Robson Insights

JIB 4109

Ashutosh Chaudhari, Drew Lawton, Garret Meek, George Vlady, Jordan Umusu

Client: Dr. Patricia Moreland

Repository: https://github.com/drewmlawton/JIB-4109

Table of Contents

List of Figures

Terminology

Term	**Definition**	**Context**
API	An interface that allows different software applications to communicate with each other, often used for handling requests between the frontend and backend.	Handles communication between the frontend (React) and backend (DRF) in the application for managing obstetric cases.
AWS (Amazon Web Services)	A cloud platform offering a variety of services, such as computing power and database hosting, to build scalable applications.	Used to host and manage the application's infrastructure in the cloud.
Cloud Architecture	A network of servers hosted in the cloud, used to run applications and store data remotely rather than on local infrastructure.	Enables the system to run on remote servers, making it scalable and reducing the need for physical hardware.
Database (PostgreSQL)	A relational database used to store and manage data for applications, providing structured access and management of information.	Stores patient data and classification results, ensuring the system can efficiently query and manage information for use in the application.
DRF (Django Rest Framework)	A high-level Python framework for building APIs, providing tools for authentication, database interaction, and more.	Manages backend requests and provides responses by interacting with the database and processing business logic.
Dynamic Architecture	Describes how the components of a system interact at runtime, focusing on the flow of data and the behavior of the system as it operates.	Illustrates how the application behaves during user interactions, like querying for patient classifications or running reports.
EC2 Instance	A virtual server provided by AWS, used to host the core application logic and services.	Runs the core application logic and manages backend services like DRF, serving the main processes of the system.
NGINX	A web server and reverse proxy used to handle HTTP/HTTPS requests, serve static content, and route requests to backend services.	Routes user requests to the appropriate backend services, ensuring smooth communication between users and the system.
React	A JavaScript library for building interactive and reusable user interface components, commonly used for frontend development.	Creates the frontend of the application, ensuring an interactive and responsive user experience for medical professionals.
RDS Instance	A managed database service provided by AWS that supports PostgreSQL and handles database scaling, backups, and replication.	Hosts the PostgreSQL database, ensuring that all patient data is securely stored and easily retrievable.
Robson Classification System	A classification system for categorizing pregnancies, specifically used to determine the necessity of cesarean sections.	Used in the system to classify pregnancies and assist in monitoring cesarean section rates for better medical decision-making.
Static Architecture	The design of the system's components, relationships, and dependencies when the system is not running, providing a high-level overview of its structure.	Defines how the system is structured at rest, showing how components like the API, database, and frontend depend on each other.

WSGI Server (Gunicorn)	A Python application server that bridges web servers (like NGINX) and backend code (like DRF), capable of managing multiple worker processes.	Facilitates backend processes and manages API requests, allowing the system to handle multiple concurrent users efficiently.

Introduction

Background

This application is designed to support medical professionals in Rwanda by enhancing the management and classification of obstetric cases. It focuses on the Robson classification system, a globally recognized method introduced by the World Health Organization for categorizing pregnancies based on specific criteria that include the mother's obstetric history, the baby's gestational age, fetal presentation, and the number of babies. The system is crucial for identifying the appropriateness of cesarean sections, helping to address increasing Cesarean section rates being observed globally.

The project's main objective is to develop a digital tool that not only increases the accuracy of patient classification but also provides medical professionals with means of data visualization and sharing to monitor and analyze Cesarean section rates effectively. By doing so, the application aims to contribute to safer, more effective maternity care practices in Rwanda.

Document Summary

The ***System Architecture*** section provides a high-level overview of the major components of our application and their interactions. The static architecture diagram illustrates the logical relationships between these components, while the dynamic architecture diagram demonstrates a specific example of runtime behavior and how the components interact during that process.

The ***Component Design*** section provides a more detailed view of both the static and dynamic view of the application. The Structural Diagram shows the static relationship of the low-level components, while the Runtime Interactions diagram reveals how the static components interact during runtime.

The ***UI Design*** section exhibits how the user interacts with the application and provides an overview for the frontend. It includes all major screens that the user will interact with and provides detailed explanations for each image.

The ***Data Storage*** section provides a detailed view of how the application stores and manages the user data. The File Use subsection explains the format of the files used, while the Data Exchange subsection explains the data format and protocol used for data transfer and addresses security concerns.

System Architecture

Introduction

This portion of the document defines each component of the system, how they interact with each other, and how they interact with our clients. First, the static architecture will show each component and outline the dependencies as well as the rationale behind each choice. Next, the dynamic architecture will introduce the user to the system and will display how each component will respond to actions taken by the user. The goal of this architecture is to provide users with a stable, scalable, long-term solution. Because the client has minimal technical experience or access to a development team, the solution must be easy to manage, cheap, and scalable.

Rationale

The rationale behind the architectural choices is driven by the need to create a system that balances usability, scalability, and security for managing sensitive patient data. The choice for a cloud-based infrastructure using AWS is due to its scalability, reliability, and robust security features. By hosting the application on EC2 and using RDS for the database, the system scales with demand and provides high availability, while automating critical functions like backups and patching. The decision to use React for the frontend and Django Rest Framework (DRF) for the backend was made to create a responsive, maintainable, and secure application. DRF's modular architecture allows for easy integration with authentication mechanisms, and React's reusable components ensure a smooth user experience.

Security considerations were central to the design decisions. Given the sensitivity of patient data, HTTPS is being utilized to secure all communication between the client and server, preventing unauthorized access to data in transit. DRF was chosen specifically for its strong support for token-based authentication, which ensures secure interaction with third-party APIs and restricts access to authenticated users. Additionally, all sensitive data, including patient records, is encrypted at rest using AWS encryption services. Role-based access control (RBAC) further enhances security by limiting user access based on their permissions, ensuring that only authorized personnel can view or modify sensitive information. These security measures ensure that our system not only meets regulatory requirements but also provides a secure environment for users.

Static System Architecture

The static system architecture depicts what components of the system depend on each other. Each arrow represents dependency.

