Beauty Consulting App

JIC 4103
Andrew Chen, Clark Cousins, David He, Sohum Rao,
William Shutt, Anand Tsogtjargal
Client: Jerrian Sari
Repo: https://github.com/sohumrao/JIC-4103-Beauty-Consulting-App

Table of Contents

Table of Figures

Terminology

Term	Definition	Context
Expo	Platform for universal React applications to be tested and deployed	Bundles the code and allows the app to be published across devices
Express.js	Minimal Node.js web app framework	Manages API routing between frontend and backend
JSON	Lightweight format for storing data	Used to format data between frontend and backend
MongoDB Atlas	Cloud-based service that manages and scales databases	Hosts the MongoDB database for the app
MongoDB	Database that stores JSON-like data	Stores user data, such as profiles and hair related details
Node.js	JavaScript runtime for server-side programming	Forms the backend of the app.
React Native	An open-source framework for building mobile apps using JavaScript and React. It allows developers to create apps for both iOS and Android platforms	Powers the app's front-end user interface
REST	An architectural style for designing networked applications, relying on stateless, client-server communication	Express API follows REST principles to manage communication between front end and back end

Introduction

Background

This project is a mobile app designed to improve beauty consultations by enhancing communication between clients and stylists. The app's key feature is a client profile creation interface that allows users to enter personal information and hair-related concerns. By enabling clients to share important details with stylists in advance, both parties can enjoy a smoother, more efficient experience. The app showcases the technology stack, with a React Native front end connected to a Node.js backend running an Express API. User data is stored securely in a MongoDB Atlas cloud database. Additionally, the Expo library ensures the app works seamlessly on both iOS and Android by bundling JavaScript through Expo Go.

Document Summary

The **System Architecture** outlines how the different components of the app function together. The front-end, built with React Native, serves as the user interface where clients enter their information. This interface communicates with the backend, which runs on Node.js and utilizes the Express framework to handle API requests. The backend processes these requests and interacts with the MongoDB Atlas database to store and retrieve data. Expo is used to ensure the app runs smoothly across both iOS and Android devices.

The **Component Design** zooms into the details the components of the system, examining how the frontend structures various flows of UI interaction and how the backend organizes routes into controllers with clusters of related business-logic interfaces.

The **Data Design** describes the design of the database schema, explaining how various data entities like accounts, messages, and appointments are organized in the MongoDB database. In addition, this section also explains how uploaded photos are stored efficiently, how the backend exposes the database through RESTful API endpoints, and how different parts of the system exchange data through HTTP requests.

The **UI Design** describes how the design aligns with the 10 Usability Heuristics to create an intuitive mobile application experience. The app's main screens include a stylist directory for clients, appointment booking interfaces, and appointment management screens, each designed with minimal, recognizable components that prioritize user understanding.

System Architecture

Introduction

The primary goal of the system is to provide users with an intuitive interface to create accounts and manage their profiles. These system architecture diagrams outline the process of storing and retrieving user details through seamless data transmission between the frontend, backend, and database layers. The application is designed to handle user profiles that include hair-related preferences and information, aiming to create a personalized experience for each user.

Rationale

Our team chose this architecture to ensure a smooth, positive user experience tailored to the needs of our project. React Native was selected for the frontend because it allows us to build a consistent, cross-platform interface for both iOS and Android users. Node.js with Express forms the backend to efficiently handle API requests and manage business logic, while MongoDB serves as the database for securely storing diverse user data, including profiles, photos, messages, and appointments in a flexible, cloud-hosted environment. Key security considerations include encrypting passwords to protect user data, using HTTPS to secure all client-server communication, and carefully managing sensitive environment variables like database credentials. These architectural decisions enable us to deliver a reliable and user-centric application while maintaining robust data protection.

Static Architecture

Diagram

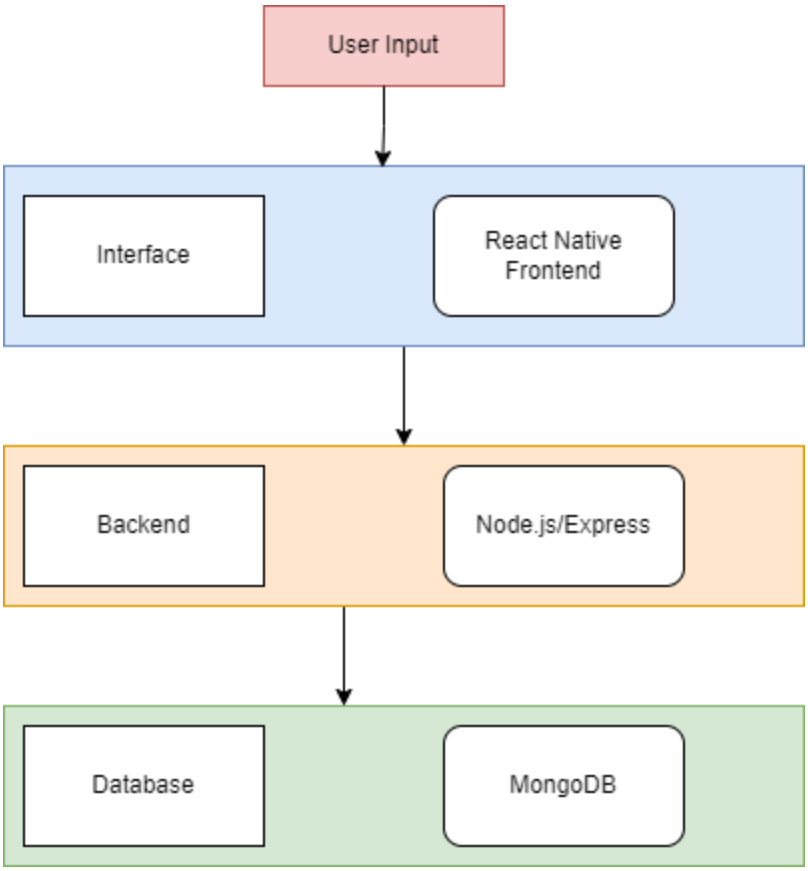

Figure 1- Static System Diagram

Our static system architecture is structured to support interaction between the user interface, backend, and database components. The system's main objective is to provide a seamless experience for users to create accounts and manage their profiles.

