

12

Building and System Deployment using Dockerization

This chapter explains how the system is built and deployed locally using Docker. It begins with an overview of the main services and their interactions within the local deployment architecture. It then provides a step-by-step guide to setting up the environment, configuring Docker Compose, and using custom scripts to build, run, and clean up the application. The chapter concludes with instructions to verify that the system is running correctly, offering a complete guide for efficient local deployment.

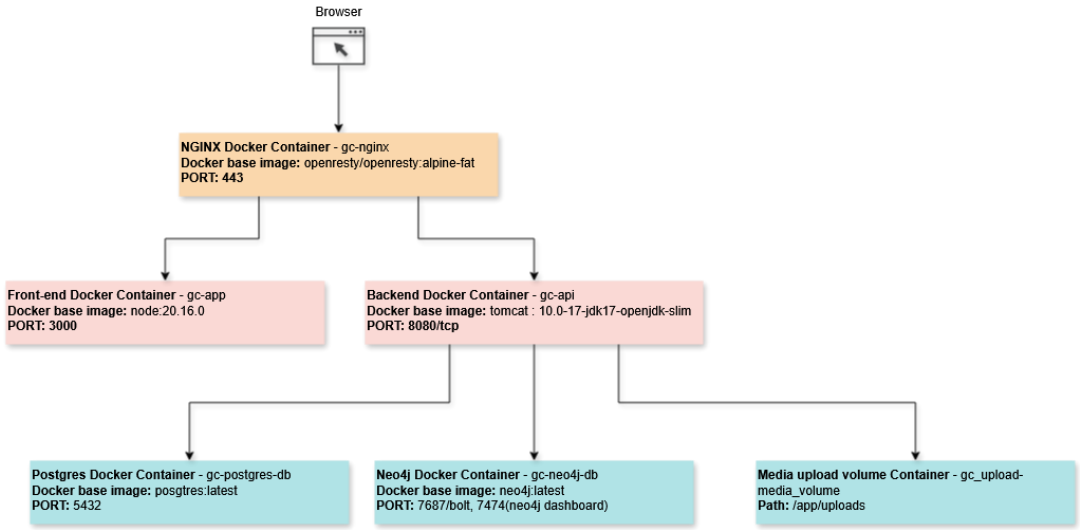
12. Building & System Deployment using Dockerization

12.1 System's Local Deployment Architecture

To support modular development and testing, the system is deployed locally using Docker. All services—frontend, backend, databases, and proxy—run inside isolated containers, coordinated by Docker Compose. This setup allows developers to replicate a production-like environment on their local machines with minimal configuration effort.

12.1.1 Overview of Services

The system consists of several interconnected components as show below:



This diagram illustrates the flow from the user's browser through the NGINX Docker container, which acts as a secure reverse proxy. Requests are routed to either the frontend service `(gc-app)` or the backend service `(gc-api)`. The backend service then interacts with supporting services: PostgreSQL for relational data, Neo4j for graph-based family tree data, and a media volume for uploaded files. All components run as isolated Docker container services.

12.1.2 How Everything Runs Together

A top-level `docker-compose.yml` file defines all the services and how they interact. When the developer runs the provided start.sh script, Docker Compose builds and starts all services in the correct order. The databases start first, followed by the backend (which depends on the databases being ready), then the frontend and NGINX.

This approach makes it easy to manage dependencies, isolate issues during development, and mirror how the application will behave when deployed on a server.

12.2 Guide to Run and Deploy the System locally

12.2.1 Prerequisite and Setup

To deploy the system locally, the following tools and configurations must be installed and set up on the host machine:

Required Tools:

- **Docker** (version 20.10 or later)
- **Docker Compose** (version 1.29 or later)
- **Java 17 SDK** (for backend build)
- **Node.js v20+ and Yarn** (for frontend build)

Project Directory Structures

To maintain modularity and clarity, the project is separated into frontend and backend components, each with a well-organized directory structure.

Backend Project Tree

The backend is structured to isolate Docker configurations for each service (Postgres, Neo4j, NGINX, Tomcat), automation scripts, and source code. Important configuration files such as neo4j.conf, nginx.conf, and pom.xml reside at the project root.

```
backend
├── docker
│   ├── neo4j
│   │   └── Dockerfile
│   ├── nginx
│   │   ├── ssl
│   │   │   ├── certs
│   │   │   │   ├── dhparam.pem
│   │   │   │   └── selfsigned.crt
│   │   │   └── private
│   │   │       └── selfsigned.key
│   │   └── Dockerfile
│   ├── postgres
│   │   └── Dockerfile
│   └── tomcat
│       └── Dockerfile
├── postman-collection
├── scripts
├── src
├── docker-compose.yml
├── neo4j.conf
├── nginx.conf
└── pom.xml
```

Frontend Project Tree

The frontend follows a standard React project layout. Docker and Yarn scripts are provided to automate builds and deployments. The src/ directory contains the main application code, while package.json handles dependency management.

```
frontend/
├── public/
├── scripts/
│   ├── bootstrap.sh
│   ├── cleanup.sh
│   ├── start.sh
│   └── stop.sh
├── src/
├── docker-compose.yml
├── Dockerfile
├── package.json
└── Readme.md
```

12.2.2 Docker compose Configuration

As proposed in High-Level architecture diagram, the core services of the Generation Connect platform—including the backend (Spring Boot), frontend (React), PostgreSQL, Neo4j, and the NGINX gateway—are containerized using Docker. Each service is built from a dedicated Dockerfile configuration as provided here in *Section 6.2*, with configurations tailored for separation of concerns, ease of maintenance, and consistent environment setup.

To manage these containers and handle inter-service communication, a single `docker-compose.yml` file is used. This configuration brings up the entire application stack in one step, handling network creation, port mappings, volume persistence, and environment variable setup automatically.

The following sections define how the platform's services are built, connected, and run together as a cohesive system, following the same sequence as configured in the `docker-compose.yml` file.

Service: nginx (Gateway Proxy)

The nginx service acts as the reverse proxy and request router for the platform. It is built using a custom Dockerfile present in the project tree at /pm455/backend/docker/nginx/ and uses the OpenResty base image for Lua scripting support.

- **Ports**: Maps port 80 of the container to the host, exposing the platform externally.
- **Volumes**:
  - `nginx.conf` is mounted as read-only to inject the configured NGINX routing and Lua authentication.
  - A volume is mounted for logging at `/var/log/nginx`.
- **Depends On:** The service depends on app (frontend) and api (backend) containers to be available.

```
services:
  nginx:
    container_name: gc-nginx
    image: gc-nginx-image
    build:
      context: backend/docker/nginx
      dockerfile: Dockerfile
    ports:
      - "80:80" // For HTTP
      - “443”:”443” //For HTTPS
    volumes:
      - ./backend/nginx.conf:/usr/local/openresty/nginx/conf/nginx.conf:ro
      - ./docker/nginx/ssl/:/etc/ssl/:ro         // mount ssl certificate in nginx volume
      - ./logs:/var/log/nginx
    depends_on:
      - app
      - api
```

Service: db (PostgreSQL)

The db service runs PostgreSQL as the primary relational database. It is built using a custom Dockerfile present in the project tree at `/pm455/backend/docker/postgres/`

- **Environment Variables**: Define the database name, user, and password.
- **Volumes**: Stores persistent data in a named volume.
- **Healthcheck**: Ensures the service is ready before dependent containers start.
- **Ports**: Exposes port 5432

```
neo4jdb:
    container_name: gc-neo4j-db
    image: gc-neo4j-db-image
    build:
      context: backend/docker/neo4j
      dockerfile: Dockerfile
    environment:
      NEO4J_AUTH: neo4j/gc-family-password
      NEO4J_dbms.default_database: gc-family-db
    ports:
      - "7687:7687"
      - "7474:7474"
    volumes:
      - "gc-neo4j-db-data:/var/lib/neo4j/data"
      - ./backend/neo4j.conf:/conf/neo4j.conf:ro
    healthcheck:
      test: [ "CMD", "cypher-shell", "-u", "neo4j", "-p", "gc-family-password", "--format",
"verbose", "RETURN 1" ]
      interval: 10s
      timeout: 5s
      retries: 10
```

Service: neo4jdb (Neo4j)

The neo4jdb service runs the Neo4j graph database, used for modeling and storing family tree relationships.

- **Authentication**: Secured using a username/password combination defined in the NEO4J_AUTH variable.
- **Configuration**: Loads a custom neo4j.conf file via a mounted volume.
- Ports: Exposes both the Bolt protocol port (7687) and the web UI (7474).
- **Healthcheck**: Uses the cypher-shell to confirm readiness.

```
  neo4jdb:
    container_name: gc-neo4j-db
    image: gc-neo4j-db-image
    build:
      context: backend/docker/neo4j
      dockerfile: Dockerfile
    environment:
      NEO4J_AUTH: neo4j/gc-family-password
      NEO4J_dbms.default_database: gc-family-db
    ports:
      - "7687:7687"
      - "7474:7474"
    volumes:
      - "gc-neo4j-db-data:/var/lib/neo4j/data"
      - ./backend/neo4j.conf:/conf/neo4j.conf:ro
    healthcheck:
      test: [ "CMD", "cypher-shell", "-u", "neo4j", "-p", "gc-family-password", "--format",
"verbose", "RETURN 1" ]
      interval: 10s
      timeout: 5s
      retries: 10
```

Service: api (Backend - Spring Boot)

The api service runs the backend application using a Tomcat + JDK 17 base image.

- Environment Variables: Define PostgreSQL and Neo4j connection URLs, usernames, and passwords.
- Volumes: A volume is mounted to persist uploaded media files.
- Depends On: The service waits for both db and neo4jdb to pass their health checks before starting.

```
  api:
    container_name: gc-api
    image: gc-api-image
    build:
      context: backend/docker/tomcat
      dockerfile: Dockerfile
    volumes:
      - "gc_uploaded-media:/app/uploads"
    ports:
      - "8080:8080"
    environment:
      # Postgres environment variables
      - SPRING_DATASOURCE_URL=jdbc:postgresql://gc-postgres-db:5432/gc-db
```

```
      - SPRING_DATASOURCE_USERNAME=gc-user
      - SPRING_DATASOURCE_PASSWORD=gc-password
      # Neo4j environment variables
      - SPRING_DATA_NEO4J_URI=bolt://gc-neo4j-db:7687
      - SPRING_DATA_NEO4J_USERNAME=neo4j
      - SPRING_DATA_NEO4J_PASSWORD=gc-family-password
      - SPRING_DATA_NEO4J_DATABASE=gc-family-db
    depends_on:
      db:
        condition: service_healthy
      neo4jdb:
        condition: service_healthy
    links:
      - db
      - neo4jdb
```

Service: app (Frontend - React)

The app service runs the React frontend using Node.js.

- **Build Context**: Uses the frontend directory and Dockerfile.
- **Ports**: Maps the development server to port 3000.
- **Volumes**: Shares the node_modules folder and source code directory for live reload and development.

```
  app:
    container_name: gc-app
    image: gc-app-image
    build:
      context: frontend/
      dockerfile: Dockerfile
    stdin_open: true
    ports:
      - "3000:3000"
    volumes:
      - "gc_node_modules:/app/node_modules"
      - "./frontend:/app"
```

Defined Volumes

Persistent volumes are defined for PostgreSQL data, Neo4j data, uploaded media discussed as server storage in High Level Architecture Diagram, and Node.js dependencies.

```
volumes:
  gc-postgres-db-data:
    name: gc_postgres_db_data_volume
  gc-neo4j-db-data:
    name: gc_neo4j_db_data_volume
  gc_node_modules:
    name: gc_node_modules_volume
  gc_uploaded-media:
    name: gc_uploaded-media_volume
```

***Note**: The configuration in the `docker-compose.yml` file must be executed in sequence, as container startup is ordered, and certain services depend on others being fully initialized in next section.*

12.2.3 Build, Start, and Cleanup Scripts prepared to run the application

Because the application involves multiple technologies, it is important to have an automated way to handle local deployment by building images, starting services, and cleaning up containers, images, and volumes. These scripts automate common tasks and ensure that all services are consistently set up and managed. Also, these scripts made development of application very easy to be completed as it was one time effort and good to manage.

The root bootstrap and cleanup scripts present in main directory triggers backend and frontend bootstrap and cleanup script respectively and make the process of local deployment and cleaning up automated. Let's see each one by one:

Backend and Infrastructure Bootstrap Script

The following script compiles the backend application using Maven and builds Docker images for all backend-related services, including PostgreSQL, Neo4j, the Spring Boot API, and NGINX.

```
# Build the backend application
mvn clean
mvn install -DskipTests

# Copy the generated JAR file to the Tomcat Docker context
cd docker/tomcat/ || exit
rm -f gc-api.jar
cd ../../
cp target/gc-api.jar docker/tomcat/

# Build PostgreSQL image
cd docker/postgres || exit
docker build -t gc-postgres-db-image -f Dockerfile .
cd ../../

# Build Neo4j image
cd docker/neo4j || exit
docker build -t gc-neo4j-db-image -f Dockerfile .
cd ../../

# Build API image
cd docker/tomcat || exit
docker build -t gc-api-image -f Dockerfile .
cd ../../

# Build NGINX image
cd docker/nginx || exit
docker build -t gc-nginx-image -f Dockerfile .
cd ../../
```

Frontend Bootstrap Script

This script builds the frontend Docker image using the provided Dockerfile in the frontend/ directory.

```
cd frontend || exit
docker build -t gc-app-image -f Dockerfile .
cd ..
```

Start Script

To start all services at once, use the following command from the root directory where docker-compose.yml is located:

```
docker compose up -d
docker compose logs -f
```

***Note**: Separate start scripts are also provided to run the frontend and backend of the Generation Connect application independently. This supports isolated development and avoids the need to load the entire application stack during individual service development.*

Cleanup Script

This script stops and removes all containers and images created during the build and run process. It also includes optional volume cleanup commands that can be uncommented as needed.

```
mvn clean

# Stop and remove PostgreSQL
cd docker/postgres || exit
docker stop gc-postgres-db
docker rm gc-postgres-db
docker images
docker rmi -f gc-postgres-db-image

# Stop and remove Neo4j
docker stop gc-neo4j-db
docker rm gc-neo4j-db
docker images
docker rmi -f gc-neo4j-db-image

# Stop and remove API
cd ../..
cd docker/tomcat || exit
docker stop gc-api
docker rm gc-api
docker images
docker rmi -f gc-api-image

# Stop and remove NGINX
cd ../..
cd docker/nginx || exit
docker stop gc-nginx
docker rm gc-nginx
docker images
docker rmi -f gc-nginx-image

# Stop and remove frontend
docker stop gc-app
docker rm gc-app
docker images
docker rmi -f gc-app-image
docker volume rm gc_node_modules

# Optional: remove Docker volumes
# docker volume rm $(docker volume ls -q)
```

***Note:** If these scripts were not provided, building the application would require running all the commands manually, which could be a tedious and error-prone process. Therefore, using scripts is the most efficient way to manage a complex, multi-technology application like Generation Connect.*

12.2.4 Running the System and Verifying Local Deployment

To run the system following steps must be followed in the sequence as below as per the requirement:

1. Clone the Repository

```
git clone https://campus.cs.le.ac.uk/gitlab/pgt_project/24_25_spring/pm455.git
cd pm455
```

2. To enable invitation emails and AI-powered with `gpt-3.5-turbo` model - features, the following configurations must be added to the `src/main/resources/application.properties`.

```
For SMTP configuration
spring.mail.username=add-your-mail-id      # Replace with actual email
spring.mail.password=add-app-password      # Replace with generated app password


openai.api.key=add-openai-key
openai.api.url=https://api.openai.com/v1/chat/completions
```

3. Run Bootstrap Script at `/pm455` which will trigger bootstrap script under backend and frontend directory automatically

```
./scripts/bootstrap.sh
```

4. Run Start script at /pm455 which will trigger project root tree's docker-compose.yml file via triggering all Dockerfile of /backend and /frontend configured as automation

```
./scripts/start.sh
```

5. Run Clean script at `/pm455` which will trigger cleanup script under backend and frontend directory automatically and clean the images and containers created in Step 2 and 3.

```
./scripts/cleanup.sh
```

6. To clean all the volumes created via running Docker, run the below command at `/pm455`, this command has been given separately and commented out in `./scripts/cleanup.sh` because it removes database, and it must be used only when required.

```
docker volume rm $(docker volume ls -q)
```

7. To proceed with `https://localhost`, accept the self-signed certificate in your browser:
   - In Chrome: Click Advanced → Proceed to localhost (unsafe)
   - In Firefox: Click Advanced → Accept the Risk and Continue

8. For more information and to add admin user to the data, check out Readme file to run the application locally, Readme_local_run.md on Gitlab.

13

Postman Collection Overview

This section provides an overview of the Postman collection used for testing APIs, outlining how requests were structured, organized, and validated.

It demonstrates the role of Postman in ensuring reliable and consistent backend functionality.

13. Postman Collection Overview

The Postman collection for *Generation Connect* is organized into two primary segments:

- **Generation Connect Consumed APIs on Frontend**
  This section includes all REST APIs actively used by the frontend application. It covers functionalities such as user authentication, family tree management, user roles and invites, media uploads, historical research tagging, and admin analytics. Each request is documented with saved examples to showcase request structure and expected responses.

- **Backend Research and Study (PoC)**
  This section was used for experimental development and verification of graph database modeling using Neo4j. It contains endpoints to manage Person nodes and create relationships like parent-child, sibling, and marriage, allowing validation of relationship traversal and graph queries.

Environment Setup and Token Handling

A mock environment as shown *Appendix 40*, named Generation Connect Mock Environment is configured with the following reusable variables:

- baseUrl – Base path for API execution (e.g., https://localhost:8080/...)
- accessToken – Stores member login token
- adminAccessToken – Stores admin login token

Pre-request Script and Test Scripts

To support secure and seamless testing, test scripts are used to automatically save JWT tokens returned after login:

```
// After successful login – save user token
var jsonResponse = pm.response.json();
pm.environment.set("accessToken", jsonResponse.accessToken);

// After successful admin login
var jsonResponse = pm.response.json();
pm.environment.set("adminAccessToken", jsonResponse.accessToken);

// Attach user token from environment
var accessToken = pm.environment.get("accessToken");
pm.request.headers.add({
    key: 'Cookie',
    value: 'accessToken=' + accessToken
});
```

14

Testing Strategy and Validation

This chapter outlines the approach to ensuring software reliability through structured testing.

It includes unit testing for individual components and integration testing using Postman to validate the interaction between system modules.

14. Testing Strategy and Validation

Testing was carried out regularly throughout the development of Generation Connect to ensure the system worked correctly and met user expectations. Postman was used to test backend APIs, and tests were grouped into two main areas:

- Generations Connect APIs used: These included features like login, sign-up, family tree creation, profile posts, media uploads, role-based access, and admin functions.
- Backend Research and Study APIs: These were used to test Neo4j-based graph features such as adding people and creating family relationships like parent-child, sibling, and spouse. These tests helped confirm the correctness of the family tree structure and relationship queries.

For each Postman collection has example saved under each requests.

UI and UX components were also tested during each feature development to make sure pages were displaying properly, user flows were smooth, and interactions worked as expected.

Testing was done after every new feature was added and at the end of each two-week development phase. If any issues were found, they were fixed right away and tested again to confirm the fix worked. This process helped maintain the stability of both the PostgreSQL and Neo4j parts of the system, as well as the frontend user experience.

Tested Feature	Date
Setup Testing	02-03-2025
Welcome Page Testing	04-03-2025
Admin, User Login and User Signup Testing	12-03-2025
Interactive Family Tree Testing	25-03-2025 to 27-03-2025
Collaborative Story Telling	06-04-2025
Research Api Testing	07-04-2025
Privacy Control Testing	07-04-2025
Send and Invite Feature Testing	13-04-2025
Tag Research Testing	13-04-2025
Admin Api	14-04-2025
Add Multi-media support	26-04-2025
Upgrade HTTP to HTTPS	04-05-2025
Integration with Open AI	09-05-2025

UI/UX Testing is often for feature enhancement. Also have added unit test cases in the GitLab repository.

15

Achieved results

This chapter outlines the results achieved from the original list of proposed features, along with additional improvements made during development. These extra changes were added to enhance user experience and make the platform more user-centered.

15. Achieved results and their Evaluation

During the initial planning phase, the project requirements were classified into three categories: Essential, Desirable, and Optional.

The project successfully achieved:

- 100% of the Essential requirements,
- Approximately 100% of the Desirable requirements (with one modification/replacement made for practical reasons), and an enhancement using AI Integration.
- Approximately 33% of the Optional requirements (one out of three features completed).

Optional requirements were primarily reserved for future development, maintaining flexibility for future platform growth.

15.1 Evaluation of Essential Requirements

The Essential Requirements represented the core functionalities of the Generation Connect platform. Full completion of these features was mandatory for the project's success.

The following essential features were successfully achieved:

Feature	Achievement	Notes
Interactive Family Trees	✅ Achieved	Users can create, visualize, and interact with multi-generational family trees, with features such as clickable nodes, tooltips, and zooming capabilities.
Collaborative Storytelling	✅ Achieved	Multiple users can contribute (as per the privileges user has) to personal stories, memories, and anecdotes to family tree profiles, supporting rich, multi-perspective storytelling.
Media Integration	✅ Achieved	Upload and manage multimedia content (images, videos, documents) linked to family and individual (in family Tree) profiles posts.
Privacy Control and Access Permissions	✅ Achieved	Users control visibility for Private and Public Tree (only viewable unless owner or contribution role is assigned) and contribution rights for each tree assigned by the owner of the tree.
Collaborative Research Tools	✅ Achieved	Search and tag historical records from external archives as a prototype for the idea.

Register User & Authenticate	✅ Achieved	Secure account creation and login implemented for both Admin and Users using JWT tokens; ensures protected access to sensitive family tree data.

Although not initially required, secure user authentication was introduced early due to the sensitive nature of family data. It ensures only authorized users can access or modify family trees, reinforcing platform security and aligning with web application best practices.

Evaluation:

All essential requirements were successfully delivered. The implemented features align with the initial motivations and performance expectations, ensuring that the platform provides a strong foundation for preserving family histories through interactive, collaborative storytelling.

15.2 Evaluation of Desirable Requirements

Desirable Requirements were intended to enhance the platform's usability, administration, and user engagement. Full delivery was not mandatory, although achieving as many as possible was an objective.

The following desirable features were achieved:

Feature	Achievement	Notes
Event Tracking and Timeline	✅ Achieved	An AI-powered chatbot was integrated using the OpenAI API to provide real-time answers about family tree members and their relationships. This complements the timeline and event tracking features, offering interactive exploration without storing user queries or responses.
Audit Logging	✅ Achieved	Admin has access to activities logs tracking user activities for security, monitoring, and compliance purposes.
Community Forums and Discussions	Not Achieved (but intentionally replaced),	After reviewing initial feedback from Principal Marker and considering the practical needs of the application, the Community Forums feature was intentionally replaced with a secure Invite System to provide a more user-centric and practical onboarding process.
Manage User on Invitation (Modification)	✅ Achieved (as a better alternative to Community	A secure Invite System was implemented using Google SMTP, allowing tree owners to invite users with assigned roles, enhancing collaboration and maintaining controlled access to family trees.

to join the family tree	Forums and Discussions)	

Evaluation:

Nearly all desirable features were successfully achieved. After reviewing initial feedback and reflecting on the practical needs of the platform, the originally planned Community Forums feature was intentionally replaced with a secure Invite System.

This decision enhanced the platform's user-centric design by providing a more practical, secure, and streamlined onboarding process for new and existing users, without compromising the project's collaborative goals.

15.3 Evaluation of Optional Requirements

Optional Requirements were identified as future enhancements intended to further enrich the platform but were not mandatory for project completion.

The table below summarizes the status of these features based on the current development phase.

Feature	Achievement	Notes
Geographical Mapping of Family History	Not Achieved	Intended for future versions; will display migration patterns and ancestral locations.
Celebration Features	Not Achieved	Future enhancement to celebrate user achievements and family events.
Usage Metrics	✅ Achieved	The admin dashboard displays real-time metrics such as total users, family trees (public and private), pending invites, active users in the last 30 days, and posts and comments, fulfilling the requirement for tracking platform engagement and activity.

Evaluation:

As planned, most optional features were reserved for future development and have not been implemented in the current phase.

However, the Usage Metrics feature was successfully delivered, providing administrators with real-time platform engagement insights.

The overall system architecture remains extendable, allowing the planned optional enhancements to be incorporated smoothly in future iterations

16

Critical Reflection and Evidence-Based Solution

This chapter provides a reflective analysis of the project, highlighting key challenges and the solutions applied. It also includes an evidence-based evaluation to assess the project's success using measurable outcomes.

16. Critical Reflection and Evidence-Based Evaluation

Throughout the Generation Connect project, a variety of challenges were encountered across the planning, implementation and integration. This section outlines the major obstacles faced during the development lifecycle and the strategies adopted to address them, providing a reflective and evidence-driven evaluation of the overall project experience.

16.1 System Deployment: Docker and Containerization

Challenge:

From the start, it was clear that containerization would be necessary due to the number of different services the system required—backend APIs, the frontend React app, Neo4j for graph data, and PostgreSQL for structured storage. However, the real challenge wasn't deciding to use Docker; it was figuring out how to set everything up in a clean, consistent way that worked reliably across machines. Managing dependencies and configurations for so many components risked becoming messy and error-prone if handled manually.

Approach:

To address this, I chose Docker as the base platform and carefully wrote individual Dockerfiles for each core component. I also used Docker Compose to orchestrate the entire stack, allowing all services to start together in a single, coordinated environment.

What made a real difference was investing time in creating a set of Bash scripts to automate the repetitive parts of the workflow. These scripts handled everything from building the Spring Boot backend and React frontend to packaging the Docker images and bringing the full system up with Docker Compose. By removing manual steps, the deployment process became much more streamlined and reliable.

Outcome:

This setup turned out to be one of the best decisions made during development. It allowed me to spin up the complete system with a single command, on any compatible machine, without worrying about mismatched configurations or missing dependencies. It also saved time when testing new features or demonstrating the system to others, because the environment was always reproducible. Overall, containerization with scripting support brought structure, predictability, and efficiency to the development workflow.

Evidence of this can be seen in:

- The Docker Compose setup shown in *Section 12.2.2*, where each service is defined with networking and volume configurations.
- The included Bash script such as bootstrap.sh, start.sh, cleanup.sh and stop.sh, which automates image builds and orchestrates startup without manual steps (see *Section 12.2.3*).
- A successful multi-service boot captured in, *Appendix 43* confirming the coordination of frontend, backend, and both databases.

16.2 Secure Traffic with NGINX and Lua Proxy

Challenge:

Routing requests in a secure and flexible way became a critical requirement, especially since the platform needed to support both public-facing functionality—such as user registration, invite acceptance, and admin login—and authenticated areas like managing family trees or accessing research tools. Simply forwarding requests based on URLs wasn't enough. There had to be a clear separation between what was accessible to the public, authenticated users, and platform administrators. Managing these access levels while maintaining clean URIs and minimizing exposure to backend services was not straightforward.

Approach:

To meet this need, I configured NGINX as a reverse proxy and lightweight API gateway. Beyond standard proxying, I used Lua scripting within NGINX to implement custom logic for JWT validation and role-based access control. This allowed the system to extract tokens from cookies, decode them, verify their integrity, and evaluate the user's role—all before the request ever reached the backend.

