AI-Enhanced Matchmaking App for Language Exchange

JID 4203

Aden Robertson, Krishnav Bose, Saahir Shaik, Townshend White, Vikranth Bellamkonda

Client: Yongtaek Kim

Repository: https://github.com/shaiksaahir/JID_4203_Language_Exchange

Table of Contents

List of Figures

Terminology

Agora.io RTC SDK: An API the application uses to help the development and functionality of the video calling feature.

API (Application Programming Interface): Software that defines how two or more applications interact with each other.

Axios: A JavaScript library used to make HTTP (HyperText Transfer Protocol) requests to and from a web browser.

Backend: The part of the application inaccessible to the user, primarily used for storing and organizing data.

CSS (Cascading Style Sheets): A programming language used for specifying the presentation and styling of a website.

Frontend: The part of the application accessible to the user — what the user sees and interacts with.

HTML (Hypertext Markup Language): A programming language used for defining the content and structure of a website.

JavaScript: A programming language used to develop web pages.

JSON (JavaScript Object Notation): A format used to structure and transmit data.

MySQL: A software used for database management built on the SQL (Structured Query Language) programming language, which creates and manipulates data relational to a database.

Node.js: A JavaScript runtime environment that supports the execution of JavaScript code within a scalable network environment. Commonly used for building dynamic web pages.

React.js: A JavaScript library for building user interfaces.

Socket.io: An API the application uses to help the development and functionality of user matchmaking.

Introduction

Background

Our application is a community interaction application designed to assist in the study of Korean for students at the Georgia Institute of Technology and English for students at Yonsei University. The application is a progressive web application, utilizing web technologies to provide a desktop-based application user experience. It is constructed using React.js for the frontend, Node.js for the backend, and MySQL for the database. It also utilizes the Socket.IO library for user chatting and matchmaking. The application includes user interactivity functionality, such as user matchmaking, chatting, and video calling, and translation features to streamline communication for those of differing languages. Our goal for this project is to provide a highly accessible interactive environment built on community engagement and the study of multiple languages.

Document Summary

The System Architecture section provides a high-level view of the application and how its major components, such as the user interface and database, interact statically and dynamically.

The Component Design section describes the application’s components alongside their static and dynamic interactions in much more detail.

The Data Storage Design section will provide a clear understanding of how data will be stored and exchanged within the application using MySQL.

The UI Design section details the major user interfaces within the application and how the user interacts with them.

System Architecture

Introduction

Our application is a progressive web application that will run on any web browser. Its system incorporates a layered architecture to assist with the accessibility of the application, detailed below with the static and dynamic architectural designs of the system. The static architecture details the system's major components and how they are organized and interact with each other. The dynamic architecture gives an example scenario to outline how the system's components interact during runtime.

Rationale

The system's architecture is designed to be highly accessible, both with how the user interacts with the system itself and how future developers will interact with the system after it launches. To achieve this, the system utilizes a layered approach to its structure. We chose a layered approach as it provides flexibility in component communication, allowing the various components of the system to quickly interact with others, streamlining user flow. It also provides flexibility when developing, allowing us to separate and finalize certain layers without having to worry about other aspects of the system, which streamlines and accelerates development.

This separation also benefits development from a security standpoint. Our system is based entirely on and around various users interacting with each other, so having user information private and secure is extremely important. Layering the components allows the system to make sure user information is secure before passing it along; each layer has a security check. This makes sure that a user's password is hashed before passing it to the database where it is more out in the public, compared to all data always being quickly accessible to every component. This also benefits those users who do not want their profile information public, as they can set a preference to have their visibility private, which provides another security check between the user interface, backend, and database.

Static Architecture

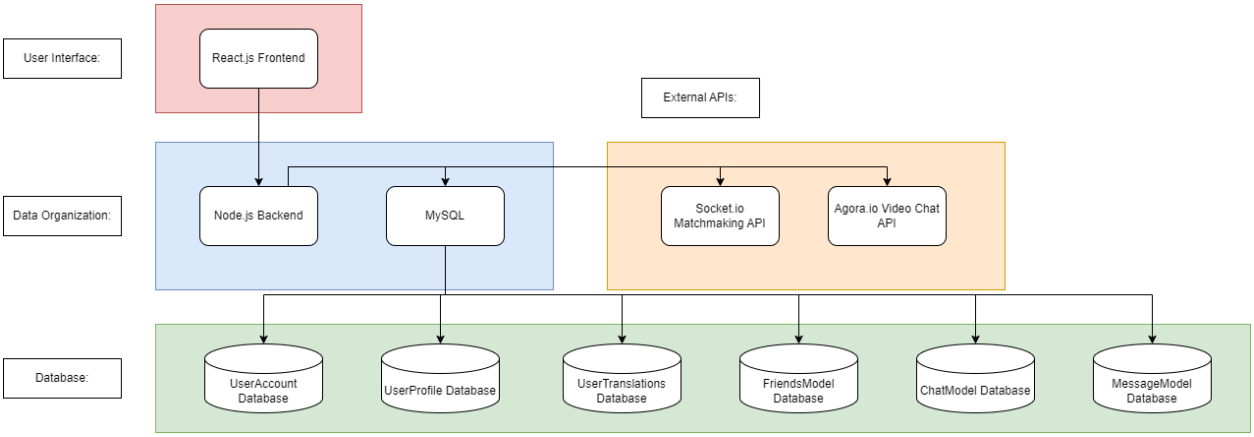

Figure 1: Static Architecture Diagram

The system's static architecture was primarily due to necessity and client feedback. This is the project's third iteration, so an architecture/structure was already established. Our client felt this architecture suited the application, and after an intensive review of the architecture, our team agreed. Thus, we decided to use what the previous iterations had built and continue to build off of it. The system is comprised of three separate layers, each encompassing various components, that all communicate with each other.

