

# National College of Ireland

BSc (Honours) in Computing

BSCCYBE4

Academic Year 2023/2024

Fagner Nunes

19216718

x19216718@student.ncirl.ie

Concealed messaging app

Technical Report

## Contents

Executive Summary.....	2
1.0 Introduction.....	2
1.1. Background.....	2
1.2. Aims.....	3
1.3. Technology.....	3
2.0 System.....	3
2.1.1. Functional Requirements.....	3
2.1.1.1. Use Case Diagram.....	4
2.1.1.2. Requirement 1 User Registration ID: UC-001.....	4
2.1.1.3. Block and Unblock a User UC-002.....	7
2.1.2. Data Requirements.....	9
2.1.3. User Requirements.....	10
2.1.4. Environmental Requirements.....	12
2.1.5. Usability Requirements.....	14
2.2. Design & Architecture.....	15
2.3. Implementation.....	18
2.3.1. Backend Implementation.....	18
2.3.2. Front-end Implementation.....	19
2.4. Graphical User Interface (GUI).....	21
3.0 Conclusions.....	21
4.0 Appendices.....	22
4.1. Project Proposal.....	22
<b>Objectives.....</b>	<b>24</b>
Functionalities.....	24
<b>Goals.....</b>	<b>24</b>
<b>Market Research.....</b>	<b>25</b>
<b>State Of The Art.....</b>	<b>26</b>
Concealed Messaging:.....	26
Discreet Communication:.....	26
Targeted User Base:.....	26
Emphasis on Privacy:.....	27
Whatsapp:.....	27
Telegram.....	28
<b>Technical Approach:.....</b>	<b>28</b>
Milestones:.....	29
<b>Project Plan.....</b>	<b>29</b>
Week 1-2: Requirement Gathering and Design.....	29

Week 3-4: Frontend Development.....	29
Week 5-6: Backend Development and Integration.....	29
Week 7: Testing and Refinement.....	30
Week 8: Deployment and Finalization.....	30
<b>Objective.....</b>	<b>30</b>
<b>Functionalities.....</b>	<b>30</b>
Must have.....	30
Should Have.....	32
Could Have.....	32
<b>Implementation Order.....</b>	<b>33</b>
Before authentication.....	33
Post authentication.....	34
<b>Design.....</b>	<b>35</b>
Wireframe for the dummy apps.....	35
Registration screens.....	35
2 steps verification screen.....	36
Chat screen.....	36
User search Screen.....	38
Profile Screen.....	39
Contact list scree.....	40
1.1. Documentation.....	40
<b>AuthContext.....</b>	<b>42</b>
Methods.....	42
<b>Calculator.....</b>	<b>43</b>
Methods.....	44
<b>Camera.....</b>	<b>45</b>
Methods.....	45
<b>ChatComponent.....</b>	<b>47</b>
Methods.....	47
<b>ChatItem.....</b>	<b>49</b>
Methods.....	49
<b>ChatScreen.....</b>	<b>51</b>
Methods.....	51
<b>ContactItem.....</b>	<b>52</b>
Methods.....	53
<b>ContactsScreen.....</b>	<b>53</b>
<b>ProfileScreen.....</b>	<b>54</b>
<b>UpdatePassword.....</b>	<b>54</b>
Methods.....	55
<b>screens/ModalComp.....</b>	<b>55</b>
<b>screens/beforeLogin/EmailVerification.....</b>	<b>57</b>
Methods.....	58
<b>screens/beforeLogin/Login.....</b>	<b>60</b>
Methods.....	61

<b>screens/beforeLogin/LoginOrRegister.....</b>	<b>62</b>
Methods.....	63
<b>screens/beforeLogin/Password.....</b>	<b>64</b>
Methods.....	65
<b>screens/beforeLogin/Register.....</b>	<b>66</b>
Methods.....	67
<b>backend.....</b>	<b>69</b>
Requires.....	69
Methods.....	69
Methods.....	74

## Executive Summary

This report details the development and functionalities of a messaging application with deep integration between backend and frontend.

Purpose: The purpose of this report is to give a deep overview of the messaging application, including how the backend server, frontend user interface, authentication mechanisms, and messaging functionalities were configured.

### Backend Overview:

- Uses Express.js for server setup.
- It implements JWT for user authentication and authorization.
- User-uploaded images are stored on AWS S3.
- Some key features will include user registration, login, image upload, updating a profile, and blocking or unblocking a user.
- A detailed API documentation of endpoints such as /send-image, /user/block/, /user/unblock/, and /contacts/get-contacts.

### Frontend Overview

- Built in React Native and seamlessly integrated with the backend APIs.
- Provides screens for user registration, login, profile management, and messaging.
- Makes navigation using React Navigation and theming with React Native Paper possible.
- It offers contact management, real-time messaging with the help of WebSocket, and image handling.

### Conclusions:

The messaging application is a good comprehensive solution to cover the key functions that are required for a modern communication platform. It comes with detailed API and frontend documentation, which means it can be developed and maintained in the future with ease.

## 1.0 Introduction

This report details the development of both backend and frontend components of a messaging application. This is a project to create a platform for messaging that will be firm, scalable, and user-friendly, built using cutting-edge technologies and best practices in software development.

### Background

The primary reasons behind the realization of this project include an increasing need for safe and reliable communication tools rich in features for both personal life and business. The messaging app will fill identified gaps with existing solutions, primarily in terms of user experience, real-time communication, and multimedia support.

The most general purpose of the project was developing a messaging application able to meet most of the common issues that occur within others: scalability, security, and ease of use. The principal intention of this project is to be at the forefront in meeting the requirements of customers today and being capable of servicing them in the future—a platform offering reliable, intuitive communication tools.

## Aims

The project will seek to achieve the following:

- Build a backend that is secure and scalable: Implement strong user authentication, real-time messaging, and multimedia handling.
- Create a User-Friendly Frontend: Design an intuitive interface that facilitates easy navigation and efficient communication.
- Ensure security and privacy of data: Adopt secure techniques pertaining to the transmission and storage of data that also protect the user's information.
- Enable Seamless Integration: Guarantee there is smooth interaction between the frontend and backend components, which results in a uniform user experience.

## Technology

The technologies that the project is using to achieve its goals are as follows:

- Backend: Express.js with JWT for authorization and AWS S3 for image storage. Real-time messaging is done via WebSocket.
- Frontend: Developed using React Native, which uses React Navigation for smooth screen transitions and React Native Paper for styling with theme consistency.
- Database: It uses a relational database managed with an ORM in managing user-related data and message history for quick retrievals.
- Security: Incorporates JWT to provide secure authentication and authorization, which guarantees integrity and confidentiality of data.