At the top, we have the frontend built with React Native, which serves as the interface where users can enter their data and view stored information. The core operations take place in the backend, implemented using Node.js/Express. This layer processes the requests from the frontend, handling the logic for account creation, profile management, directory, appointments, and messaging. The backend then communicates with the database, MongoDB, to store and retrieve user data. MongoDB is responsible for securely managing the stored information, ensuring data integrity and accessibility.

This architecture was chosen to ensure the application remains secure and scalable. The modularity of this structure also allows for easy updates and scalability as the application grows.

Dynamic Architecture

Diagram

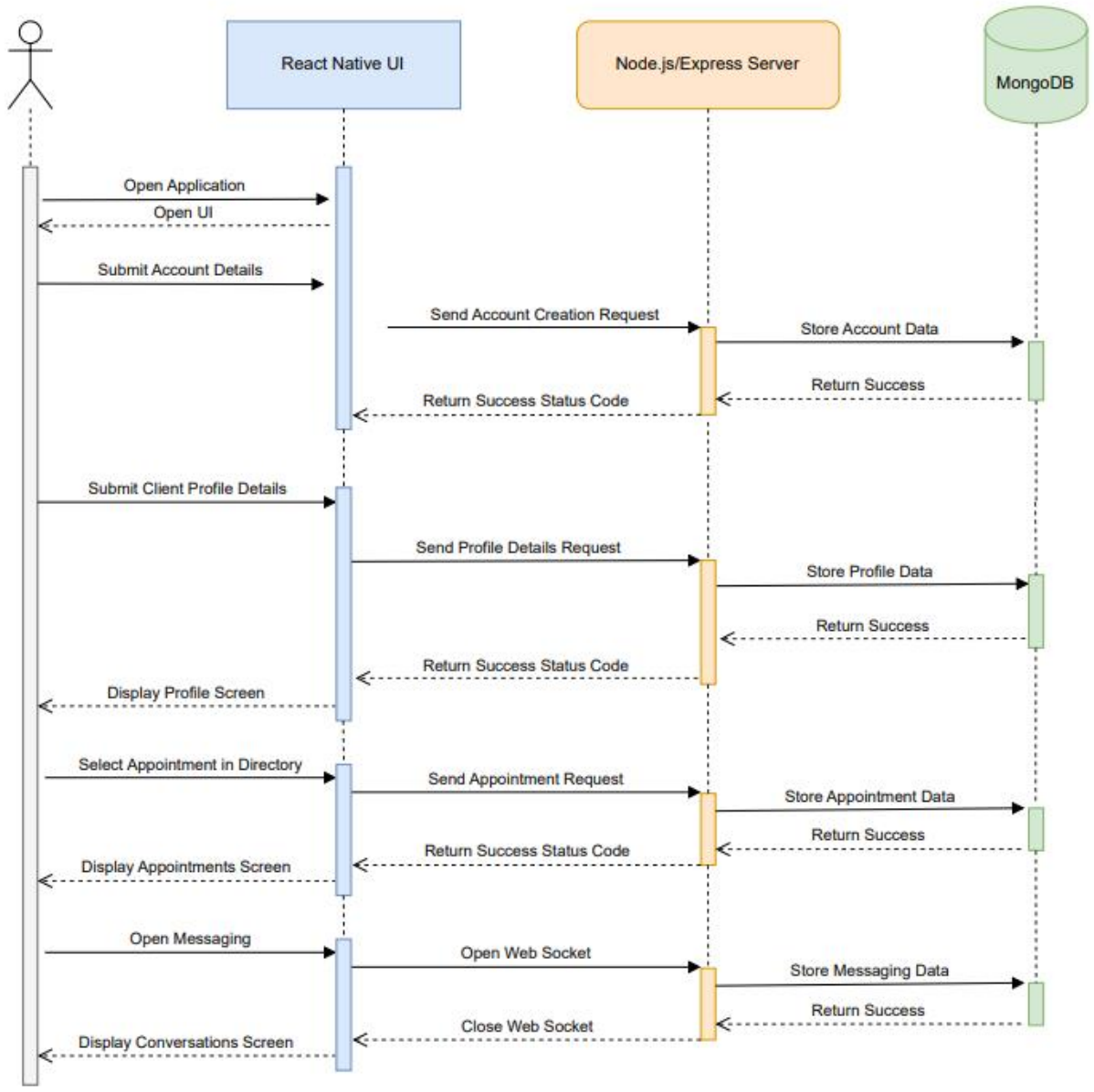

Figure 2 - Dynamic System Diagram

We chose this UML diagram type for the dynamic system architecture because it effectively captures the sequence of interactions between the **React Native UI**, **Node.js/Express** backend, and **MongoDB** database during key user workflows. By focusing on specific features such as account creation, profile submission, appointment scheduling, and messaging, this diagram highlights the core functionality and runtime behavior of the application. These features were selected because they represent the most critical interactions for the user experience and demonstrate how different system components collaborate to deliver seamless, end-to-end functionality. This level of detail allows us to clearly convey the flow of data and the

responsibilities of each architectural layer, ensuring a comprehensive understanding of the system's dynamic behavior.

When the user first launches the application, the **React Native UI** component renders the user interface. This component is responsible for displaying data as well as taking data as input to be transmitted to the underlying backend server. In this specific use case, the **React Native UI** handles data input for account creation and beauty client profile data, as well as displaying said data after it has been stored.

The backend layer of the application is the **Node.js/Express Server** component. This component is responsible for handling requests from the **React Native UI** and carrying storage/retrieval of data. In the depicted scenario, the server handles an account creation request by storing the data and returning a success status to the frontend. The logic for the client profile data is carried out the same way, with specific error handling logic implemented for both situations. The **React Native UI** component receives status codes from the Node.js/**Express Server** component and either displays an error, or allows the user to proceed with their profile creation, appointment creation, and messaging processes.