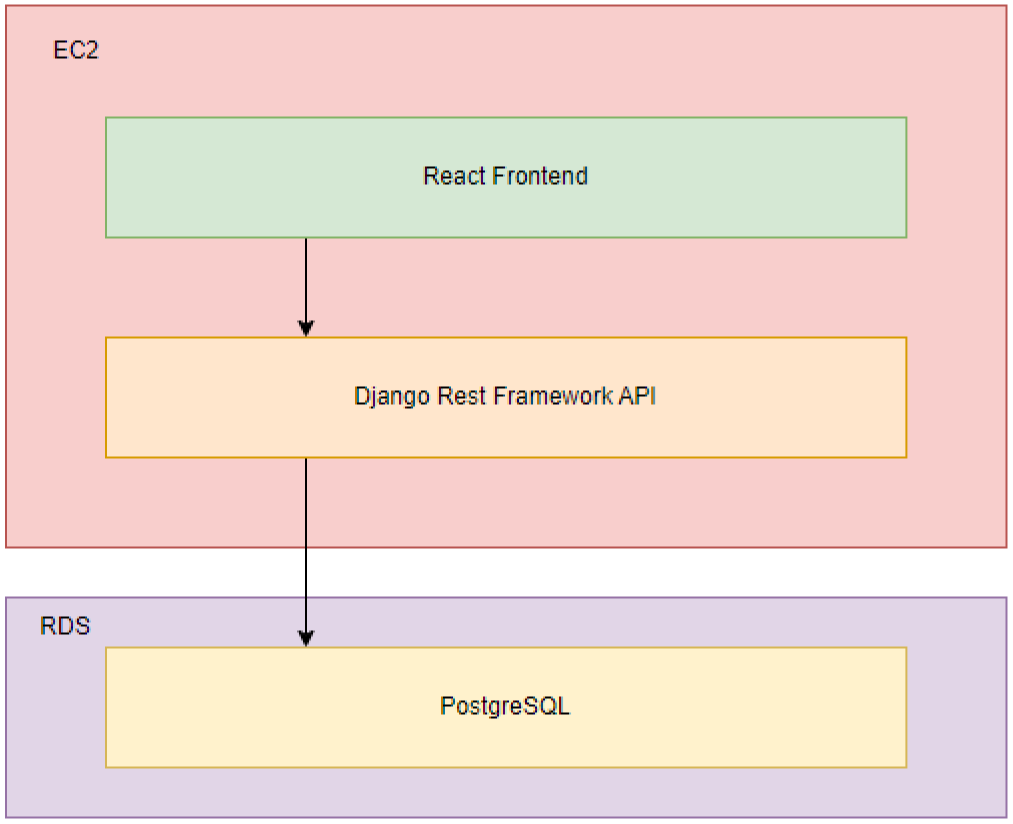


Figure 1.1 – Static Design Diagram

The cloud architecture consists of two AWS instances. First, an EC2 (Elastic Compute Cloud) instance which serves as the applications core, housing the application logic. Second, an RDS (Relational Database Service) instance, hosting the PostgresSQL database.

The first component is the frontend. React was chosen as the frontend framework utilized create an easy to use, interactive, and responsive user experience. React components make it easy to reuse common elements and seamlessly call the rest API. Being the most widely used and supported frontend framework, our developers would be able to resolve any issues quickly by reaching out for community support.

The next layer is the API. This component handles all the backend logic involved with querying our database and presenting that data to the frontend in a way that helps the user achieve their goals. DRF was chosen for this component because it comes with out of the box authentication and easy integration with most major database services.

Our bottom layer is the database. The database will store all the data needed by our application. The underlying RDS instance will handle all the database management including backups, scaling, patching, and replication.