## 2.0 System

### 2.1.1. Functional Requirements

- User Authentication
  - Users shall be able to register with a unique email and password.
  - The system shall validate the email during registration by sending a confirmation code.
  - Users shall be able to log in using their registered email and password.
  - Authentication tokens (JWT) shall be used to manage user sessions securely
- User Profile Management

- Users shall be able to update their profile information, including their name, description, and profile picture.
- The profile picture shall be uploaded to an AWS S3 bucket, and the URL shall be updated in the user's profile.
- **Contact Management**
  - Users shall be able to add, delete, and search for contacts using their email.
  - The system shall display a list of user contacts, including their name and profile picture.
- **Messaging**
  - Users shall be able to send and receive text messages in real-time.
  - Users shall be able to send images along with text messages.
  - The system shall store message history and allow users to view past conversations.
- **Blocking Users**
  - Users shall be able to block and unblock other users.
  - Messages from blocked users shall not be delivered.
- **Notification Settings**
  - Users shall be able to update their notification preferences, including sound, vibration, and notifications.
- **Password Management**
  - Users shall be able to update their password.
  - The system shall enforce password strength criteria, such as a minimum length of 8 characters, including at least one capital letter and one number

The functions outlined in this report so far ensure that the messaging application will be secure, reliable, and user-friendly, providing a solid foundation for future enhancements.

#### 2.1.1.1. Use Case Diagram

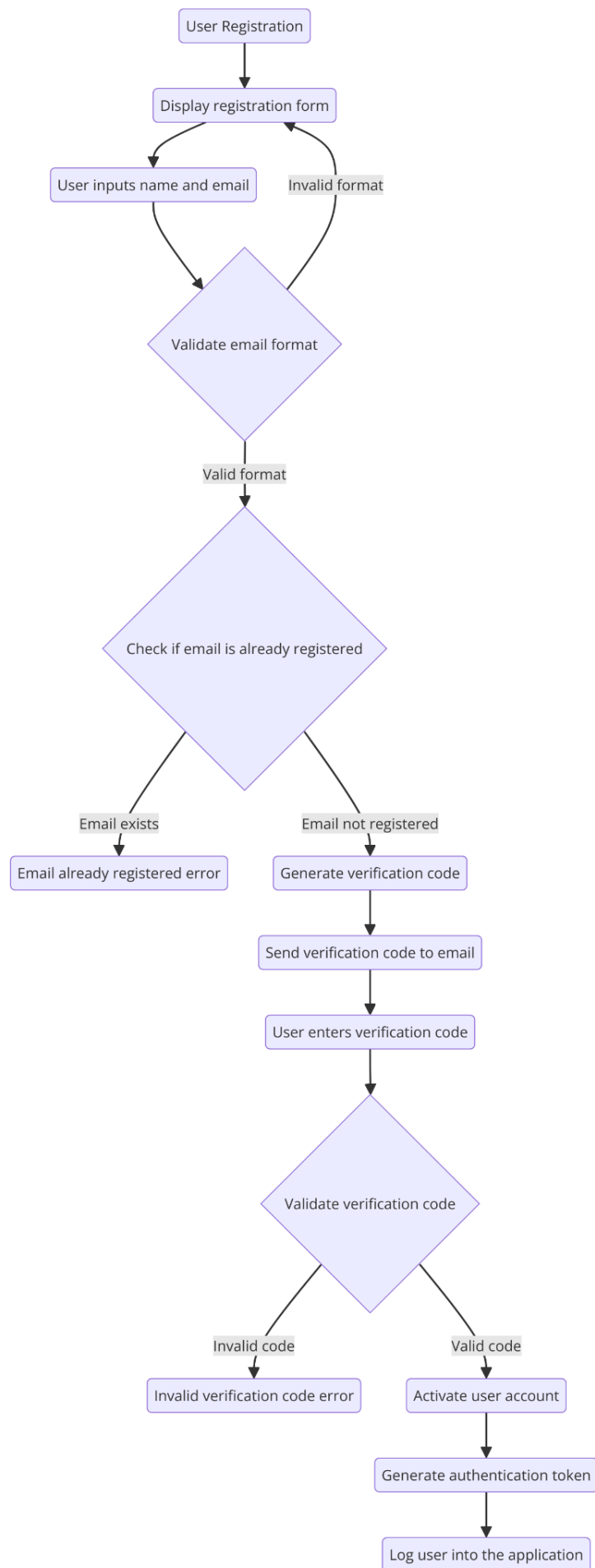
#### 2.1.1.2. Requirement 1 User Registration ID: UC-001

User registration is of high priority for the messaging application. The feature allows new users to register by entering their name and email address.

This use case will explain the process that a new user registers for the application, receives the verification code in his/her email, and then verifies that email to complete their registration process.

#### Use Case Diagram

Actors: User, System





## Flow Description

### Precondition

- The system is in an initialization state.
- A user on the registration screen of the mobile application.

### Activation

The user inputs his/her name and email address from the registration page and submits the registration form.

### Main flow

1. Initially, the system presents the registration screen to the user.
2. The user inputs his/her name and email address and submits it.
3. The system verifies that the email format is correct.
4. The system checks if such an email is already registered.
  - a. If the email is registered, then an error message to the user (see E1)
5. The system creates a verification code and sends it to the user's email address.
6. The user receives the email and types in the verification code into the application.
7. The system authenticates the verification code.
  - a. In case of the code being invalid, the system shows an error message (see E2)
8. The user account status is activated, and registration is completed.
9. An authentication token is generated and the user is logged into the application.

### Alternate flow

#### A1 : Email already registered

- An error message is displayed stating the email is already taken.
- the use case returns to step 2

### Exceptional flow

#### E1 : <title of E1>

- The system signals the user that the format of the email is badly formatted.
- The user corrects the email format and resubmits the form.
- The use case continues at step 3.

#### E2 : Anomaly in verification code

- The system signals to the user that the verification code he entered doesn't match what has been sent.
- The user requests a new verification code and re-enters it.
- This use case follows step 7.

### Termination

- The system displays the main application screen, where the user is logged in and the application is ready to use.

### Post condition

- The system will wait for the next registration workflow.

List further functional requirements here, using the same structure as for Requirement1.