The database layer of the application, **MongoDB**, has the task of housing the actual data of the application. In this scenario, account and client profile data is sent from the **Node.js/Express Server** when it handles requests from the frontend, and the data is stored in **MongoDB.** The database is hosted remotely using the MongoDB Atlas cloud service, where it is available for retrieval at any time by the other components of the application.

Component Design

Introduction

The component design of our app offers a detailed view of how key elements such as account management, client profiles, and photo uploads are handled. This section elaborates on the interactions between the backend services that support user accounts, and client data using MongoDB. The static component diagram illustrates the relationships between these controllers, the front-end React Native interface, and the MongoDB database. Meanwhile, the dynamic component diagram demonstrates how the system handles critical processes, such as account creation, by depicting runtime interactions between components. These views maintain the conceptual integrity of the system's architecture while ensuring efficient data handling and scalability.

The conceptual integrity between the component diagram and the architecture diagrams was maintained through consistent color coding and alignment of responsibilities across components. Each layer in the static component diagram—frontend, backend, and database—is visually distinct with its own color (blue, yellow, and green respectively), mirroring the separation of concerns depicted in the system architecture diagrams. This alignment ensures clarity in how the components interact and reflect their roles in the system. The logical flow of data, from user-facing features to backend controllers and database entities, is preserved across both diagrams, emphasizing the modularity and clear boundaries of the architecture.

Static Component

Diagram

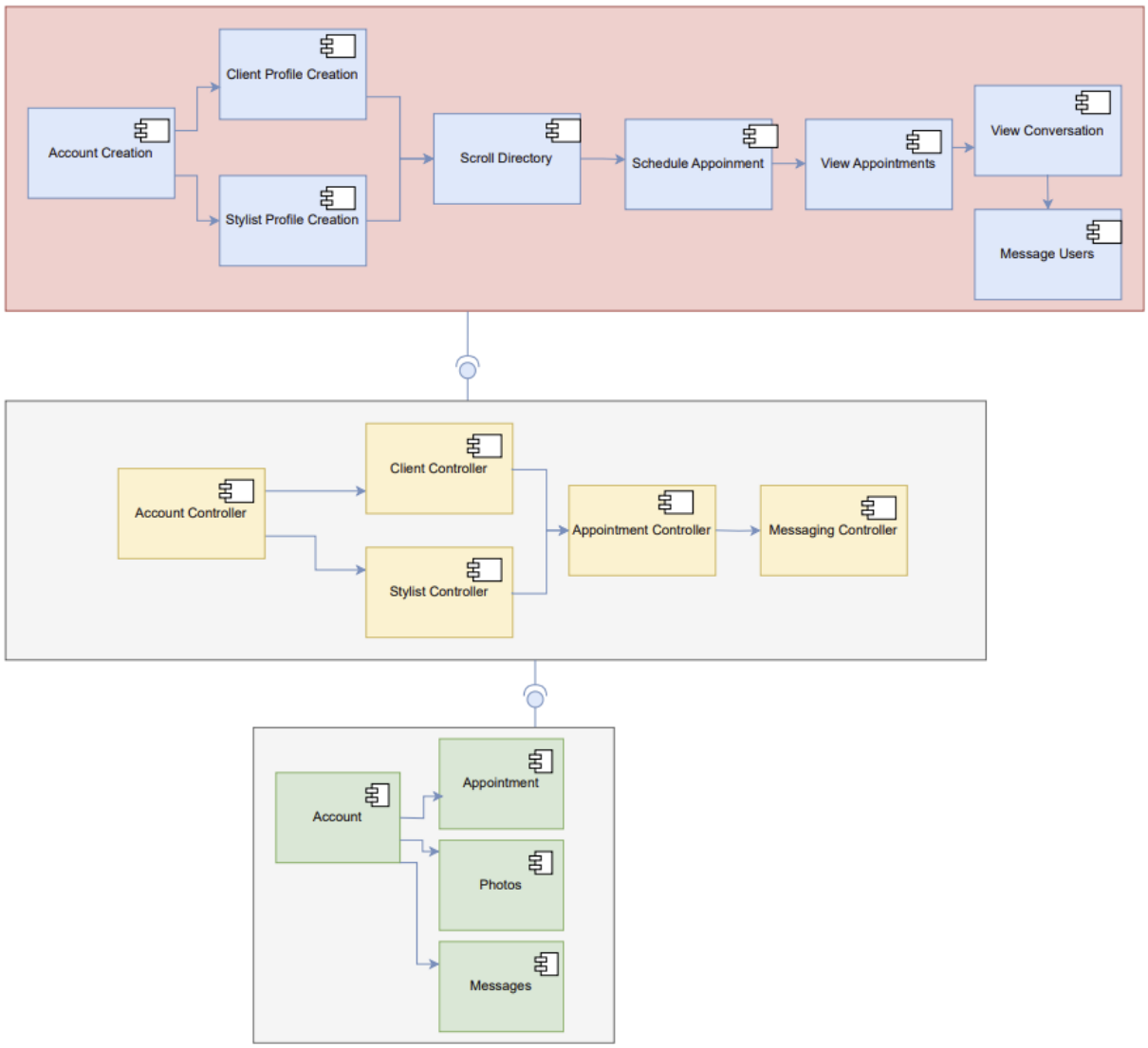

Figure 3 - Static Component Diagram

The diagram depicts the relationships between key components involved in profile creation, user account management, system navigation, messaging, and appointment scheduling. In the top section, Account Creation feeds directly into Client Profile Creation and Stylist Profile Creation, signifying that user authentication is required before profile creation can occur. These profile creation components link to key features such as the Scroll Directory, Schedule Appointment, and View Appointments screens, which rely on completed user profiles to enable browsing stylist options, booking appointments, and managing existing bookings. Additionally, the View Conversation and Message Users screens allow users to access and participate in messaging functionality.

The middle section illustrates the backend logic, with distinct controllers handling different aspects of the application: the Account Controller manages account creation and authentication, the Client and Stylist Controllers handle user profile creation and updates, the Appointment Controller processes appointment scheduling and management, and the Messaging Controller supports sending, retrieving, and storing user messages. These controllers ensure that data is processed accurately and communicated seamlessly between the frontend and the database.