- Public endpoints like `/auth/signup`, `/auth/login`, and `/family-tree/manage-users/accept` were explicitly allowed without token validation.
- Protected paths such as `/admin/**` and `/family-tree/**` required valid JWT tokens and were gated based on role: Admin or User.
- The gateway handled redirection and rejection (401 or 403) instantly if the role check failed, keeping the backend clean and stateless.

The routing rules and Lua script are stored and versioned as part of the project infrastructure (see *Section 11*), ensuring that any team member or reviewer could understand the access control logic directly from the configuration files.

Outcome:

This approach created a strong first line of defence. It reduced load on backend services by blocking invalid or unauthorized requests at the gateway itself and ensured that Admin- and User-facing areas remained strictly isolated. It also made the security logic transparent and easy to maintain, as most access control conditions were written in Lua right inside the NGINX configuration.

To validate this setup:

- I tested routing behaviour by accessing protected endpoints without a token and with invalid roles (resulted in 403), and with valid credentials (successful) as shown in *Appendix 44*, *Appendix 46* and *Appendix 45*.
- Screenshots of these test scenarios are included in *Appendix 47* and *Appendix 45*, just right against `` `gc-nginx` ``.
- A snippet of the NGINX location block and embedded Lua role validation logic is shown in *Section 11.5's* code snippet.

16.3 Microservice Security: Spring Security Framework

Challenge:

While the NGINX gateway provided a solid layer of protection at the system's edge, it was essential that backend microservices enforced their own authentication and authorization policies. Relying solely on the gateway was risky—if someone managed to bypass or misroute around it, the backend still needed to reject unauthorized access. The challenge was to implement robust, role-based security without creating tightly coupled or overly complex configurations that could impact development flexibility or performance.

Approach:

To solve this, I integrated Spring Security into the backend services. The configuration was designed to validate JWT tokens embedded in cookies and establish role-based permissions for each endpoint. A custom filter (`JwtAuthFilter`) was used to extract the token, decode it using a service (JwtService), and load the user context. This happened early in the request lifecycle, before any controller logic was executed.

Key steps in the security setup included:

- Defining public endpoints like `/auth/signup`, `/auth/login`, and `/family-tree/manage-users/accept` which did not require authentication.
- Securing protected endpoints such as `/person/**`, `/family-tree/**`, and `/admin/**` which required valid JWT tokens and checked for roles like ROLE_USER or ROLE_ADMIN.
- Applying stateless session policies using `SessionCreationPolicy.STATELESS`, ensuring no server-side session data was stored.
- Configuring CORS policies within `WebMvcConfigurer` to allow secure communication between the frontend and backend, especially during cross-origin requests.

The code for this setup, including the security configuration class and the filter chain, is documented and versioned as part of the project (see *Section 5.3* and *Section 8.1*).

Outcome:

This internal security layer provided a safety net, making sure that even if the gateway was misconfigured or bypassed, the backend wouldn't process unauthorized requests. It also ensured consistent role validation and predictable behaviour across endpoints, regardless of where a request originated.

To demonstrate and validate this:

- I conducted tests using Postman and browser dev tools by removing or tampering with tokens, confirming that access was denied with proper HTTP error codes (401 or 403).
- I also tested session behaviour across browser reloads to confirm that the stateless configuration was enforced correctly.
- Code snippets of the Spring Security config and the custom JWT filter are included in *Appendix 49* , and screenshots of access validation tests are presented for various scenarios

from *Appendix 44-48*, for forbidden for success using postman call and logs against `'gc-api'`

Together, these steps helped the backend services uphold security independently while supporting a clean, modular, and scalable architecture.

16.4 Architectural Challenge: Introducing a Hybrid Database Model

Challenge:

In the early stages of development, the plan was to manage all application data—including user profiles, access roles, posts, and family trees—within PostgreSQL. However, when it came to building the actual family tree structure with its deep, interconnected relationships, the limitations of a relational schema quickly became apparent. Modelling parent-child links, marriages, and extended connections across generations in SQL became overly complex, error-prone, and rigid. At the same time, switching entirely to a graph database like Neo4j wasn't feasible, especially for structured data like user details and access control.

Approach:

To address this, I designed a hybrid database model. PostgreSQL remained responsible for handling transactional and structured data (e.g., user accounts, access permissions, metadata), while Neo4j was introduced solely to manage the family tree's graph structure. To test the feasibility of this approach, I built a working proof-of-concept where each family tree entry in PostgreSQL linked to corresponding nodes and relationships in Neo4j.

This architectural shift required restructuring parts of the backend—especially the service layer and DTOs—and upgrading some Spring Boot modules to support dual-database handling. Fortunately, the backend was modular from the start, which allowed for this transition without major rewrites.

Outcome:

The hybrid model allowed the application to benefit from the strengths of both worlds. PostgreSQL ensured reliable and efficient handling of user-centric data, while Neo4j allowed dynamic, bidirectional traversal of family relationships. This not only simplified development going forward but also enhanced performance and maintainability.

To support this architecture:

- Sample family trees were created and tested using Cypher and SQL in parallel.
- The relationship between `familyTreeId` in PostgreSQL and root nodes in Neo4j is documented in *Appendix 50*.
- Sequence diagrams demonstrate this integration *Appendix 15* and *Appendix 16*.

16.5 Backend Integration: Neo4j Repository Management

Challenge:

Adopting Neo4j introduced a completely new way of thinking about data. Unlike SQL, which operates in rows and tables, Neo4j's Cypher Query Language (CQL) focuses on nodes and relationships. Modelling family members and their connections—like parents, spouses, and children—was no longer about foreign keys, but about directional, traversable paths. This shift came with a steep learning curve, especially around managing bidirectional relationships and avoiding data inconsistencies.

Approach:

To get familiar with Neo4j, I started small—creating simple person nodes using Spring Data Neo4j and testing basic relationships like FATHER_OF and SPOUSE_OF. Then I gradually increased complexity, introducing conditional logic for roles, gender-specific relationships, and bidirectional linking. It took several iterations to stabilize the logic and avoid cyclic or orphaned nodes.

Each stage involved writing test nodes, creating mock family branches, and verifying the results through Cypher queries and Neo4j Browser visualization. The repository layer was gradually refined to support efficient traversal and updates while ensuring relationships stayed consistent.

Outcome:

The result was a robust Neo4j repository layer capable of handling diverse and complex family relationships. Operations like adding a child, updating a person's profile, or fetching ancestors now performed reliably and reflected accurate graph paths.

- Sample Cypher queries and annotated test data are provided in Section 8.2's *12th point and Appendix 50*'s Neo4j Graph.
- A detailed timeline of iteration and refinement efforts is documented in the reflective merge requests available on GitLab—refer to Merge Request Link 1 and Link 2

This stage not only deepened my understanding of graph data modelling but also made the core family tree logic scalable and future-proof.

16.6 Frontend Visualization: Dynamic Graph Rendering

Challenge:

Selecting and integrating a suitable frontend library to visualize the evolving family tree posed notable difficulties. While D3.js offered powerful graph visualization capabilities, integrating it smoothly with Reacts component lifecycle proved challenging. Moreover, many existing libraries did not support persistence, leading to data loss on page refresh.

Approach:

After evaluating several frontend libraries, I identified a static D3 family tree module that provided a clean, node-based layout. Although it wasn't connected to any backend by default, I modified it extensively to include:

- API integration with the Neo4j backend.
- Support for dynamic updates via user interaction.
- Callback hooks that triggered save/update operations through HTTP requests.

This allowed users to add or remove people from the tree and see the changes reflected both visually and in the database. It also ensured the tree could be reloaded with the same state preserved, making the experience reliable.

Outcome:

A fully functional, real-time family tree visualization was achieved. It synced seamlessly with the Neo4j backend and allowed for dynamic interactions, even with deep relationship chains. The visualization worked with zoom, pan, and tooltip support, enhancing usability across generations.

- Sample renders and interface screenshots are included in *Appendix 51 and in Section 10*.
- The callback and data sync logic is described in detail in Section 9.1's point *11*.

This solution brought the platform to life—turning raw data into an engaging, interactive experience for users documenting their family history.

Other challenges are identified below and described in detail with Evaluation in the following *Sections 8.3* and *Section 8.4*.

17

Conclusion and Future Work

This final chapter summarizes the overall project outcomes and reflects on the development journey. It discusses deviations from the original plan, offers suggestions for future enhancements, and provides a comparative analysis with existing systems to highlight innovation and areas for growth.

17. Conclusion and Future Work

17.1 Summary

The Generation Connect project set out with the aim of creating an interactive and collaborative family history platform that would not only preserve but also celebrate the unique stories and relationships that define family heritage. Through the course of development, the project successfully implemented all core functionalities, fulfilled the essential and desirable objectives, and laid a strong, extensible foundation for future expansion.

17.2 Self Reflection

This journey has been much more than just building a technical solution; it has been a deep personal learning experience. When I first conceptualized the project, I envisioned a tool where families could collaboratively build their family trees, add stories, and share memories. However, as development unfolded, it became clear that just building a functional platform wasn't enough—it had to be intuitive, secure, and meaningful for the people who would use it.

Throughout this project, I not only strengthened my technical capabilities—working across technologies like React, Spring Boot, Neo4j, PostgreSQL, Docker, Open AI Api and NGINX—but also developed a much deeper understanding of user-centered design principles. Every decision, from including the interactive tree to implementing secure role-based access and invite-based collaboration, was rooted in real-world concerns about usability, privacy, and security.

There were steep learning curves at every corner. Implementing containerized deployment using Docker, integrating graph data visualization dynamically in React, designing a hybrid database model that balanced relational and graph data, configuring NGINX Lua scripting for role validation—all demanded research, trial-and-error, and adaptability. Yet, each of these challenges pushed me to grow beyond my initial technical skillset.

A particularly important realization came when I decided to introduce secure authentication mechanisms—even though this was not part of the university's original requirements. Handling sensitive family information meant that proper security could not be compromised and taking that initiative gave me confidence in prioritizing ethical and practical considerations over merely meeting baseline objectives.

Flexibility became one of the core lessons learned. While some features evolved based on practical constraints (such as replacing the planned Community Forums with a secure Invite System), these deviations turned out to be improvements rather than compromises, always aligned with the central vision of making the platform as useful and secure as possible.

Managing development alongside continuous reporting and reflection also taught me the importance of agile thinking, clear communication, and consistent documentation—skills that are equally as important as technical expertise in any real-world project environment.

Looking back, Generation Connect represents not just a fully working web application, but also my personal growth as a developer, researcher, and problem-solver. It mirrors my ability to take an idea from a blank canvas to a tangible, working system that addresses real user needs thoughtfully. This experience has given me a renewed excitement for future challenges and a strong foundation for building not only better systems but also better experiences for users.

17.3 Deviations from Initial Plans

Although Generation Connect largely stayed faithful to its original vision, several important deviations from the initial plan emerged organically during the project lifecycle. These changes were driven by a deeper understanding of real-world needs, evolving user priorities, and technical feasibility. Rather than setbacks, they became opportunities to strengthen the system's design, security, and usability.

One notable deviation in terms of component addition was the integration of AI, which is used not only for upcoming events and timeline features but also to gather additional insights based on the family tree.

The other key deviations are shown and summarized below:

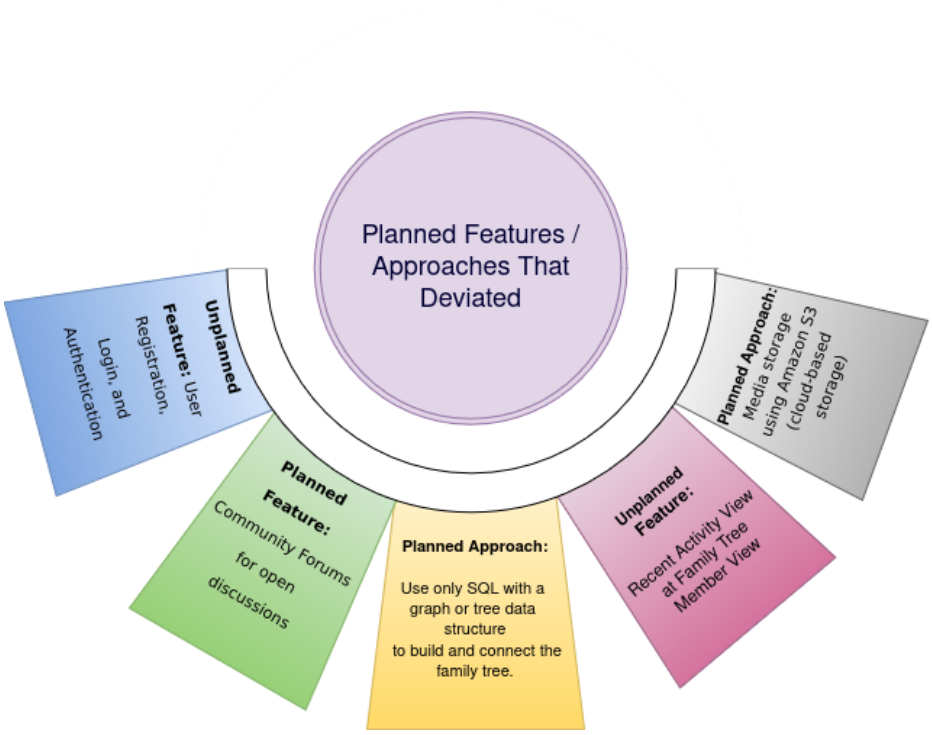


Figure 54 Deviation from Initial Plans

Deviation 1: Additional Functionality - User Registration, Login, and Authentication

Current Implementation	Reason for Deviation	Reflection
Secure user authentication was implemented using JWT, with tokens stored in HTTP-only cookies to safeguard against common web vulnerabilities such as cross-site scripting attacks.	While the university's requirements emphasized privacy controls, explicit user authentication was not mandated, allowing for a publicly accessible platform. However, to ensure meaningful privacy and protect sensitive family data, secure user registration and authentication were deemed essential. This feature, though not originally classified as a core requirement, directly supported the platform's broader privacy objectives and was therefore integrated during development.	The decision to incorporate secure user authentication, despite its absence from the initial requirement list, demonstrated a comprehensive understanding of real-world data protection practices. By addressing privacy at a foundational level, the platform was strengthened not only in terms of academic fulfilment but also in alignment with ethical standards for managing sensitive information. This proactive approach highlights a commitment to long-term system integrity and user trust.

Deviation 2: Planned Feature - Community Forums for open discussions

Current Implementation	Reason for Deviation	Reflection
Replaced with a Secure Invite System to join family trees for collaboration.	Following initial feedback from the Principal Marker and an assessment of the application's practical requirements, the Community Forums feature was intentionally replaced with a secure Invite System. This approach offered a more focused, user-centric, and practical onboarding experience by facilitating controlled access and collaboration within specific family trees.	Substituting community forums with an invite-based system enhanced usability by prioritizing direct collaboration over open discussion. This decision aligned the platform more closely with its core purpose—private and secure family history sharing—while improving user experience and maintaining data confidentiality.

Deviation 3: Planned Approach: Only Sequential Query Language to be used with Graph or Tree Data Structure to create family tree and connect all the persons with each other.

Current Implementation	Reason for Deviation	Reflection
Configured database as hybrid model of Graphical and Relational Database Management System. Apart from database maintained for Family tree member sand their relationship, every database is configured using PostgreSQL. Even creation of Family tree is maintained at RDBMS level.	Neo4j database - Graphical Database Management System enhanced and made the system intelligent and very much scalable for the future scops which might arise.	Easy to be maintained, rearranged and changed for a graphical or tree-based data. Also, Neo4j usage make the system heavy loaded therefore this hybrid configuration made the system light weighted

Deviation 4: Planned Approach- Media storage using Amazon S3 (cloud-based storage)

Current Implementation	Reason for Deviation	Reflection
Revised to use local server storage by creating and mounting a volume within Generation Connect's API server container.	While Amazon S3 was initially considered for its scalability and durability, further evaluation revealed that its integration added unnecessary complexity for the current stage of the application. Additionally, storage and cost limitations posed challenges for ongoing media maintenance in a low-user, development-focused environment.	Adopting a local storage strategy offers a practical and resource-efficient solution aligned with current application demands. This approach simplifies implementation while meeting present media storage needs. The system design allows for future expansion by maintaining compatibility with a hybrid storage model—supporting a transition to cloud services like Amazon S3 when user growth and data volume necessitate it. This decision demonstrates effective prioritization of simplicity and maintainability during early-stage deployment without compromising future scalability.

Deviation 5: Additional New Feature – Recent Activity View (for Family Tree Member View)

Current Implementation	Reason for Deviation	Reflection
This feature has been developed as an enhancement beyond the original scope. It makes use of audited data planned for admin portal, via displaying the last user activities specific to the family tree.	The existing audit logging API developed for the admin portal was identified as a reusable component. It was adapted to provide a recent activity view tied to the user's connected family tree, enhancing the user experience by increasing visibility into collaborative actions.	This enhancement added meaningful value by improving visibility into ongoing contributions. It was implemented without major architectural changes and fits naturally into the existing audit logging system.

17.4 Suggestions for Future Work

Generation Connect has been thoughtfully designed not just for where it is today, but for where it's going. As the platform grows, there are clear opportunities to expand its capabilities, deepen user engagement, and broaden its impact. The following ideas represent key areas of future development, each grounded in real user needs and long-term platform goals.

Generation Connect is designed with future scalability and engagement in mind. The following areas highlight key directions for enhancement:

a. **Additional Frontend Enhancement:**
   Currently, the family tree node supports profile pictures through image URLs only. Due to time constraints, the functionality for direct image upload was not implemented, as it required additional handling for integration with a very complicated library i.e family-chart [20]. Given more time, this feature could have been incorporated to enhance user convenience and flexibility.
   Use already developed backend API's which are not yet integrated with frontend such as deleted, edit post, comments, remove media from post, and many more.

b. **Historical Visualization:**
   Integrate mapping tools (e.g., Leaflet) to display ancestral migration and historical locations, offering users a geographical context to their family stories.

c. **Expanded Research Sources:**
   Connect to additional archives like the U.S. National Archives and third-party genealogy APIs to broaden research opportunities without disrupting current APIs.

d. User Engagement Features:
Introduce milestone celebrations (e.g., birthdays, completed branches) and enhance forums with threaded discussions and moderation to build a stronger community.

e. Technical Optimization:
Implement Redis caching (already planned) for faster response times and prepare backend for cloud storage (e.g., Amazon S3) to support growing media needs.

f. Research Collaboration:
Create an open-access area for external researchers to view and share insights, encouraging wider participation before account registration.

g. Mobile Accessibility:
Develop a mobile app with offline access, media uploads, and real-time notifications to support on-the-go usage.

h. Scalable Tree Visualization:
Enhance the D3.js family tree for better performance with large datasets and improved responsiveness across devices.

18

References

18. References

1. "Generations Connect: A Collaborative Family History Platform," *University of Leicester* By P. Severi (pgs11) [Online] Available: https://campus.cs.le.ac.uk/teaching/proposals/v23626.generations_connect_a_collaborative_family_history_platform. [Accessed: 27- Feb -2025].

2. "Intergenerational projects to preserve family heritage," *Assisted Living Locators*, By Katie Krupka May, 10 2024. [Online]. Available: https://assistedlivinglocators.com/articles/intergenerational-projects-to-preserve-family-heritage [Accessed: 27- April -2025].

3. "3 places to share family history documents and images online," *Family Tree Magazine*, By Sunny Jane Morton 2024. [Online]. Available: https://familytreemagazine.com/storytelling/3-places-to-share-family-history-documents-and-images-online. [Accessed: 27- April -2025].

4. "What Is Genealogy Software? An Introduction to Its Features and Benefit," *Evidentia Software*, By Edward Thompson September 13, 2024. [Online]. Available: https://evidentiasoftware.com/genealogy-software-best-features-benefit. [Accessed: 27- April -2025].

5. "How to Organize Your Family History Research with family genealogy software," *Evidentia Software*, By Edward Thompson October 25, 2024. [Online]. Available: https://evidentiasoftware.com/family-genealogy-software-research. [Accessed: 27- April -2025].

6. "How to Choose the Best Genealogy Software: Key Features to Consider," *Evidentia Software*, By Edward Thompson October 13, 2024. [Online]. Available: https://evidentiasoftware.com/choose-best-genealogy-software-for-family. [Accessed: 27- April -2025].

7. "How Tree Genealogy Software Can Help You Discover Long-Lost Relatives," *Evidentia Software*, By Edward Thompson September 27, 2024. [Online]. Available: https://evidentiasoftware.com/how-tree-genealogy-software-can-help. [Accessed: 27- April -2025].

8. The National Archives (UK), "Website," [Online]. Available: https://discovery.nationalarchives.gov.uk. [Accessed: 27 - April - 2025].

9. "Discovery for developers: about the application programming interface (API)," The National Archives (UK), [Online]. Available: https://www.nationalarchives.gov.uk/help/discovery-for-developers-about-the-application-programming-interface-api. [Accessed: 27 - April - 2025].

10. "API Documentation - National Archives Developer Resources," The National Archives (UK) [Online]. Available: https://discovery.nationalarchives.gov.uk/API/sandbox/index#!/SearchRecords/SearchRecords_GetRecords. [Accessed: 27 - April - 2025].

11. "Online Research Tools and Aids," National Archives (USA), [Online]. Available: https://www.archives.gov/research/start/online-tools. [Accessed: 27 - April - 2025].

12. “National Archives Catalog," Website [Online]. Available: https://catalog.archives.gov. [Accessed: 27 - April - 2025].

13. “Swagger API Documentation," NextGen Catalog API 0.2.0 OAS3 [Online]. Available: https://catalog.archives.gov/api/v2/api-docs. [Accessed: 27 - April - 2025].

14. Confinity, "Confinity - Family Tree Application," [Online]. Available: https://www.confinity.com/culture/best-online-platforms-for-family-history-preservation. [Accessed: 27 - April - 2025].

15. iMeUsWe, "iMeUsWe - Indian Heritage Family Tree," [Online]. Available: https://www.imeuswe.in/the-future-of-genealogy-how-to-create-family-tree-using-imeuswe [Accessed: 27 - April - 2025].

16. Ancestry, "Ancestry - Family History and Genealogy," [Online]. Available: https://www.ancestry.com. [Accessed: 27 - April - 2025].

17. “Features – Gramps Project,” Gramps Blog, [Online]. Available: https://gramps-project.org/blog/features/. [Accessed: 15 - May – 2025].

18. How to dockerize Spring Boot + React apps," *Medium,* By Luiz Gustavo De O. Costa May 10, 2022. [Online]. Available: https://luizcostatech.medium.com/how-to-dockerize-spring-boot-react-apps-1a4aea1acc44. [Accessed: 27 - April - 2025].

19. "Docker overview," Docker Documentation. [Online]. Available: https://docs.docker.com/get-started/docker-overview/. [Accessed: 27 - April - 2025].

20. Donatso, “Family Chart – Interactive Family Tree Library,” GitHub, [Online]. Available: https://github.com/donatso/family-chart. [Accessed: 15 - May - 2025].

21. Sidharth M., “How NGINX handles thousands of concurrent requests,” Medium, [Online]. Available: https://medium.com/@_sidharth_m_/how-nginx-handles-thousands-of-concurrent-requests-202ca1a1cc44. [Accessed: 15 - May – 2025].

22. “NGINX as a Reverse Proxy,” NGINX Documentation, [Online]. Available: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/. [Accessed: 15 - May - 2025].

23. “HTTP Load Balancing,” NGINX Documentation, [Online]. Available: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/. [Accessed: 15 - May - 2025].

24. “Gateway Architecture Overview,” NGINX Gateway Fabric, [Online]. Available: https://docs.nginx.com/nginx-gateway-fabric/overview/gateway-architecture/. [Accessed: 15 - May - 2025].

25. “Understanding the location directive in NGINX,” DigitalOcean Tutorials, [Online]. Available: https://www.digitalocean.com/community/tutorials/nginx-location-directive. [Accessed: 15 - May - 2025].

26. “Beginner’s Guide,” NGINX.org, [Online]. Available: https://nginx.org/en/docs/beginners_guide.html. [Accessed: 15 - May - 2025].

27. “Server Names,” NGINX.org, [Online]. Available: https://nginx.org/en/docs/http/server_names.html. [Accessed: 15 - May - 2025].

28. “NGINX Integration,” FluxNinja Aperture Documentation, [Online]. Available: https://docs.fluxninja.com/aperture-for-infra/integrations/gateway/nginx. [Accessed: 15 - May - 2025].

29. “SSL Module,” NGINX.org, [Online]. Available: https://nginx.org/en/docs/http/ngx_http_ssl_module.html. [Accessed: 15 - May - 2025].

30. Farhan, “Spring Security JWT Authentication & Authorization,” Medium, [Online]. Available: https://medium.com/code-with-farhan/spring-security-jwt-authentication-authorization-a2c6860be3cf. [Accessed: 15 - May - 2025].

31. StackHawk, “What is CORS?,” [Online]. Available: https://www.stackhawk.com/blog/what-is-cors/. [Accessed: 15 – May – 2025].

32. Neo4j, “Cypher Query Language Overview,” [Online]. Available: https://neo4j.com/product/cypher-graph-query-language/. [Accessed: 15 - May - 2025].

33. “Napkin AI – Project Chat Summary,” Napkin.ai, [Online]. Available: https://app.napkin.ai/page/CgoiCHByb2Qtb25lEiwKBFBhZ2UaJGFiNTVlMzI2LWIwODEtNGY4ZS05MDJlLWY4ZGVjMTUzMzZjMw?s=1. [Accessed: 15 - May - 2025].

34. “MVC Framework - Introduction,” GeeksforGeeks, [Online]. Available: https://www.geeksforgeeks.org/mvc-framework-introduction/. [Accessed: 15 - May - 2025].