Dynamic System Architecture

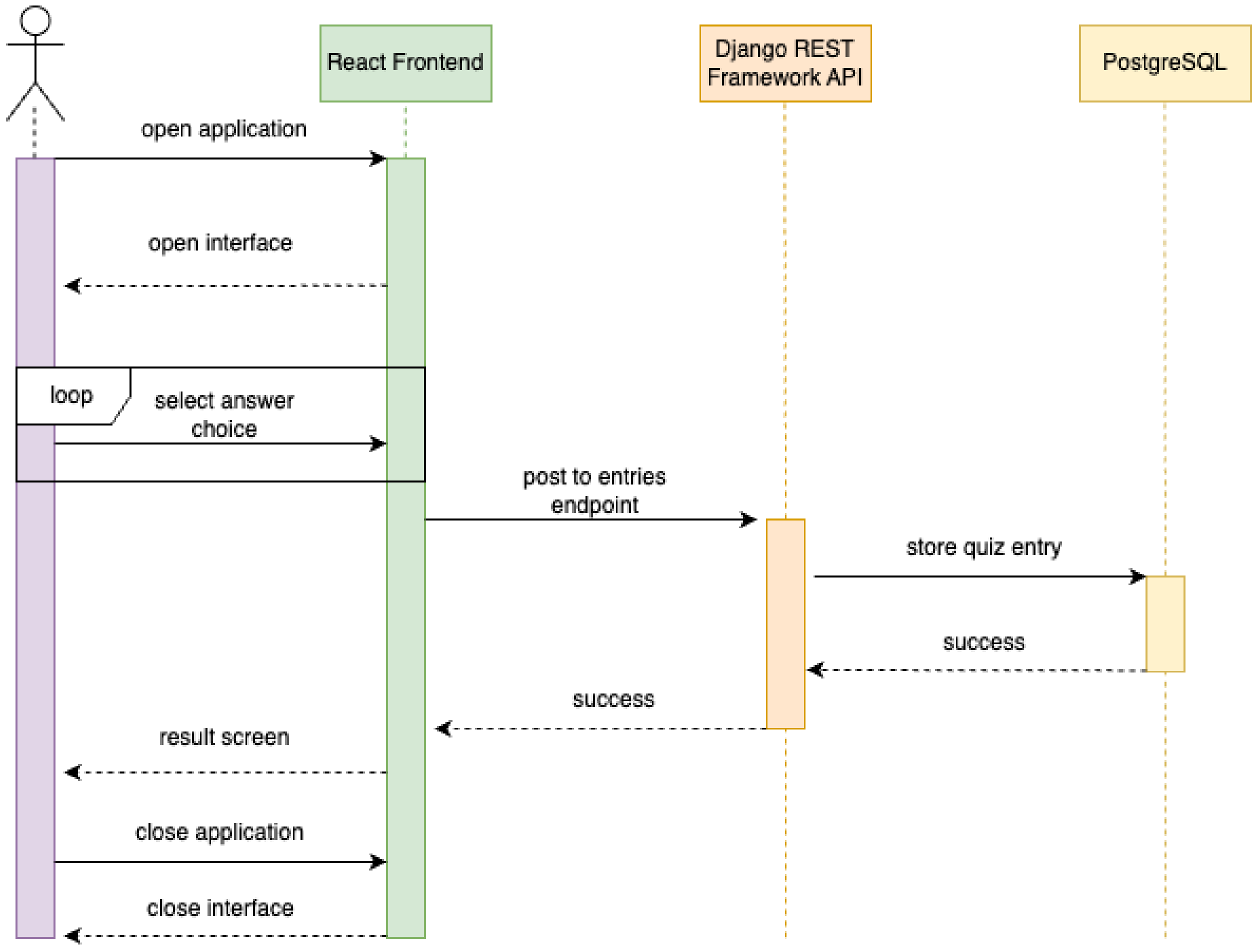


Figure 1.2 - Dynamic Design Diagram

The dynamic design diagram is a system sequence diagram that shows the control flow between our application layers for a particular scenario–taking the Robson classification quiz–and illustrates how data is exchanged between the application layers. Upon opening the application, the user is presented with a series of questions. The user repeatedly selects answer choices to complete the quiz. Once the user completes the sufficient number of questions to accurately classify the pregnancy, the React frontend posts to our entries endpoint, which stores the result of the quiz along with the user's profile information. When the database layer successfully stores this data, it notifies the backend, and the entries endpoint returns successfully. Finally, the user is presented with the results screen. This scenario represents the key functionality of our application, the classification process and storage of that data, and it utilizes each layer of our architecture while showcasing state management and API integration, making it an important aspect to depict.

Component Design

Introduction

The following section will expand on the above system architecture, showcasing each component in more detail.   To illustrate the static components of our system, we have assembled a class diagram (Figure 2.1). This diagram highlights the core classes within each architectural component, their relationships, and their responsibilities, ensuring a clear understanding of the system's foundational building blocks. For the dynamic side, we have chosen to use an interaction overview diagram (Figure 2.2) illustrates the runtime interplay between the frontend and backend. This diagram outlines how the system processes various user inputs, showcasing the flow of data and control through the architecture. Together, these diagrams provide a structural and behavioral view into our application.

Static Elements

In Figure 2.1, we present the static detailed components of our system, structured in alignment with the MVC (Model-View-Controller) architecture. The diagram illustrates the relationships between the Presentation Layer, Application Layer, and Data Layer, with dependency relationships depicted using <<create>> and <<use>> arrows. The <<create>> arrows show which components are responsible for instantiating others, such as the Server creating Invite objects and triggering email notifications through the Email Client. The <<use>> arrows represent dependency relationships, such as the Groups component relying on the Server for backend operations, or the Server interacting with the UserProfile and Invite components to manage group permissions and invitations. In the Presentation Layer, components like Home, Results, and Groups leverage React hooks (useEffect, useState) to manage state and interact with the Server for API calls, such as fetching groups or creating invitations. In the Application Layer, the Server serves as the central orchestrator, managing operations like authentication, group membership, and notifications by interfacing with the Data Layer models and the Email Client. Lastly, the Data Layer defines core entities like User, Group, Invite, and UserProfile, establishing key attributes and relationships, such as linking a user to a group and defining permissions (is_admin, can_view).