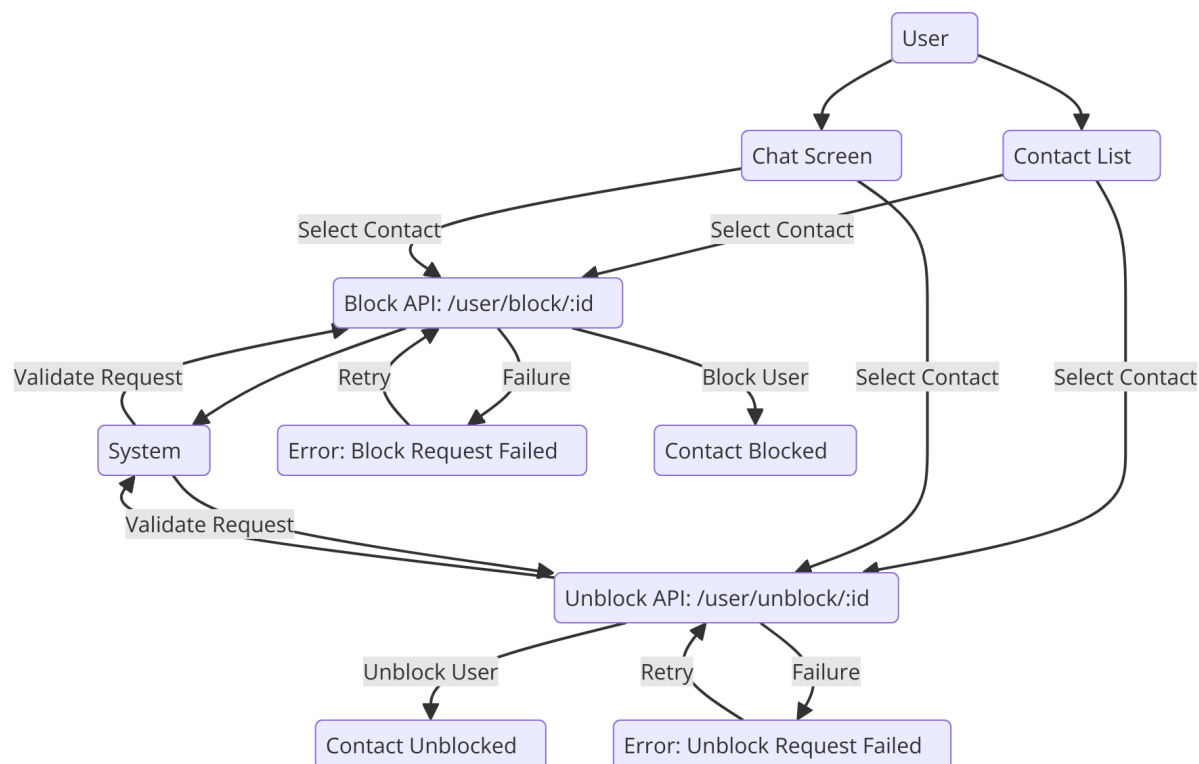
#### 2.1.1.3. Block and Unblock a User UC-002

The blocking and unblocking feature is significant in attaining a safe and controlled environment within the messaging application. This will lead to user control for the personal management of interaction, protection against unwanted interactions, and security. Hence, the user experience and security themselves have a high-priority level requirement .

Enable users to block or unblock any other user inside the application, so that the blocked user is not allowed to send messages to or view the profile of the blocking user.

### Use Case Diagram

Actors: User, System



### Precondition:

- The user is logged into the application and has a valid session.

- The user is on the contact or chat screen where they can select the contact to block or unblock.

#### Activation:

- User decides to block/unblock a contact and from the blocked/unblocked contact profile or chat screen, initiates the action of blocking or unblocking that contact.

#### Main Flow:

1. System shows a contact list or chat screen
2. User chooses a contact to block
3. System sends a block request to Backend API (/user/block/ )
4. Backend validates the request and blocks the user.
5. The system will display an error message.
6. Update the contact status to "Blocked" and send a message to the user that says "action successful"

#### Alternative Flow A1 Unblock a User

- Preconditions
  - The user has already blocked another contact.
  - The user selects a contact that has been blocked before and chooses to unblock them.
  - The system sends an unblock request to the backend API (/user/unblock/ ).
  - The backend validates the request and unblocks the user.
  - It displays an error message if the request fails (E2).
  - The system updates the contact status to unblocked and notifies the user of the successful action.
  - The use case continues from step 5 of the main flow.

#### Exceptional Flow: E1: Block Request Failure

- The system shows an error message, implying that the block request failed.
- The user can retry blocking the contact.
- UC resuming from Main Flow step 2

#### E2: Unblock request failure

- A system error message is shown: "An error occurred processing the unblock request"
- The user is allowed to re-perform the action of unblock the contact.

- UC resuming from Alternate Flow step 2

#### End

- The new status for contact is shown, and it returns on the contact list or chat screen.

#### Post-condition:

- System goes to the waiting state prepared to accept the next user action to be related with blocking or unblocking contact.

### 2.1.2. Data Requirements

This part explains the data requirements for the messaging application, the necessary data entities, their attributes, and the relationship that exists between them. The data requirements ensure that all important information in the system is captured and managed appropriately.

#### Data Entities and Attributes

- User
  - UserID: Unique identifier for the user (Primary Key)
  - Name: Full name of the user
  - Email: Email address of the user (Unique)
  - Password: Hashed password for authentication
  - ProfilePictureURL: URL of the user's profile picture stored in AWS S3
  - Description: User's profile description
  - Sound: Notification sound setting (Boolean)
  - Notification: Notification setting (Boolean)
  - Vibration: Vibration setting (Boolean)
  - AccountStatus: Status of the user's account (active, blocked, etc.)
  - Code: Email confirmation code
  - CodeTimestamp: Timestamp when the verification code is generated
  - LoginAttempts: Number of login attempts
  - LoginAttemptTimestamp: Timestamp of the last login attempt
- Contact
  - ContactID: Unique identifier for the contact (Primary Key)
  - UserID: Unique identifier for the user (Foreign Key)
  - ContactUserID: Unique identifier for the contact user (Foreign Key)
- Message
  - MessageID: Unique identifier for the message (Primary Key)

- SenderID: Unique identifier for the sender (Foreign Key)
- ReceiverID: Unique identifier for the receiver (Foreign Key)
- MessageContent: Text content of the message
- ImageLink: URL of the image associated with the message, if any
- Timestamp: Time when the message is sent
- Delivered: Status of the message (Boolean)
- Read: Status of the message (Boolean)
- Chat
  - ChatID: Unique identifier for the chat (Primary Key)
  - UserID: Unique identifier for the user (Foreign Key)
  - Messages: List of messages associated with the chat

### Relationships

- **User-Contact:** One-to-Many (A user can have several contacts)
- **User-Message:** One-to-Many (A user can send and receive multiple messages)
- **Message-Chat:** One-to-Many (Chat can contain many messages)

### Data Management

- **Storage:** User profile pictures and the images related to the message are stored in AWS S3. The URL is stored in the database.
- **Security:** Hash user passwords before storing them. Employ authenticated tokens for secure session management, such as JWT.
- **Backup and Recovery:** Regular backup of databases to prevent data loss and recover them in case of failure.