The lower section represents the database layer, which stores data across four distinct collections: Account, Appointment, Photos, and Messages. The Account collection stores user credentials and profile information, while the Appointment collection tracks all scheduled appointments. The Photos collection manages user-uploaded images, and the Messages collection stores conversation data for the messaging feature. The modular design of the database ensures efficient storage, retrieval, and management of data, while maintaining clear boundaries between different types of information. This cohesive structure supports the app's core functionalities and ensures the system is both scalable and maintainable.

Dynamic Component

Diagram

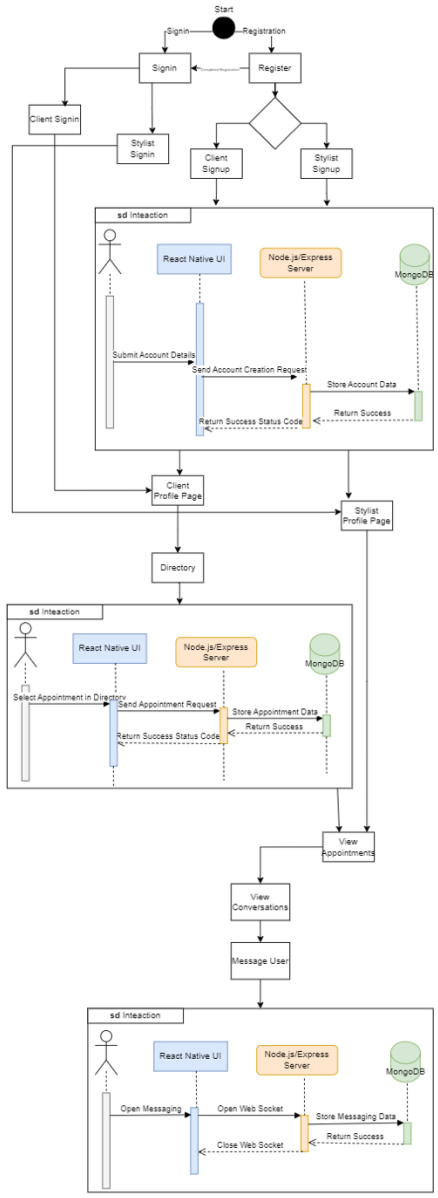

Figure 4 - Dynamic Component Diagram

This dynamic component design incorporates **SD Interaction** diagrams to showcase the intricate communication and workflows between the React Native UI, Node.js/Express Server, and MongoDB during critical features like account creation, appointment scheduling, and messaging. Each SD Interaction block provides a detailed view of the sequence of actions, method calls, and data exchanges. For example, during account creation, the SD Interaction diagram separates the workflows for clients and stylists by branching at the decision point between "Client Signup" and

"Stylist Signup." This distinction ensures that user-specific details, such as profile information or stylist business details, are routed through their respective backend controllers and stored appropriately in the database.

Similarly, in appointment scheduling, the SD Interaction diagrams illustrate how clients navigate the directory to select stylists, while stylists manage availability and existing appointments. These flows are clearly separated to reflect the unique roles of clients and stylists, ensuring accurate backend validation and data storage for each case. For the messaging feature, the SD Interaction diagrams emphasize real-time communication by showing how WebSocket connections are opened and closed, with messages exchanged and persistently stored in the database for future retrieval. These detailed diagrams highlight the synchronization and role-specific decisions embedded in the workflows, ensuring that both clients and stylists have tailored experiences while maintaining system cohesion.

Data Design

Introduction

We use MongoDB, a document-based NoSQL database, to store all user data. This includes authentication details, client profiles, business details, user-uploaded photos, appointments, messages, and more.

Figure 5 provides an overview of the major entities in our database, Clients and Stylists, in addition to auxiliary entities such as Photos, Appointments, and Chats. We chose to use a UML Class Diagram as we are using Mongoose, a library for Object Relational Modeling, to organize our database collections and calls. Unfortunately, the UML diagram does not capture where we have decided to embed objects inside documents as opposed to create mappings with IDs. However, this is acceptable as these semantics are only relevant for MongoDB performance optimizations and are unimportant towards understanding the relationships between entities.