19

Appendices

19. Appendices

Appendix 1 Public Endpoint without Authentication

1.A User Signup

```
# REQUEST BODY
{
    "emailId": "johnDoe@gc.ac.in",
    "firstName": "Jhon",
    "lastName": "Doe",
    "dateOfBirth": "1995-12-05",
    "password": "JohnDoe#1997",
    "country": "UK",
    "city": "Leicester",
    "contactNumber": "+44 9988998899"
}
```

```
# RESPONSE BODY
{
    "responseStatusCode": "CREATED",
    "responseMessage": "Signup successful for: johnDoe@gc.ac.in"
}
```

1.B User Login

```
USER LOGIN

# REQUEST BODY
{
    "username": "johnDoe2@gc.ac.in",
    "password": "JohnDoe#1997"
}
```

```
# RESPONSE BODY
{
    "accessToken":
"eyJhbGciOiJIUzI1NiJ9.eyJyb2xlcyI6WyJST0xFX1VTRVIiXSwic3ViIjoiam9obkRvZTJAZ2MuYWMuaW4iLCJpYXQiOj
E3NDY0Mjc4ODgsImV4cCI6MTc0NjQ4Nzg4OH0.x_MbMvcLNS0K60cyUGYBXKr7tXO6nN2P-NKMBqNoMGo"
}
```

```
ADMIN LOGIN

# REQUEST BODY
{
    "username": "admin@gc.ac.in",
    "password": "2025%generation-connect",
    "isAdmin": true
}
```

```
# RESPONSE BODY
{
    "accessToken":
"eyJhbGciOiJIUzI1NiJ9.eyJyb2xlcyI6WyJST0xFX0FETUlOIl0sInN1YiI6ImFkbWluQGdjLmFjLmluIiwiaWF0IjoxNz
Q2NDI4NTA5LCJleHAiOjE3NDY0ODg1MDl9.EpfefRZcMbQE6wdcwxIMQL61yLY6BwowmwOUcyMoQu8"
}
```

1.C User Logout

```
 Query param: user with userId
```

```
# RESPONSE BODY
{
```

```
    "responseStatusCode": "OK",
    "responseMessage": "Logout successful"
}
```

1.D Accept Sent invite

```
# REQUEST BODY
"76e007c3-cb76-4672-bc69-25c71a152bde"

# RESPONSE BODY
Invite accepted successfully.
```

Appendix 2 Administrative APIs

2.A User Metrics

```
# RESPONSE BODY
{
    "pendingInvites": 0,
    "averagePostsPerTree": 5,
    "familyTreeOwners": 1,
    "averageCommentsPerTree": 0,
    "totalNonAdminUsers": 4,
    "appAdmins": 1,
    "publicFamilyTrees": 0,
    "totalPosts": 5,
    "totalFamilyTrees": 1,
    "totalComments": 0,
    "recentFamilyTreesLast30Days": 1,
    "privateFamilyTrees": 1,
    "activeUsersLast30Days": 2
}
```

2.B Audit Tree List

```
# RESPONSE BODY
[
    {
        "familyTreeId": 23,
        "familyTreeName": "test1",
        "memberUserId": "johnDoe@gc.ac.in",
        "familyTreeDescription": "",
        "familyTreeAccessLevel": "Private"
    }
]
```

Appendix 3 Family Tree Generation

3.A Create family Tree

```
# REQUEST BODY
{
    "familyTreeName": "Jon Doe Family Tree",
    "familyTreeDescription": "Jon Doe Family Tree description details",
    "familyTreeAccessLevel": "PRIVATE",
    "memberUserId": "johnDoe@gc.ac.in"
}
```

```
# RESPONSE BODY
{
    "familyTreeId": 25,
    "familyTreeName": "Jon Doe Family Tree",
    "memberUserId": "johnDoe@gc.ac.in",
    "familyTreeDescription": "Jon Doe Family Tree description details",
    "familyTreeAccessLevel": "Private"
}
```

3.B Add/Update Person

```
# REQUEST BODY
{
    "id": "c98305ea-d74b-43ff-9b18-fe93ce5ab0d3",
    "rels": {},
    "data": {
        "firstName": "Jon",
        "lastName": "Done",
        "gender": "M",
        "label": "",
        "desc": "",
        "birthdate": "",
        "avatar": ""
    },
    "main": true
}
```

```
# RESPONSE BODY
{
    "personId": "c98305ea-d74b-43ff-9b18-fe93ce5ab0d3",
    "familyTreeId": 24,
    "firstName": "Jon",
    "lastName": "Done",
    "birthdate": null,
    "gender": "M",
    "firstNode": null,
    "userId": null,
    "label": "",
    "desc": "",
    "avatar": "",
    "properties": null,
    "createdBy": null,
    "createdDate": null,
    "lastModifiedBy": "johnDoe@gc.ac.in",
    "lastModifiedDate": "2025-05-05T07:47:18.224149741Z",
    "version": 1
}
```

3.C Get family Tree Member

```
[
    {
        "id": "94b01b62-6906-4d99-a210-3f9d8ab51520",
        "familyTreeId": 1,
        "data": {
            "firstName": "John",
            "lastName": "Doe",
            "gender": "M",
            "lastModifiedBy": "johnDoe@gc.ac.in",
            "label": "",
            "desc": "",
            "avatar": ""
        },
        "relationship": {
            "spouses": [
                "cf0b16d8-1cc2-4629-93ab-edbdf25cd84a"
            ]
```

```
        }
    },
    {
        "id": "cf0b16d8-1cc2-4629-93ab-edbdf25cd84a",
        "familyTreeId": 1,
        "data": {
            "firstName": "Alice",
            "lastName": "Doe",
            "gender": "F",
            "lastModifiedBy": "johnDoe@gc.ac.in",
            "label": "",
            "desc": "",
            "avatar": ""
        },
        "relationship": {
            "spouses": [
                "94b01b62-6906-4d99-a210-3f9d8ab51520"
            ]
        }
    }
]
```


3.D List all Users trees


```
# RESPONSE BODY
[
    {
        "familyTreeId": 23,
        "familyTreeName": "test1",
        "memberUserId": "johnDoe@gc.ac.in",
        "familyTreeDescription": "",
        "familyTreeAccessLevel": "Private"
    }
]
```


3.E Delete Family Tree

```
# RESPONSE BODY

Family tree deleted successfully.
```

3.F Delete person

```
# RESPONSE BODY

Person with ID e24eeaae-bbff-42f6-afcd-fb31c0d2f01c deleted successfully.
```

3.G Chat AI

```
# Request Body

"Tell me about the history of Alice Smith's family."

# RESPONSE BODY

"Alice Smith's family dates back to the early 19th century..."
```

Appendix 4 Post and Comment Sharing with Media (Family & Individual Profiles)

4.A Add Post

```
# REQUEST BODY
{
    "familyTreeId": 24,
    "postContent": "post for Jon Doe Family Tree.",
    "createdBy": "johnDoe@gc.ac.in",
    "createdDate": "2025-05-05T07:51:18"
}
```

```
# RESPONSE BODY
{
    "id": 1003,
    "personId": null,
    "familyTreeId": 24,
    "postContent": "post for Jon Doe Family Tree.",
    "createdBy": "johnDoe@gc.ac.in",
    "createdDate": "2025-05-05T07:51:18",
    "lastModifiedBy": null,
    "lastModifiedDate": null,
    "userProfilePost": false
}
```

4.B Add Comment

```
# REQUEST BODY
{
    "postId": 1,
    "commentContent": "Congratulation Alice!!",
    "createdBy": "johnDoe@gc.ac.in",
    "createdDate": "2025-05-05T20:48:32"
}
```

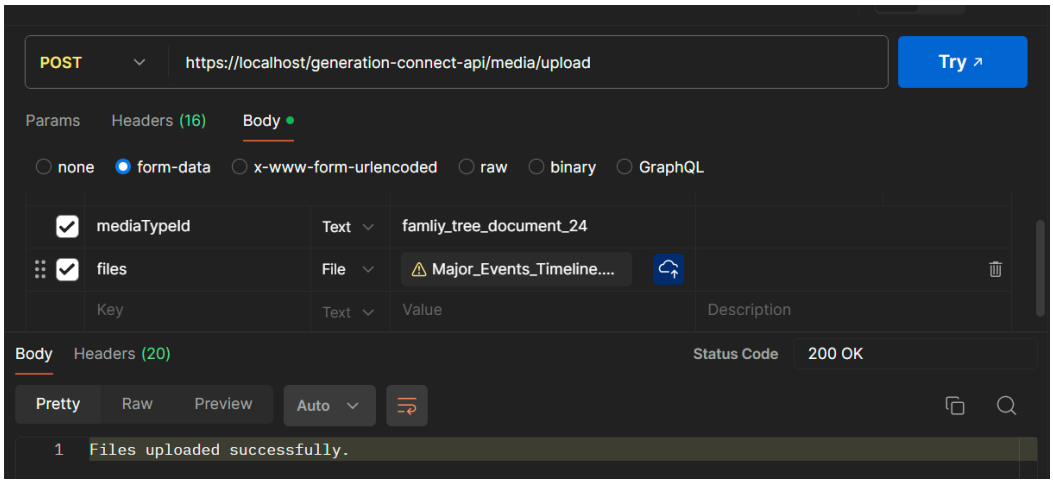
```
# RESPONSE BODY
{
    "id": 3,
    "postId": 1,
    "commentContent": "Congratulation Alice!!",
    "createdBy": "johnDoe@gc.ac.in",
    "createdDate": "2025-05-05T20:48:32",
    "lastModifiedBy": null,
    "lastModifiedDate": null
}
```

4.C Get Comments

```
# RESPONSE BODY
[
    {
        "id": 3,
        "postId": 1,
```

```
        "commentContent": "Congratulation Alice!!",
        "createdBy": "johnDoe@gc.ac.in",
        "createdDate": "2025-05-05T20:48:32",
        "lastModifiedBy": null,
        "lastModifiedDate": null
    }
]
```


4.D Upload Media



4.E Get Media by type


```
# RESPONSE BODY
[
    {
        "id": 1,
        "fileName": "0a9447c8-ee24-4be7-a264-ec243f802432_Screenshot from 2025-05-05 01-14-
26.png",
        "fileType": "image/png",
        "fileUrl": "/media/files/family_profile_post_3/0a9447c8-ee24-4be7-a264-
ec243f802432_Screenshot from 2025-05-05 01-14-26.png",
        "fileSize": 47076,
        "uploadedBy": "UploaderName",
        "uploadedAtFormatted": "2025-05-05 20:57:28",
        "description": null
    }
]
```


Appendix 5 Family tree Profile Wall

5.A Get Family Posts


```
# RESPONSE BODY

[
    {
        "id": 1002,
        "personId": null,
        "familyTreeId": 24,
```

```
        "postContent": "post for Jon Doe Family Tree.",
        "createdBy": "johnDoe@gc.ac.in",
        "createdDate": "2025-05-05T07:51:18",
        "lastModifiedBy": null,
        "lastModifiedDate": null,
        "userProfilePost": false
    }
]
```


5.B Get Family Tree Recent Activities


```
# RESPONSE BODY

[
    {
        "id": 708,
        "familyTreeId": 24,
        "tableName": "Person",
        "actionType": "UPDATE",
        "recordId": "c98305ea-d74b-43ff-9b18-fe93ce5ab0d3",
        "recordName": "Jon Done",
        "createdBy": "johnDoe@gc.ac.in",
        "createdDate": "2025-05-05T07:47:18.596272"
    },
    {
        "id": 707,
        "familyTreeId": 24,
        "tableName": "Person",
        "actionType": "UPDATE",
        "recordId": "c98305ea-d74b-43ff-9b18-fe93ce5ab0d3",
        "recordName": "Jon Done",
        "createdBy": "johnDoe@gc.ac.in",
        "createdDate": "2025-05-05T07:46:11.079202"
    },
    {
        "id": 702,
        "familyTreeId": 24,
        "tableName": "FamilyTree",
        "actionType": "ADD",
        "recordId": "24",
        "recordName": "Jon Doe Family Tree",
        "createdBy": "johnDoe@gc.ac.in",
        "createdDate": "2025-05-05T07:40:49.395731"
    }
]
```


Appendix 6 Individual Profile of Person in Family Tree

6.A Get Person Posts


```
# RESPONSE BODY

[
    {
        "id": 1004,
        "personId": "e9724ae9-e352-4a43-8d31-7adf46236435",
        "familyTreeId": null,
        "postContent": "Post for user profile",
        "createdBy": "johnDoe@gc.ac.in",
```

```
        "createdDate": "2025-05-05T08:02:48",
        "lastModifiedBy": null,
        "lastModifiedDate": null,
        "userProfilePost": false
    },
    {
        "id": 1005,
        "personId": "e9724ae9-e352-4a43-8d31-7adf46236435",
        "familyTreeId": null,
        "postContent": "Post for user profile",
        "createdBy": "johnDoe@gc.ac.in",
        "createdDate": "2025-05-05T08:02:48",
        "lastModifiedBy": null,
        "lastModifiedDate": null,
        "userProfilePost": false
    }
]
```


Appendix 7 Family Tree Historical Research

7.A Search Historical Records

```
# Request parameter
Query param: searchQuery
```


```
# RESPONSE BODY

{
    "records": [
        {
            "altName": "",
            "places": [],
            "corpBodies": [],
            "taxonomies": [],
            "formerReferenceDep": "",
            "formerReferencePro": "",
            "heldBy": [
                "Somerset Heritage Centre (South West Heritage Trust)"
            ],
            "context": "SOMERSET ARCHAEOLOGICAL AND NATURAL HISTORY SOCIETY COLLECTION.North
Petherton deeds etc.North Petherton.",
            "content": "",
            "urlParameters": null,
            "department": "",
            "note": "",
            "adminHistory": "",
            "arrangement": "",
            "mapDesignation": "",
            "mapScale": "",
            "physicalCondition": "",
            "catalogueLevel": 9,
            "openingDate": "",
            "closureStatus": "",
            "closureType": "",
            "closureCode": "",
            "documentType": "",
            "coveringDates": "1644",
            "description": "20 Chas. I, June I. Alexander Bullpan of Mooreland in the parish of
Northpetherton, yeoman, to William Smyth or Perry of the same, husbandman, Hester, his wife, &
John Smyth or Perry their son, for £52 OF cottage & 3 acres of land & pasture adjoining at
Mooreland on the Northeast side of Northmoore in the parish of Northpetherton with the
appurtenances, with common of pasture in North moore, Kingscliffe & Hethfeilde belonging, all
late in the occupation of George Coombe & Edethe, his wife; for lives of 3 lessees. Rent 6/-
p.a. payable at Michaelmas & Annunciation with suit to the Courts of the said Alexander at
```

```
Mooreland & doing of repairs in housing, hedging, ditching, etc., as required. Heriot 20/-.
Proviso as to non-payment of heriot if deaths are not successive. Covenants as a \"houseboote
ploughboote fyreboote and hedgeboote\" to be taken by lessees without waste making; peaceable
possession; saving harmless against George & Edethe Coombe or claimants under them; non-payment
of rents service. Constitution of Mathew Yeondole & Iohn Chapell of Northpetherton Alexander's
Attornies to take & deliver \"season\". Mark of William Smyth on fold-tag. Traces of red wax
adhering to parchment tag. Parchment, indented. Endorsements (1) as to execution; witnesses,
Mathew Yeondole, John Chappell, the mark of John Smyth or Perry, the elder; (2) \"Katherine the
wife of Tho: Knap/Katherine theire daughter / Jon theire sonne. / a bond in 100 s for quiet
enjoymt [See also DD\\SAS\\H/47/3/6]",
            "endDate": "31/12/1644",
            "numEndDate": 16441231,
            "numStartDate": 16440101,
            "startDate": "01/01/1644",
            "id": "5a002f03-af5f-49a1-98cb-9aae7a71a831",
            "reference": "DD\\SAS\\H/47/3/5",
            "score": 7.40403652,
            "source": "600",
            "title": "COUNTER PART LEASE"
        },
….
]
}
```


7.B Tag Record for Family Tree


```
# REQUEST BODY

{
    "records": [
        {
            "altName": "",
            "places": [],
            "corpBodies": [],
            "taxonomies": [],
            "formerReferenceDep": "",
            "formerReferencePro": "",
            "heldBy": [
                "Somerset Heritage Centre (South West Heritage Trust)"
            ],
            "context": "SOMERSET ARCHAEOLOGICAL AND NATURAL HISTORY SOCIETY COLLECTION.North
Petherton deeds etc.North Petherton.",
            "content": "",
            "urlParameters": null,
            "department": "",
            "note": "",
            "adminHistory": "",
            "arrangement": "",
            "mapDesignation": "",
            "mapScale": "",
            "physicalCondition": "",
            "catalogueLevel": 9,
            "openingDate": "",
            "closureStatus": "",
            "closureType": "",
            "closureCode": "",
            "documentType": "",
            "coveringDates": "1644",
            "description": "20 Chas. I, June I. Alexander Bullpan of Mooreland in the parish of
Northpetherton, yeoman, to William Smyth or Perry of the same, husbandman, Hester, his wife, &
John Smyth or Perry their son, for £52 OF cottage & 3 acres of land & pasture adjoining at
Mooreland on the Northeast side of Northmoore in the parish of Northpetherton with the
appurtenances, with common of pasture in North moore, Kingscliffe & Hethfeilde belonging, all
late in the occupation of George Coombe & Edethe, his wife; for lives of 3 lessees. Rent 6/-
p.a. payable at Michaelmas & Annunciation with suit to the Courts of the said Alexander at
Mooreland & doing of repairs in housing, hedging, ditching, etc., as required. Heriot 20/-.
```

```
Proviso as to non-payment of heriot if deaths are not successive. Covenants as a \"houseboote
ploughboote fyreboote and hedgeboote\" to be taken by lessees without waste making; peaceable
possession; saving harmless against George & Edethe Coombe or claimants under them; non-payment
of rents service. Constitution of Mathew Yeondole & Iohn Chapell of Northpetherton Alexander's
Attornies to take & deliver \"season\". Mark of William Smyth on fold-tag. Traces of red wax
adhering to parchment tag. Parchment, indented. Endorsements (1) as to execution; witnesses,
Mathew Yeondole, John Chappell, the mark of John Smyth or Perry, the elder; (2) \"Katherine the
wife of Tho: Knap/Katherine theire daughter / Jon theire sonne. / a bond in 100 s for quiet
enjoymt [See also DD\\SAS\\H/47/3/6]",
            "endDate": "31/12/1644",
            "numEndDate": 16441231,
            "numStartDate": 16440101,
            "startDate": "01/01/1644",
            "id": "5a002f03-af5f-49a1-98cb-9aae7a71a831",
            "reference": "DD\\SAS\\H/47/3/5",
            "score": 7.40403652,
            "source": "600",
            "title": "COUNTER PART LEASE"
        },
….
]
}
```


```
# RESPONSE BODY
Record already tagged to this family tree.
```

7.C Get Tagged Research Records


```
# RESPONSE BODY
{
    "records": [
        {
            "altName": "",
            "places": [],
            "corpBodies": [],
            "taxonomies": [],
            "formerReferenceDep": "",
            "formerReferencePro": "",
            "heldBy": [
                "Somerset Heritage Centre (South West Heritage Trust)"
            ],
            "context": "SOMERSET ARCHAEOLOGICAL AND NATURAL HISTORY SOCIETY COLLECTION.North
Petherton deeds etc.North Petherton.",
            "content": "",
            "urlParameters": null,
            "department": "",
            "note": "",
            "adminHistory": "",
            "arrangement": "",
            "mapDesignation": "",
            "mapScale": "",
            "physicalCondition": "",
            "catalogueLevel": 9,
            "openingDate": "",
            "closureStatus": "",
            "closureType": "",
            "closureCode": "",
            "documentType": "",
            "coveringDates": "1644",
            "description": "20 Chas. I, June I. Alexander Bullpan of Mooreland in the parish of
Northpetherton, yeoman, to William Smyth or Perry of the same, husbandman, Hester, his wife, &
John Smyth or Perry their son, for £52 OF cottage & 3 acres of land & pasture adjoining at
Mooreland on the Northeast side of Northmoore in the parish of Northpetherton with the
appurtenances, with common of pasture in North moore, Kingscliffe & Hethfeilde belonging, all
late in the occupation of George Coombe & Edethe, his wife; for lives of 3 lessees. Rent 6/-
```

```
p.a. payable at Michaelmas & Annunciation with suit to the Courts of the said Alexander at
Mooreland & doing of repairs in housing, hedging, ditching, etc., as required. Heriot 20/-.
Proviso as to non-payment of heriot if deaths are not successive. Covenants as a \"houseboote
ploughboote fyreboote and hedgeboote\" to be taken by lessees without waste making; peaceable
possession; saving harmless against George & Edethe Coombe or claimants under them; non-payment
of rents service. Constitution of Mathew Yeondole & Iohn Chapell of Northpetherton Alexander's
Attornies to take & deliver \"season\". Mark of William Smyth on fold-tag. Traces of red wax
adhering to parchment tag. Parchment, indented. Endorsements (1) as to execution; witnesses,
Mathew Yeondole, John Chappell, the mark of John Smyth or Perry, the elder; (2) \"Katherine the
wife of Tho: Knap/Katherine theire daughter / Jon theire sonne. / a bond in 100 s for quiet
enjoymt [See also DD\\SAS\\H/47/3/6]",
            "endDate": "31/12/1644",
            "numEndDate": 16441231,
            "numStartDate": 16440101,
            "startDate": "01/01/1644",
            "id": "5a002f03-af5f-49a1-98cb-9aae7a71a831",
            "reference": "DD\\SAS\\H/47/3/5",
            "score": 7.40403652,
            "source": "600",
            "title": "COUNTER PART LEASE"
        },
….
]
}
```


Appendix 8 Manage users in the Family Tree

8.A Add Existing User to the family tree

```
# REQUEST BODY
{
    "familyTreeId": 24,
    "addedMemberUserId": "johnDoe2@gc.ac.in",
    "role": "Viewer",
    "memberAddedBy": "johnDoe@gc.ac.in"
}
```

```
# RESPONSE BODY
User added successfully
```

8.B Send Invite

```
# REQUEST BODY
{
    "familyTreeId": 24,
    "invitedEmail": "johnDoe2   @gc.ac.in",
    "role": "Owner",
    "inviterUserId": "johnDoe@gc.ac.in"
}
```

```
# RESPONSE BODY
Invite sent successfully..
```

8.C Update Role

```
# REQUEST BODY
{
```

```
    "role": "Contributor",
    "familyTreeId": 24,
    "addedMemberUserId": "johnDoe2@gc.ac.in",
    "memberAddedBy": "johnDoe@gc.ac.in"
}
```

```
# RESPOSE BODY
User added successfully
```

8.D Remove User

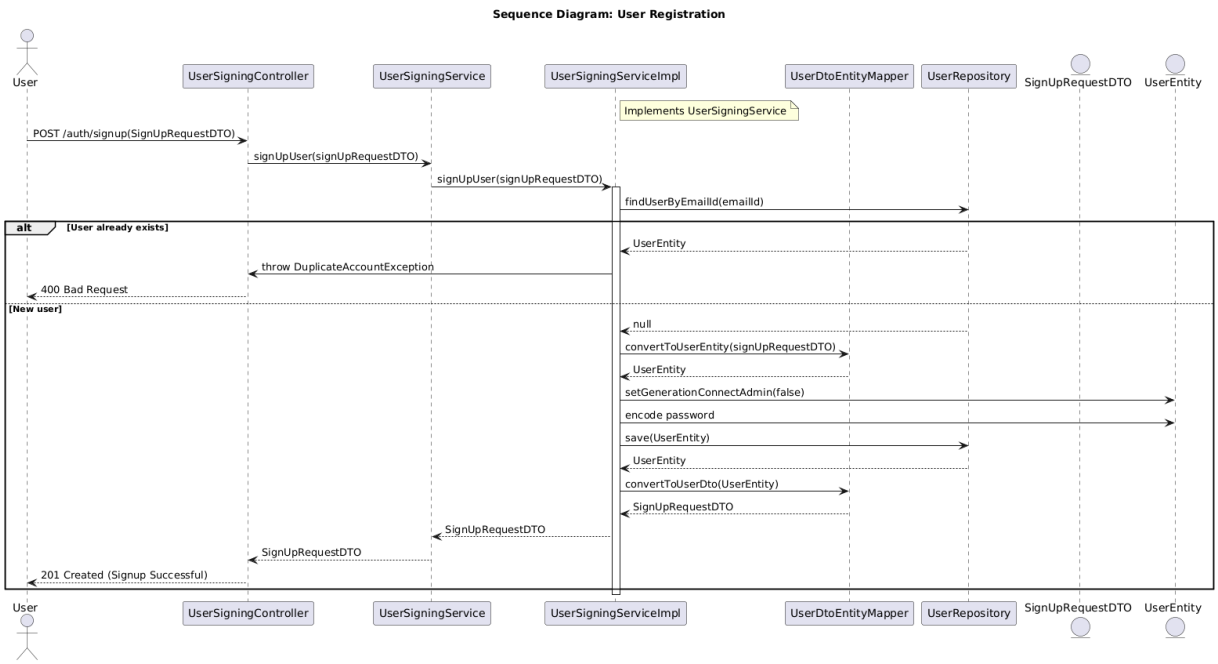
```
# REQUEST BODY
{
    "memberAddedBy": "johnDoe@gc.ac.in",
    "familyTreeId": 24,
    "addedMemberUserId": "johnDoe2@gc.ac.in"
}
```

```
# RESPONSE BODY
User removed successfully
```

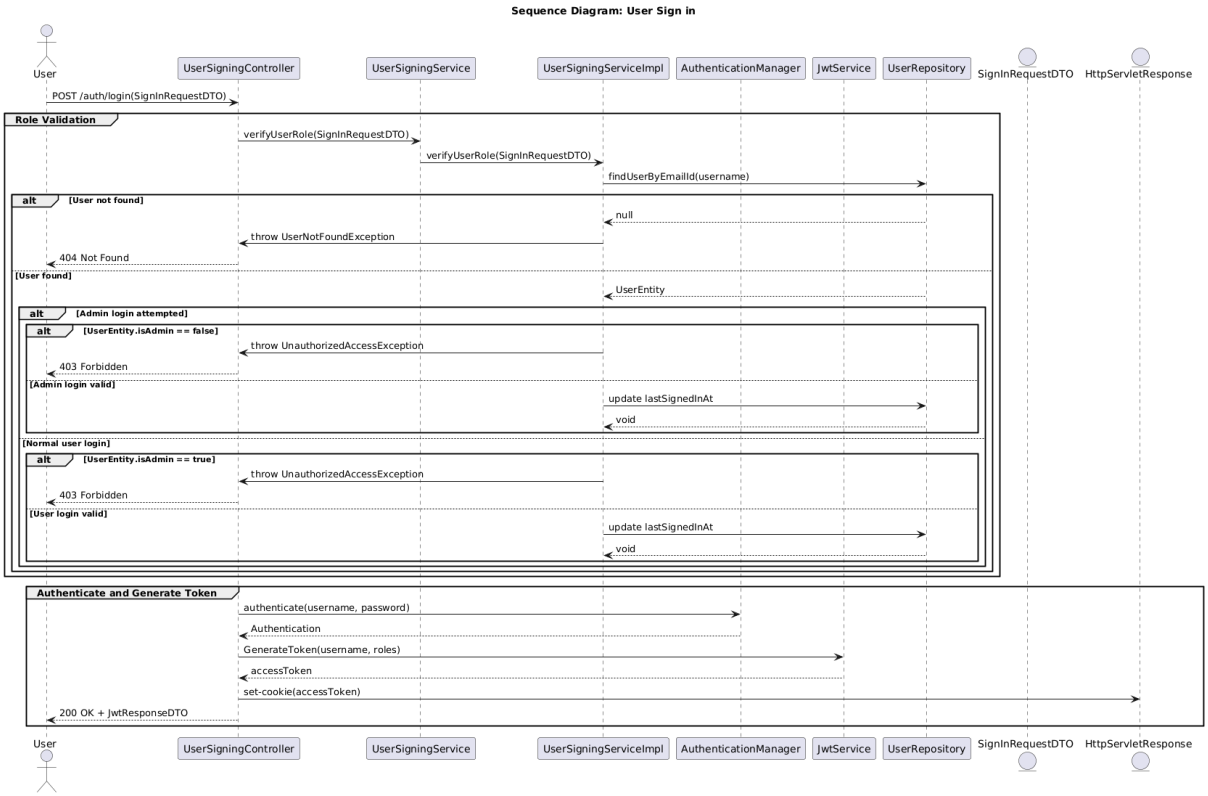
8.E Get user

```
# RESPONSE BODY
[
    {
        "memberAddedBy": "johnDoe@gc.ac.in",
        "familyTreeId": 24,
        "addedMemberUserId": "johnDoe@gc.ac.in",
        "role": "Owner"}]
```