The first layer in the system is the User Interface. This layer details the layout and visuals of the application. This is the only layer the user directly interacts with. It is built using React.js, additional JavaScript, CSS, and HTML. These are all standard frontend languages that allow us to develop an interactive user experience.

The second layer in the system is the Data Organization. This layer provides all of the system's logic, outlining the necessary communication needed between all three layers. It primarily focuses on the management of data — how it is organized and manipulated. It is built using Node.js, which is helpful given the online user interaction within the application. The Node.js logic fetches the user input from the User Interface and then interacts with the Socket.io API, Agora.io API, and MySQL in order to perform the application's needs. The external Socket.io API is utilized when the user needs to match with another user within the bounds of our matchmaking logic. The external Agora.io API is utilized for video calls. MySQL outlines the construction and management of the databases while allowing for the quick retrieval of stored data. After using the required tools, the Node.js logic then updates the User Interface, so the user quickly gets the necessary feedback.

The third layer in the system is the Database. The layer is comprised of all of the necessary databases for the system, all of which have the necessary data organized and retrievable by the

Node.js logic. There is a lot of user data, which is why it is prudent for the system to utilize a database management system such as MySQL. Our current databases include UserAccount, UserProfile, UserTranslations, FriendsModel, ChatModel, and MessageModel. The UserAccount houses all of the user's login information, including an encrypted password for user protection. The UserProfile houses all user preferences for matchmaking. The UserTranslations stores the user's translation log. The FriendsModel stores the user's friends list. The ChatModel houses the user's previous chats, and the MessageModel stores the actual messages in each of those chats. All of this data is given by the user in the User Interface, retrieved by the Data Organization layer, and finally sent through MySQL to populate the correct database.

Dynamic Architecture

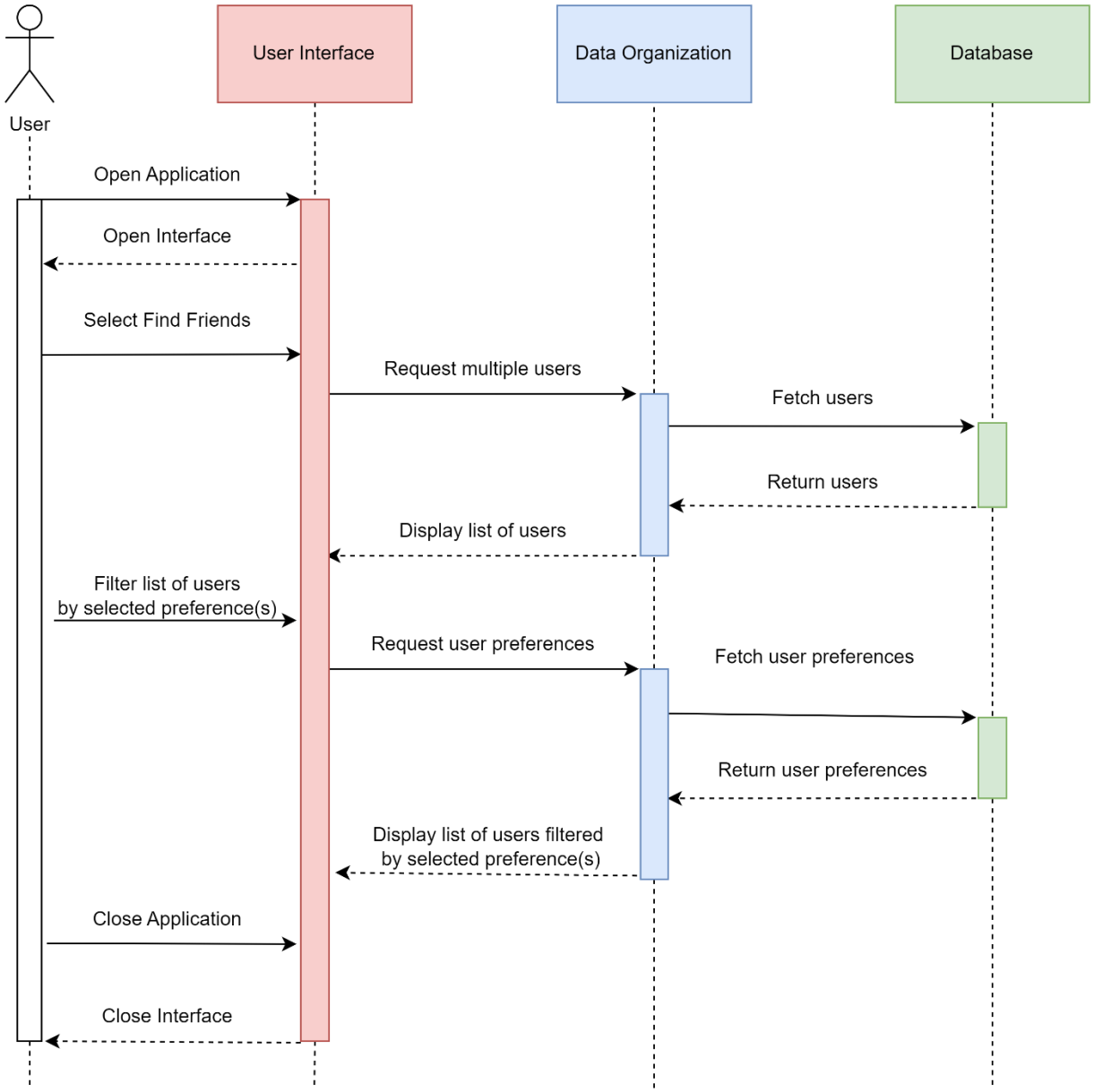

Figure 2: Dynamic Architecture Scenario

The system's dynamic architecture is represented in the above system sequence diagram in Figure 2, which depicts the system's runtime behavior and flow of control between the different components as a user performs a common task.