Data Design Diagram

Diagram

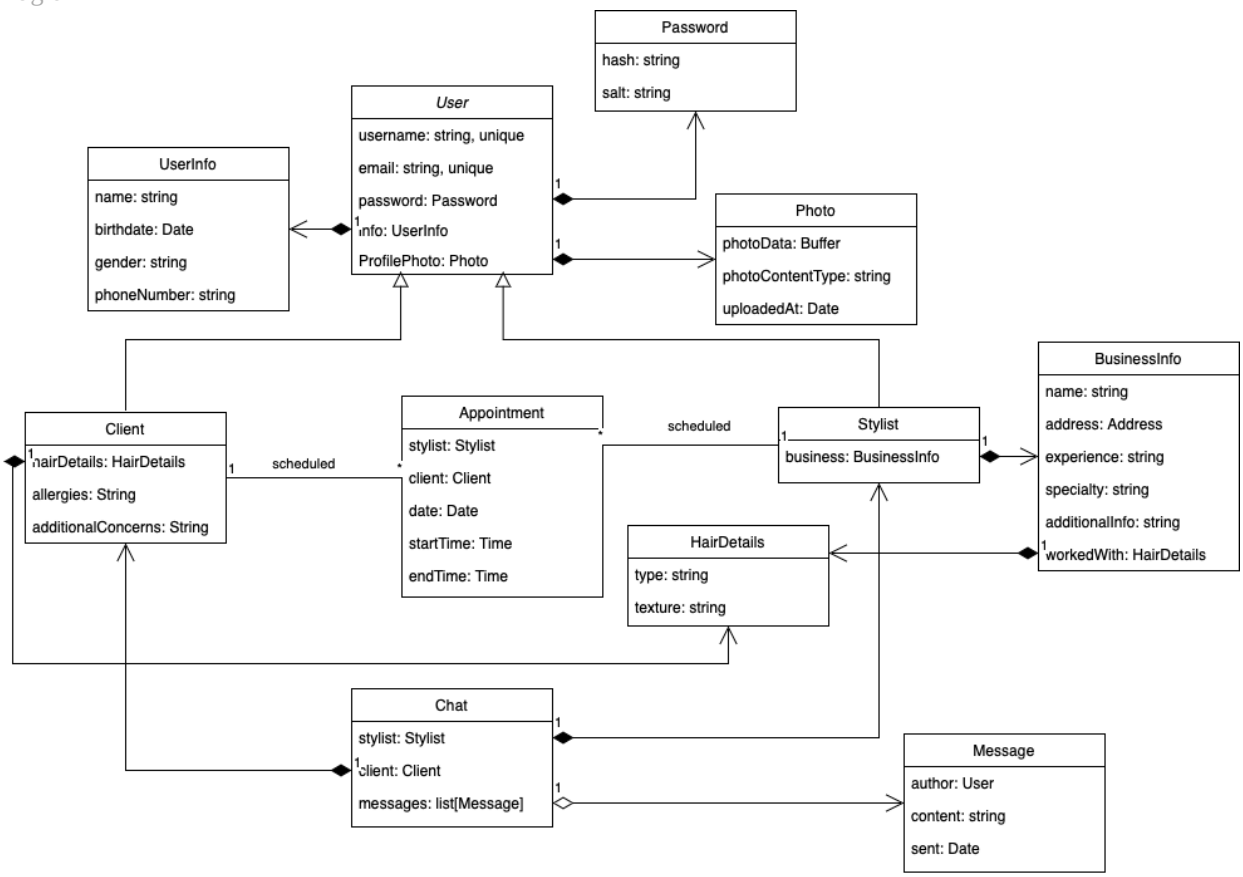

Figure 5 - Database Use Diagram

File Use

As mentioned before, our application stores all user data in the MongoDB database. This includes user-uploaded photos, which are compressed before storage as individual MongoDB documents (binary buffer, file format, upload date). In the future, it would be ideal to migrate photo storage to GridFS, a file chunking specification provided by MongoDB, so that our database can store images larger than 16MB.

Additional environment variables required for running the backend, such as the MongoDB connection string and email credentials for reset password, are stored in a config.env file that is colocated with the rest of the code.

Data Exchange

Client applications communicate with our backend via HTTP requests to RESTful API endpoints. Requests are sent with JSON bodies. Successful responses use JSON bodies while unsuccessful responses use a raw body. Status codes are attached to all responses so that client applications can easily determine the appropriate display update to the user.

Our backend is connected to our MongoDB cluster using a MongoDB connection string, which facilitates MongoDB's provided protocols for communicates.

Security Considerations

Since our backend will be hosted on the Internet, anyone on the Internet will be able to send requests to our backend and database. In theory, this means that anyone can access the passwords and personal data of our users if our API endpoints are not properly secured. Before deploying our app to end users, the following protections will need to be implemented:

1. Passwords will be encrypted so that they cannot be reverse engineered should a malicious actor gain access to the database. It is also possible that we may migrate our secure authentication to a 3rd-party platform such as Okta.
2. Requests to protected resources will be verified by JSON Web Tokens (issued after login/authentication) to ensure that access to sensitive information is authorized.
3. All requests must be communicated over a secure channel i.e. HTTPS

Eventually, our app will store Personally Identifiable Information such as our users' names and ages, as well as aspects of their personal health information such as their allergies and past medical treatments relevant to hairstyling. In addition to securely storing our database credentials, extra care will need to be taken to ensure that our app complies with all legal requirements for storing and protecting sensitive user information.

UI Design

Introduction

In this section, we present the User Interface of our mobile application. The UI consists of several different screens, each of them with their own styling, interactive properties, and navigation functionality to other screens. When designing the UI, we were careful to follow the 10 Usability Heuristics for UI design, a set of design principles which guided us in the creation of an app that is efficient, aesthetically pleasing, and highly usable.

UI Walkthrough

Upon opening the app, the user is prompted with the sign-in screen and can sign into an existing account or create a new one. The figure below shows the user flow of creating a new account, which brings the user to the sign-up landing screen. This screen is where the user makes the choice to create a profile as either a beauty client or a stylist.