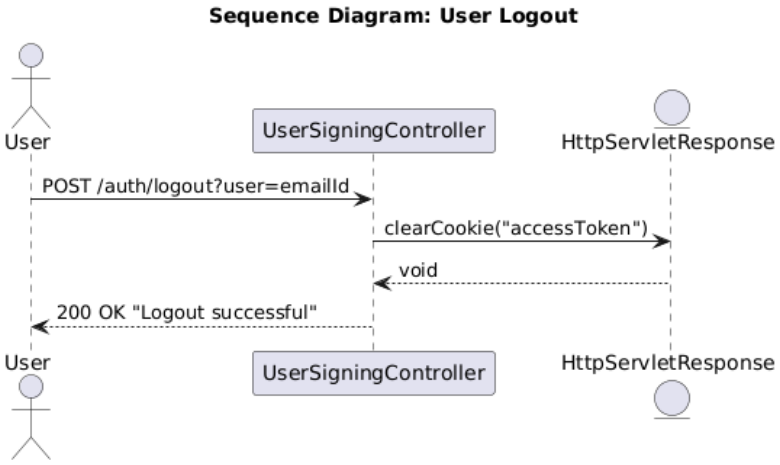
Appendix 9 Sequence Diagram: User Signup



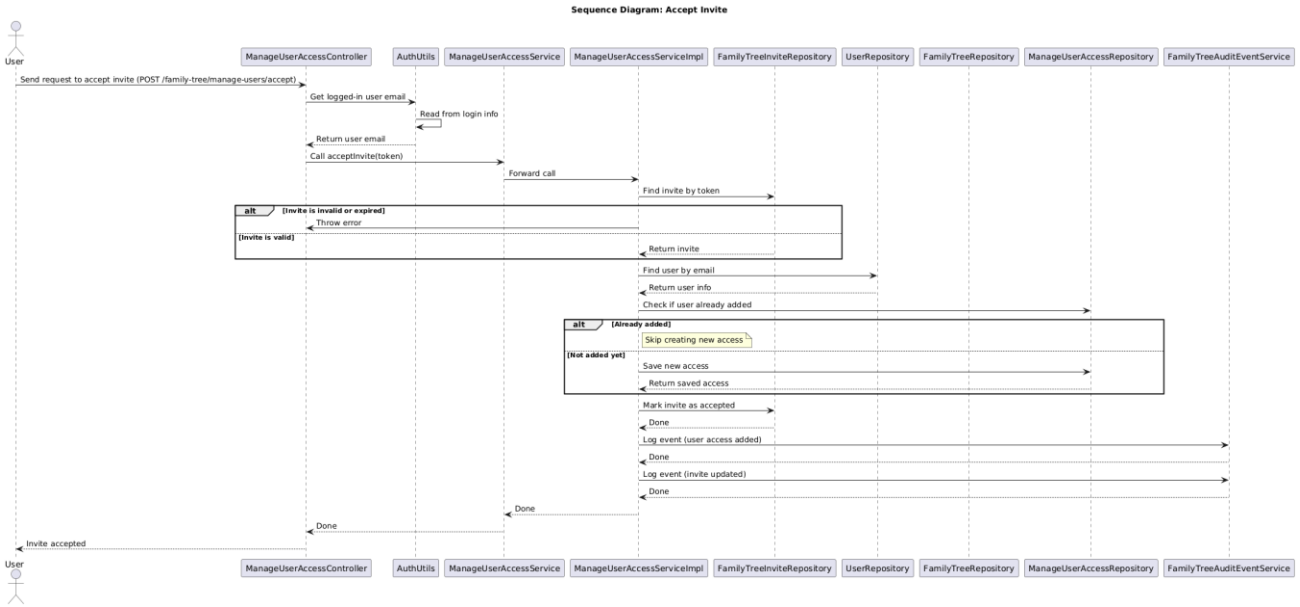
Appendix 10 Sequence Diagram: User Login



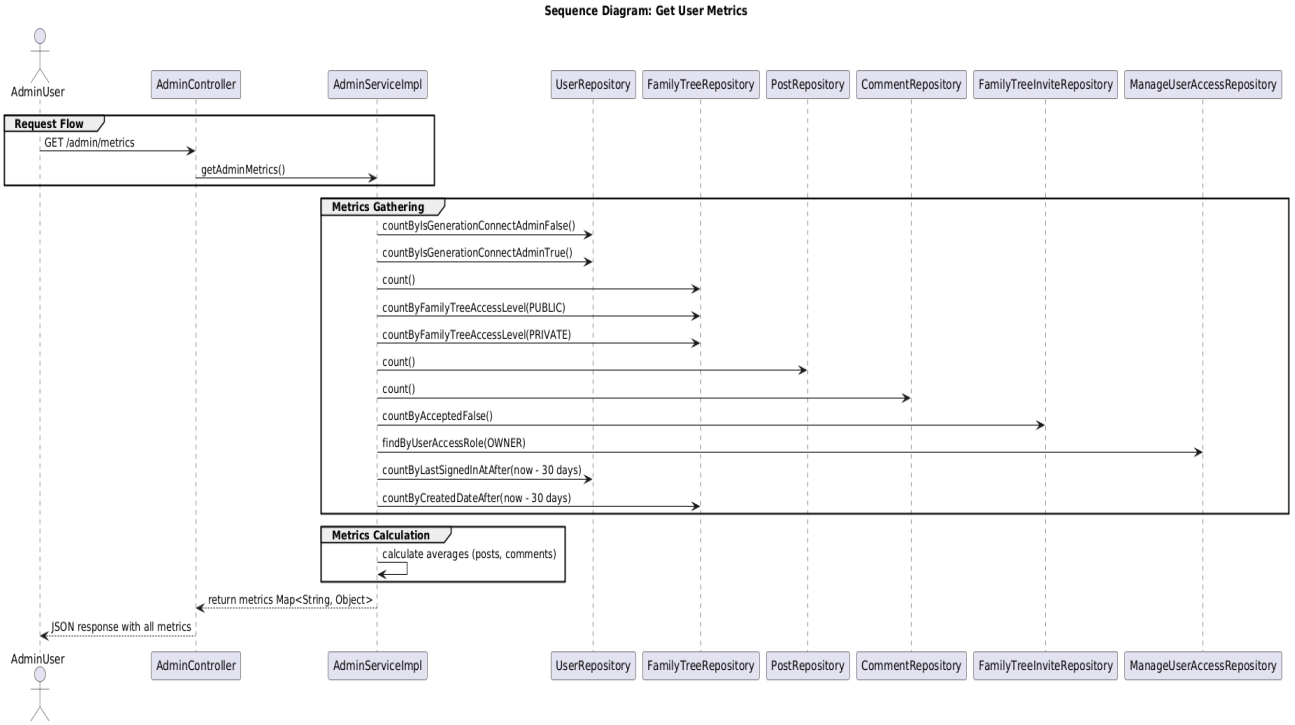
Appendix 11 Sequence Diagram: User Logout



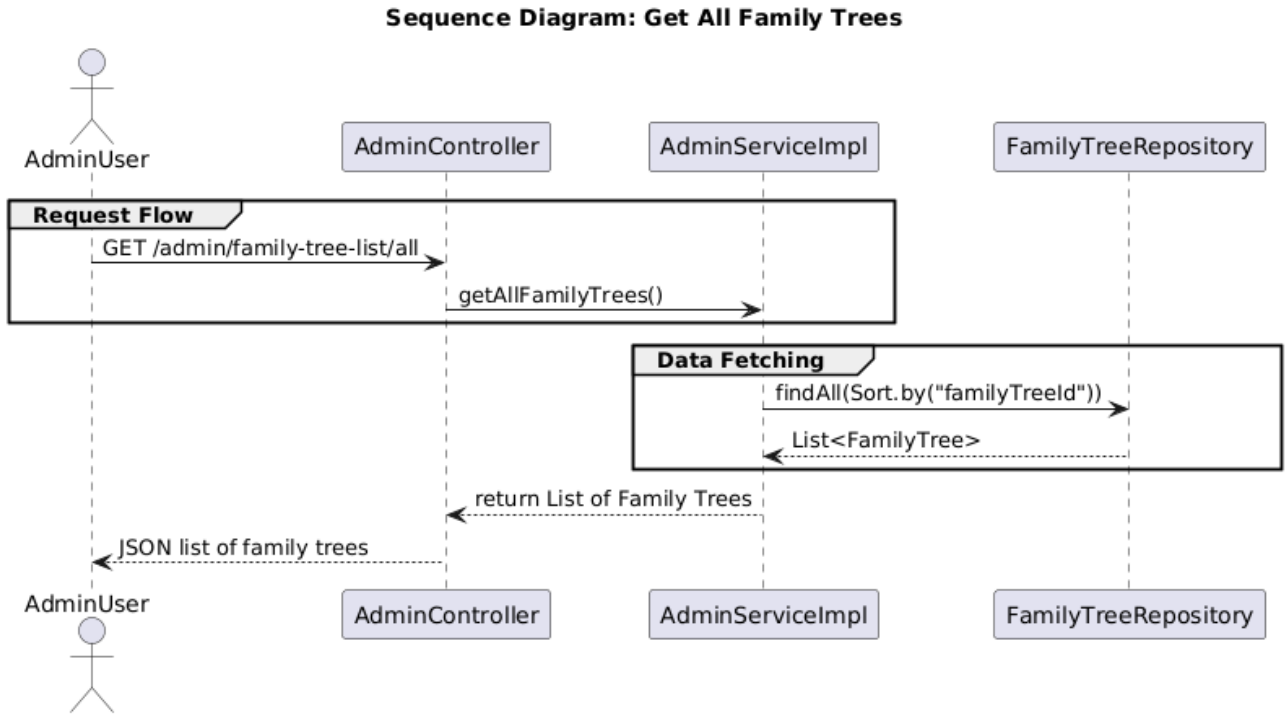
Appendix 12 Sequence Diagram: Accept Sent Invite



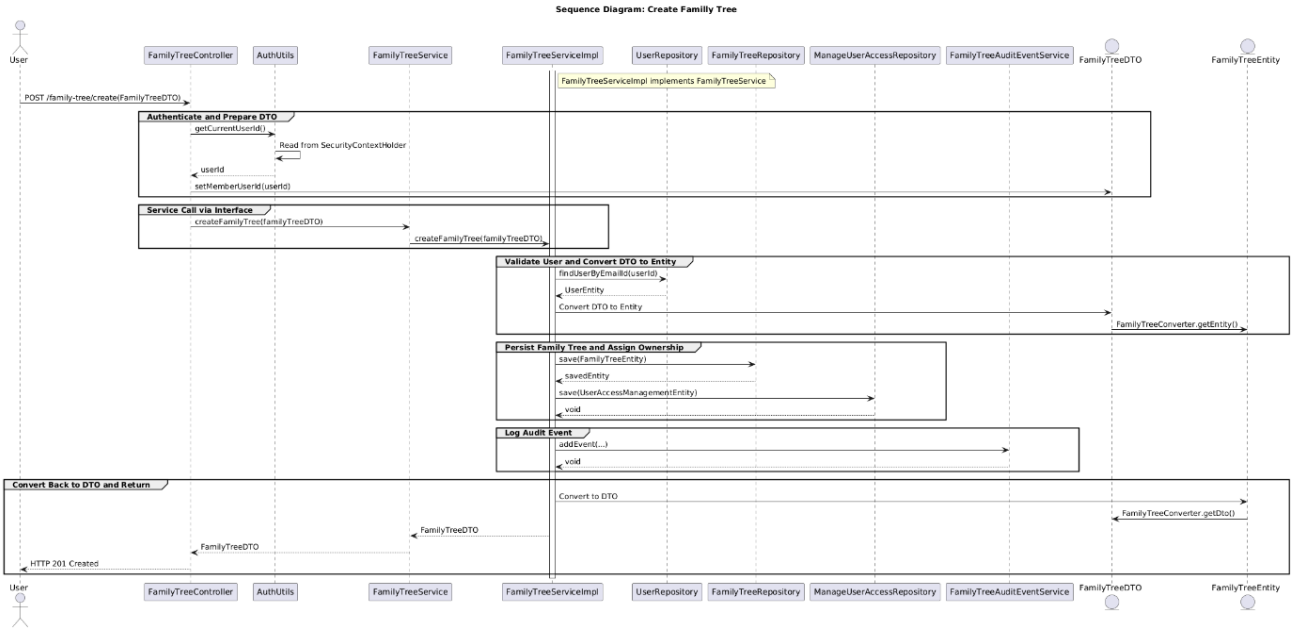
Appendix 13 Sequence Diagram: User Metrics



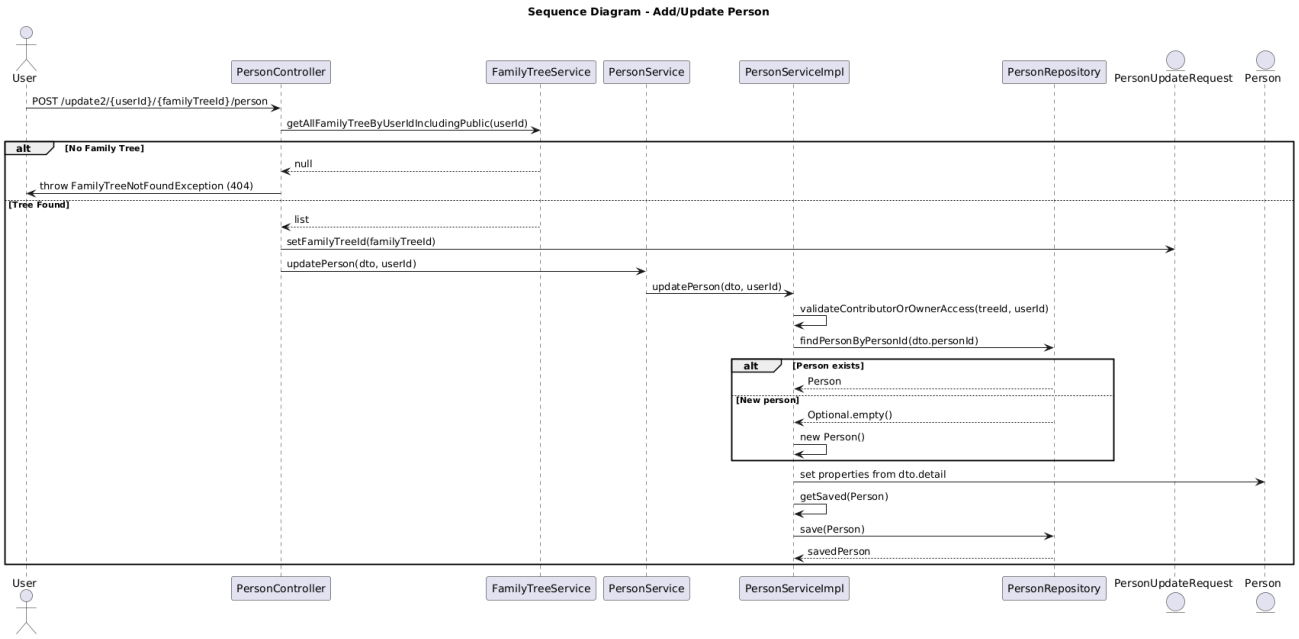
Appendix 14 Sequence Diagram: Audit Tree List



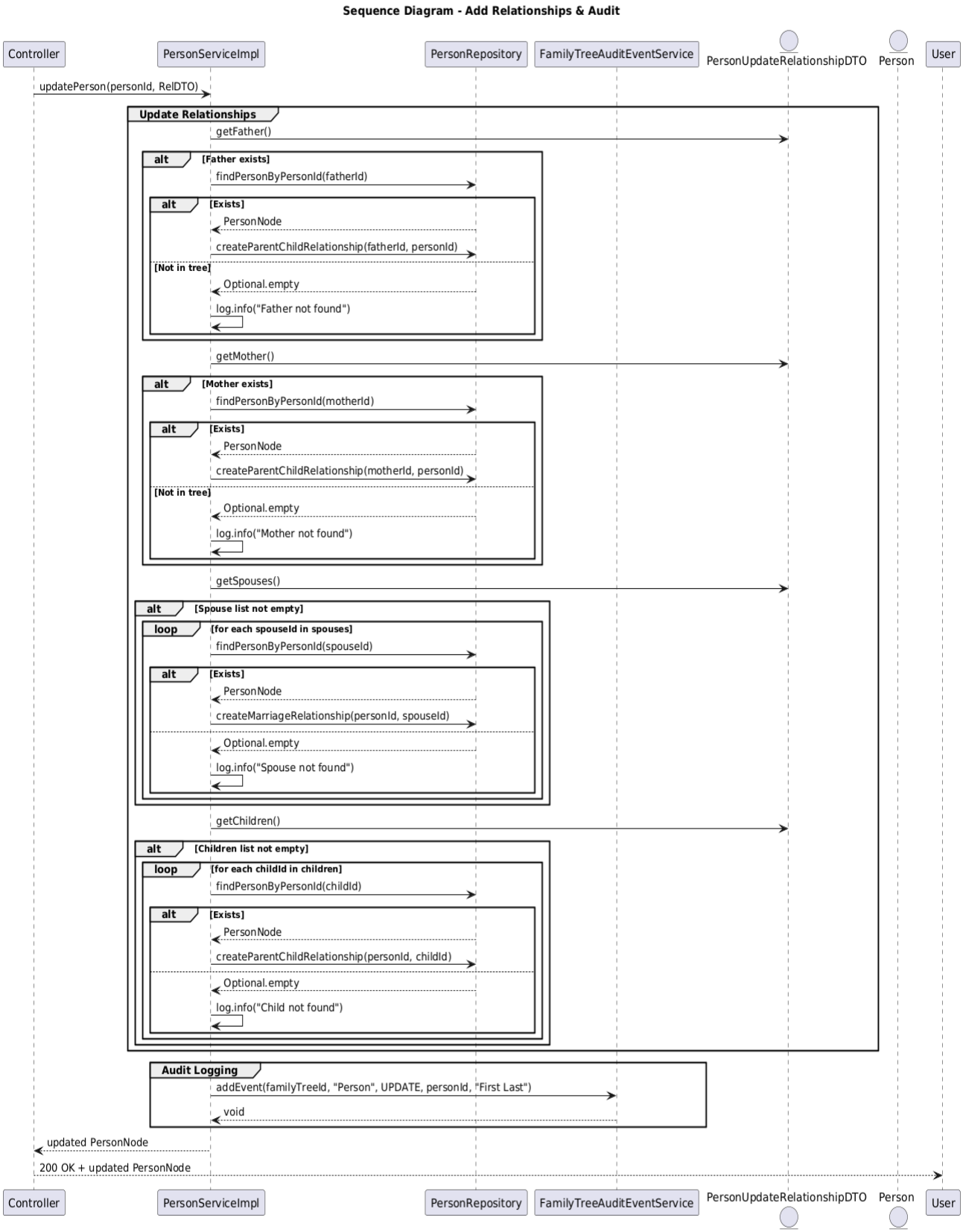
Appendix 15 Sequence Diagram: Create Family Tree



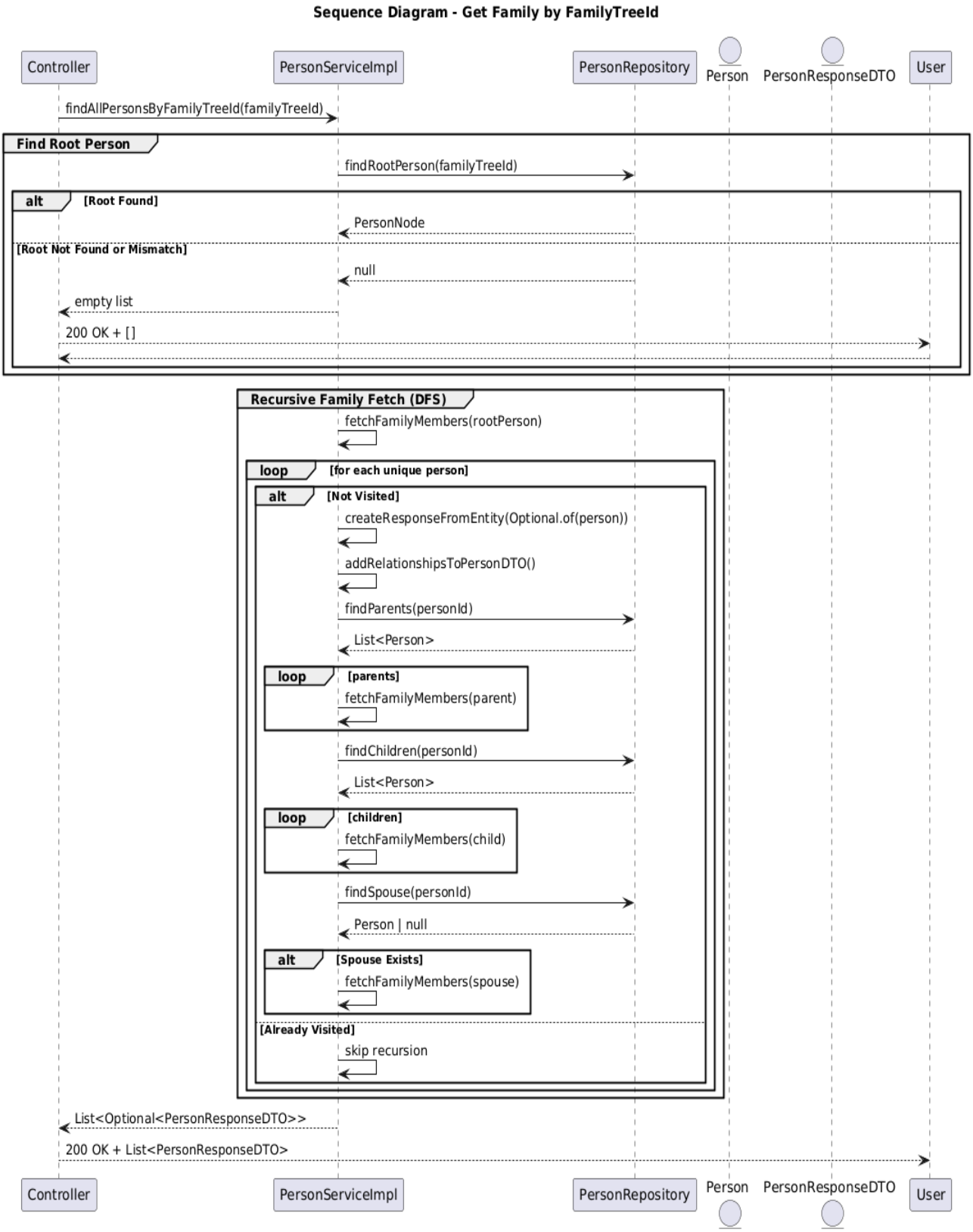
Appendix 16 Sequence Diagram: Add/Update Person



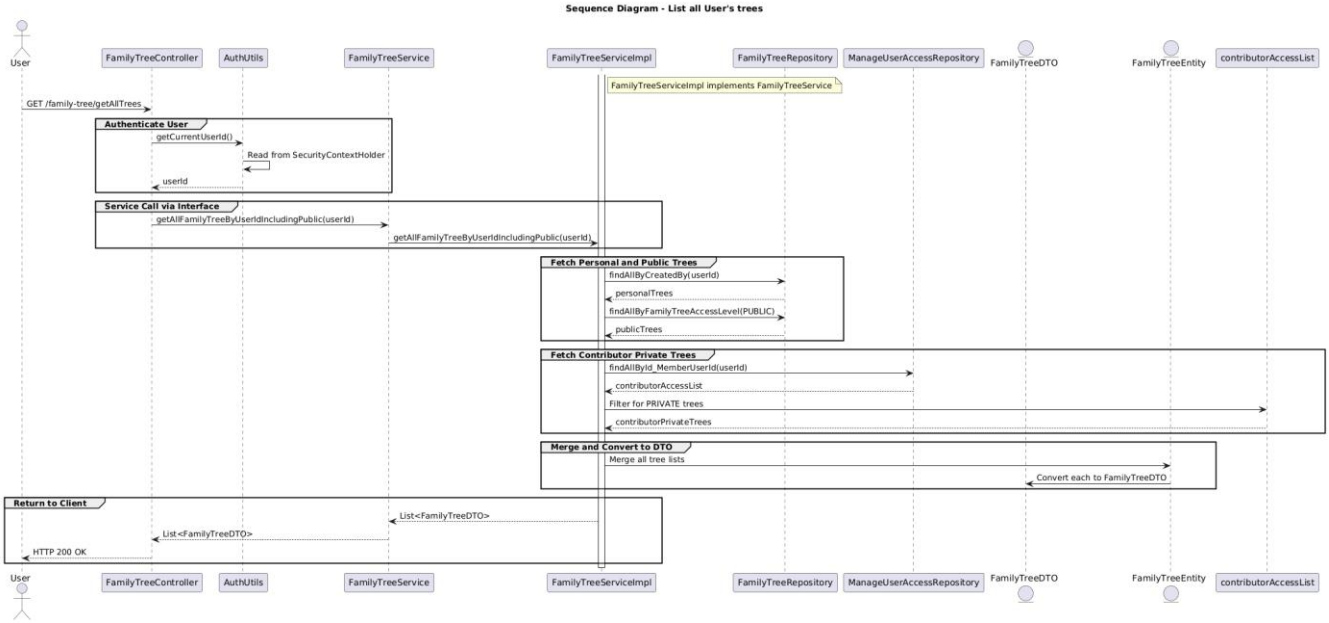
Continued flow of relationships in next page…


Sequence Diagram - Add Relationships & Audit
Controller
PersonServiceImpl
PersonRepository
FamilyTreeAuditEventService
PersonUpdateRelationshipDTO
Person
User
updatePerson(personId, RelDTO)
Update Relationships
getFather()
alt [Father exists]
findPersonByPersonId(fatherId)
alt [Exists]
PersonNode
createParentChildRelationship(fatherId, personId)
[Not in tree]
Optional.empty
log.info("Father not found")
getMother()
alt [Mother exists]
findPersonByPersonId(motherId)
alt [Exists]
PersonNode
createParentChildRelationship(motherId, personId)
[Not in tree]
Optional.empty
log.info("Mother not found")
getSpouses()
alt [Spouse list not empty]
loop [for each spouseId in spouses]
findPersonByPersonId(spouseId)
alt [Exists]
PersonNode
createMarriageRelationship(personId, spouseId)
Optional.empty
log.info("Spouse not found")
getChildren()
alt [Children list not empty]
loop [for each childId in children]
findPersonByPersonId(childId)
alt [Exists]
PersonNode
createParentChildRelationship(personId, childId)
Optional.empty
log.info("Child not found")
Audit Logging
addEvent(familyTreeId, "Person", UPDATE, personId, "First Last")
void
updated PersonNode
200 OK + updated PersonNode