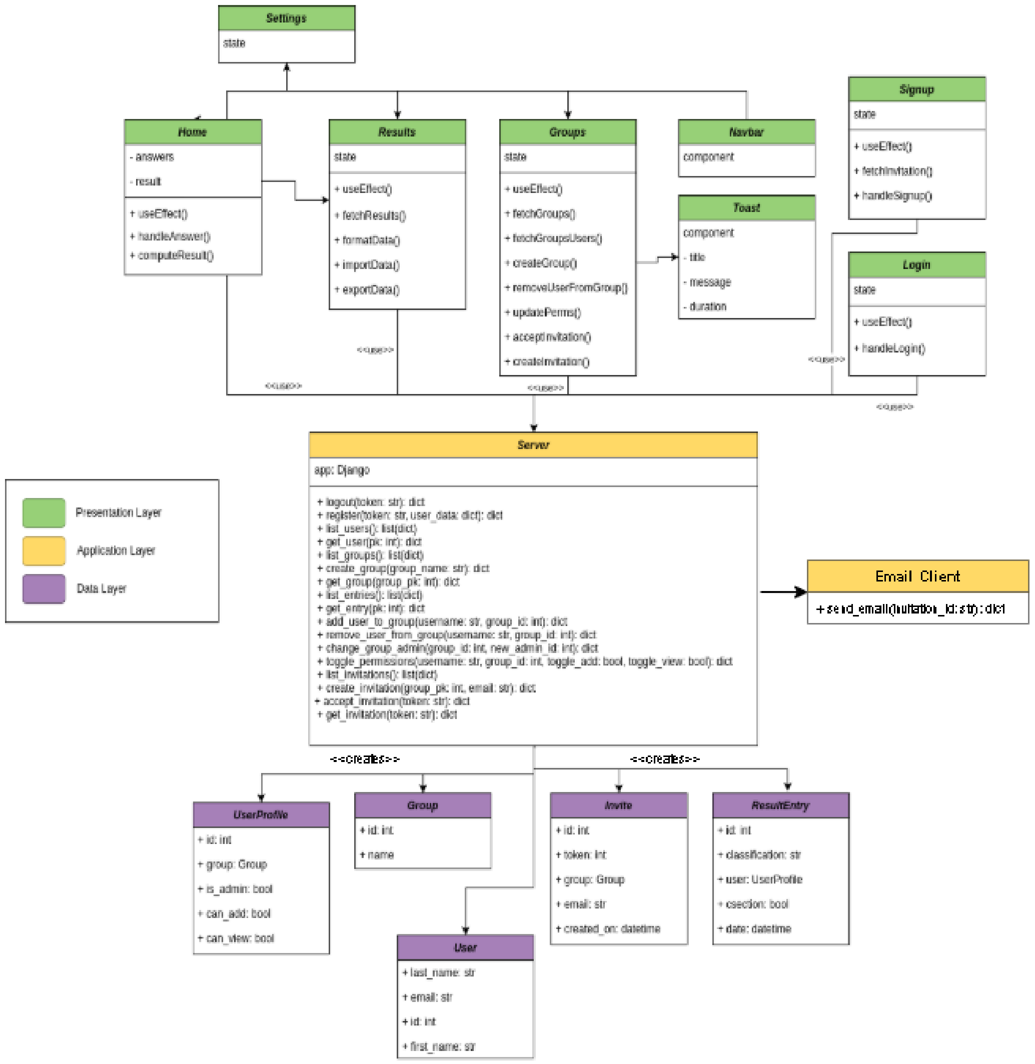


Figure 2.1 - Class Diagram

Dynamic Elements

Figure 2.2 is an interaction overview diagram showing the functionality of the group membership management screen. It shows the interaction flow when a user decides to add or remove a user from/to a group. If the user wants to add a user not currently on the platform, it creates an invitation within the appropriate group and then sends that new user an invitation email. If the user wishes to add a user already on the platform (said user belongs to differing groups), an invitation is created for the invited user to examine once they open the application. If a user wants to remove a member, it will simply delete the corresponding user profile from the group. After both scenarios, the displayed collection of users is updated and rendered again.

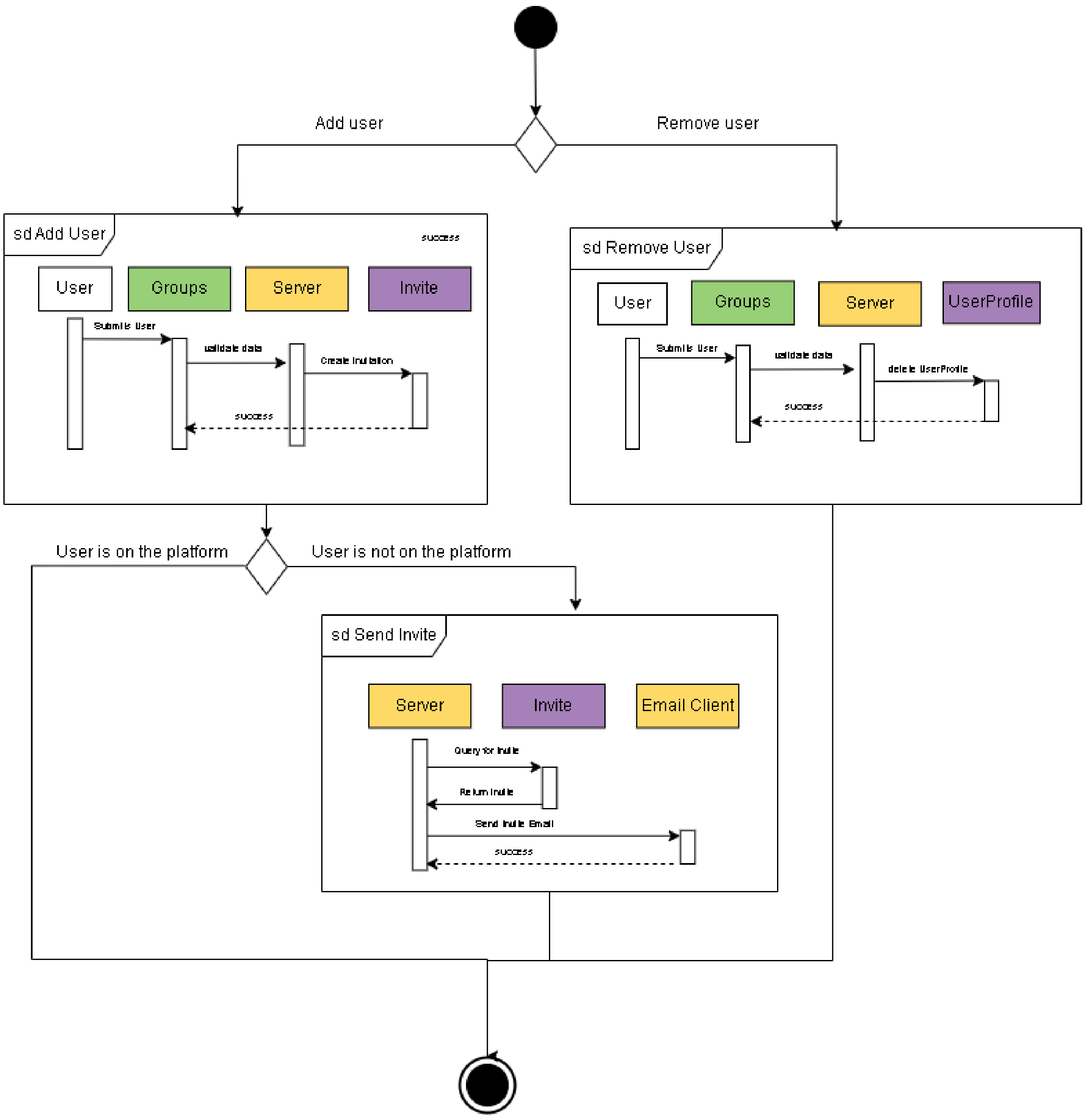


Figure 2.2 – Interaction Diagram

Data Design

Introduction

This section will outline what data our app is storing, how the data is structured, and how we transfer that data from client to server. For more specific information about our API calls, refer to the section on user interface design.