Upon the user's opening of the application, the User Interface initiates and is ready for use. Note: the application itself requires a user to log in to their account or register for an account, but since

this feature is extremely commonplace, we decided to omit it to focus on our application's specific features. Continuing along with the above scenario, the user selects "Find Friends." The User Interface processes this input and sends a request to the Data Organization component for multiple user profiles. The Data Organization component then fetches users from the Database, which then returns the requested users. After successfully fetching the data, the Data Organization component puts the users in a list and presents it to the User Interface for display. After viewing the displayed list of users, this user wants to filter out certain users based on a selected user preference or preferences. The User Interface processes this input and sends a request to the Data Organization component for user preferences. The Data Organization component then fetches the required user preferences from the Database and uses those preferences to shrink the user list to only contain those with the selected preferences, which is then presented to and displayed on the User Interface. After the user completes these tasks, they close the application, where the User Interface then closes the interface.

We chose the common tasks of finding potential friends and filtering the given list by preferences because of its importance regarding the application while being straightforward. The application is built on community involvement, and these tasks present a streamlined depiction of how the system functions with the goal of community involvement in mind. We also chose to display the dynamic architecture with a system sequence diagram due to its ease of access and clear distinction regarding representing different system components and their interactions.

Component Design

Introduction

Whereas the previous section delved into the various layers of our application's system and how they interact, this section details the components of those layers, both in terms of how they are designed and how they interact with one another. This is detailed below with the static and dynamic elements of the system. The static elements detail each individual component, breaking down what comprises each layer and how every aspect interacts with the rest of the system. The dynamic elements give an example scenario to outline how those specific component aspects interact during runtime. The diagrams below depict the same layers as the System Architecture — the User Interface, Data Organization, and Database layers — color-coded accordingly.