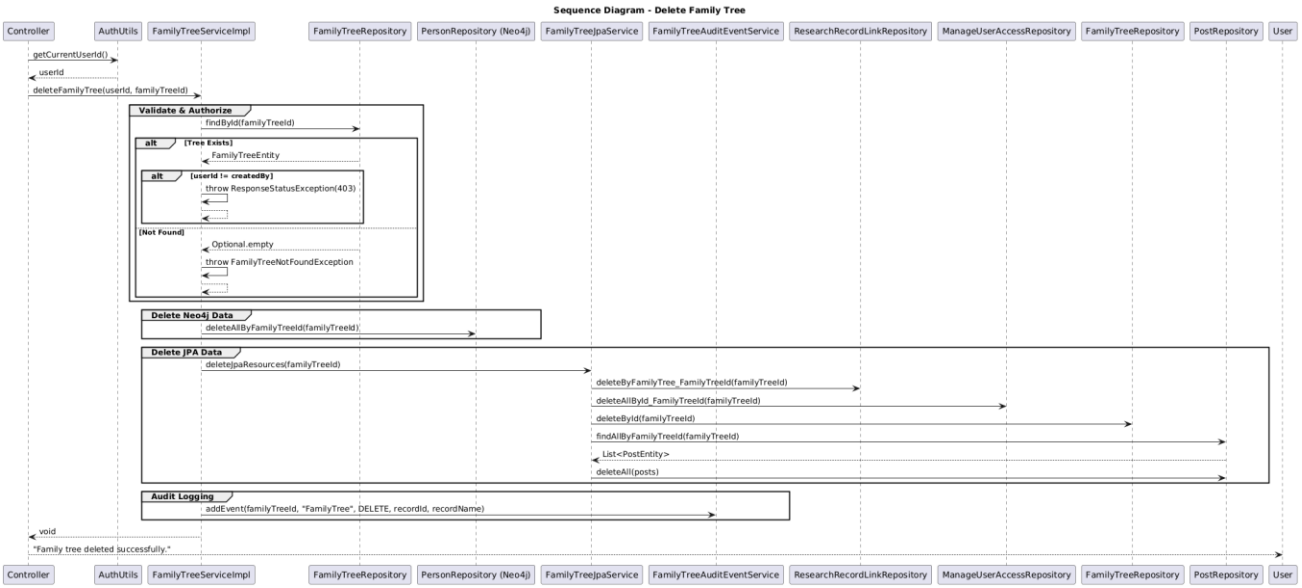
Appendix 17 Sequence Diagram: Get family Tree member



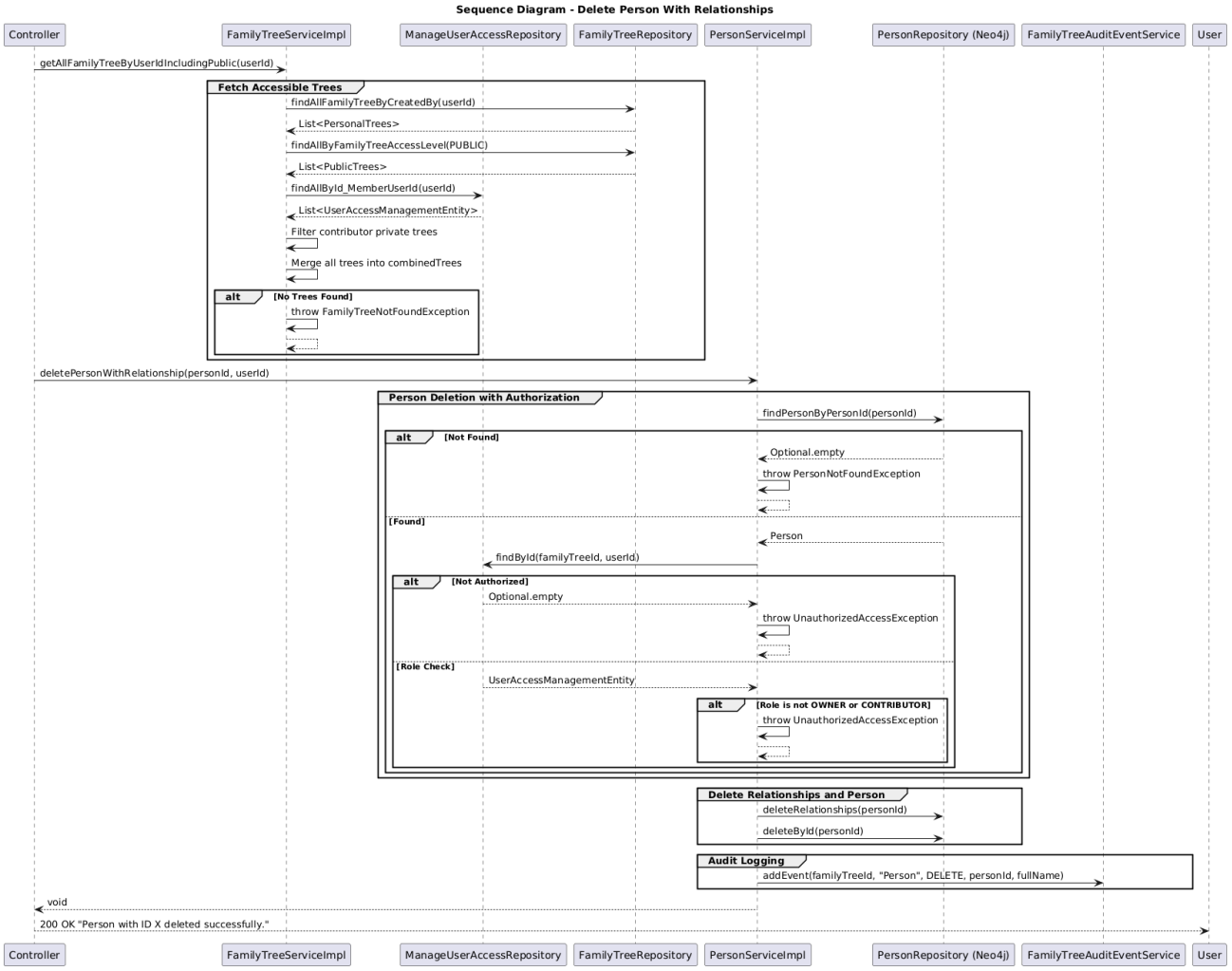
Appendix 18 Sequence Diagram: List all User's trees



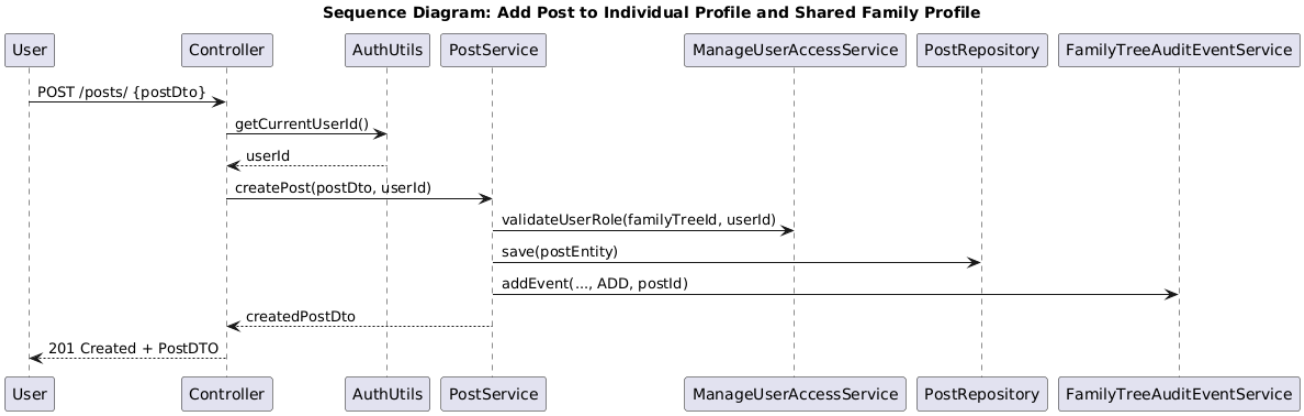
Appendix 19 Sequence Diagram: Delete family tree



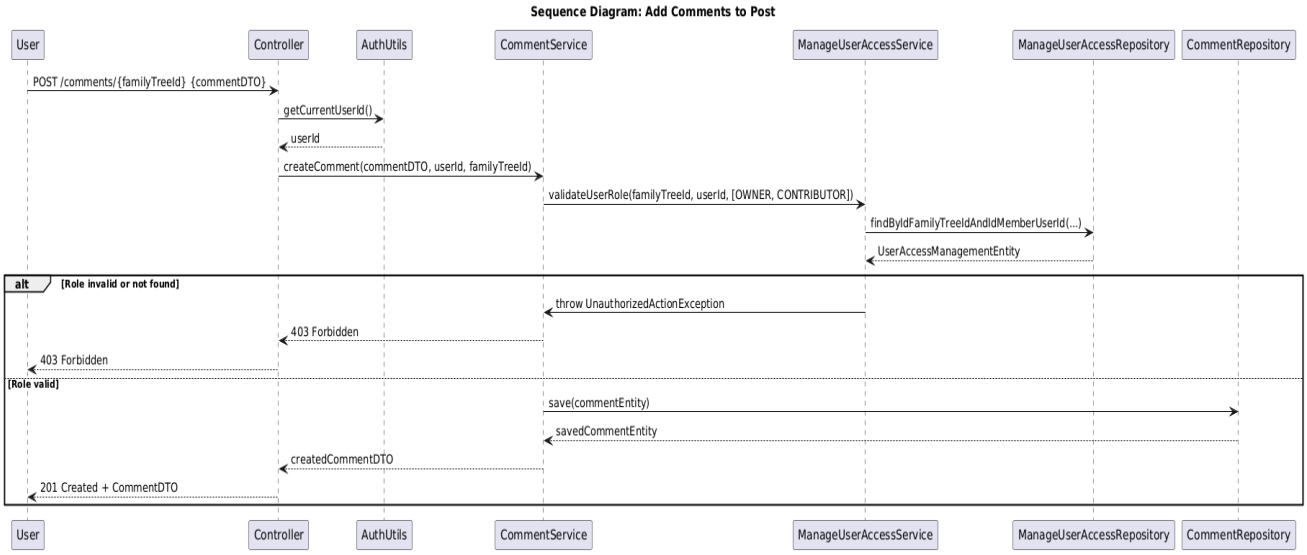
Appendix 20 Sequence Diagram: Delete person



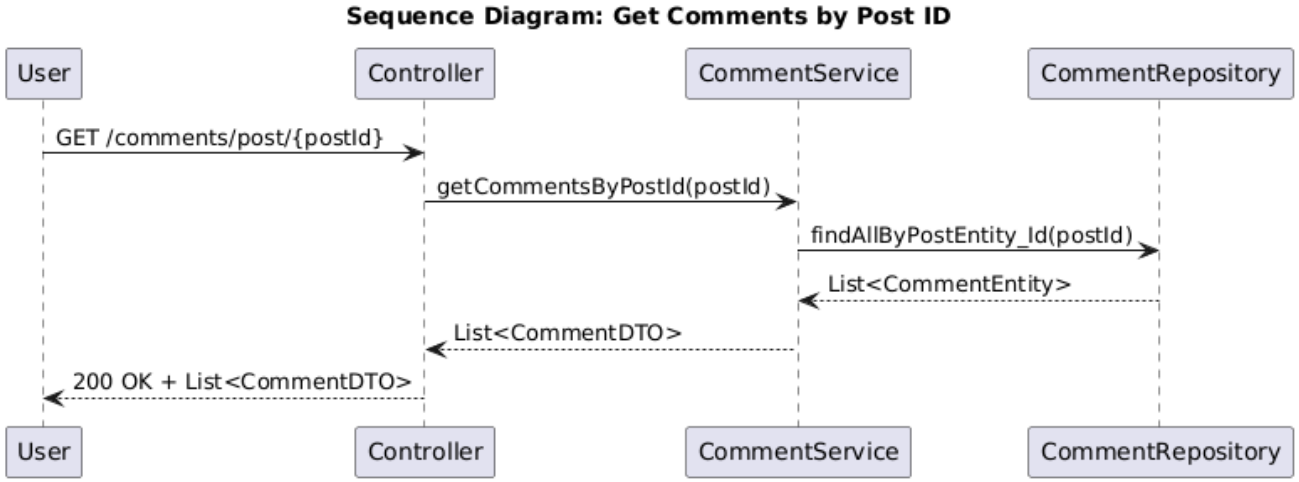
Appendix 21 Sequence Diagram: Add Post



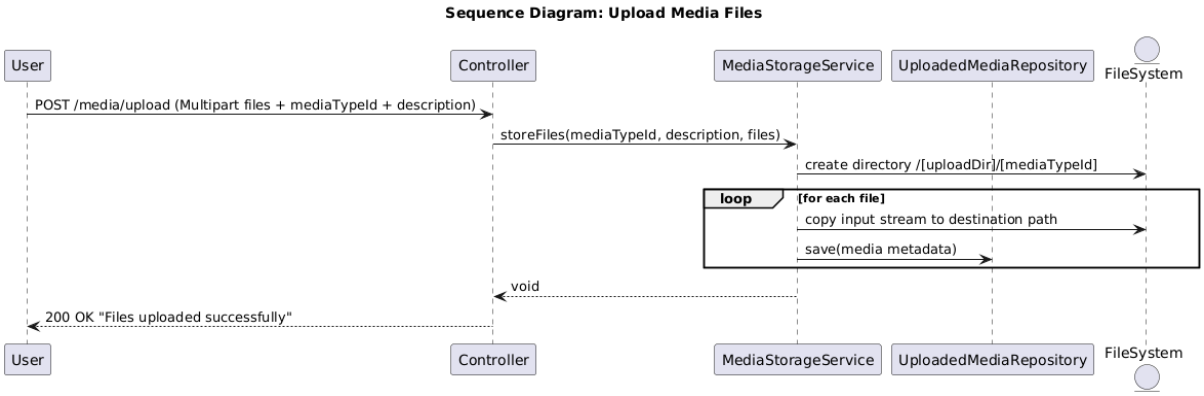
Appendix 22 Sequence Diagram: Add Comment



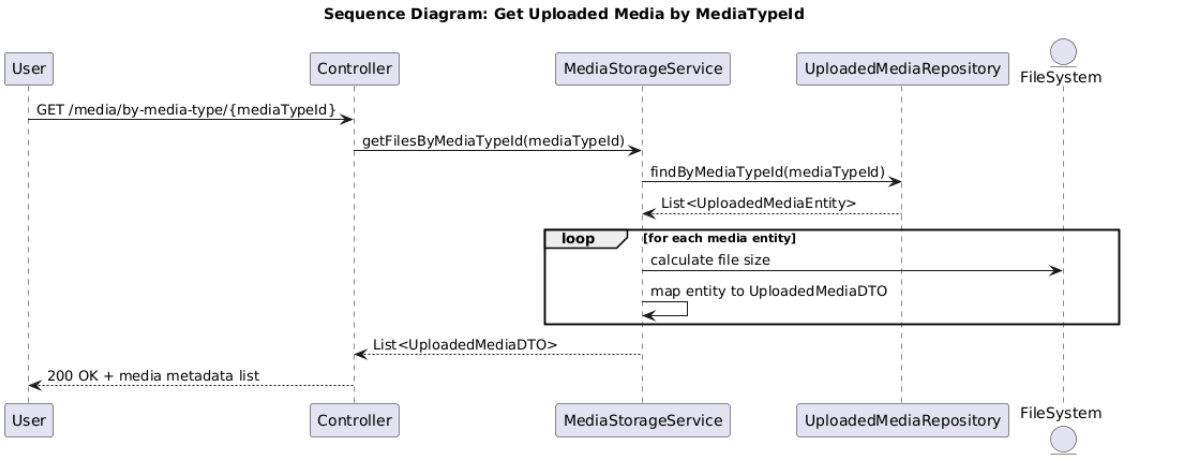
Appendix 23 Sequence Diagram: Get Comments by Post



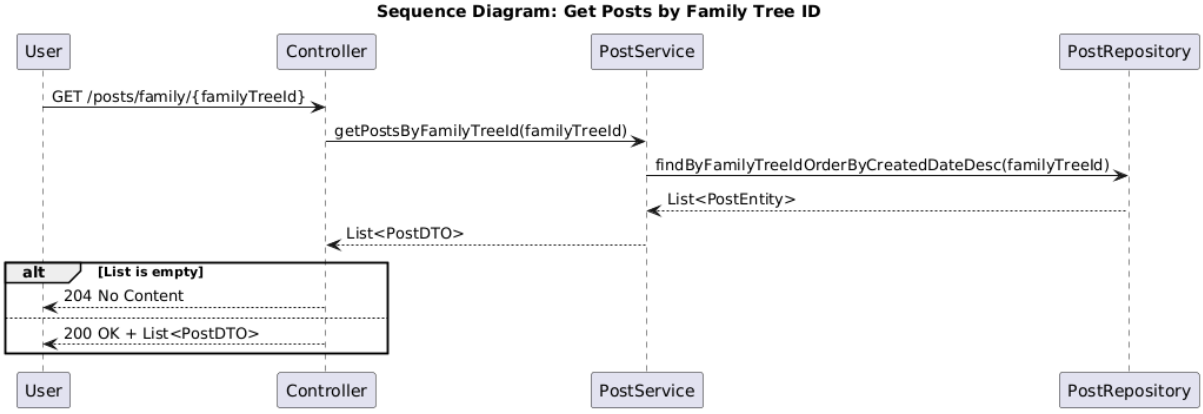
Appendix 24 Sequence Diagram: Upload Media



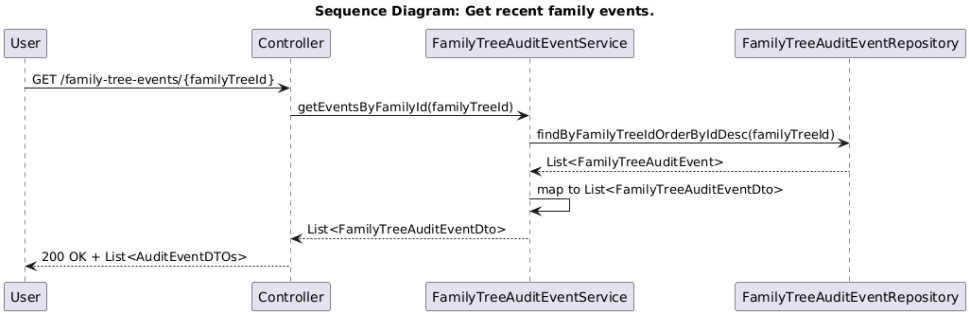
Appendix 25 Sequence Diagram: Get media by media type



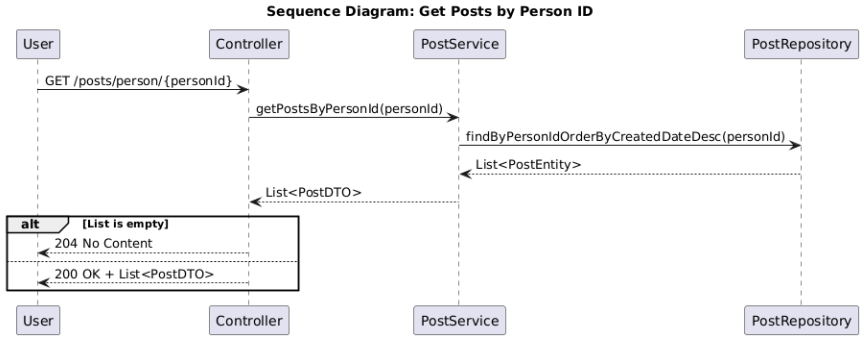
Appendix 26 Sequence Diagram: Get family Post



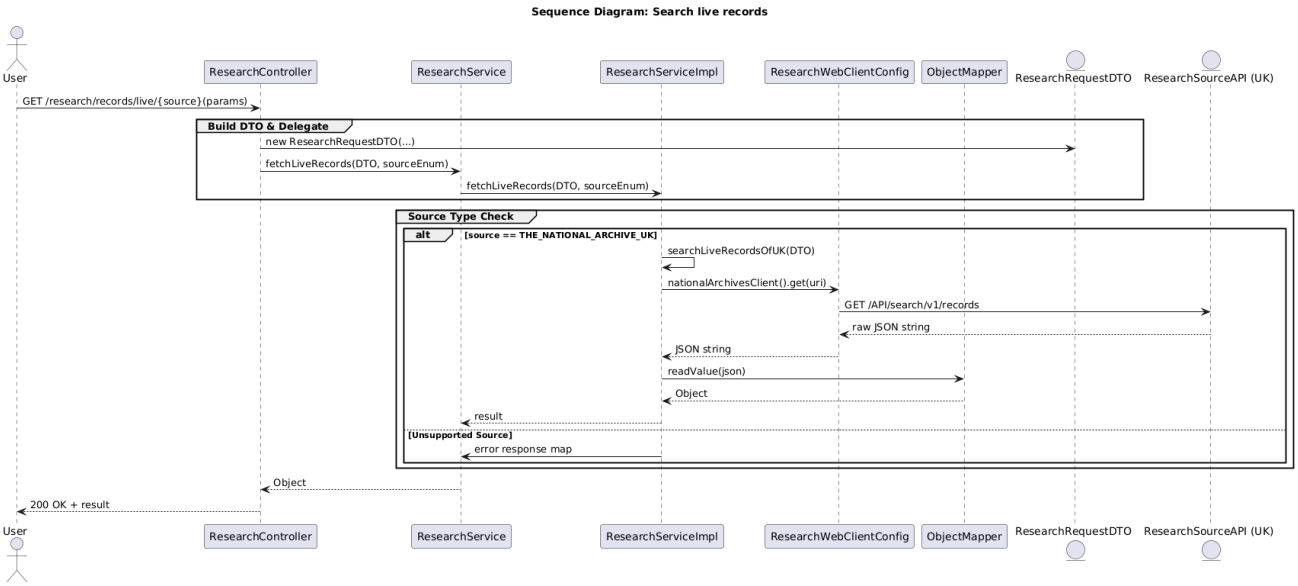
Appendix 27 Sequence Diagram: Get family tree Recent Activities



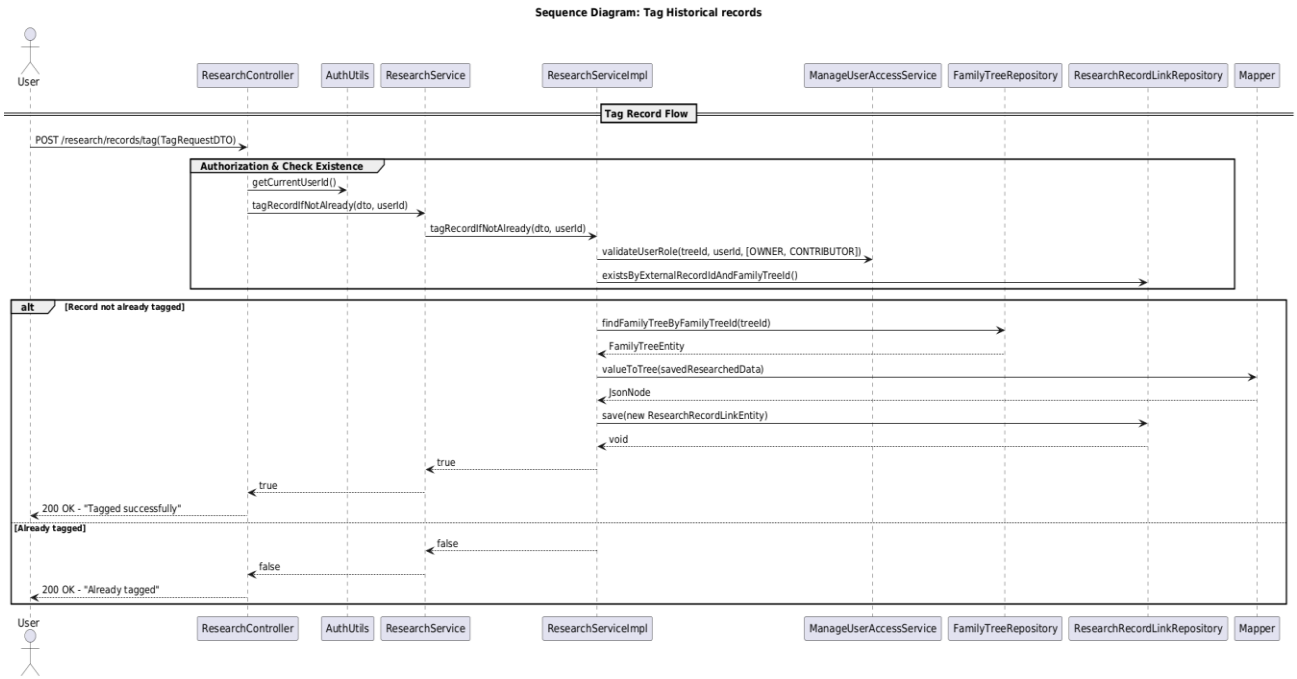
Appendix 28 Sequence Diagram: Get person posts



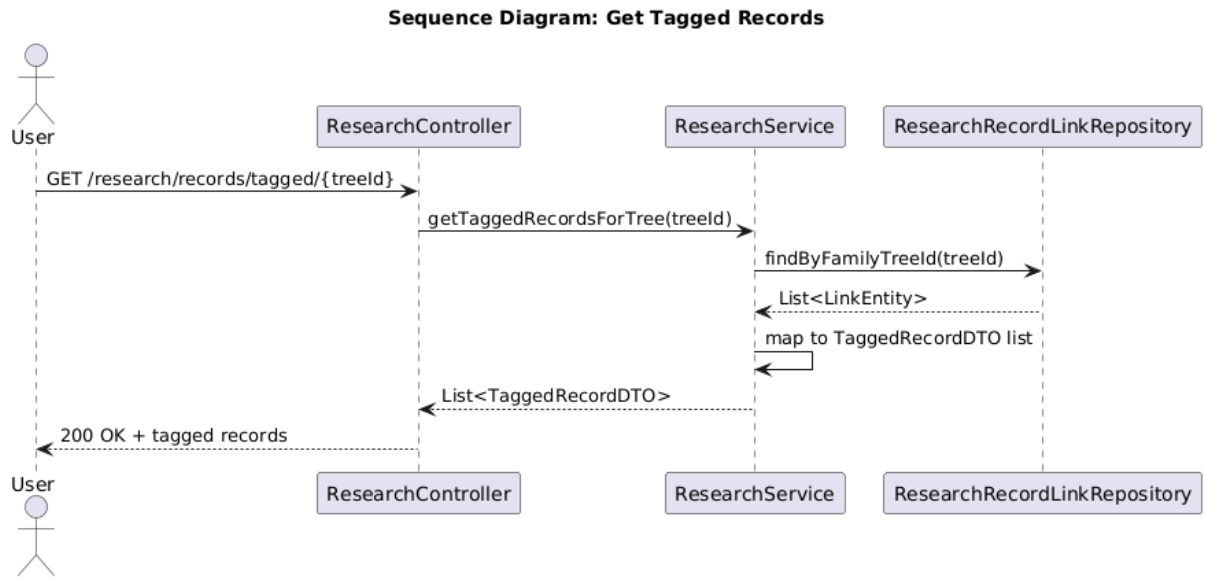
Appendix 29 Sequence Diagram: Search Historical records