Database Use

Below we have provided an Entity Relationship (ER) diagram. This diagram shows entities (rectangle), entity attributes (oval), and relationships between entities (diamond).

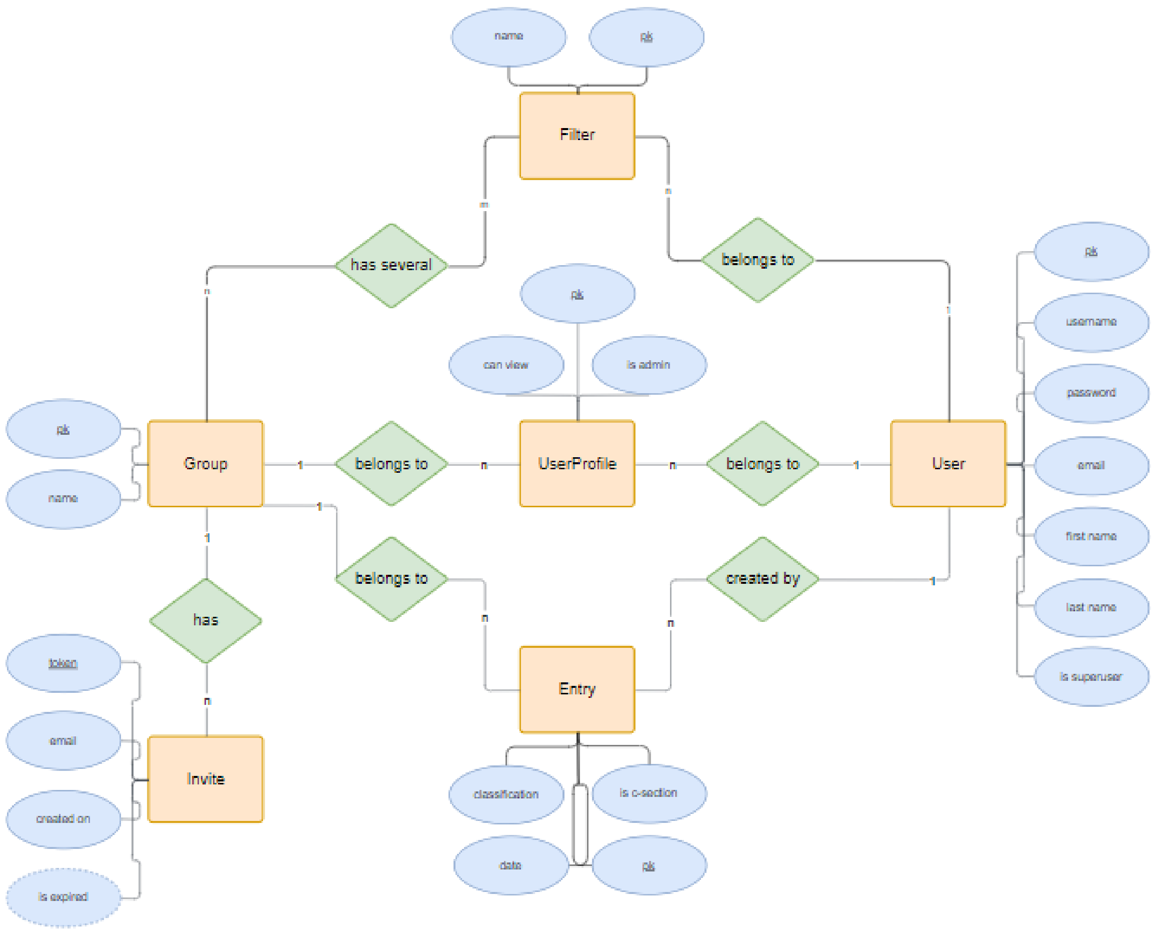


Figure 3.1 - Entity Relationship Diagram

As the diagram shows, there are six entities: Group, User, UserProfile, Filter, Entry, and Invite.

1. Group:

Groups are how we segment users. Each group has a name and an integer primary key. With this design, two different groups can have the same name.

2. User:
   Like most apps, we have a user entity. This entity stores data that belongs to each individual user and helps with our authentication logic.
3. UserProfile:
   This entity serves as a bridge between users and groups. Because each user can be in multiple groups but have different roles in each group, we needed an entity between them to describe the role of the user in the group (admin, can view permissions)
4. Filter:
   Filter provides our users with an easy way to aggregate data across groups. Each filter has a list of groups. If the user selects a filter, we display all the entries in the filter's groups.
5. Entry:
   The entry entity represents the data our client is interested in tracking – what classification each pregnancy falls into and whether it was a c-section.
6. Invite:
   Invite helps with how we manage our user's registration process. A group admin can send an invite to a potential member. An invite is added to the database and a link with the invite token is sent to the email. We then use this token in the registration URL to know if the user was invited and to what group they were invited.

File Use

A quarterly report template intended to be downloaded, filled out, and uploaded, is available as a CSV file on our application. All icons come from external libraries such as Lucide, so we have no need for additional images.

Data Exchange

We chose to use a REST API to send data from server to client. We are formatting this transaction of data using JavaScript Object Notation (JSON).