### Data Validation

- **User Registration:** Make sure email addresses are unique, and the password strength is ensured.
- **Message Delivery:** Ensure both the existence of the sender and receiver before the message is sent.
- **Profile Update:** Before a profile picture is uploaded to S3, validate for the format and size of the picture.

## 2.1.3. User Requirements

### User Requirements

These are the functionalities and features which the messaging application is supposed to have for users' expectations. These requirements are very important since they validate that the application will be user-friendly, secure, and efficient for the end user.

### Registration and Login

- The user should be able to register with their email and a strong password.
- The system should then send a verification code to their email for account activation.

### User Login

- Users must be allowed to log in with their registered email and password.

### Profile Edit

- Users can modify their profile information, e.g. name, description, and profile picture.
- The pictures need to be uploaded and stored securely in AWS S3.

### Contact Search

- Users will be able to search other users by email and add them to contacts.
- Users should be able to see their contact list with profile pictures and status information.
- Users should be able to delete contacts from the contact list of a user.

### Messaging

- Users should send and receive text messages in real time.
- Users should send and receive images in association with sent text messages.
- The system should maintain history of messages and allow users to see previous conversations.

### Block and Unblock Users

- Users should be allowed to block each other in order to prevent unwanted messages.
- Users should have the power to unblock previously blocked users.
- When a user is blocked, that user should be unable to message or view the profile of the individual who had blocked them.

### Notification Settings

- Users will be permitted to manage their preference settings for notifications in sound, vibration, or just notifications.

### Password Management

- A password change interface should be provided.
- The system should enforce a good password policy in terms of size, composition, etc.

### Usability

- The user interface shall be intuitive and user-friendly.
- An experienced user shall perform all available system functions with no more than minimal training.
- The application shall provide error messages and indicate appropriate corrections.

### Security and Privacy

- User data should be transmitted over secure channels, HTTPS.
- Hash passwords of users and not save plain passwords.
- Use authentication tokens (JWT) for secure session management.

### Performance

- Must be responsive and load fast.
- Handle up to 10,000 concurrent users without performance degradation.

Such user requirements assure that messaging applications meet the expectations and needs of their users by delivering a secure, effective, and easy-to-use platform for communication.

#### 2.1.4. Environmental Requirements

The environmental requirements of the messaging application are the multiple factors that ensure the system works effectively, reliably, and securely within its specified environment. These requirements are critical in maintaining high performance, scalability, and usability to deliver a seamless user experience.

- **Hardware and Software Environment**

Amazon EC2 hosts the back-end server, providing a robust and highly scalable environment for server-side operations. This assures that the server can support heavy traffic and scale efficiently as the number of users grows. In the backend, PostgreSQL is used for database management, guaranteeing safe and efficient storage and retrieval of data. Using AWS S3 to store images uploaded by users ensures efficiency in multimedia management; moreover, it offers high availability and durability.

The frontend is developed using React Native, which provides cross-platform compatibility between iOS and Android. Hence, the application will have huge coverage without having to maintain separate codebases for different platforms. It employs React Navigation for easy screen transitions and React Native Paper for theming consistency, along with a huge number of UI components to make up a captivating user interface.

- **Network and Security**

The application requires stable and very fast internet connectivity to support real-time messaging and seamless interactions between the frontend and the backend. WebSockets, in some way, require real-time communication, which involves a reliable structure in the network so messages are properly and timely delivered.

Security, being very important to this aspect, has various measures put in place by the application to ensure user data is protected. Interchanges between frontend and backend are over HTTPS; hence, data is already encrypted during transmission. User passwords will be hashed prior to storage, while on the user side, authentication tokens will be used in managing user sessions securely. This ensures no unwanted access or probable breaches into users' data.

- **Usability and Accessibility**

The application is intuitive and user-friendly, thus allowing users to easily navigate through the different features with a very minimal learning curve. It is consistent in layout, color schemes, and typography, which changes the user experience into an appealing and trouble-free experience with the applications. Responsive design guarantees perfect performance of all operations on different screen sizes and orientations, offering the same experience on all kinds of devices.

It has inbuilt accessibility features to make the application useable for differently challenged people. This includes screen readers, high contrast modes, keyboard navigation, and many more to list accessibilities according to guidelines including WCAG. It also gives provision for customized settings against the notifications to be received by the user for setting up their preferences of alerts based on requirements and context.



- **Performance and Scalability**

The application should be capable of supporting 10,000 concurrent users without any performance degradation. Herein, infrastructure resilience and scalability are required; this would allowed to happen via cloud services like Amazon EC2 and AWS S3. Regular monitoring and optimization have to be anticipated with respect to server performance and the database in order for the servers and applications to keep functioning in a quick and reliable way across all responses. The backups are quite frequent, and there is a scheme on disaster recovery in case the system fails, mover of all, to ensure the integrity and availability of data. Regular backups of database systems and user-uploaded content are intended. In such a case, if an outage occurs, measures are put in place to fast track the restoration of service. The environmental needs of the messaging application are what make it efficiently and securely operable in the very environment within which it is called to serve. Based on robust infrastructure in the cloud, tight security measures, and adhering to best practices in both usability and accessibility, the application takes solid first steps toward delivery of a quality user experience. These requirements provide a sound basis for attaining the long-term objectives of an application with respect to success and scalability and, therefore, to cater effectively to the dynamic requirements of the users.

#### **2.1.5. Usability Requirements**

Usability requirements explicitly deal with issues that guarantee making the messaging application intuitive, efficient, and pleasant to the users, for it improves both user experience and satisfaction.

The application should provide an intuitive interface, which would allow users to easily flow through a variety of functionalities. Best practices in UI and UX design need to be followed, whereby consistency in layout, color schemes, and typing is ensured. A responsive-design technique should be implemented by the application, since it would scale out quite well at different screen sizes and orientations of mobiles or other types of devices.