Static Elements

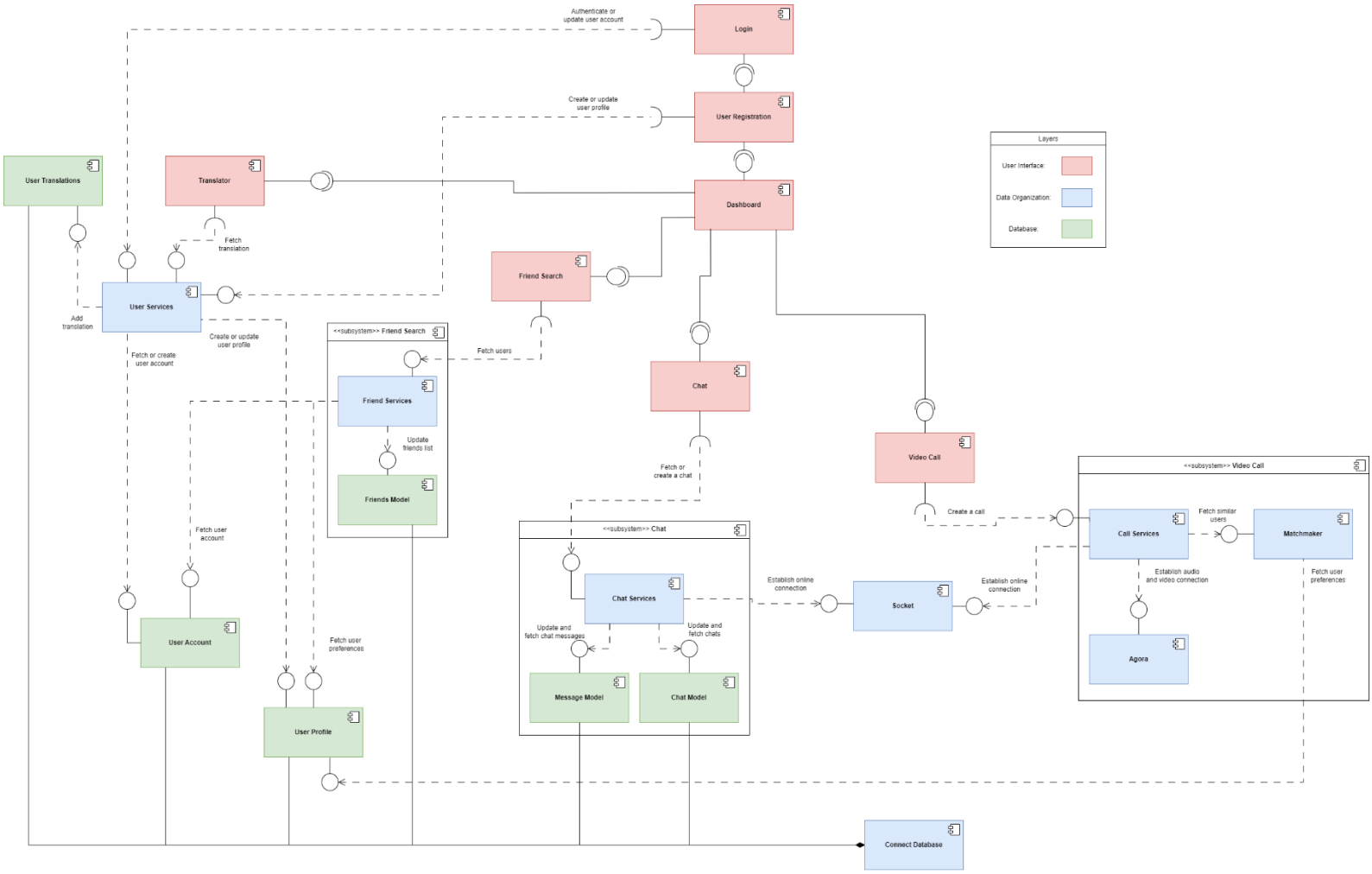

Figure 3: Static Elements Diagram

The component diagram above depicts the components of our application and their static relationships with each other. A component diagram was chosen due to its detailed breakdown of the system and how the various parts of the system interact. This diagram includes various frontend components, represented by the red User Interface layer, that define and control the application's interface. Backend components, represented by the blue Data Organization layer, communicate with user input from the frontend components in order to accurately fetch or update the needed information and data, utilizing various APIs and databases to do so. Those databases, represented by the green Database layer, are called on by the backend components to return or update their respective data. All of the frontend components flow from each other, which is why they each have an association connection. Since they flow from each other, the previous component represents the required interface. The backend components connect to the frontend components via dependency because, although the frontend components are required to initiate the backend components, shown by the required interface semicircle, the frontend

components require the data retrieval of the backend components to determine how the interface is filled. The database components are connected to the backend components also through dependency because, although the backend components detail how the data should be utilized, they must get that data from the database components. These database components are all compositions from the Connect Database backend component, as without that component, the whole system would not be able to utilize the databases.