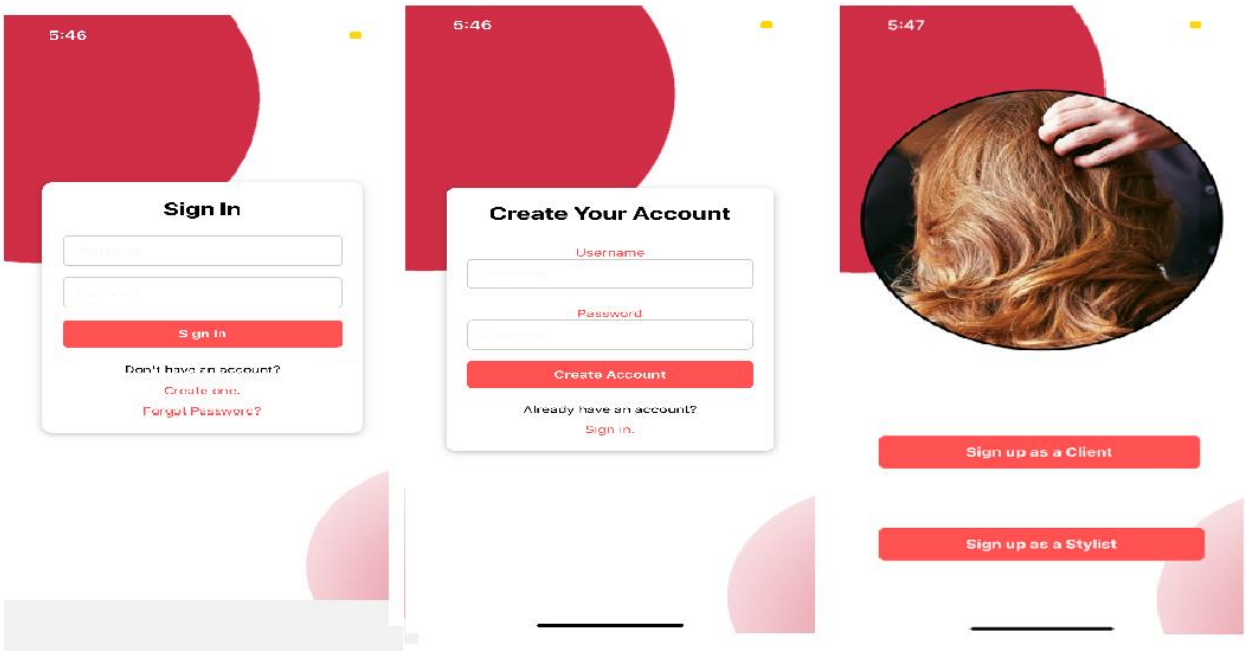

Figure 6 - Account Creation Screens

Once a user has created their account, they will need to go through the sign-up process. This involves several screens that allow the user to fill out their personal information. For clients, the inputs are regarding their hair history, allergies, and any concerns they may have. For stylists,

their sign-up process involves entering their business information as well as their experience with hair.

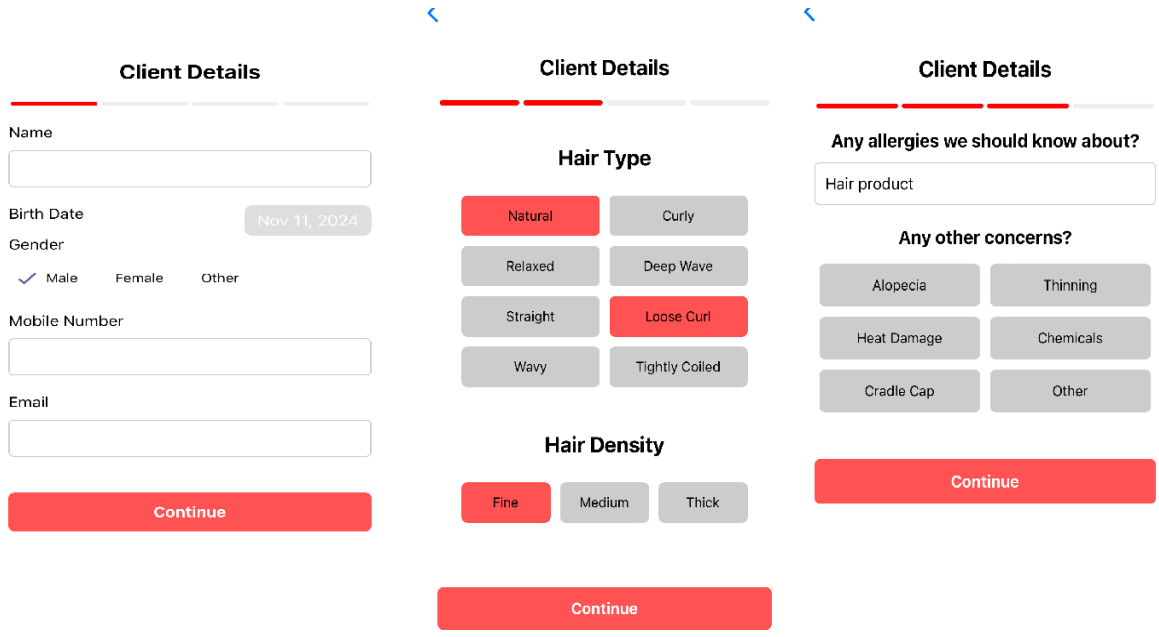

Figure 7 - Client Sign-Up Screens

Figure 7 shows the sequence of screens in the sign-up process for clients. Figure 8 shows the sign-up process for stylists. When designing these screens, we kept in mind Usability Heuristic 1: Visibility of System Status; the bar at the top of the screen makes it clear to the user how far along they are in the sign-up process. These screens also incorporate Heuristic 3: User Control and Freedom. The user is free to either continue or return to the previous screen at any point, providing them with a feeling of control and confidence.