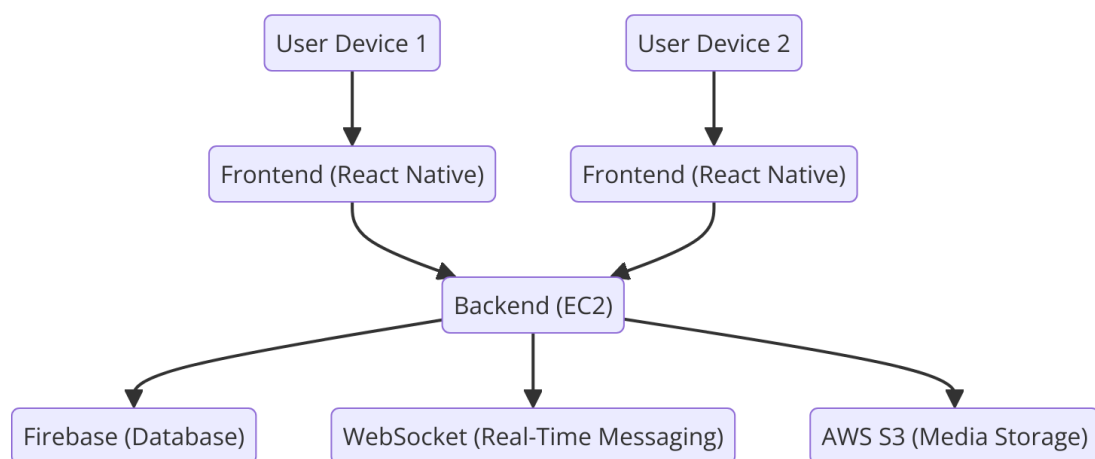
New users shall be able to sign up and start using the application with minimal instructions. The processes for registration and login should be simple and intuitive, with step-by-step instructions and clear hints. After that, a user shall be able to update his profile, add contacts, send messages, and edit settings succinctly and with minimal training. Experienced users shall be able to use all features of the system following less than two hours of training in total. The average number of errors users make per day should not be more than two after this training.

Such error messages and feedback should be informative and user-friendly, aiming to help the user understand what has gone wrong and how to set things right. For example, when one is registering and puts an incorrect email format, there should be a clear system message of what the error is and how one can get the format right.

The application should provide accessibility features, from screen readers to high contrast modes and keyboard navigation for users with disabilities. This will need to be compliant with accessibility standards, including WCAG. Customize notification settings should also be provided to allow the user to set preferences for alerts with a sound, vibration, or visible display. In such a way, everyone could customize this according to personal requirements and contexts.

In other words, the set of requirements for usability ensures a messaging application that is accessible, effective, and easy to handle, synonymous with a good user experience for all.

## Design & Architecture



## Data Structure

- User Object

```
{
  "userID": "unique_identifier",
  "name": "user_name",
  "email": "user_email",
  "password": "hashed_password",
  "profilePictureURL": "url_to_profile_picture",
  "description": "user_description",
  "sound": true,
```

```

    "notification": true,
    "vibration": true,
    "accountStatus": "active",
    "code": "verification_code",
    "codeTimestamp": "timestamp",
    "loginAttempts": 0,
    "loginAttemptTimestamp": "timestamp"
  }

```

- Message Object

```

{
  "messageID": "unique_identifier",
  "senderID": "sender_user_id",
  "receiverID": "receiver_user_id",
  "messageContent": "text_message",
  "imageLink": "url_to_image",
  "timestamp": "message_timestamp",
  "delivered": true,
  "read": false
}

```

- Contact Object

```

{
  "contactID": "unique_identifier",
  "userID": "user_id",
  "contactUserID": "contact_user_id"
}

```

Server-client architecture is aimed to achieve modularity, scalability, and maintainability in this messaging application. There are the involvement of two large sub-components of the system: the backend server and the frontend client application. In this architecture, there is a clear separation of concerns that requires every component to be developed, maintained, and scaled independently.

It hosts the backend server on Amazon EC2 for a robust and highly scalable environment of server-side operations. User data, contact lists, and message histories are stored in Google Firebase to leverage its real-time database capabilities and seamless interaction with the rest of Google Cloud services. Media files, such as images users upload, are kept in Amazon S3 buckets for large file storage and retrieval efficiently and securely.

Real-time messaging is facilitated through WebSocket's ability to communicate instantly between peers.

Finally, JSON Web Tokens are used to provide secure user authentication and session management in the system; this is a robust and scalable security solution.

The frontend client is written using React Native, allowing for core fluency and native feel across iOS and Android. It uses React Navigation to handle transitions between screens, which assures really smooth transitions and a clear user interface. Finally, it makes use of React Native Paper for having both consistent UI components and theming. On the other hand, the components at the backend level contain user management, which handles user registration, login, profile update, and JWT-based authentication. Contact management provides for adding, deleting, and searching of contacts. Through messaging service, it facilitates message sending and receiving, storing message history, providing real-time communication by WebSocket. It provides a media storage component. This component uploads the media to AWS S3 and retrieves URLs from stored media, hence effectively managing the part related to the multimedia content of this module.

It has multiple screens at the front-end, which help the app to perform various functionalities. The authentication screens include login, registration, and e-mail verification—giving a secure, easy means through which users can log in to your application. It also has profile screens so that users are able to view and edit information, thereby enhancing personalization.

Messaging screens offer full functionality to a user for communicating with contacts, sending text messages, and sharing images. Contact screens detail all of a user's contacts, with provisions for contact searching and management; settings screens could update notification settings and preferences.

The algorithm of user authentication begins with the inputting of the user's e-mail and password. The system checks the format of the email and whether the email is registered or not. If the email exists and is registered, it matches the password with the hashed one stored in Firebase. On success, a JWT is generated for the authenticated user, allowing safe access to the application.

First, a WebSocket between the client and server has to be opened. On sending the message by a user, the client sends it through the established WebSocket. The server

receives the message from the client, saves it in Firebase, and broadcasts it to the receiver if he/she is online. The server further updates the status of the message delivered/read based on what the receiver does. The algorithm to upload media is to select a media file for upload. At the front end-client, it compresses the media file and sends it to the server at the back. Upon receiving the file, the back-end will upload the same to AWS S3. Following this, store the URL of the uploaded media in Firebase. As shown above, this URL will be used in the user's profile or message to efficiently handle and retrieve multimedia.

On the aspect of data structures, class User has a number of attributes: unique identifier, name, email, hashed password, profile picture URL description, notification settings, account status, verification code, timestamps for generating code and logging. Class Message contains unique identifier, senderID and receiverId, actual content of the message, image link, timestamp, and delivery/readability status fields to. The Contact object shall hold a unique identifier, user ID and contact user ID to associate users with their contacts. It is through this detailed design and architecture that the messaging application would be robust, scalable, and easy for maintenance. Assisted by the use of Google Firebase in data storage, Amazon EC2 for hosting in the backend, and AWS S3 for media storage, it would make sure of high performance with good scalability. Adopting modern frameworks and technologies provides a smooth and reliable user experience.

### Implementation

The messaging mobile application is primarily composed of two elements: the server backend and the frontend mobile application. These two components have been done such that each takes care of particular elements of application functionality, making the experience by the user smooth.

#### 2.1.6. Backend Implementation

The back-end is done in Node.js, while the web framework is Express.js. It exposes multiple endpoints for user registration, authentication, messaging, and image upload functionality. Listed below are principal components alongside their implementation:

- **User Registration and Authentication:**

The backend uses JSON Web Tokens to manage authentication tokens. Upon registration and login, the user will receive a JWT token from the server.