Data Security

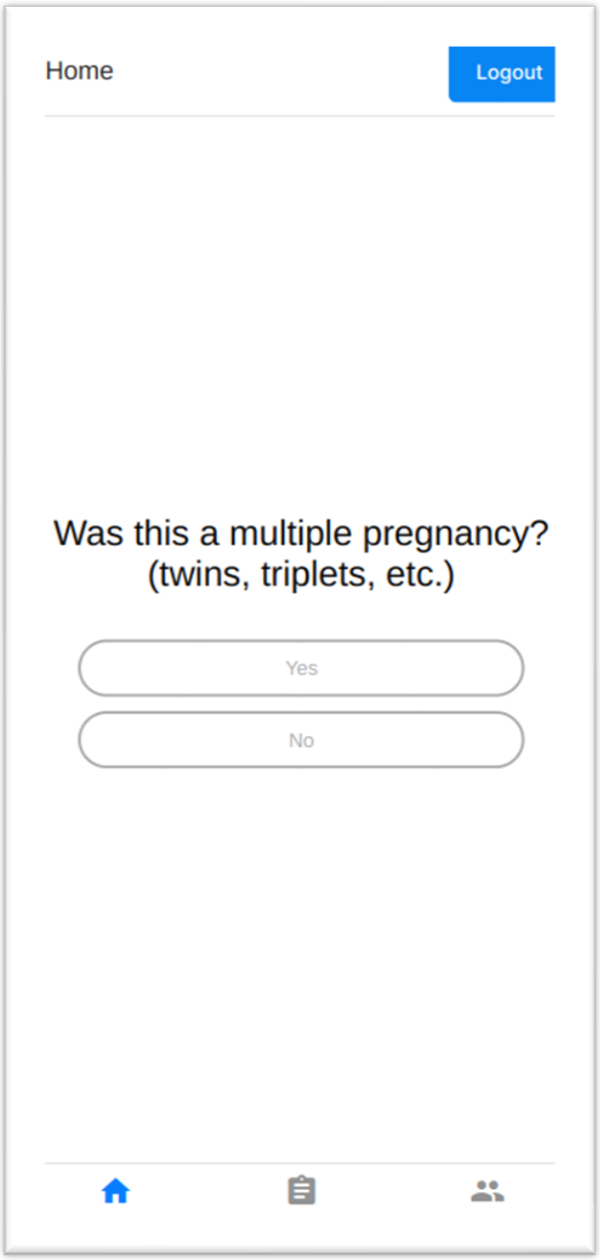
All communication between the client and the server is conducted over HTTPS, ensuring that data in transit is encrypted and protected from unauthorized access or tempering. User authentication is required to access the application. We employ token-based authentication using Django REST Framework's built-in authentication system. This ensures that only authenticated users can access protected resources. User credentials are never stored in plain text; instead, we use a secure hashing algorithm to store password hashes. Role-based access control is implemented to ensure that users can only access the data and perform actions they are authorized to. Each user is assigned specific permissions based on their role (e.g., admin, regular user), which are enforced by backend API. Personally Identifiable Information, such as usernames and email addresses, is stored in the database. It is protected through database encryption and restricted access via the application's authentication system. When transferring sensitive data, such as survey results, the application ensures that only authenticated and authorized users can access the endpoints.

UI Design

Introduction

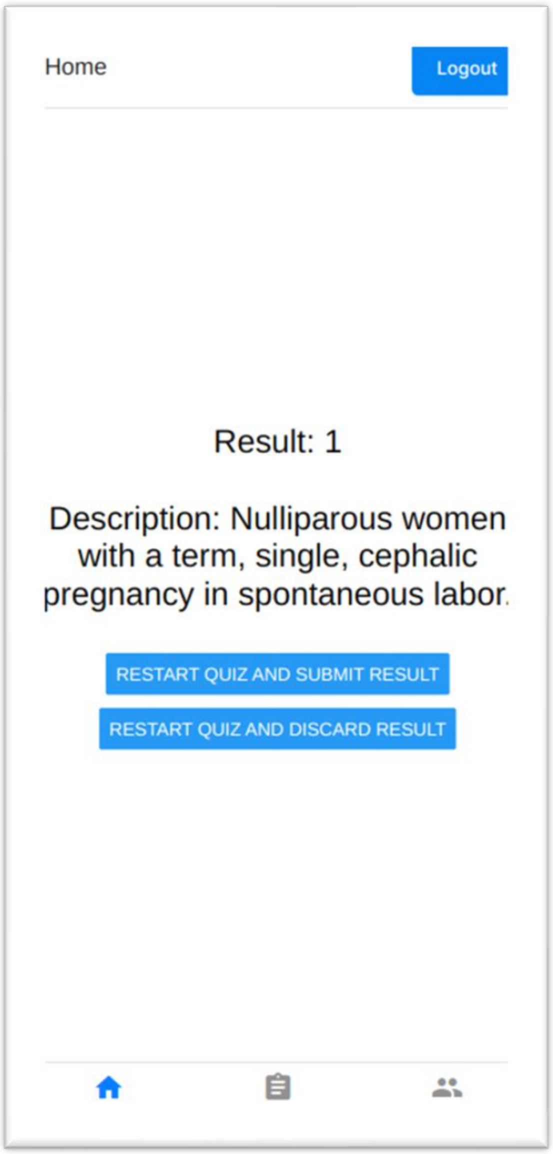
This section will highlight the core User Interface (UI) within our application. This UI will allow users to classify their patients via the Robson Classification, import and export data to and frm the application, view aggregated patient data for any group that user belongs to, invite others to the application, and lastly create and/or update any managed groups or group configuration views. We've created a simple component library to standardize visual elements throughout the app, with elements such as dropdowns, input fields, and checkboxes as examples.

When the application is first opened, a user is shown a screen prompting the user to fill out a patient questionnaire as can be seen in Figure 4.1. The questionnaire uses familiar medical terminology relevant to users' expertise, aligning with the Match Between System and the Real World heuristic. Once the questionnaire has been completed, the Robson group classification of that patient and a description of the classification is displayed immediately following, along with the ability to save or discard this data (Figure 4.2). This follows a similar theme seen throughout the app where all user actions have near immediate feedback such that the application's status is clear throughout the entirety of use. This design approach adheres to the Visibility of System Status heuristic.



Was this a multiple pregnancy? (twins, triplets, etc.)

Figure 4.1: Application Homepage Screen


Home

Result: 1

Description: Nulliparous women with a term, single, cephalic pregnancy in spontaneous labor.