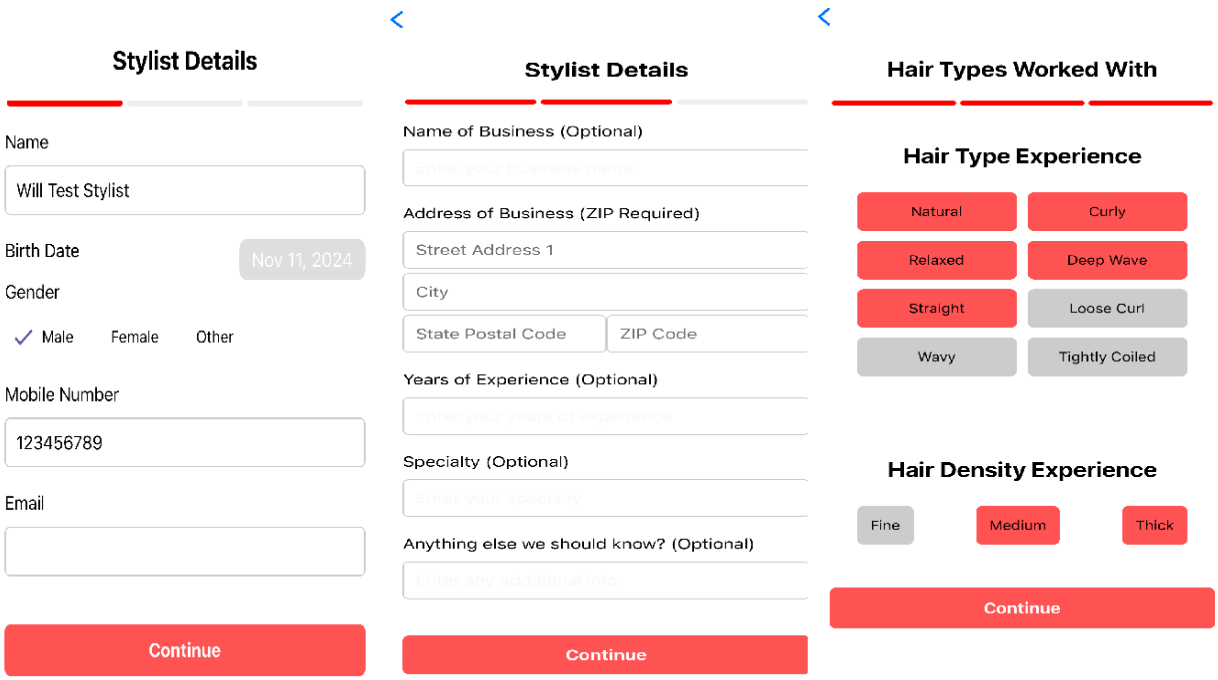

Figure 8 - Stylist Sign-Up Screens

After either a successful sign-up process or signing-in to an existing account, the user has access to the main pages of the application. Namely, the stylist directory, which exclusively clients see, and the appointments screen, which is visible to both clients and stylists with minor differences. For clients, the stylist directory is their hub for stylist listings near them, displaying each stylist's profile picture and business information. From this page, clients can book appointments by pressing the "Book Now" button, which will take them to the booking screen. Figure 9 below shows the directory screen and the subsequent appointment booking page.

In these screens we emphasized Heuristic 8: Aesthetic and Minimalist Design as well as Heuristic 6: Recognition Rather than Recall. The designs of these screens are meant to contain only the necessary information (a stylist's name, picture, business, experience) rather than bogging the screen down with superfluous components. The screens also make it easy for a new user to recognize what they need to do to book an appointment with a stylist. "Book Now" and the calendar component are meant to be instantly recognizable and easily convey to users their purpose.

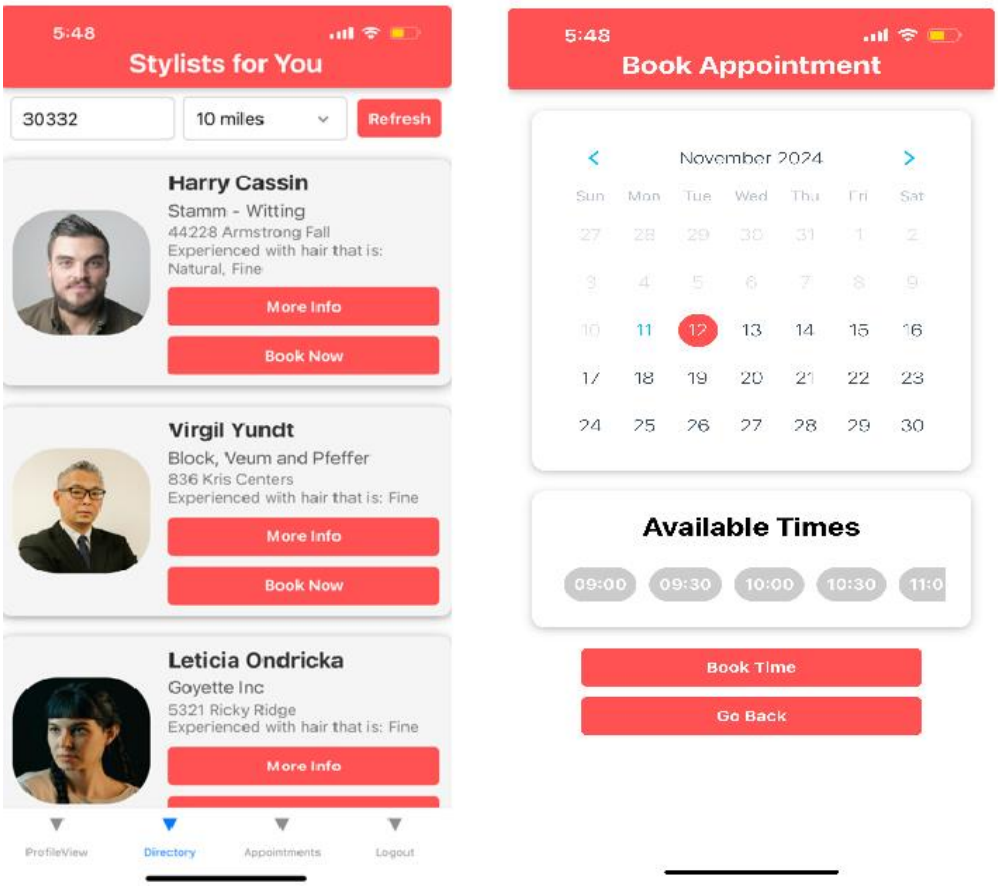

Figure 9 - Directory and Appointment Booking Screens

The appointments page is where users can view their scheduled appointments. For stylists, they can view their upcoming clients to understand their hair types and allergies before the appointment. Figure 10 shows the upcoming appointments page from the stylist's perspective, and the client view that appears when the "Client Info" button is pressed. These screens especially incorporate Heuristic 8: Aesthetic and Minimalist Design. The main appointment screen only includes the basic info like client name and appointment date/time. If a stylist wants to see more information, they can do this by choice with the "Client Info" button.