The `createUser` and `signIn` functions handle user registration and login, respectively. It is calling functions imported via `db.js` that interface with a PostgreSQL database for storing and retrieving user information.

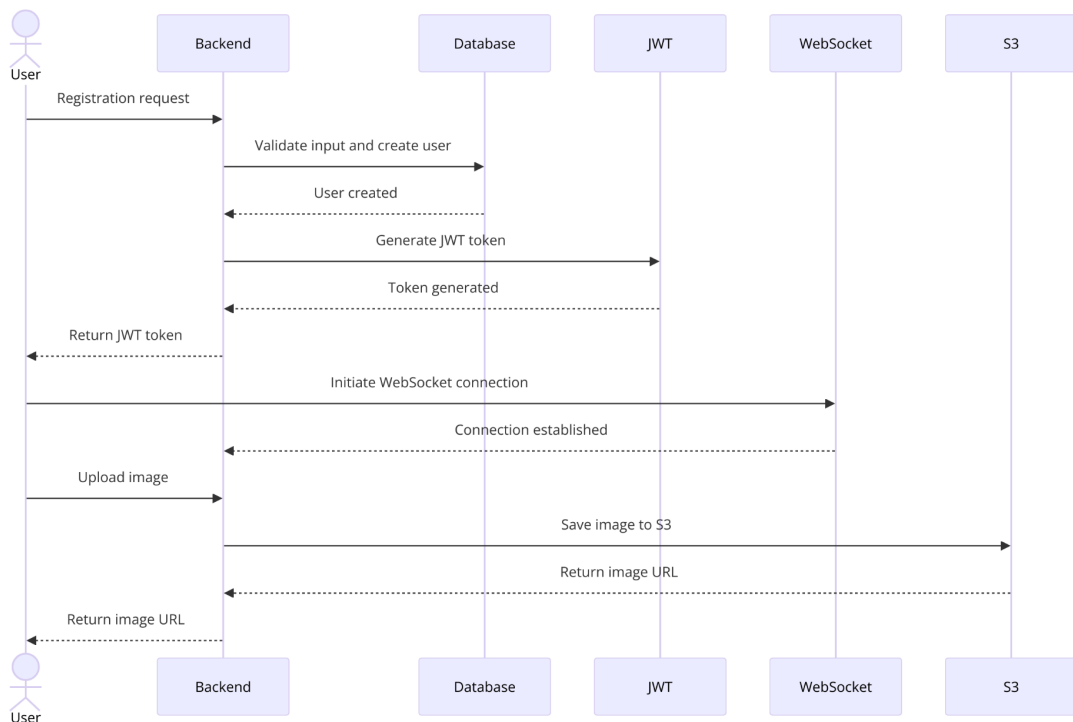
- **Messaging Functionality:**

It uses WebSockets to enable real-time messaging. A WebSocket server would listen for new connections and handle user messaging.

When a user sends a message, the WebSocket server will capture this message, process it, and forward it to the target recipient. Messages are also saved in the database for persistence using the `saveMessage` function from `db.js`. Image Upload:

The application provides support for image upload by handling file uploads via Multer and image storage via AWS S3. At the `send-image` endpoint, it will handle an upload of images to an S3 bucket and return the URL of the image to the client.

The file upload is handled by the middleware `upload.single('photo')` from `multer`, and the `AWS.S3` service is configured to interact with the S3 bucket. User Profile Management: Exposed endpoints update user profiles, including their profile pictures and settings. It ensures consistency in maintaining a user's preferences through functions like `UpdateUserProfilePicture`, `updateToggles`, and `updateDescription`, which manage these updates.



### 2.1.7. Front-end Implementation

It has a React Native frontend for the cross-platform mobile application. Some of the major features are user authentication, real-time messaging, and profile management. Following is the list of high-level components and their respective implementation:

- **Messaging Functionality:**

Navigation across different screens of the app is managed with the help of react-navigation. It comprises stack and tab navigators for navigation among authentication, chat, contacts, user profile info screens.

- **User Authentication:**

Authentication context is handled via React's Context API. It keeps track of the state related to whether a user is logged in or not, and also user information via the AuthContext.

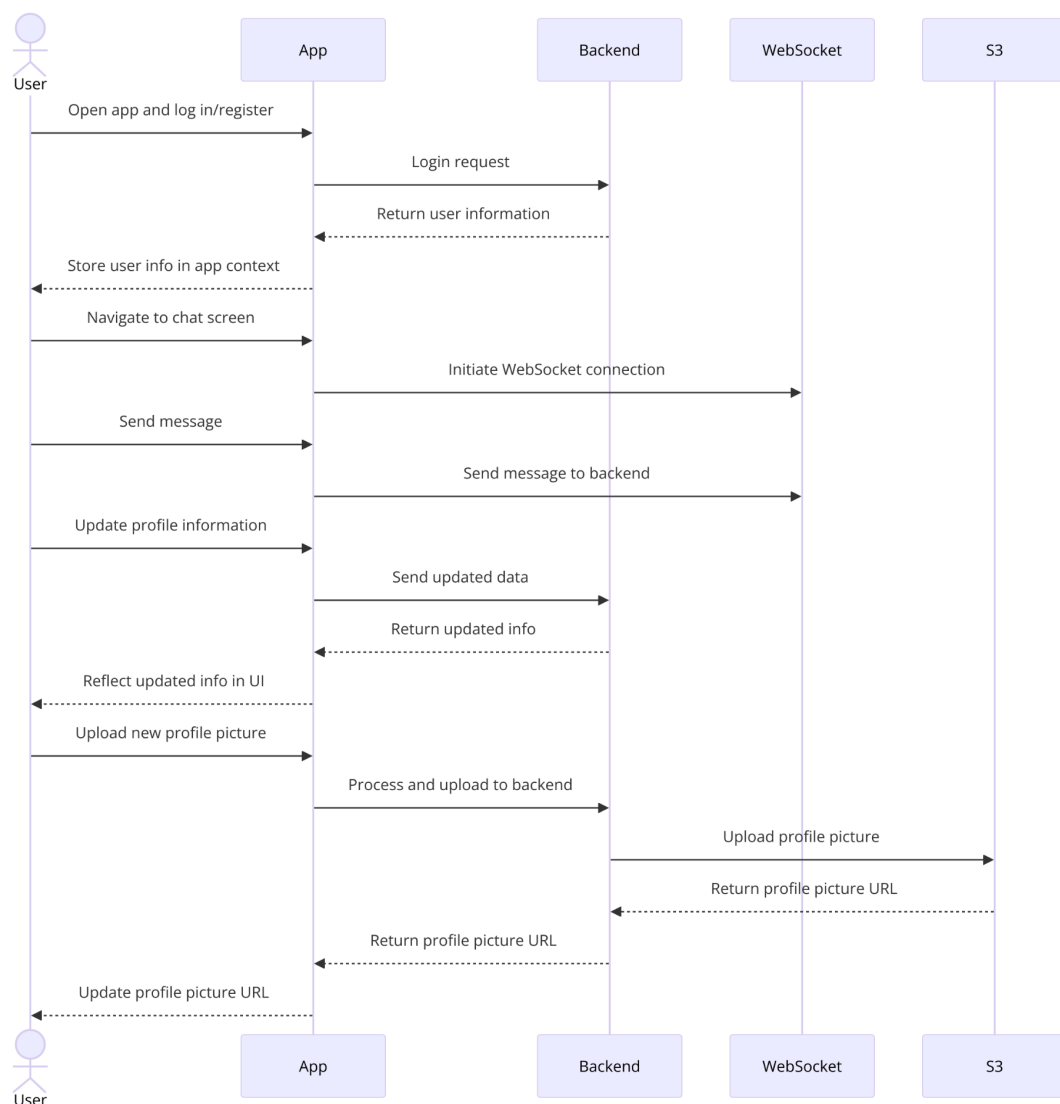
It uses AsyncStorage to persist user tokens and session data on the device, for seamless user experiences across app launches.