Logout
RESTART QUIZ AND SUBMIT RESULT
RESTART QUIZ AND DISCARD RESULT

Figure 4.2: Completed questionnaire Screen

If a user decides to save their questionnaire data, they can later view this data by clicking on the Results tab button at the bottom of the screen. The User Control and Freedom heuristic is also supported by the users' ability to discard data at the end of the questionnaire in case mistakes are made or data is no longer relevant. All primary navigation occurs through this bottom navigation bar which is recognizable to users across apps and is easy to interact with meeting the Consistency and Standards as well as the Recognition Rather Than Recall heuristic. On this page, the user can see aggregated classification data across differing groups that the user belongs to and can toggle between these groups using the dropdowns immediately above the visualizations. Additionally, the user can click on any graph on the screen to get quantitative and classification specific information as seen in Figure 4.4. The Flexibility and Efficiency of Use heuristic is addressed by the hot bar at the top of the screen allowing the users to export or import spreadsheet data and as per the Help Users Recognize, Diagnose, and Recover From Errors heuristic the system will also display alert popups on the screen if the data formatting if user or server errors occur to ensure that the user always stays informed about any issues that arise.

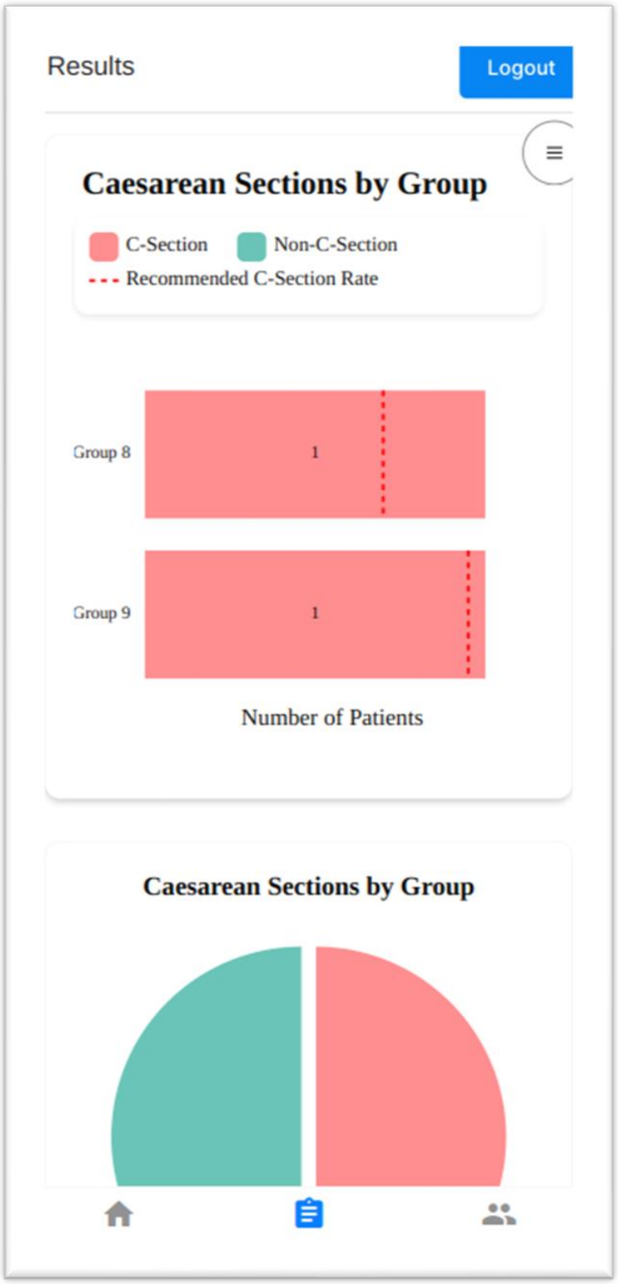
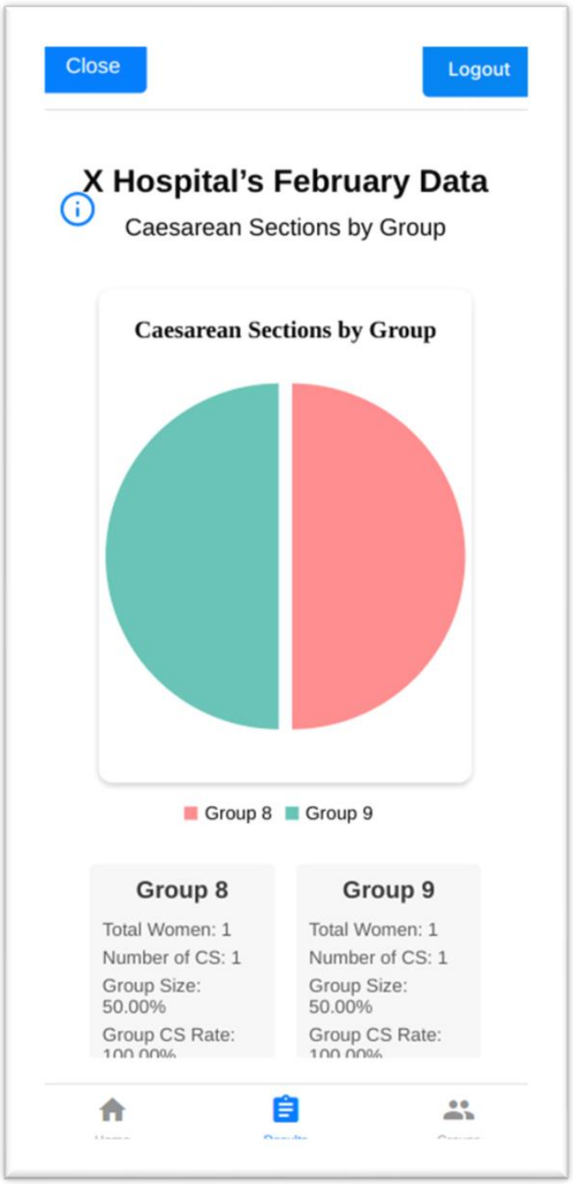

Results
Logout
Caesarean Sections by Group
C-Section
Non-C-Section
Recommended C-Section Rate
Group 8
1
Group 9
1
Number of Patients
Caesarean Sections by Group

Figure 4.3: Data Visualization Screen



X Hospital's February Data

Caesarean Sections by Group

Caesarean Sections by Group

🟥 Group 8 🟩 Group 9

Group 8	Group 9
Total Women: 1	Total Women: 1
Number of CS: 1	Number of CS: 1
Group Size: 50.00%	Group Size: 50.00%
Group CS Rate: 100.00%	Group CS Rate: 100.00%

Figure 4.4: More information Screen

A user can also navigate to the last tab button at the bottom of the screen to manage and/or create groups or group configuration views (for visualization). Via the use of dropdowns, a user can select the group they wish to modify, and given they have access, they can add, remove and modify permissions on an individual level for each member within that group. Below the group management section is a similar section that allows a user to create and update configurations for data viewing, enabling one to select/deselect varying groups within for a high-level data overview.