Dynamic Elements

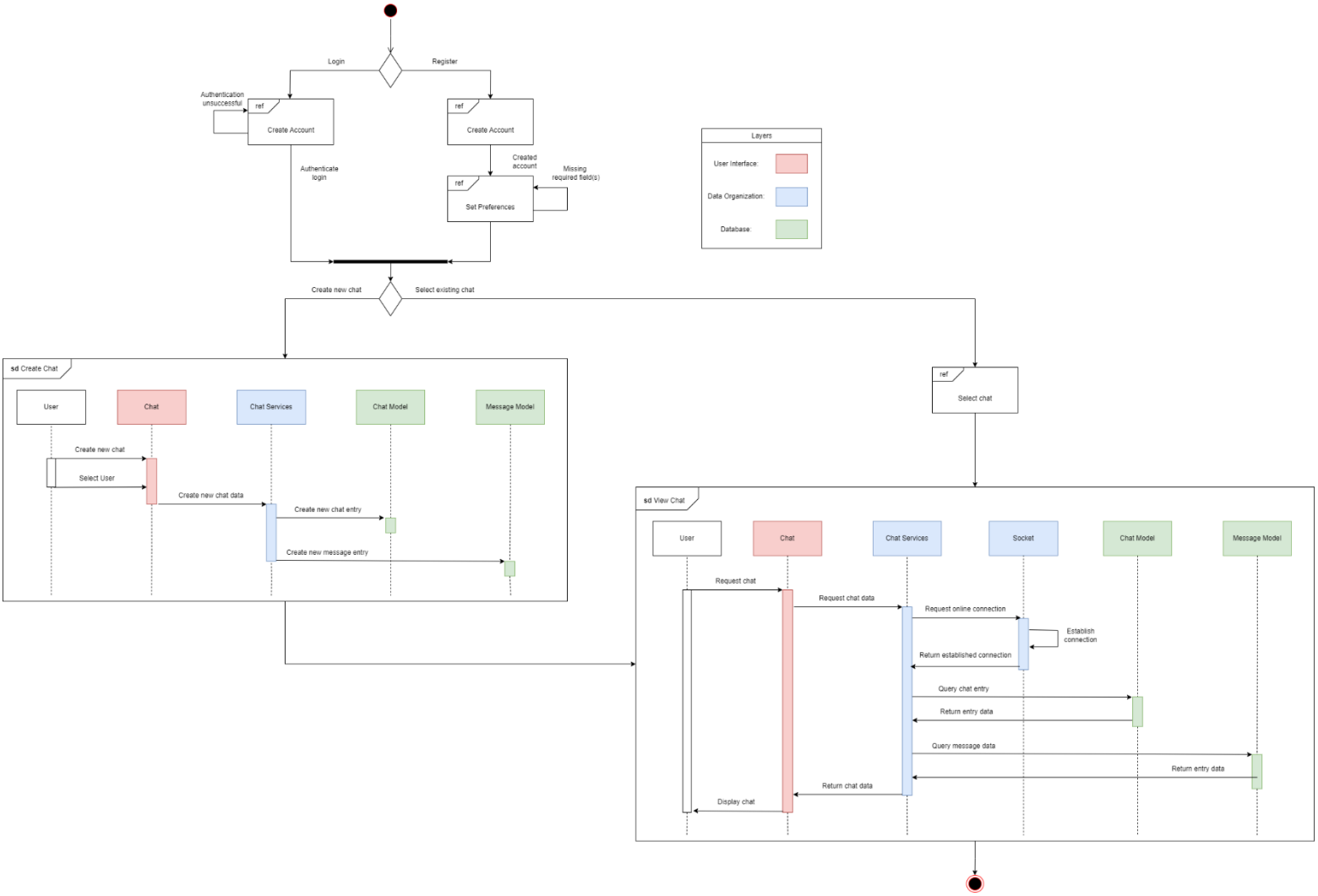

Figure 4: Dynamic Elements Scenario

The interaction overview diagram above depicts the standard workflow in the application. An interaction overview diagram was chosen to most effectively illustrate the multiple options the user has when going through the application. The diagram details how a user may use the chat feature, a feature chosen to better illustrate the difference in choice within the application as well. After going through the standard account login or registration, the user can either create a new chat or select an existing one. If they choose to create one, they then select the user they want to chat with. This signals the Chat Services component to create a new chat entry in both the Chat Model and Message Model databases. Should a user choose to select an existing chat, they would simply select a displayed previous chat. Both options feed to the user viewing their chat, which sees the user requesting a chat. This signals the Chat Services component to request an online connection from the Socket API, which establishes and returns a connection. After the

connection is established, the Chat Services component then requests the correct chat and message data and returns it, which is then displayed to the user.

Data Design

Introduction

The entity relationship diagram below depicts the data design of our application. It shows the types of entities, attributes, and relationships between entities, with the entities being the databases needed for our application. These databases are organized and managed through MySQL, which we query for retrieval and updating of the various entities. Many entities require a primary key to distinguish the specific user their attributes correlate to as well as to distinguish that there are no repeated users per attribute section, shown by an underlined attribute. Other entities require a foreign key, which is obtained from another entity to distinguish any multiple sections of attributes per user, shown by a gray attribute. An entity relationship diagram was chosen due to its ease of understanding regarding, in many cases, complex data design.

Database Use

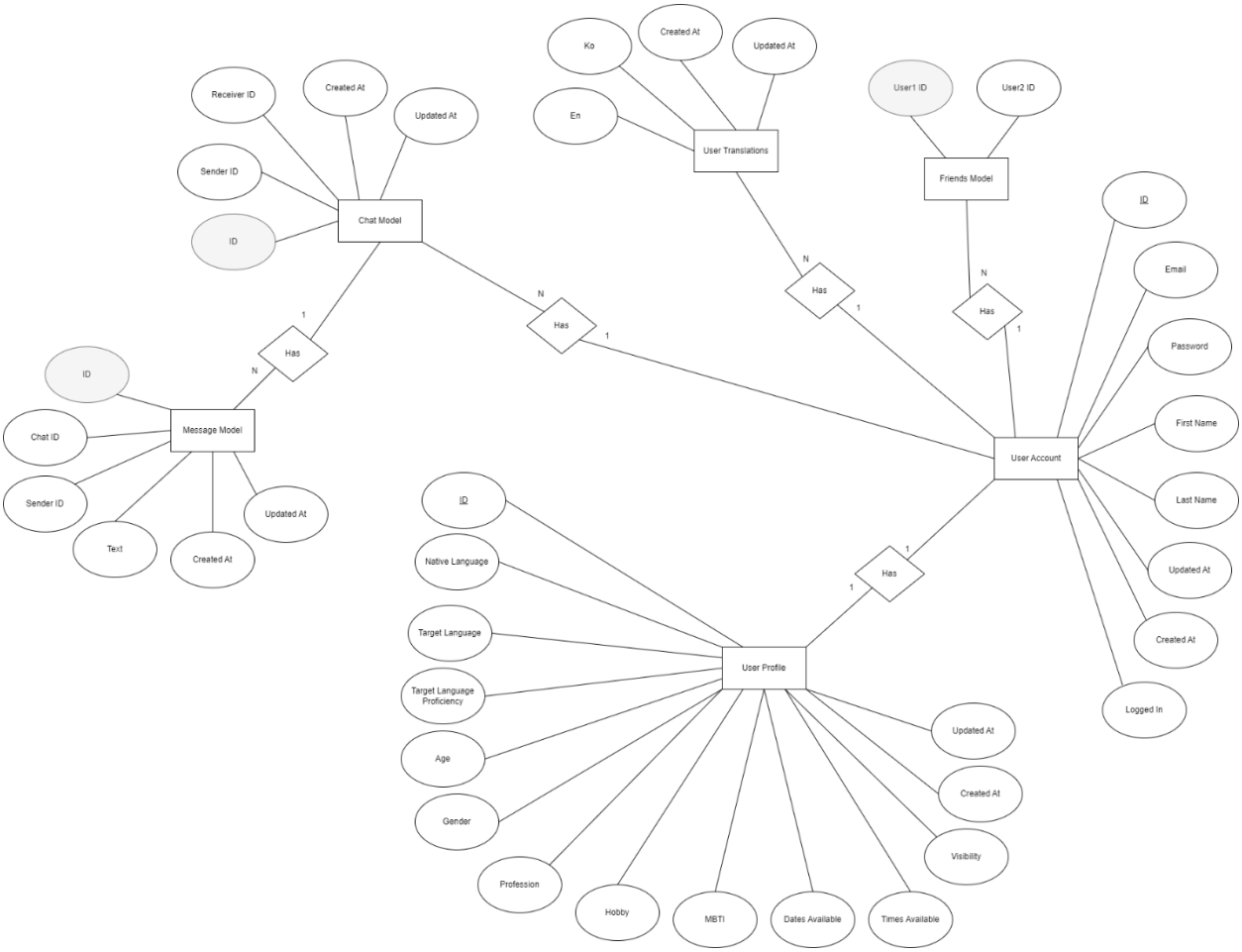

Figure 5: Database Use Diagram

Here is a list of all major entities:

- User Account: Contains user information regarding account creation and what information is needed to login to the application.
- User Profile: Contains user preferences and information regarding their description, which will be used in matchmaking and other features.
- Friends Model: Acts as a friends list for a user; details which other users a user is friends with.
- User Translations: Stores translations the user requests to save.
- Chat Model: Stores user chats.
- Message Model: Stores user messages correlating with a chat.