- **Chat functionality**

The chat component establishes a connection with the backend with a WebSocket for real-time messaging. Messages are sent using the `sendMessage` function on the opened WebSocket connection, and the chat history is fetched through a `getChatsAndMessages` function from the backend server. It keeps updating the chat UI based on new messages received, ensuring that a new user has an up-to-date conversation history.

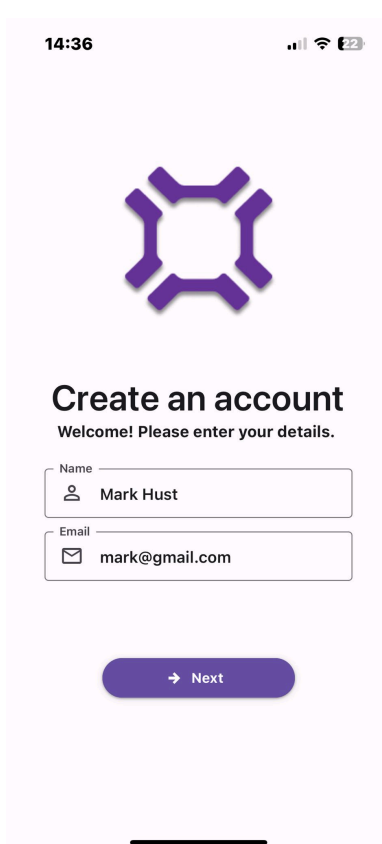
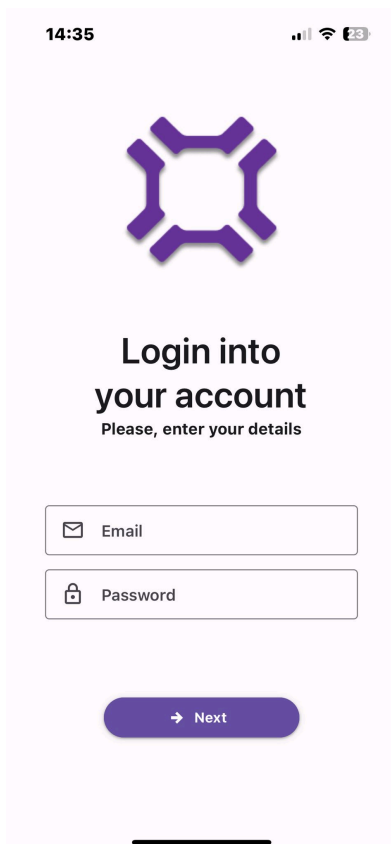
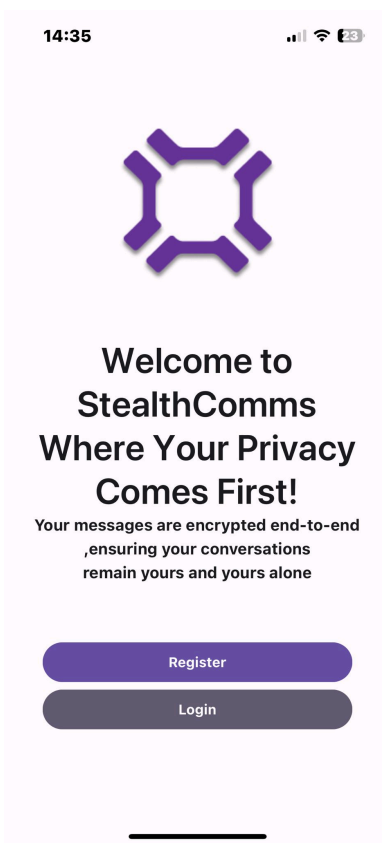
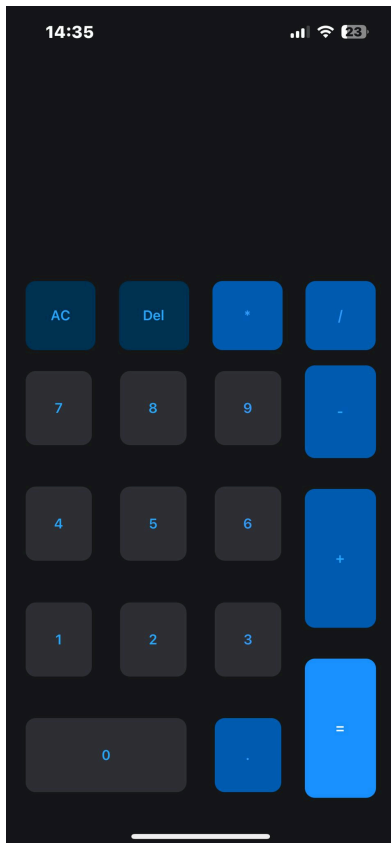
- **Profile Management**

Finally, this allows users to update information such as the username and description in profiles and even profile pictures. Functions `updateUsername` and `updateDescription` are responsible for sending a request with the new username and description back to the backend server. `ImagePicker` and `ImageManipulator` are used to handle image selection and processing before uploading the profile picture.









## Graphical User Interface (GUI)



14:37



14:37

22

## Check your email

We sent a code to fagnernunes@gmail.com

4

3

9

0

→ Next

Didn't receive the code? [Click here!](#)

1

2  
ABC

3  
DEF

4  
GHI

5  
JKL

6  
MNO


7  
PQRS

8  
TUV


9  
WXYZ

.