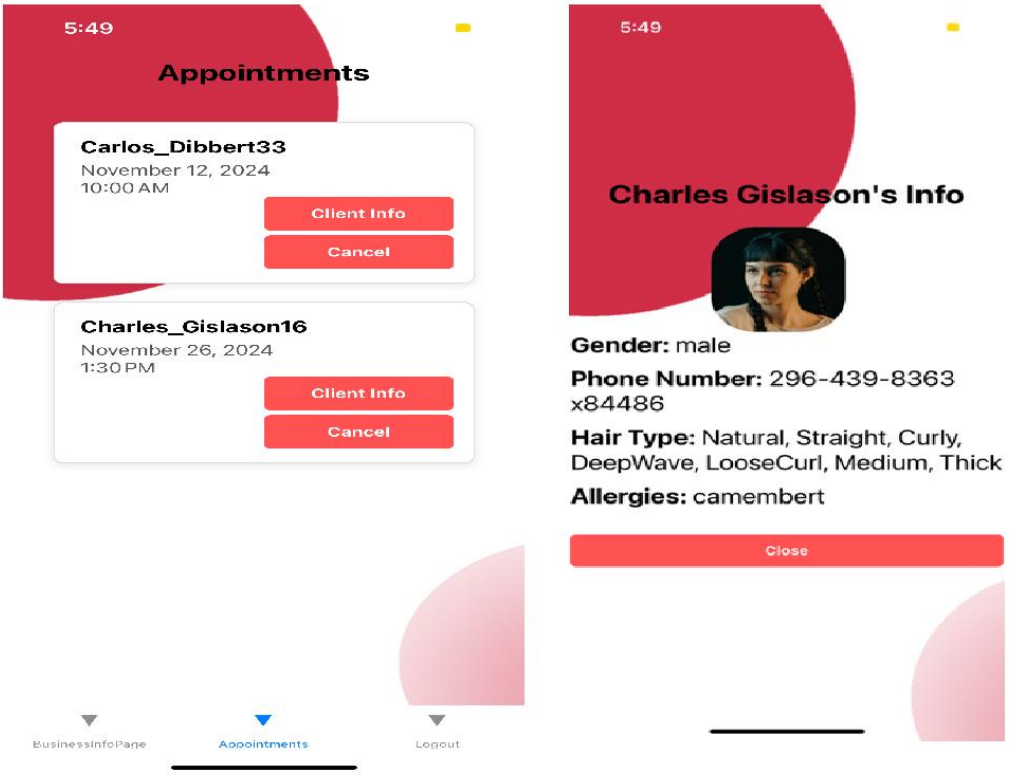

Figure 10 - Appointments and Client View Screen

The messaging feature allows clients and stylists to communicate in real time, ensuring clear and responsive interactions. After selecting a stylist from the directory or viewing appointments, clients can initiate a conversation to discuss specific hair needs or preferences. Stylists can also use this feature to confirm details with clients or provide personalized advice. The conversation list screen allows users to easily manage ongoing chats, while the individual chat screen prioritizes visibility of the most recent messages. These designs ensure that users can communicate effectively while maintaining an intuitive experience.

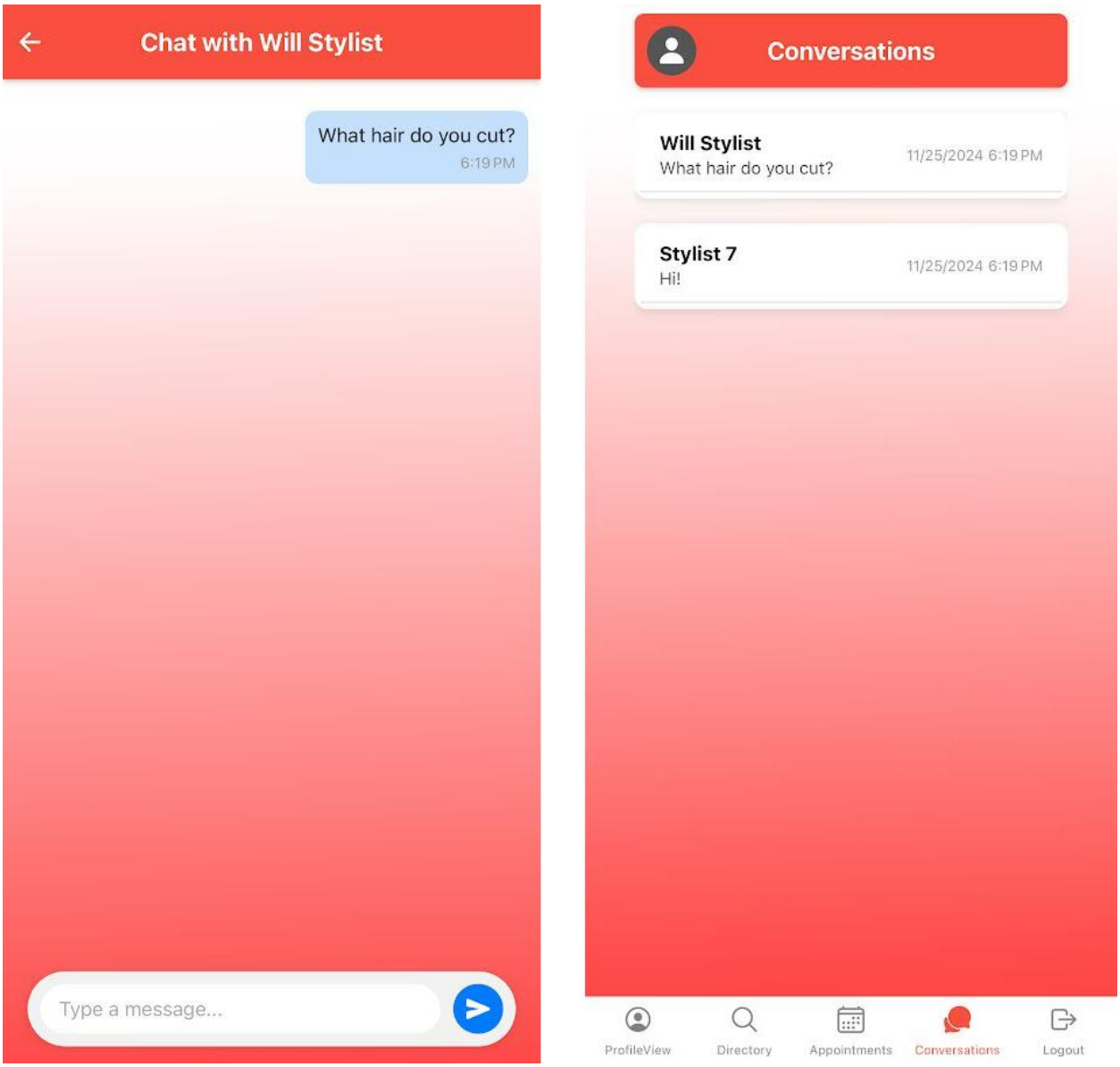

Figure 11 - Ongoing Conversations and Messaging Screens