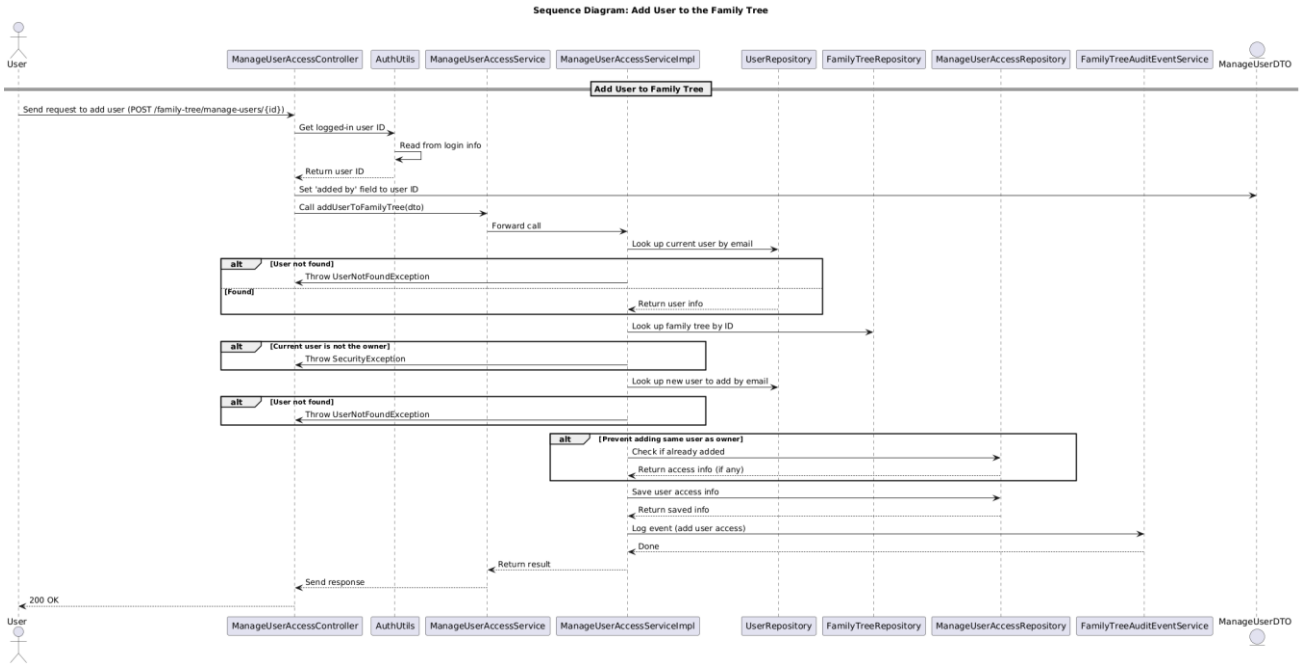
Appendix 30 Sequence Diagram: Tag Record to Family Tree



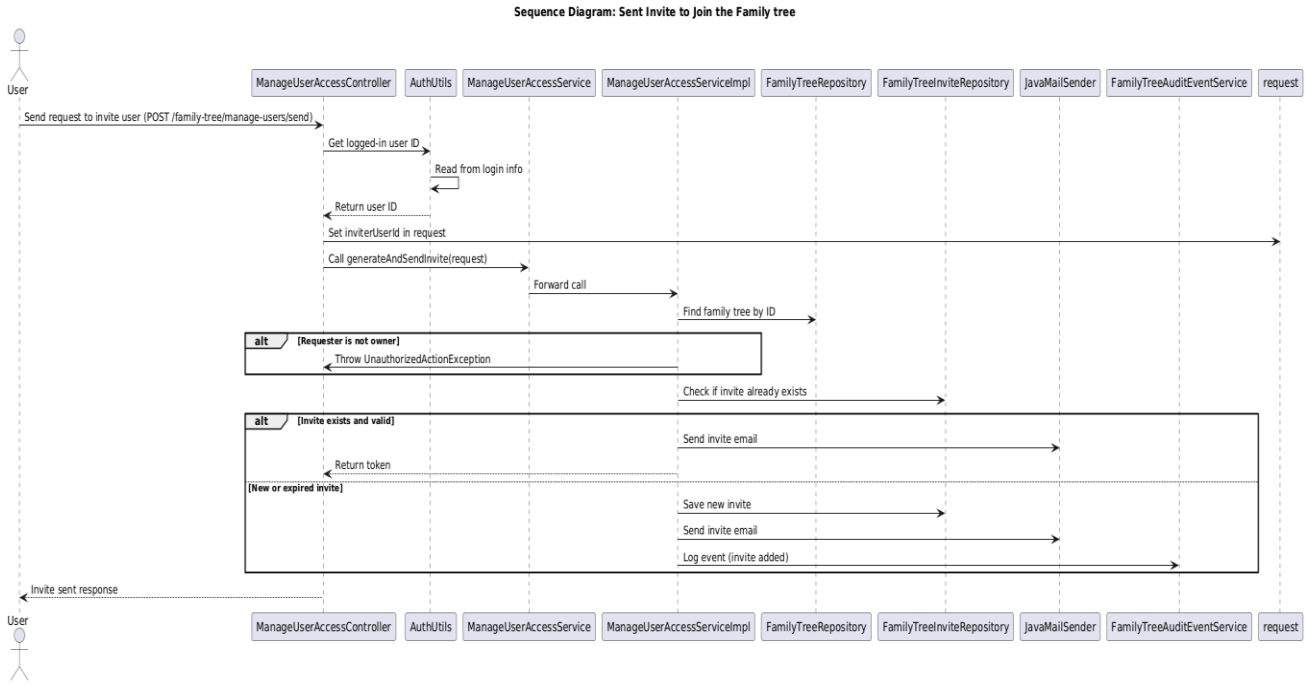
Appendix 31 Sequence Diagram: Get Tagged Research Records



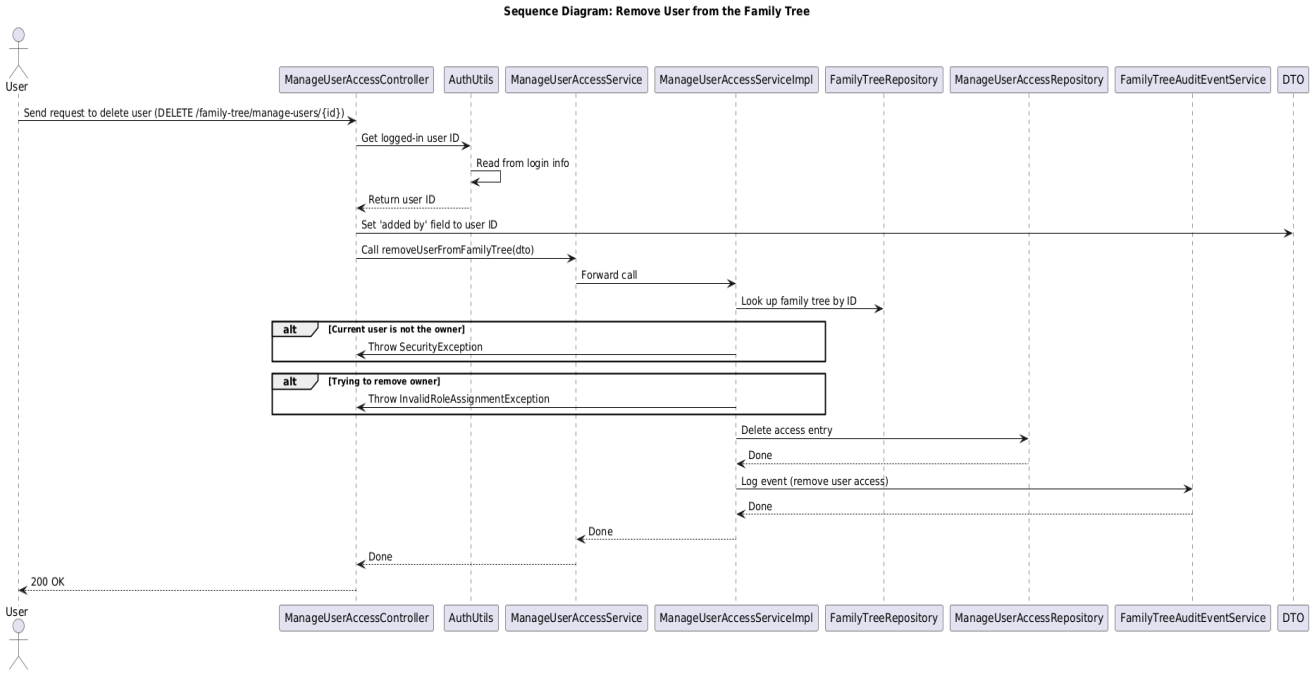
Appendix 32 Sequence Diagram: Add/Update Role



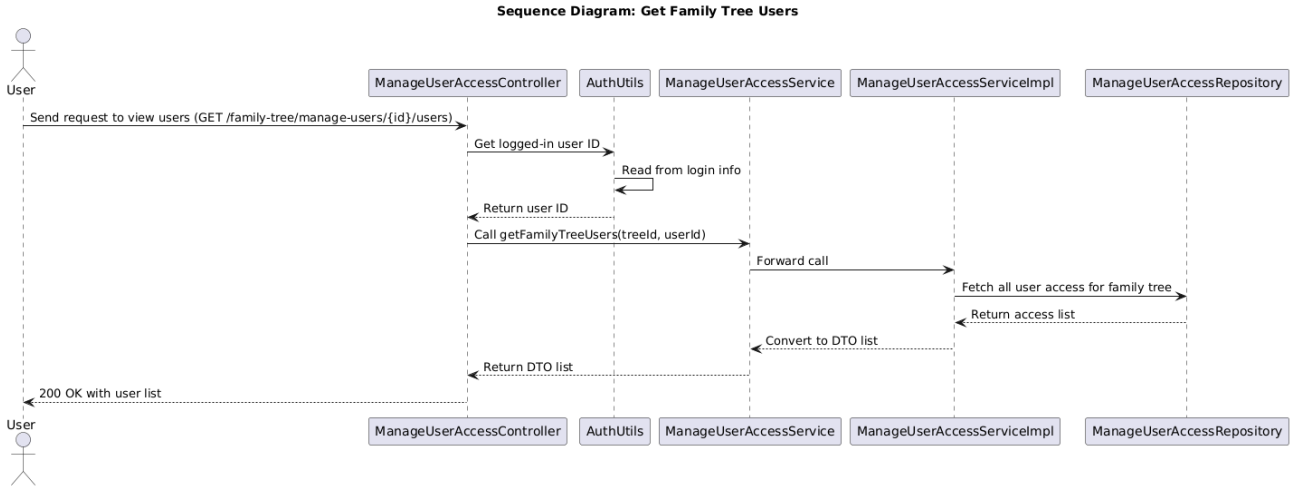
Appendix 33 Sequence Diagram: Send Invite



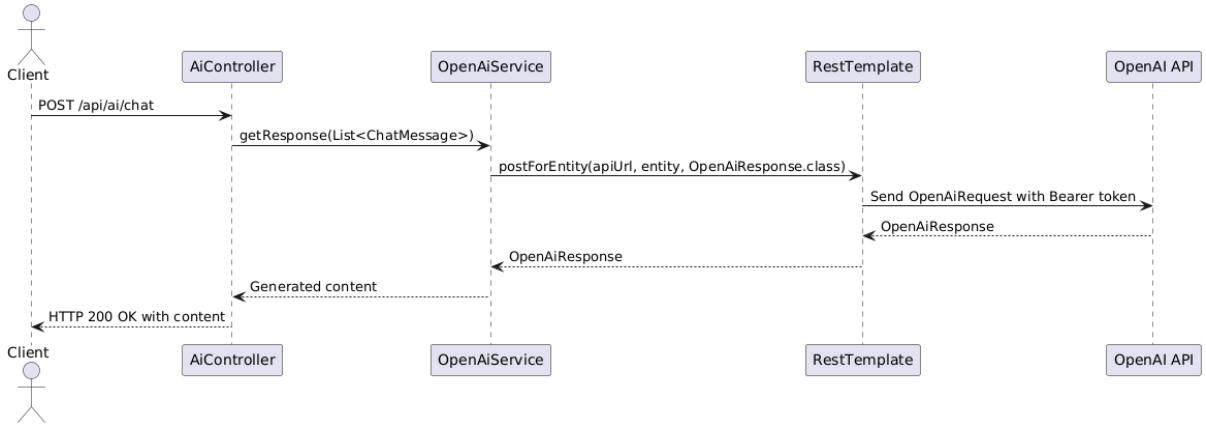
Appendix 34 Sequence Diagram: Remove User