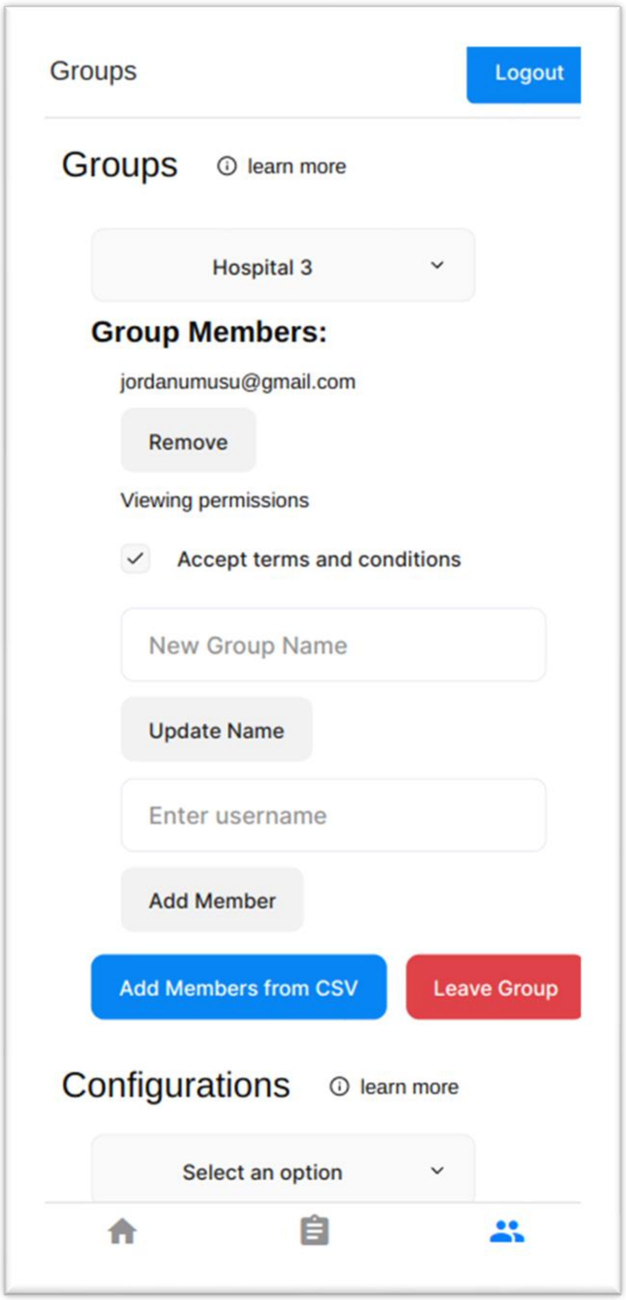


Figure 4.5: Group Management Screen

The group management screen allows for users to create, manage, and configure their groups within the app which facilitates the means for sharing a user's survey results with other doctors/administrators. A configuration is a means of saving a specific view of some combination of groups for quickly revisiting in the future. Pressing the create Group/Configuration buttons prompts a text field to open to enter and confirm the name of the new group/configuration. And once selecting an already created group/configuration using the drop-down menu, a user can view different users within the group, invite new users to the group, remove users from the group, or edit the viewing permissions of users within the group. The interface applies the Aesthetic and Minimalist Design heuristic by displaying only essential controls, such as dropdown menus for group selection and buttons for creating or modifying configurations. Tooltips explain group permissions and configuration options, ensuring adherence to the Help and Documentation heuristic.

Appendix

Overview of the Robson Classification System

The Robson Classification System, also known as the Ten-Group Classification System (TGCS), is a globally recognized method for categorizing women into clinically relevant groups based on obstetric characteristics. Developed by Dr. Michael Robson in 2001, this system aims to facilitate the analysis and comparison of cesarean section rates within and between different hospitals and healthcare settings.

By providing a standardized framework, the Robson Classification allows for the identification of specific groups contributing most to the overall cesarean section rate. This, in turn, helps healthcare providers develop targeted strategies to optimize obstetric care and reduce unnecessary cesarean deliveries.

The Robson Classification System divides pregnant women into ten mutually exclusive and totally inclusive groups based on six obstetric characteristics:

1. Parity (nulliparous or multiparous)
2. Previous cesarean section (presence or absence)
3. Gestational age (preterm or term)
4. Fetal presentation (cephalic, breech, or transverse)
5. Number of fetuses (singleton or multiple)
6. Onset of labor (spontaneous, induced, or pre-labor cesarean section)

Below is a detailed description of each group:

Group 1 - Description: Nulliparous women with a single, cephalic pregnancy, at term, in spontaneous labor.

Group 2 - Description: Nulliparous women with a single, cephalic pregnancy, at term, who had labor induced or were delivered by pre-labor cesarean section.

Group 3 - Description: Multiparous women without a previous cesarean section, with a single, cephalic pregnancy, at term, in spontaneous labor.

Group 4 - Description: Multiparous women without a previous cesarean section, with a single, cephalic pregnancy, at term, who had labor induced or were delivered by pre-labor cesarean section.

Group 5 - Description: Multiparous women with at least one previous cesarean section, with a single, cephalic pregnancy, at term.

Group 6 - Description: Nulliparous women with a single, breech pregnancy.

Group 7 - Description: Multiparous women with a single, breech pregnancy.

Group 8 - Description: Women with multiple pregnancies (twins, triplets, etc.).

Group 9 - Description: Women with a single pregnancy in a transverse or oblique lie.

Group 10 - Description: Women with a single, cephalic pregnancy, at a gestational age less than 37 weeks.