File Use

Our application uses two image files for styling, a PNG for the logo and a JPG for the user profile picture. There are no other files our application utilizes, nor does it create any temporary files.

Data Exchange

Our application uses Axios to communicate between a user's browser and the server. JSON is used to format and transmit the data itself when sending to and receiving from the server.

Since our application is based on user communication, there are notable security concerns within the application that have been addressed. Our application will go live in the future, meaning that our server will be hosted on the Internet. This could raise concerns regarding the spread of user information, even if it is not necessarily sensitive information. To combat this, the client plans to release the application through the Georgia Institute of Technology Web Hosting Services, a service that will provide the server with a secure channel to protect user data. The only piece of information that is sensitive is a user's password, which is concerning since it must be stored in a database to be accessed and authenticated. Immediate encryption is done to every user's password, utilizing a hashing function that encrypts the data within the server. A password is authenticated when a user logs in to the application utilizing Bcrypt, a JavaScript library that specializes in hashing and authenticating passwords.

UI Design

Introduction

This section outlines the user interface (UI) design of the application, which users interact with to create a profile with preferences, find friends, call/chat with other users, and translate between English and Korean, among other smaller features. We discuss our UI design decisions and describe all major pages users will interact with.

Walkthrough

Upon opening the application, users are greeted with the opening page with two options: Login or Register. Selecting Login brings users to the Login page to input their email address and password. Selecting Register brings users to two pages, with the first one allowing them to input their name, email, and a new password.

Figure 6: Opening Page

The second registration page is much more important to the functionality of the entire application. This is the Set Profile page, where users detail their preferences, like available dates and times to meet, and other demographic information, like age. This information is utilized throughout the entire application, being the basis of our matchmaking system. Some of the information is a lot more important to the integrity of our matchmaking, so it is required for a user to input it, denoted by the asterisk (*).

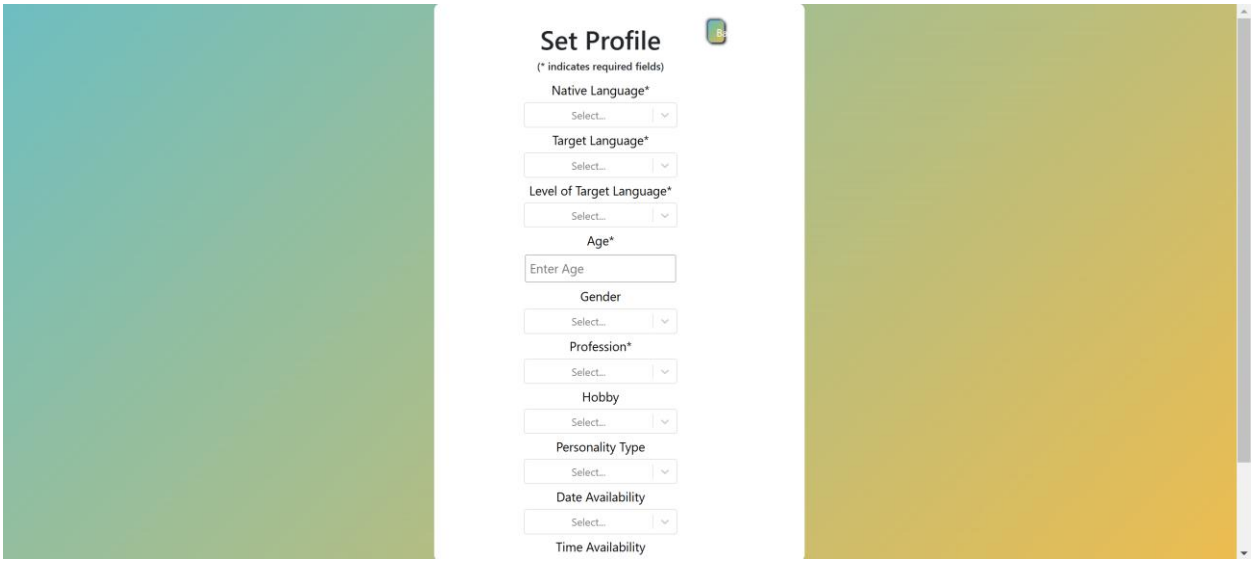

Figure 7: Set Profile Page

After logging in or registering a profile, users are directed to the Dashboard page, the central page of the entire application. From here, users can view questions and answers about the application (from the "?" button in the top right), find friends, chat or call other users, access the translator, re-set their profile preferences (bringing them to the previous Set Profile page), and logout of the application (bringing them to the Opening Page). The features with unmentioned pages each have their own designated page the user will navigate to. For the sake of brevity, further page descriptions will focus on the pages directly involved with this iteration of the project.