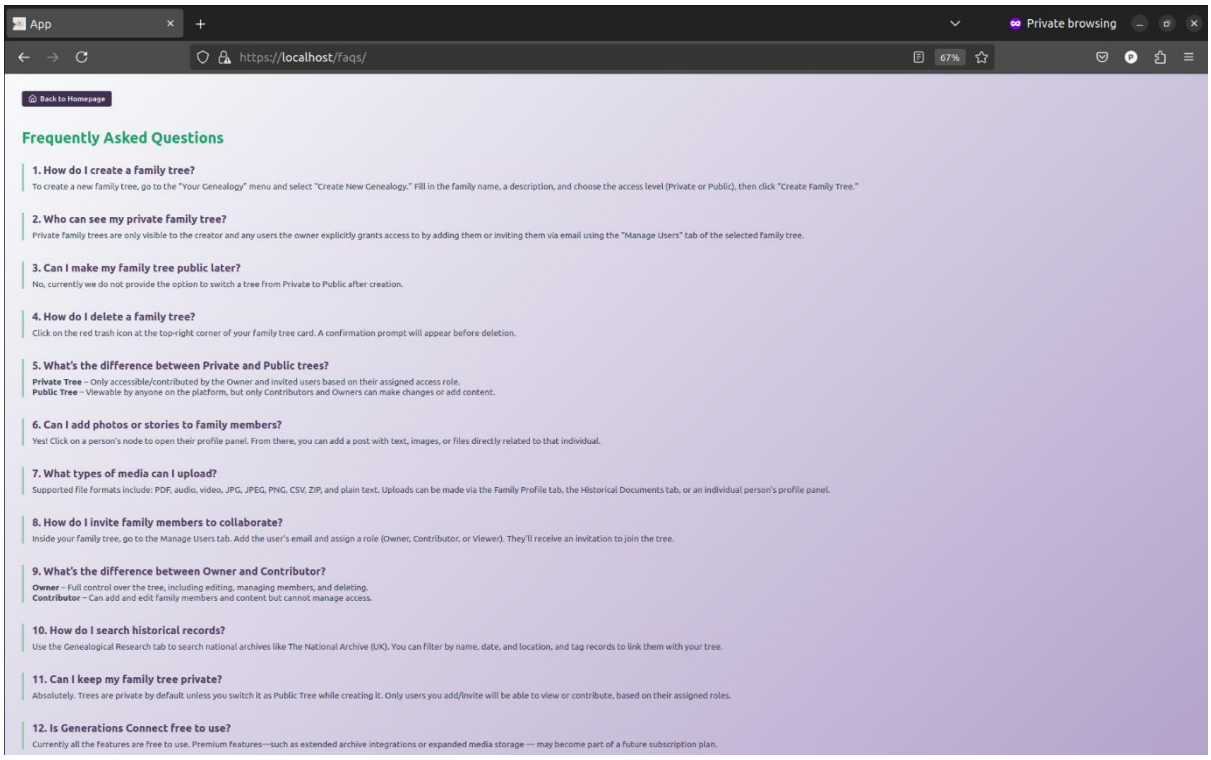
Appendix 35 Sequence Diagram: Get Users



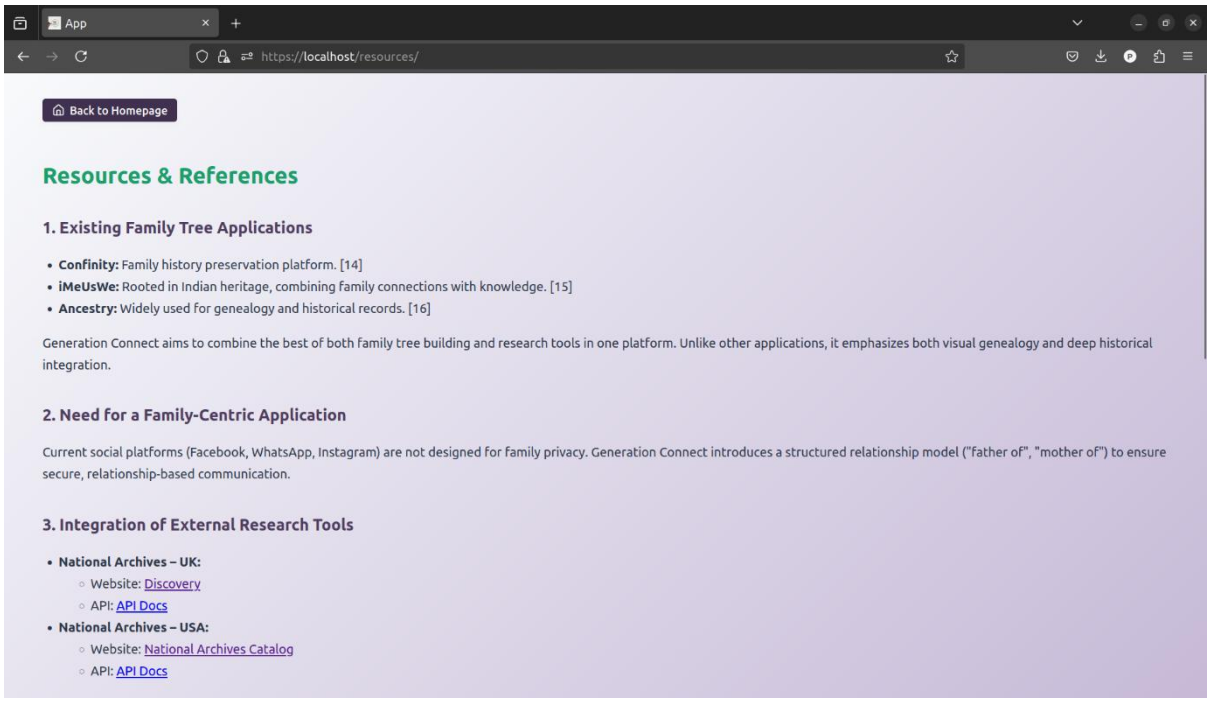
Appendix 36 Sequence Diagram: AI Chat



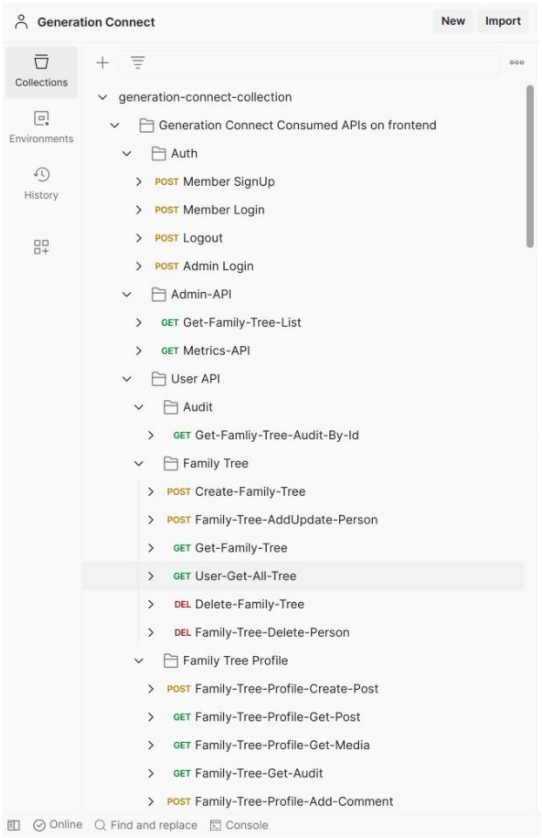
Appendix 37 FAQ's Page

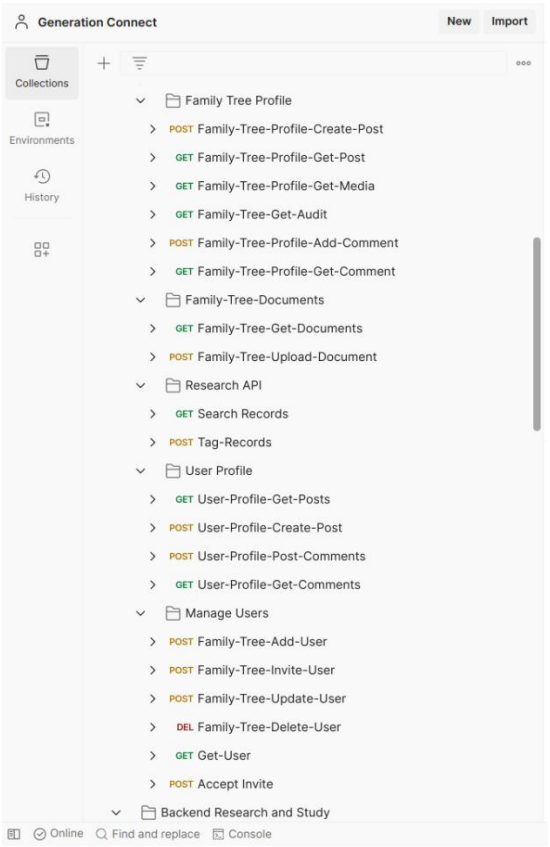


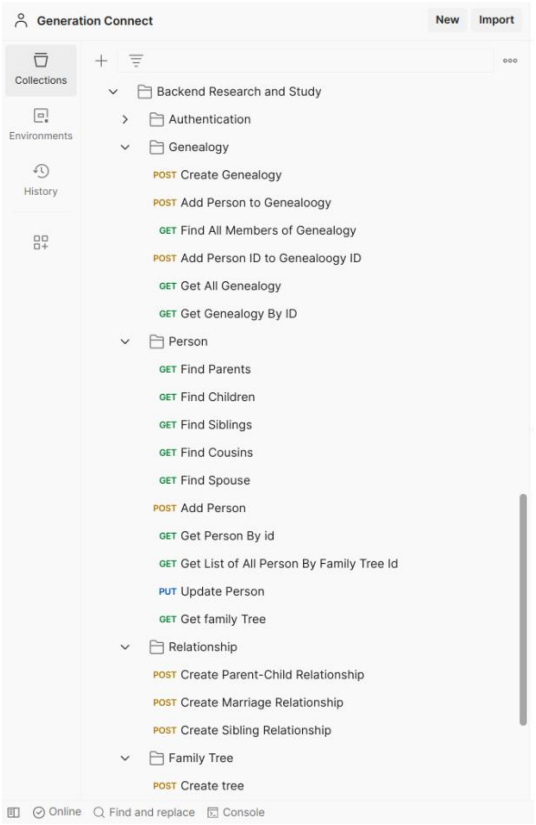
Appendix 38 Resource Page



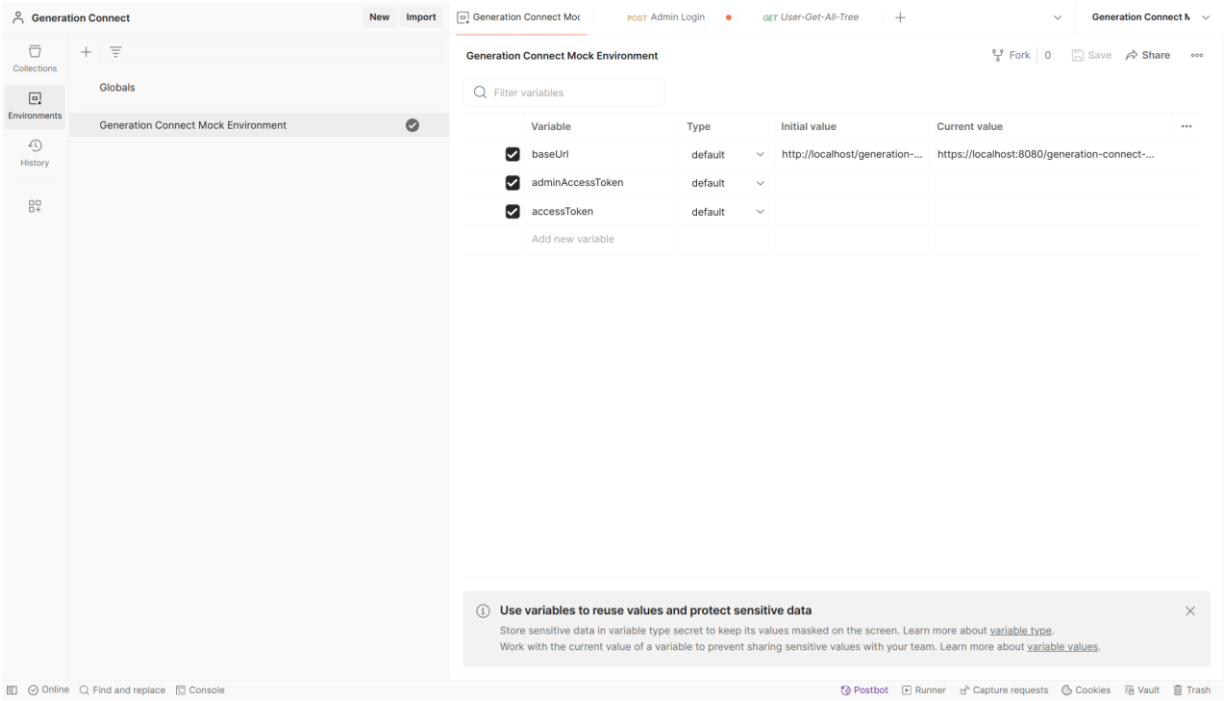
Appendix 39 Postman Collection



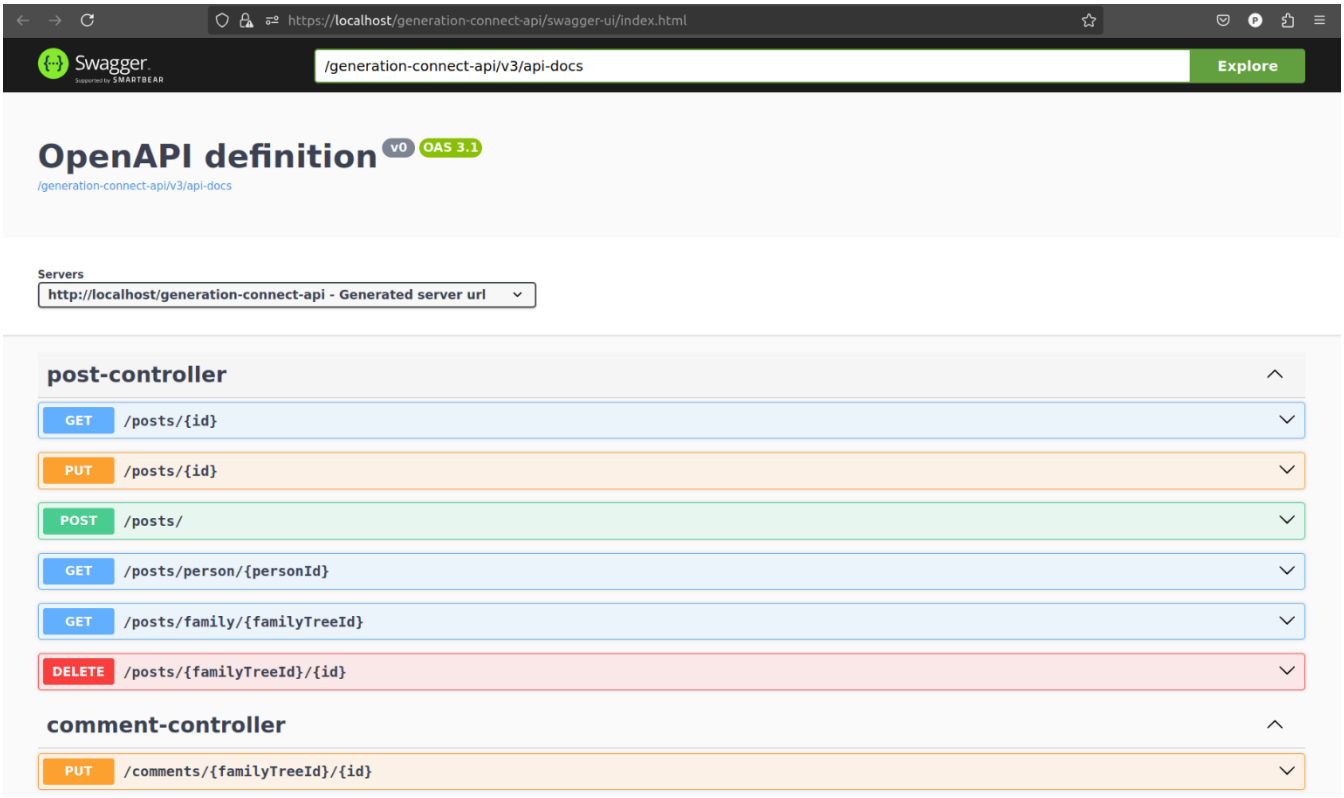


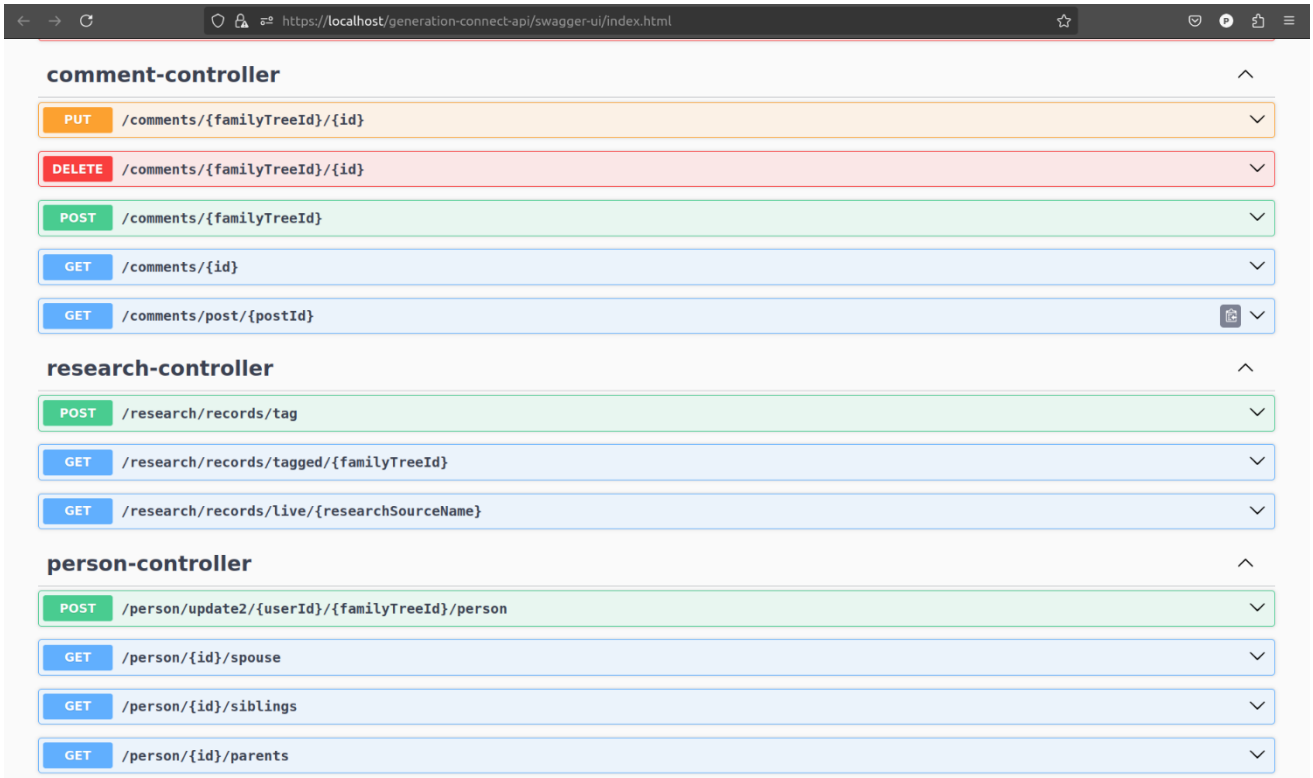


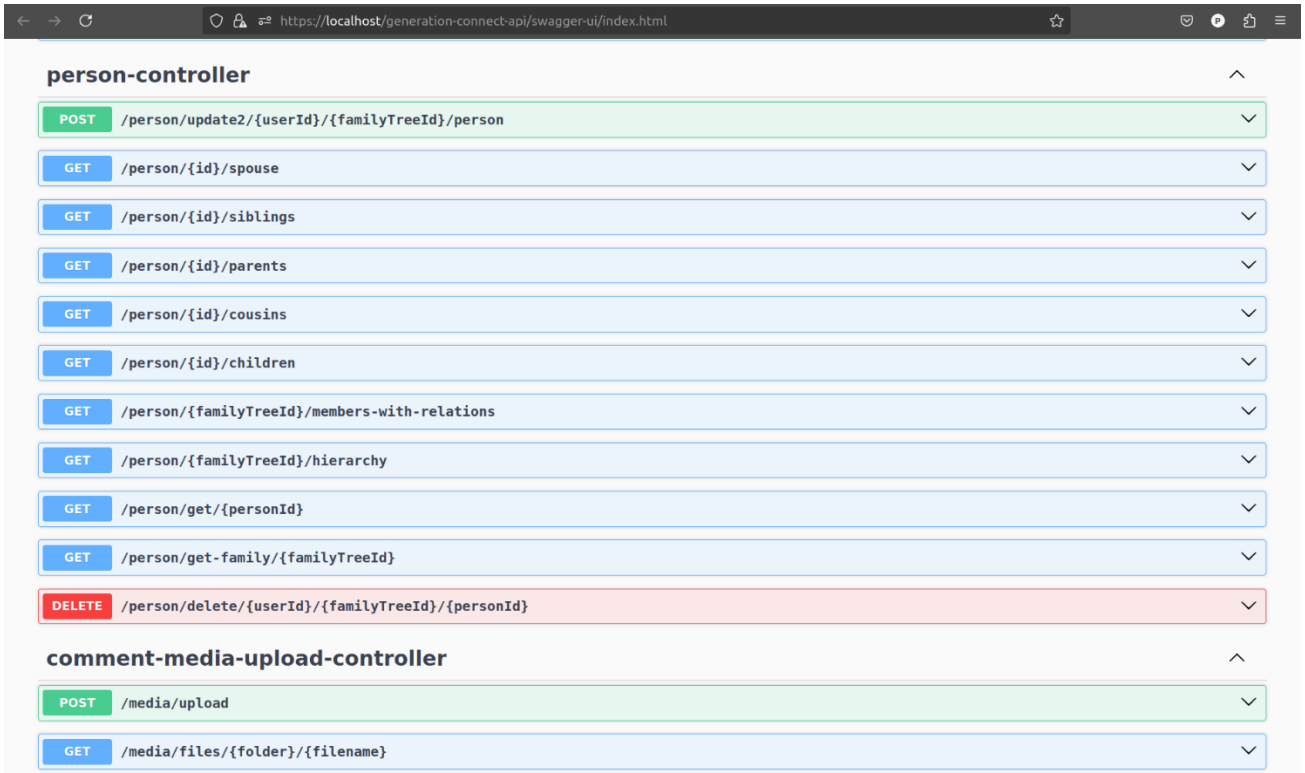
Appendix 40 Postman Environment

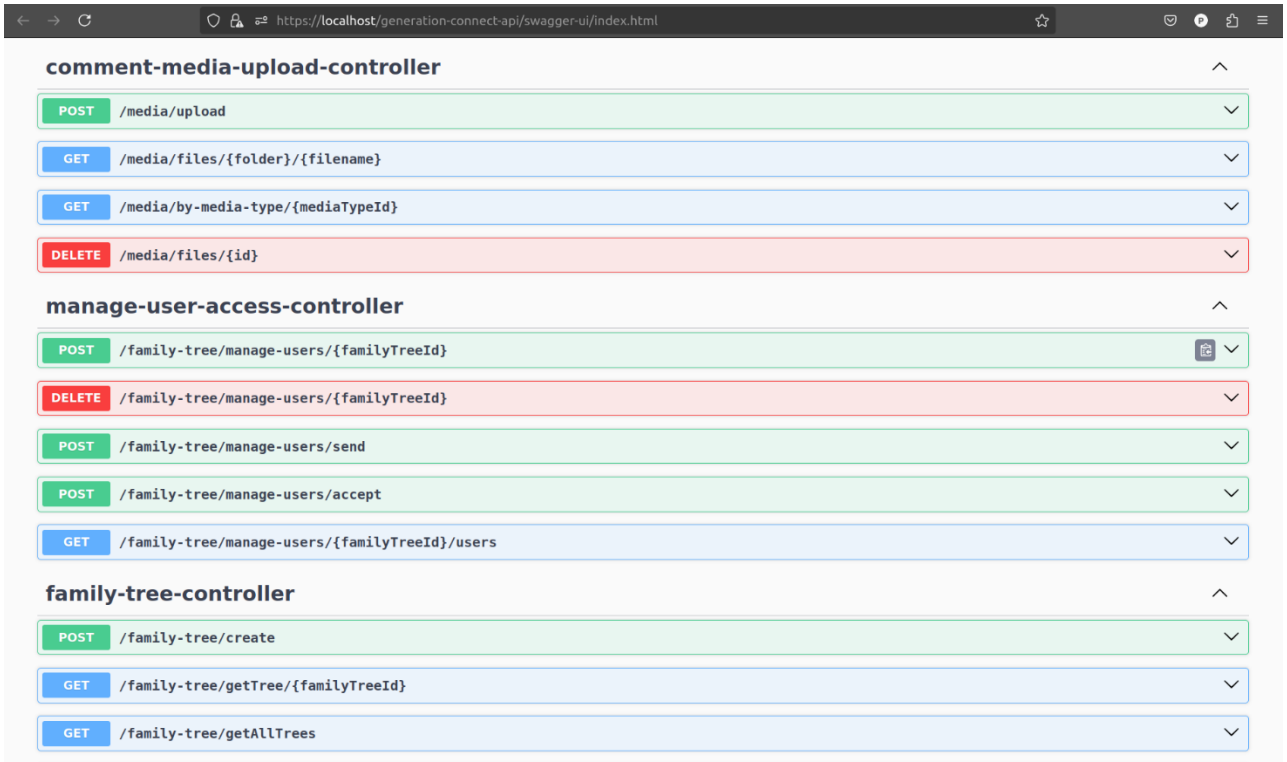


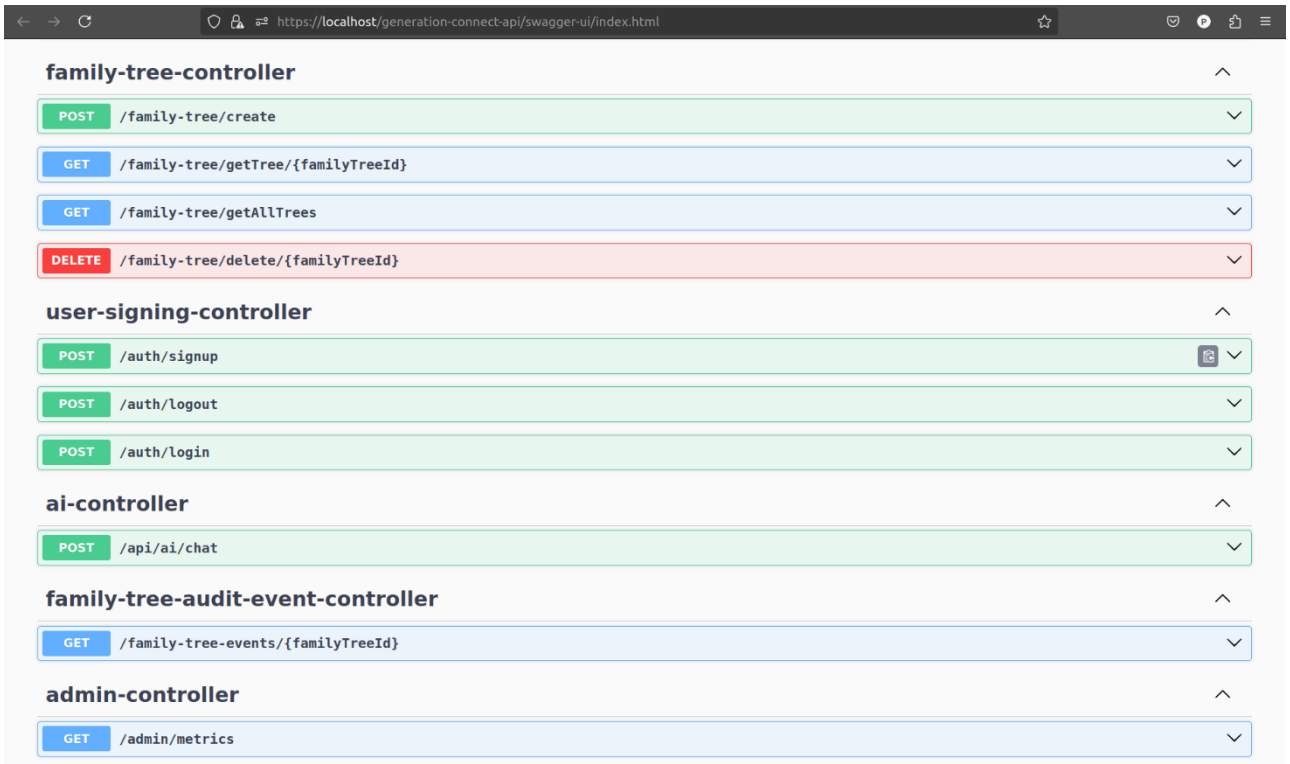
Appendix 41 Swagger API Documentation Page

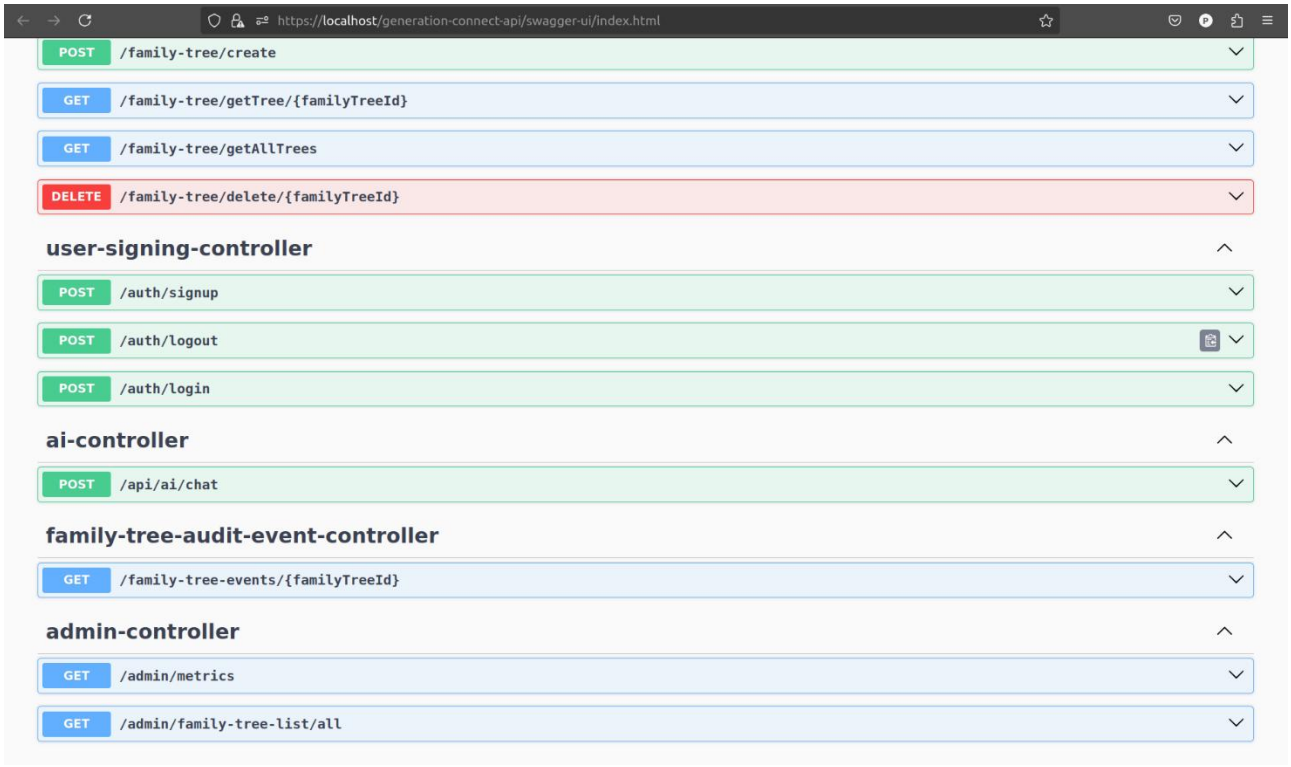



https://localhost/generation-connect-api/swagger-ui/index.html
comment-controller
PUT /comments/{familyTreeId}/{id}
DELETE /comments/{familyTreeId}/{id}
POST /comments/{familyTreeId}
GET /comments/{id}
GET /comments/post/{postId}
research-controller
POST /research/records/tag
GET /research/records/tagged/{familyTreeId}
GET /research/records/live/{researchSourceName}
person-controller
POST /person/update2/{userId}/{familyTreeId}/person
GET /person/{id}/spouse
GET /person/{id}/siblings
GET /person/{id}/parents

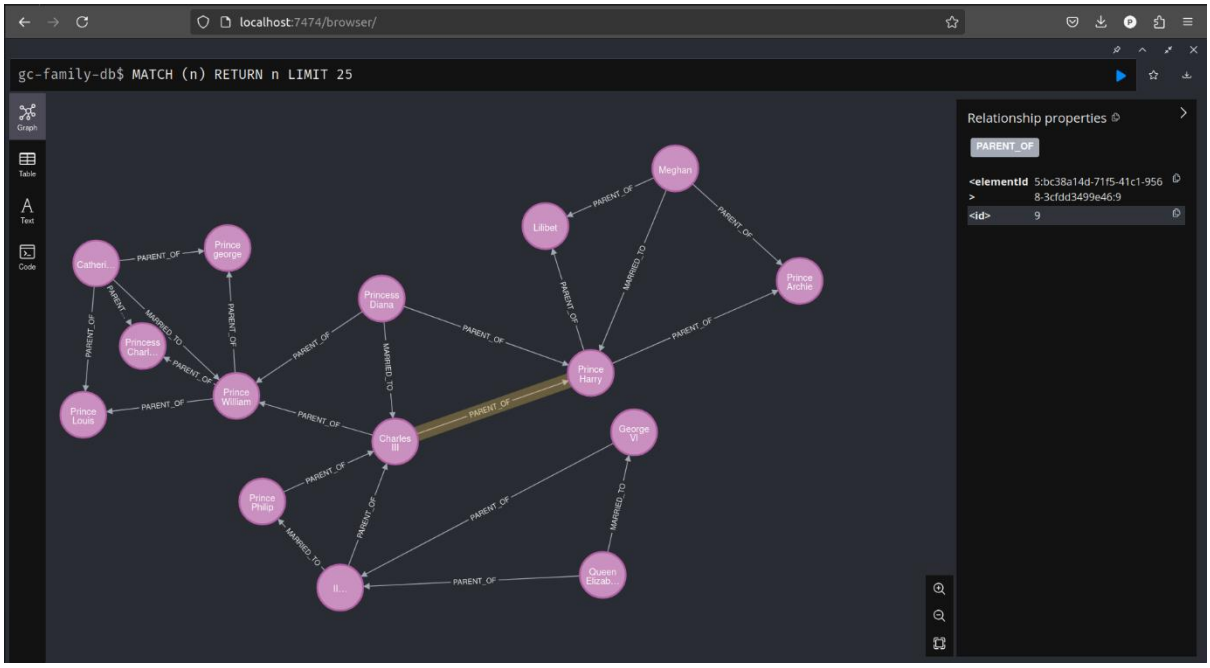

https://localhost/generation-connect-api/swagger-ui/index.html
person-controller
POST /person/update2/{userId}/{familyTreeId}/person
GET /person/{id}/spouse
GET /person/{id}/siblings
GET /person/{id}/parents
GET /person/{id}/cousins
GET /person/{id}/children
GET /person/{familyTreeId}/members-with-relations
GET /person/{familyTreeId}/hierarchy
GET /person/get/{personId}
GET /person/get-family/{familyTreeId}
DELETE /person/delete/{userId}/{familyTreeId}/{personId}
comment-media-upload-controller
POST /media/upload
GET /media/files/{folder}/{filename}


https://localhost/generation-connect-api/swagger-ui/index.html
comment-media-upload-controller
POST /media/upload
GET /media/files/{folder}/{filename}
GET /media/by-media-type/{mediaTypeId}
DELETE /media/files/{id}
manage-user-access-controller
POST /family-tree/manage-users/{familyTreeId}
DELETE /family-tree/manage-users/{familyTreeId}
POST /family-tree/manage-users/send
POST /family-tree/manage-users/accept
GET /family-tree/manage-users/{familyTreeId}/users
family-tree-controller
POST /family-tree/create
GET /family-tree/getTree/{familyTreeId}
GET /family-tree/getAllTrees


https://localhost/generation-connect-api/swagger-ui/index.html
family-tree-controller
POST /family-tree/create
GET /family-tree/getTree/{familyTreeId}
GET /family-tree/getAllTrees
DELETE /family-tree/delete/{familyTreeId}
user-signing-controller
POST /auth/signup
POST /auth/logout
POST /auth/login
ai-controller
POST /api/ai/chat
family-tree-audit-event-controller
GET /family-tree-events/{familyTreeId}
admin-controller
GET /admin/metrics



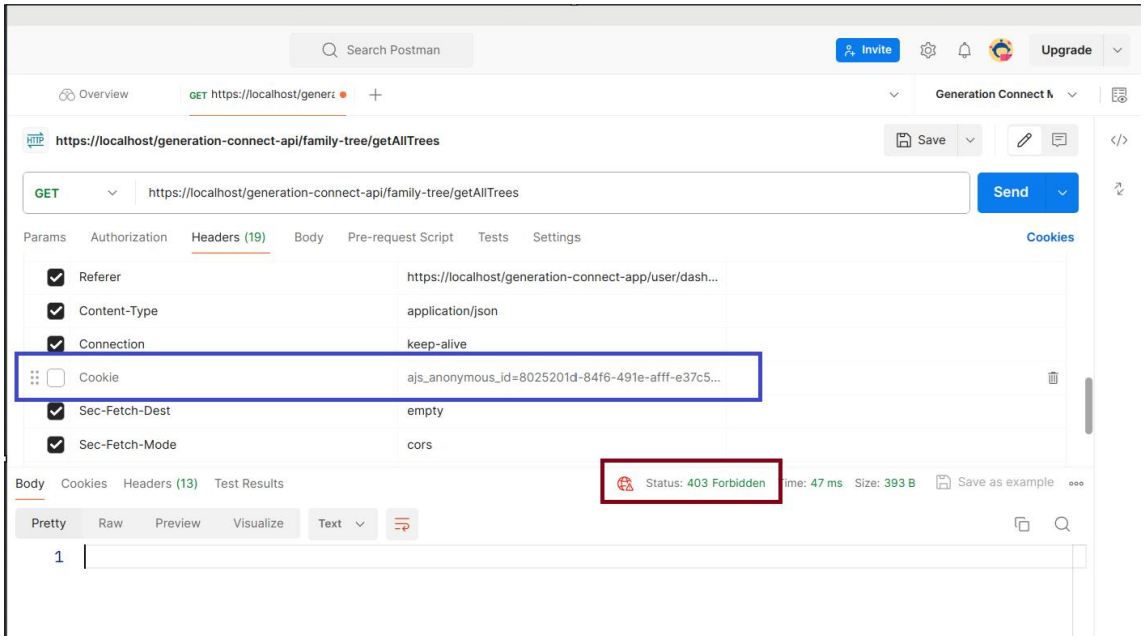
Appendix 42 Neo4j Database Browser Dashboard



Appendix 43 Docker containers and Used Images

```
naina@naina-pc:~$ docker container ls
CONTAINER ID   IMAGE                  COMMAND                  CREATED       STATUS                   PORTS
                                                                                                    NAMES
c8ce1bef5417   gc-nginx-image         "openresty -g 'daemo…"   2 hours ago   Up 2 hours               0.0.0.0:80->80/t
cp, [::]:80->80/tcp, 0.0.0.0:443->443/tcp, [::]:443->443/tcp                                        gc-nginx
c3500ec82410   gc-api-image           "java -jar /app/gc-a…"   2 hours ago   Up 2 hours               0.0.0.0:8080->80
80/tcp, [::]:8080->8080/tcp                                                                         gc-api
87b55e731cbc   gc-neo4j-db-image      "tini -g -- /startup…"   2 hours ago   Up 2 hours (healthy)     0.0.0.0:7474->74
74/tcp, [::]:7474->7474/tcp, 7473/tcp, 0.0.0.0:7687->7687/tcp, [::]:7687->7687/tcp   gc-neo4j-db
995ac05a9c67   gc-app-image           "docker-entrypoint.s…"   2 hours ago   Up 2 hours               0.0.0.0:3000->30
00/tcp, [::]:3000->3000/tcp                                                                         gc-app
7d5fb993afd1   gc-postgres-db-image   "docker-entrypoint.s…"   2 hours ago   Up 2 hours (healthy)     0.0.0.0:5432->54
32/tcp, [::]:5432->5432/tcp                                                                         gc-postgres-db
naina@naina-pc:~$ docker image ls
REPOSITORY             TAG       IMAGE ID       CREATED       SIZE
gc-nginx-image         latest    7fa3da7c1203   2 hours ago   446MB
gc-api-image           latest    754fe58a7fa3   2 hours ago   598MB
gc-app-image           latest    58d0588df314   2 hours ago   2.34GB
gc-postgres-db-image   latest    5e96d64d433e   5 days ago    438MB
<none>                 <none>    9c2c7ce9a8bc   10 days ago   446MB
<none>                 <none>    9d5b5f5f8ea7   10 days ago   601MB
<none>                 <none>    7d080fa710e4   10 days ago   2.6GB
<none>                 <none>    c775cafedde2   10 days ago   2.34GB
<none>                 <none>    5ac9ecf03b14   10 days ago   601MB
gc-neo4j-db-image      latest    57341db91278   2 weeks ago   581MB
naina@naina-pc:~$
```