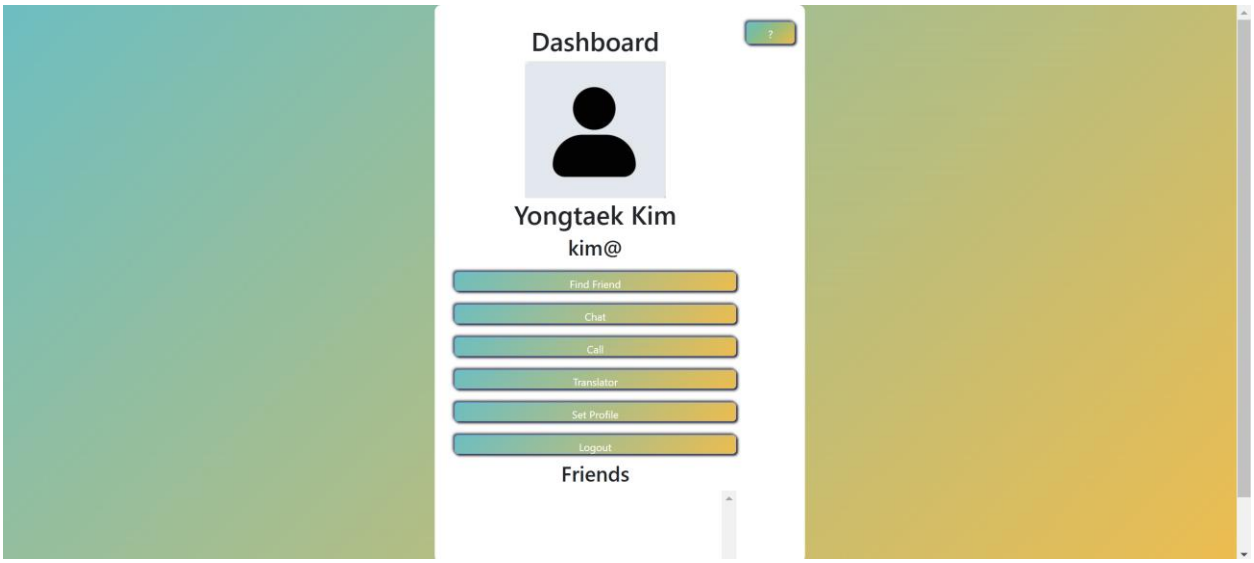

Figure 8: Dashboard Page

Selecting the Find Friend button brings users to the Find Friends page, where users can view every other user in the application who has their profile visibility set to “Show.” They can filter these users by their name or by any one of their preferences from the Set Profile page. Selecting a user adds them to the user’s friends list, which populates at the bottom of the Dashboard page. Users who a user recently called will show up under the Recent Chat Partners section. The Back button in the top right will bring users back to the Dashboard page.

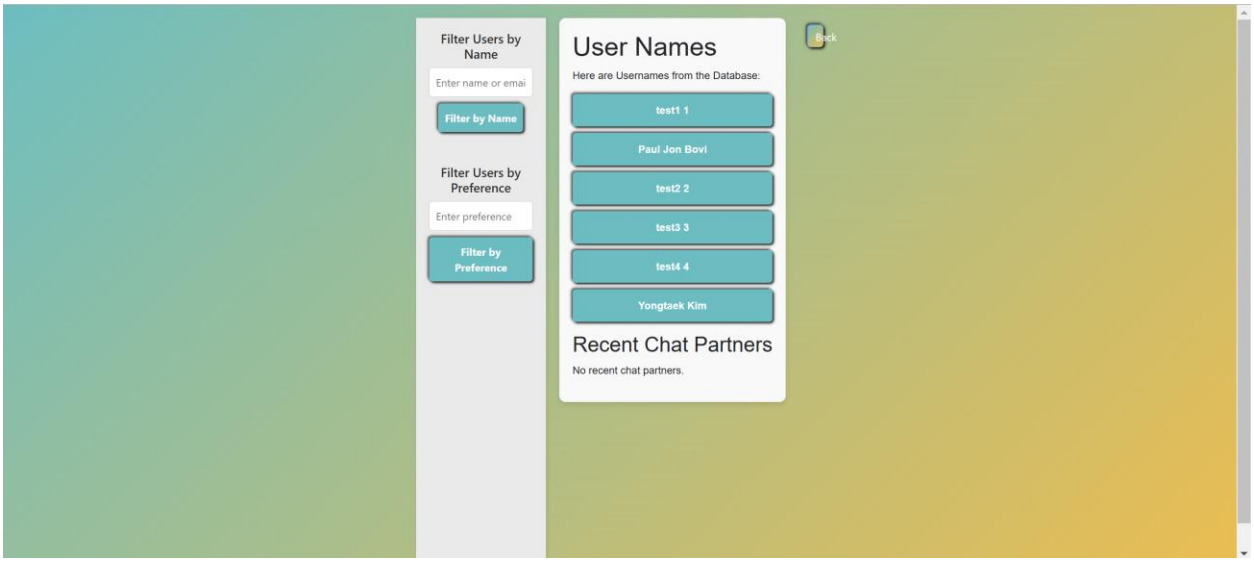

Figure 9: Find Friends Page

From the Dashboard page, selecting the Call button brings users to the Video Call page, where users are able to join a video room, select their video and audio preferences, and determine their preferred number of chat partners. Selecting the Join Room button populates the page with another user who joins/joined the same room. Selecting the Back button navigates users back to the Dashboard page.

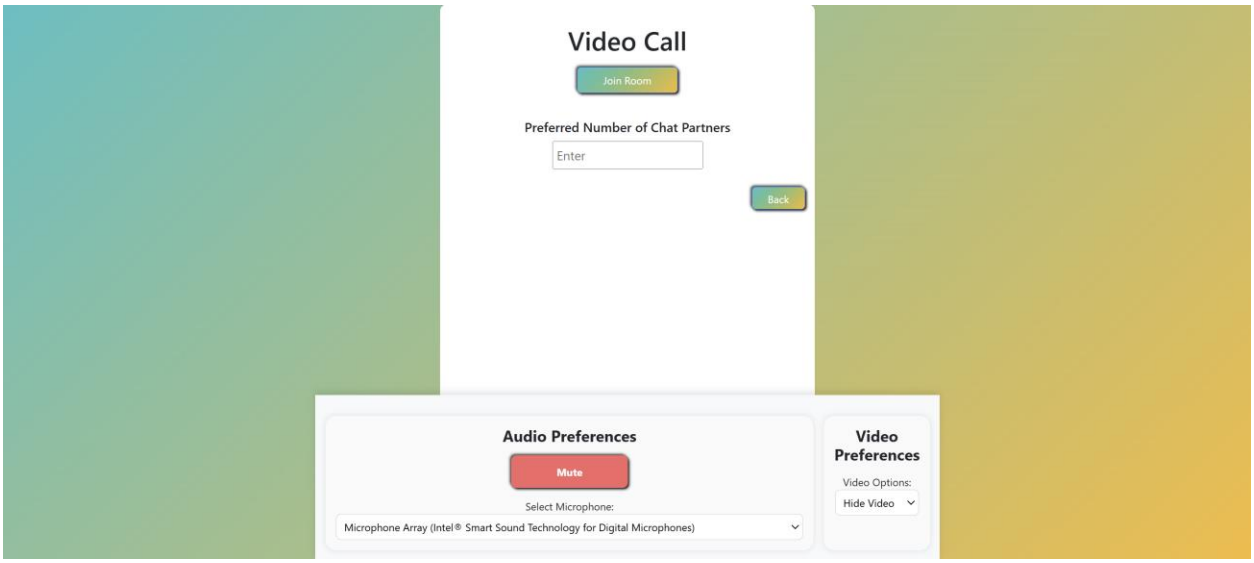

Figure 10: Video Call Page

These pages were designed to follow the Ten Usability Heuristics for UI Design.

The first heuristic, Visibility of System Status, is incorporated with active responses to the user: Each user can see that they are logged in and an active user throughout every page (notably with the dashboard, which populates with data the user previously inputted) and every button has feedback by accomplishing what it is said to do (notably with the audio and video preferences, which visually and audibly provide feedback during a call).

The second, Match Between the System and the Real World, is followed throughout with terminology such as “dashboard,” “call/chat,” and “update profile” following familiar wording to users and standard conventions in other messaging and video-calling applications.

The third, User Control and Freedom, is followed through the ease of accessibility in the application. Users can easily log out, change their profile preferences, and access various features without feeling restricted. Scenarios never required backtracking from the user, as each function was designed around a loop with the user quickly finding themselves back at the dashboard following each task. Back buttons are always present as well to navigate the user back to the dashboard.

The fourth, Consistency and Standards, is followed by distinct and differential wording, also enhanced by following conventions, such as with the difference in terminology between “chat” and “call.”

The fifth, Error Prevention, is followed with temporary text and needed errors. This is most common in the Set Profile page where temporary text displays when a user has not created an

input and asterisks are present on required sections, like age. Should a user not input a required section, the application will throw them an error, directing them back to the missing input field.

The sixth, Recognition Rather than Recall, is followed through familiar sights and wording. The dashboard is a great example because since it is the central page of our application, users can use it as the basis to navigate the rest of the application. They can easily find their way back with back buttons and can easily navigate to other pages with distinct page and button names.

The seventh, Flexibility and Efficiency of Use, is followed through the streamlining of many processes, such as the login feature which navigates the user directly to the dashboard instead of asking them to set their profile preferences again. Every back button brings the user to the dashboard page typically, since many main features do not require multiple pages, and those that do should rarely need to send users back to their preceding page. The application's design recognizes that many users would want to start back at the central dashboard page, allowing for flexibility in what feature they pursue next.

The eighth, Aesthetic and Minimalist Design, is followed through the minimalist design on each page, displaying only what is needed and breaking up core features by their respective page. Our application further follows the same color scheme and layout, typically, providing a consistent aesthetic.

The ninth, Help Users Recognize, Diagnose, and Recover from Errors, is followed with each of the application's needed error messages, such as the aforementioned Set Profile page's required input sections error message, presented in an informative and simple manner.

The tenth, Help and Documentation, is followed by the Help Page, accessible by the "?" button on the dashboard. It lists common questions and answers to help facilitate navigation.

Appendix

Agora RTC SDK

Resource URL: https://docs.agora.io/en/video-calling/get-started/get-started-sdk?platform=react-js

Response Format: JSON (event-based response in callbacks).

Requires Authentication: Yes (APP_ID, TOKEN, and CHANNEL required).

Rate Limit: 10,000 minutes free per month, with increasing rates afterwards.

Parameters:

- APP_ID: A unique identifier for the application's Agora project.
- TOKEN: An authentication string for secure communication towards the identified channel.
- CHANNEL: The name of the video call room to join.

Example Response:

{

"APP_ID": "50a71f096ba844e3be400dd9cf07e5d4",

"TOKEN": "007eJxTYHAvnsHy/o3/iukFLVElUaIv22+mXGhNmSr95+C2c3cjusMUGEwNEs0N0wwszZISLUxMUo2TUk0MDFJSLJPTDMxTTVNMJr+2TW8IZGSYxLaXmZEBAkF8bobcxJLkjNzE7My8dAYGANE0JDs=",

"CHANNEL": "matchmaking"

}