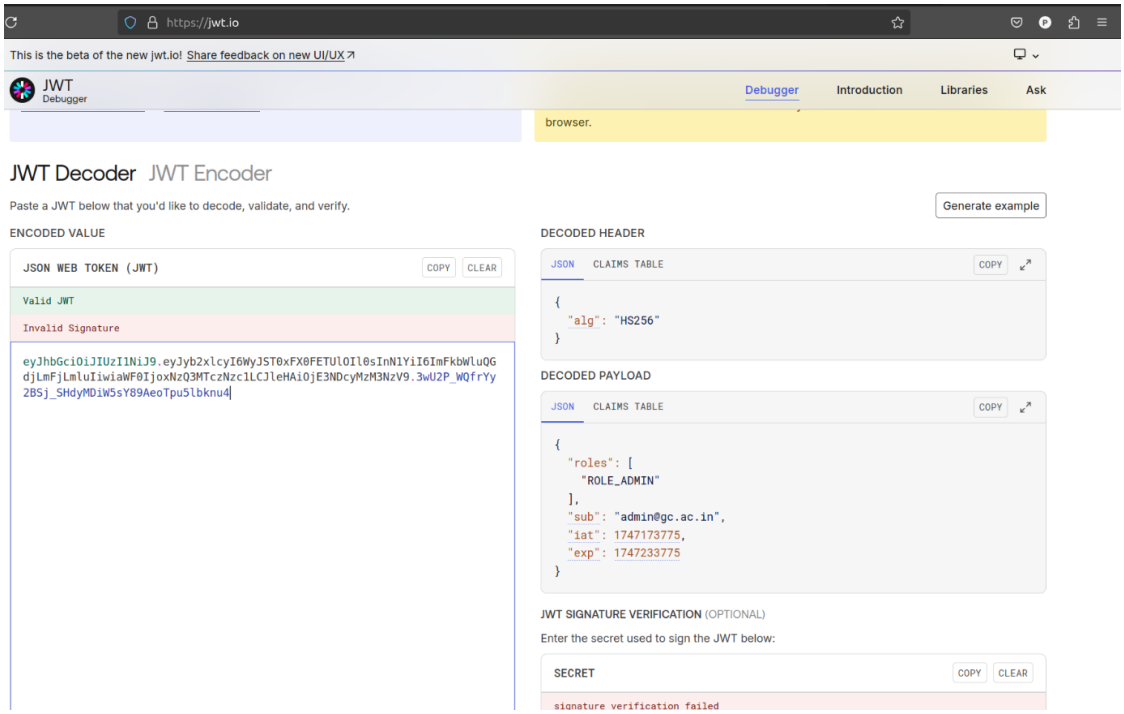
Appendix 44 Get All Trees Without Cookie (403 Forbidden)

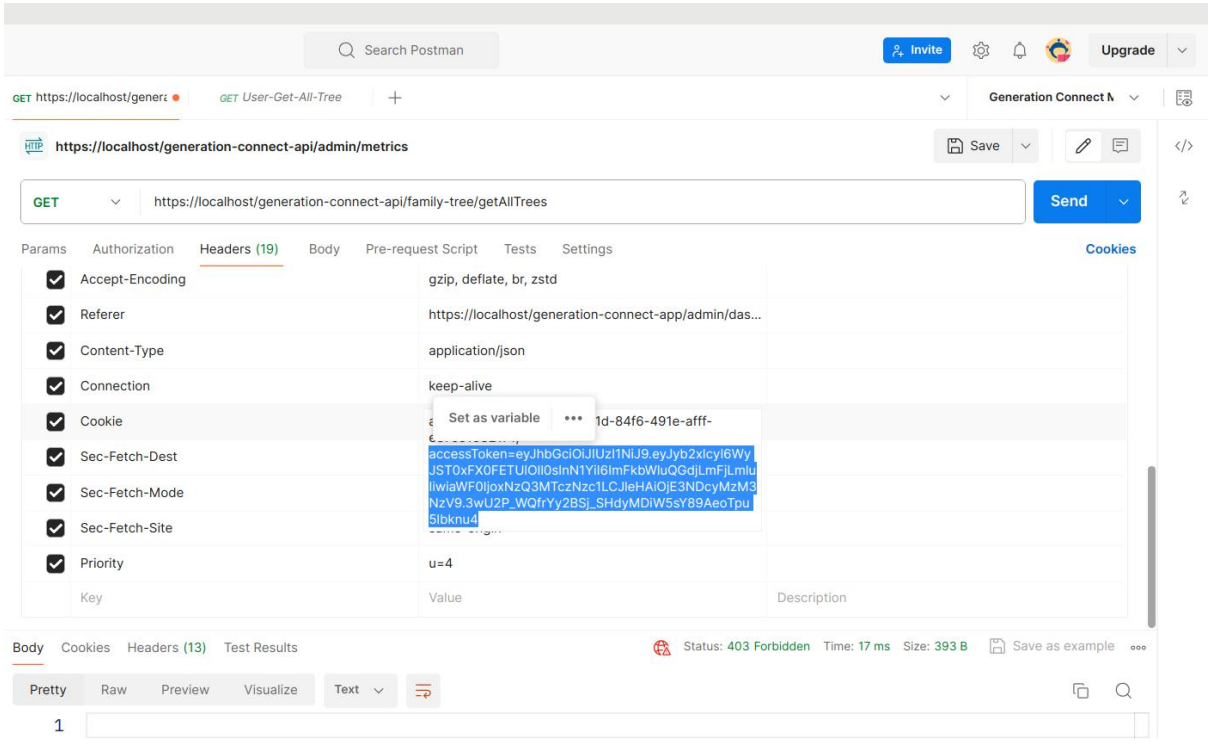


Appendix 45 Get All Trees With Valid Token in Cookie

```
gc-api          |         fte1_0.created_date>?
gc-api          | Server port: 8080 [DEBUG] Using 'application/json', given [*/*] and supported [application/json, application/*+json, application/yaml]
gc-api          | Server port: 8080 [DEBUG] Writing [{pendingInvites=0, averagePostsPerTree=0.4, familyTreeOwners=2, averageCommentsPerTree=0.0, totalNon (truncated)...]
gc-api          | Server port: 8080 [DEBUG] Completed 200 OK
gc-nginx        | 172.18.0.1 - - [13/May/2025:23:25:23 +0000] "GET /generation-connect-api/admin/metrics HTTP/1.1" 200 303 "https://localhost/generation-connect-app/admin/dash
board" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:138.0) Gecko/20100101 Firefox/138.0"
```

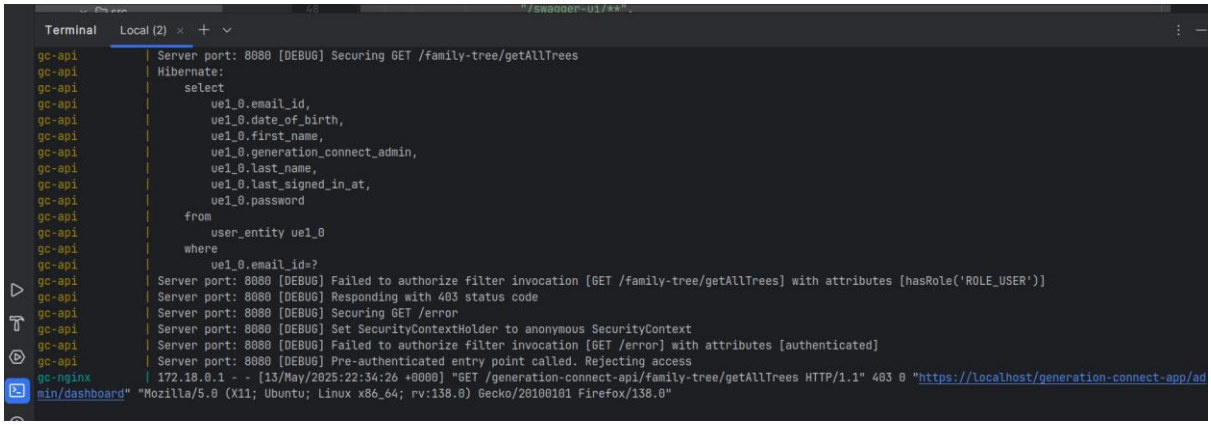
Appendix 46 Failed Get All Trees Request Using Admin Access Token





Same will happen when accessing admin api using user's accessToken

Appendix 47 Spring Boot Security and NGINX: 403 Forbidden Error



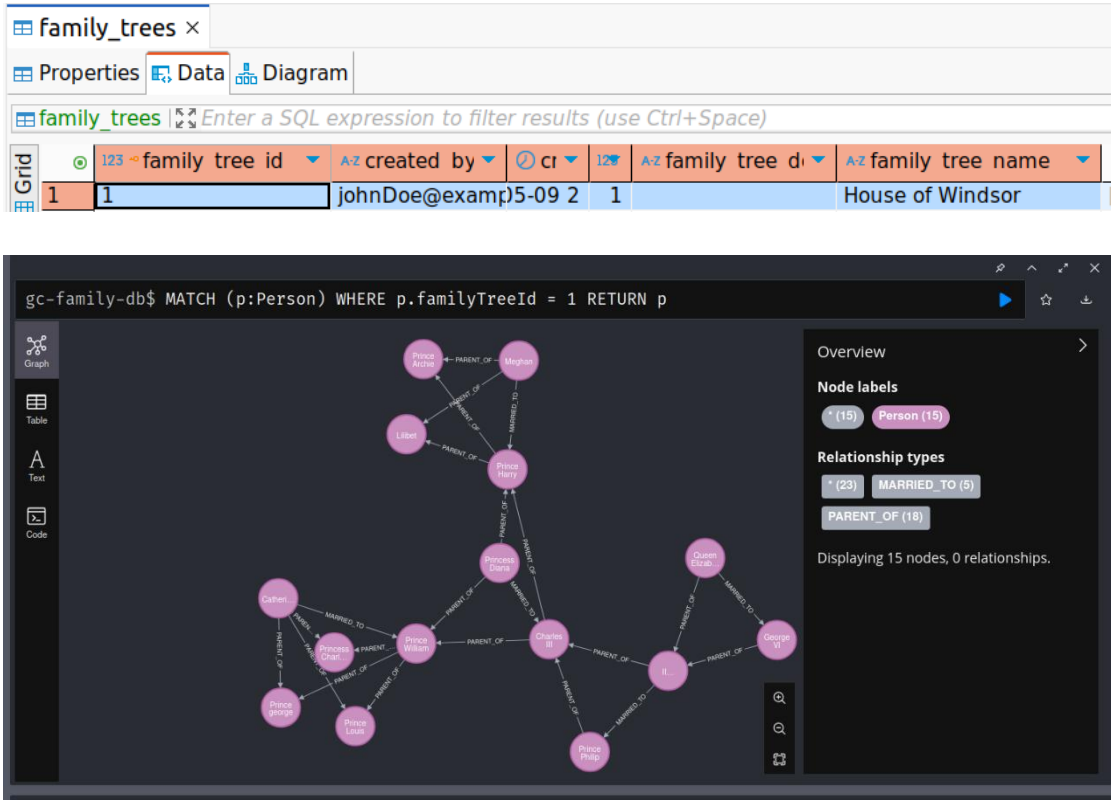
Appendix 48 Successful API Log for Get Metrics

Appendix 49 Spring Security Configuration with JWT Filter

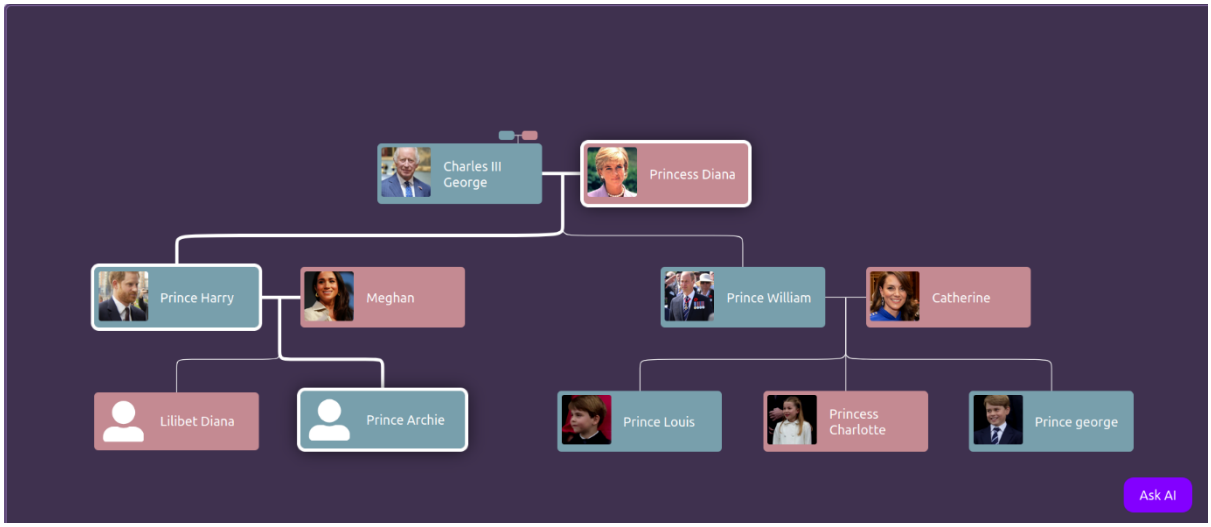
```
package com.generation.connect.config;

@Configuration
public class SecurityConfig implements WebMvcConfigurer {
    private final GCUserDetailsService userDetailsService;

    private final JwtAuthFilter jwtAuthFilter;

    public SecurityConfig(GCUserDetailsService userDetailsService, JwtAuthFilter jwtAuthFilter)
{
        this.userDetailsService = userDetailsService;
        this.jwtAuthFilter = jwtAuthFilter;
    }

    @Bean
```

```
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http
                .csrf(csrf -> csrf.disable()
                )
                .cors().configurationSource(corsFilter()).and()
                .authorizeRequests()
                .requestMatchers("/auth/login",
                        "/auth/signup",
                        "/auth/logout",
                        "/swagger-ui/**",
                        "/family-tree/manage-users/accept",
                        "/resources/**").permitAll()
                .and()
                .authorizeRequests()
                .requestMatchers("/person/**").hasRole("USER")
                .requestMatchers("/family-tree/**").hasRole("USER")
                .requestMatchers("/family-tree/manage-users/**").hasRole("USER")
                .requestMatchers("/relationship/parent-child?**").hasRole("USER")
                .requestMatchers("/research-archives/**").hasRole("USER")
                .requestMatchers("/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
                .and()
                .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and()
                .authenticationProvider(authenticationProvider())
                .addFilterBefore(jwtAuthFilter,
UsernamePasswordAuthenticationFilter.class).build();
    }

    @Bean
    public UrlBasedCorsConfigurationSource corsFilter() {
        CorsConfiguration config = new CorsConfiguration();
        config.setAllowCredentials(true);
        config.addAllowedOriginPattern("*");
        config.addAllowedOrigin("http://localhost");
        config.addAllowedOrigin("https://localhost");
        config.addAllowedHeader("*");
        config.addAllowedMethod("*");
        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", config);
        return source;
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public AuthenticationProvider authenticationProvider() {
        DaoAuthenticationProvider authenticationProvider = new DaoAuthenticationProvider();
        authenticationProvider.setUserDetailsService(userDetailsService());
        authenticationProvider.setPasswordEncoder(passwordEncoder());
        return authenticationProvider;

    }

    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration config)
throws Exception {
        return config.getAuthenticationManager();
    }

    @Bean
    public UserDetailsService userDetailsService() {
        return userDetailsService;
    }
}
```

Appendix 50 Connection Between family_tree Table and Person Node



Appendix 51 Family Tree Rendering Overview



Appendix 52 Family Tree card Input Validations

```
function postSubmit(props) {
    if(!datum?.data?.firstName){
      console.log("form submit :" + JSON.stringify(datum?.data?.firstName));
      this.showOverlayMessage("First name is mandatory");
      return;
    }
    const todayDate = new Date().toISOString().split("T")[0];
    const birthdate = datum?.data?.birthdate;
    if(birthdate && birthdate > todayDate) {
      this.showOverlayMessage("Birthdate cannot be in the future.");
      return;
    }

    const demiseDate = datum?.data?.demiseDate;
    if (birthdate && demiseDate) {
      if (demiseDate < birthdate) {
        this.showOverlayMessage("Demise date cannot be before birthdate.");
        return;
      }
      if (demiseDate > todayDate) {
        this.showOverlayMessage("Demise date cannot be in the future.");
        return;
      }
    }

    if (this.addRelativeInstance.is_active) this.addRelativeInstance.onChange(datum)
    else if (!props?.delete) this.openFormWithId(datum.id);

    if (!this.is_fixed) this.closeForm()

    this.store.updateTree({})

    this.updateHistory()
    if (this.datumCallback) {
      this.datumCallback(datum, props, this.addRelativeInstance.is_active)
    }

  }

…

EditTree.prototype.showOverlayMessage = function(message, timeout = 2000) {
  const overlay = document.getElementById("overlay-message");
  const text = document.getElementById("overlay-text");
  const closeBtn = document.getElementById("overlay-close");

  if (overlay && text && closeBtn) {
    text.textContent = message;
    overlay.style.display = "flex";
    overlay.style.backgroundColor = "#e33b5a";
    overlay.style.color = "white";
    overlay.style.borderBlockColor = "white";

    if (this._overlayTimeout) {
      clearTimeout(this._overlayTimeout);
    }

    this._overlayTimeout = setTimeout(() => {
      overlay.style.display = "none";
    }, timeout);

    closeBtn.onclick = () => {
      overlay.style.display = "none";
      clearTimeout(this._overlayTimeout);
    };}}
```

Appendix 53 Add/Update Person to the tree with code flow

Controller

```
@RestController @RequestMapping("/person")
public class PersonController {

    private final PersonService personService;

    private final FamilyTreeService familyTreeService;

    public PersonController(PersonService personService, FamilyTreeService familyTreeService) {
        this.personService = personService;
        this.familyTreeService = familyTreeService;
    }
    @PostMapping("/update2/{userId}/{familyTreeId}/person")
    public ResponseEntity<Person> updatePerson(
            @RequestBody PersonUpdateRequest personUpdateRequest,
            @PathVariable Long familyTreeId,
            @PathVariable String userId     ) {
        if (familyTreeService.getAllFamilyTreeByUserIdIncludingPublic(userId) == null) {
            throw new FamilyTreeNotFoundException("No family tree found connected to userId " +
userId + " and " + familyTreeId + " familyTreeId.");
        }
        personUpdateRequest.setFamilyTreeId(familyTreeId);
        Person updatedPerson = personService.updatePerson(personUpdateRequest, userId);
return ResponseEntity.ok(updatedPerson);
    }
}
```

Interface

```
public interface PersonService {
    Person updatePerson(PersonUpdateRequest personUpdateRequest, String userId);
}
```

Interface Implementation

```
@Slf4j @Service public class PersonServiceImpl implements PersonService {

    private final PersonRepository personRepository;

    private final ManageUserAccessRepository manageUserAccessRepository;

    private final FamilyTreeAuditEventService familyTreeAuditEventService;

    public PersonServiceImpl(PersonRepository personRepository,
                             ManageUserAccessRepository manageUserAccessRepository,
FamilyTreeAuditEventService familyTreeAuditEventService) {
        this.personRepository = personRepository;
        this.manageUserAccessRepository = manageUserAccessRepository;
        this.familyTreeAuditEventService = familyTreeAuditEventService;
    }
@Override public Person updatePerson(PersonUpdateRequest personUpdateRequest, String userId) {

    validateContributorOrOwnerAccess(personUpdateRequest.getFamilyTreeId(), userId);

    Optional<Person> personForUpdate =
personRepository.findPersonByPersonId(personUpdateRequest.getPersonId());
    Person person = personForUpdate.orElseGet(Person::new);
    person.setPersonId(personUpdateRequest.getPersonId());
    person.setFamilyTreeId(personUpdateRequest.getFamilyTreeId());
    person.setFirstName(personUpdateRequest.getDetail().getFirstName());
    person.setLastName(personUpdateRequest.getDetail().getLastName());
    person.setBirthdate(personUpdateRequest.getDetail().getBirthdate());
    person.setDemiseDate(personUpdateRequest.getDetail().getDemiseDate());
```

```
    person.setGender(personUpdateRequest.getDetail().getGender());
    person.setAvatar(personUpdateRequest.getDetail().getAvatar());
    person.setLabel(personUpdateRequest.getDetail().getLabel());
    person.setDesc(personUpdateRequest.getDetail().getDesc());
    person.setCountryOfCitizenship(personUpdateRequest.getDetail().getCountryOfCitizenship());
    person.setAwardReceived(personUpdateRequest.getDetail().getAwardReceived());
    person.setPlaceOfBirth(personUpdateRequest.getDetail().getPlaceOfBirth());
    person.setEducatedAt(personUpdateRequest.getDetail().getEducatedAt());
    person.setOccupation(personUpdateRequest.getDetail().getOccupation());
    person.setPositionHeld(personUpdateRequest.getDetail().getPositionHeld());
    person.setEmployer(personUpdateRequest.getDetail().getEmployer());
    person.setResidence(personUpdateRequest.getDetail().getResidence());

person.setLanguagesSpokenWritten(personUpdateRequest.getDetail().getLanguagesSpokenWritten());
    person.setSignificantEvent(personUpdateRequest.getDetail().getSignificantEvent());
    person.setOwnerOf(personUpdateRequest.getDetail().getOwnerOf());
    Person saved = getSaved(person);

    PersonUpdateRelationshipDTO updateRelationship =
personUpdateRequest.getPersonRelationshipDTO();
    if (updateRelationship != null) {
        if (updateRelationship.getFather() != null) {
            Optional<Person> relatedPersonOpt =
personRepository.findPersonByPersonId(updateRelationship.getFather());
            if (relatedPersonOpt.isEmpty()) {
                log.info("The related person with ID {} is not part of the family tree.",
updateRelationship.getFather());
            } else {
                personRepository.createParentChildRelationship(updateRelationship.getFather(),
saved.getPersonId());
            }
        }
        if (updateRelationship.getMother() != null) {
            Optional<Person> relatedPersonOpt =
personRepository.findPersonByPersonId(updateRelationship.getMother());
            if (relatedPersonOpt.isEmpty()) {
                log.info("The related person with ID {} is not part of the family tree.",
updateRelationship.getMother());
            } else {
                personRepository.createParentChildRelationship(updateRelationship.getMother(),
saved.getPersonId());
            }
        }
        if (!CollectionUtils.isEmpty(updateRelationship.getSpouses())) {
            for (String spouseId : updateRelationship.getSpouses()) {
                Optional<Person> relatedPersonOpt =
personRepository.findPersonByPersonId(spouseId);
                if (relatedPersonOpt.isEmpty()) {
                    log.info("The related person with ID {} is not part of the family tree.",
spouseId);
                } else {
                    personRepository.createMarriageRelationship(saved.getPersonId(), spouseId);
                }
            }
        }
        if (!CollectionUtils.isEmpty(updateRelationship.getChildren())) {
            for (String childId : updateRelationship.getChildren()) {
                Optional<Person> relatedPersonOpt =
personRepository.findPersonByPersonId(childId);
                if (relatedPersonOpt.isEmpty()) {
                    log.info("The related person with ID {} is not part of the family tree.",
childId);
                } else {
                    personRepository.createParentChildRelationship(saved.getPersonId(),
childId);
                }
            }
        }
    }
```

```
    log.info("Updating person with ID {} by user {} with", person.getPersonId(), userId);
    familyTreeAuditEventService.addEvent(person.getFamilyTreeId(), "Person", ActionType.UPDATE,
            person.getPersonId(), person.getFirstName() + " " + person.getLastName());
    return person;
}
@Transactional(propagation = Propagation.REQUIRES_NEW)
private Person getSaved(Person person) {
    AtomicReference<Person> saved = new AtomicReference<>(personRepository.save(person));
    return saved.get();
}

private List<Optional<PersonResponseDTO>> mapPersonsToResponseDTO(List<Person> persons) {
    List<Optional<PersonResponseDTO>> responseDTOs = new ArrayList<>();
    for (Person person : persons) {
        responseDTOs.add(createResponseFromEntity(Optional.of(person)));
    }
    return responseDTOs;
}

public Optional<PersonResponseDTO> createResponseFromEntity(Optional<Person> person) {
    if (person.isEmpty()) {
        return Optional.empty();
    }

    PersonResponseDTO dto = new PersonResponseDTO();


    PersonDetailsDTO detailsDTO = new PersonDetailsDTO();
    dto.setPersonDetailsDTO(detailsDTO);

    dto.setPersonId(person.get().getPersonId());
    dto.setFamilyTreeId(person.get().getFamilyTreeId());

    detailsDTO.setFirstName(person.get().getFirstName());
    detailsDTO.setLastName(person.get().getLastName());
    detailsDTO.setBirthdate(person.get().getBirthdate());
    detailsDTO.setDemiseDate(person.get().getDemiseDate());
    detailsDTO.setGender(person.get().getGender());
    detailsDTO.setUserId(person.get().getUserId());
    detailsDTO.setCreatedBy(person.get().getCreatedBy());
    detailsDTO.setLastModifiedBy(person.get().getLastModifiedBy());
    detailsDTO.setAvatar(person.get().getAvatar());
    detailsDTO.setDesc(person.get().getDesc());
    detailsDTO.setLabel(person.get().getLabel());
    detailsDTO.setCountryOfCitizenship(person.get().getCountryOfCitizenship());
    detailsDTO.setAwardReceived(person.get().getAwardReceived());
    detailsDTO.setPlaceOfBirth(person.get().getPlaceOfBirth());
    detailsDTO.setEducatedAt(person.get().getEducatedAt());
    detailsDTO.setOccupation(person.get().getOccupation());
    detailsDTO.setPositionHeld(person.get().getPositionHeld());
    detailsDTO.setEmployer(person.get().getEmployer());
    detailsDTO.setResidence(person.get().getResidence());
    detailsDTO.setLanguagesSpokenWritten(person.get().getLanguagesSpokenWritten());
    detailsDTO.setSignificantEvent(person.get().getSignificantEvent());
    detailsDTO.setOwnerOf(person.get().getOwnerOf());
    dto.setFirstNode(person.get().getFirstNode());

    return Optional.of(dto);
}
void validateContributorOrOwnerAccess(Long familyTreeId, String userId) {
    UserAccessManagementEmbeddedId accessId = new UserAccessManagementEmbeddedId();
    accessId.setFamilyTreeId(familyTreeId);
    accessId.setMemberUserId(userId);

    UserAccessManagementEntity access = manageUserAccessRepository.findById(accessId)
            .orElseThrow(() -> new UnauthorizedAccessException("User does not have access to
this family tree."));

    UserAccessRoleEnum role = access.getUserAccessRole();
    if (role != UserAccessRoleEnum.OWNER && role != UserAccessRoleEnum.CONTRIBUTOR) {
```

```
        throw new UnauthorizedAccessException("Only OWNER or CONTRIBUTOR can perform this
action.");
    }
}
```

Repository

```
@Repository public interface PersonRepository extends Neo4jRepository<Person, String> {

    Optional<Person> findPersonByPersonId(String personId);

    @Query("MATCH (p1:Person), (p2:Person) " +
            "WHERE p1.personId = $parentId AND p2.personId = $childId " +
            "MERGE (p1)-[:PARENT_OF]->(p2)")
    void createParentChildRelationship(@Param("parentId") String parentId, @Param("childId")
String childId);

    @Query("MATCH (p1:Person), (p2:Person) " +
            "WHERE p1.personId = $person1Id AND p2.personId = $person2Id " +
            "MERGE (p1)-[:MARRIED_TO]-(p2)")
    void createMarriageRelationship(@Param("person1Id") String person1Id, @Param("person2Id")
String person2Id);
```