0




14:37




14:37

22

Password

 Test123456

Confirm

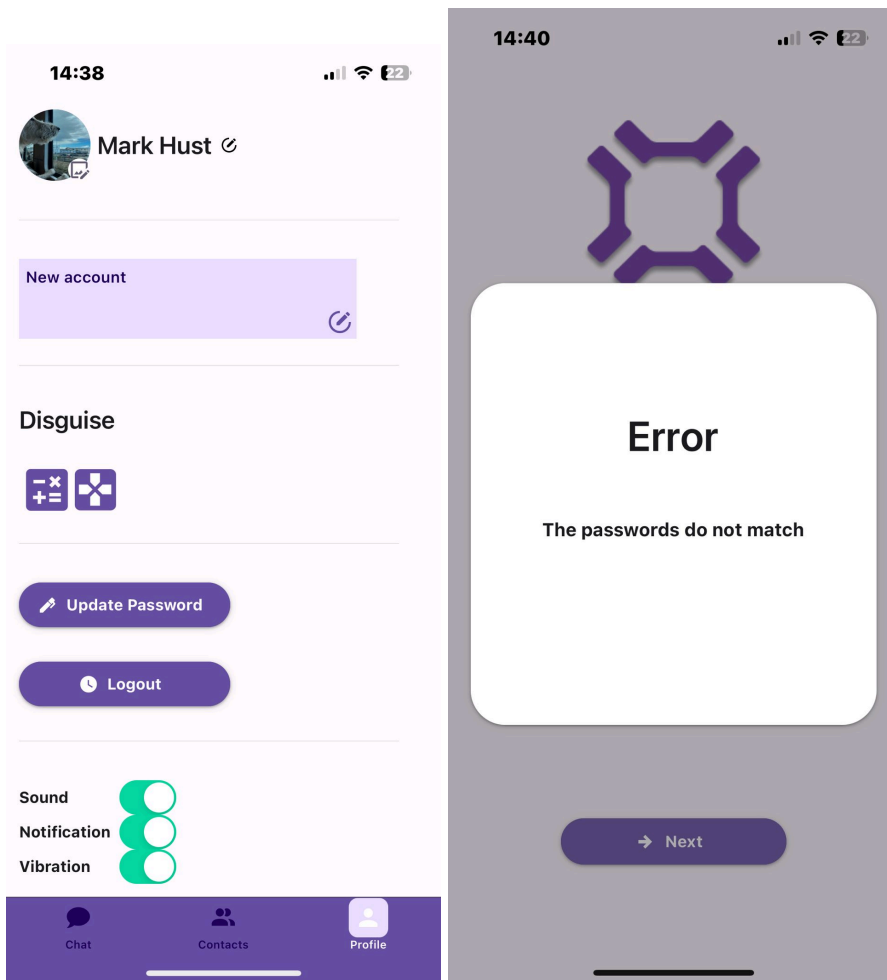
 Test123456

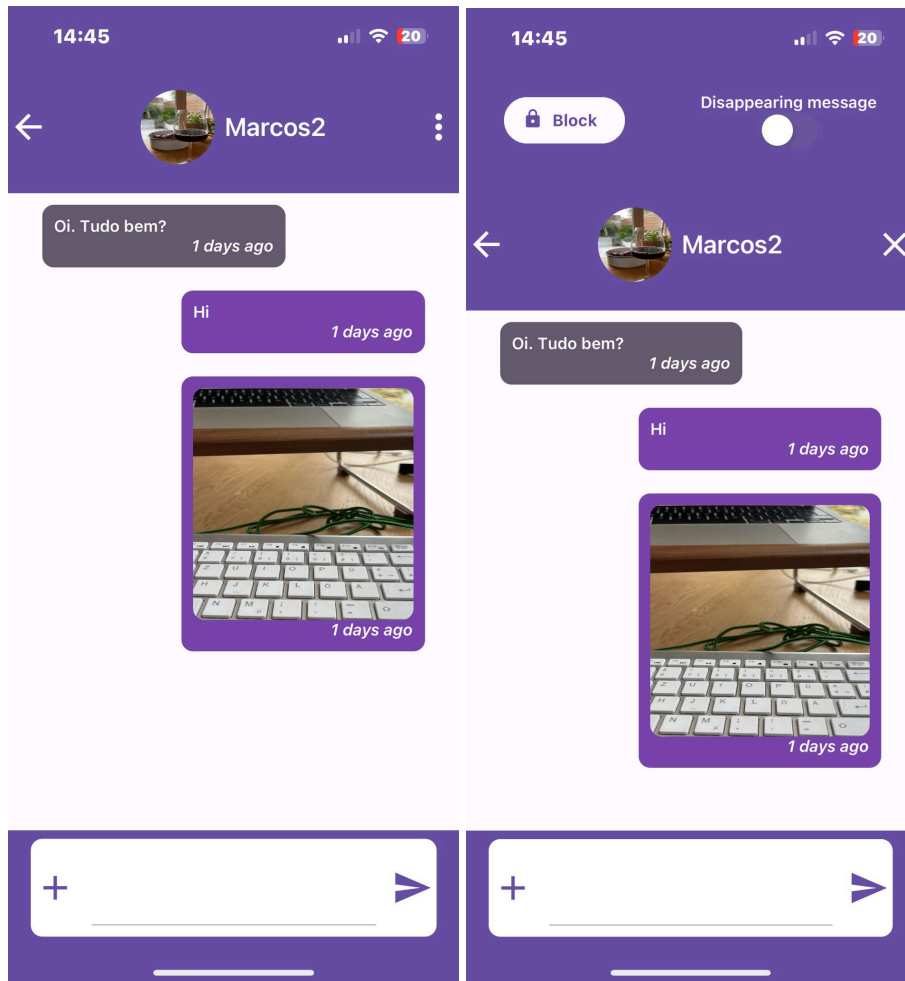
✓ Must have numbers

✓ Must be at least 8 Chars long

✓ Must have at least 1 capital letter

→ Next





### 3.0 Conclusions

This report has presented in detail the full development and implementation of a messaging mobile application using seamless integration of its back-end with its front-end. Given these views, the project had been able to achieve its aims of building a secure, scalable, user-friendly platform that employs modern technologies and best practices in the development of software.

The Node.js and Express.js-developed backend provides a robust framework that enables swift and efficient user authentication, real-time messaging, and multimedia management. With JSON Web Tokens provided for secure user sessions and AWS S3 for reliable image storage, this backend ensures data security and scalability. Detailed API documentation supports future development and maintenance, thereby making the backend strong enough to withstand changes by users.

It has a user-friendly and responsive user interface, developed with React Native at the frontend. This facilitates smooth transitions with React Navigation and React Native Paper for clean and consistent theming. Further enhanced with WebSocket integration, it provided

real-time messaging capabilities that made it possible for users to communicate without any glitches. It also had comprehensive features on profile management, ensuring that users can easily personalize their experience.

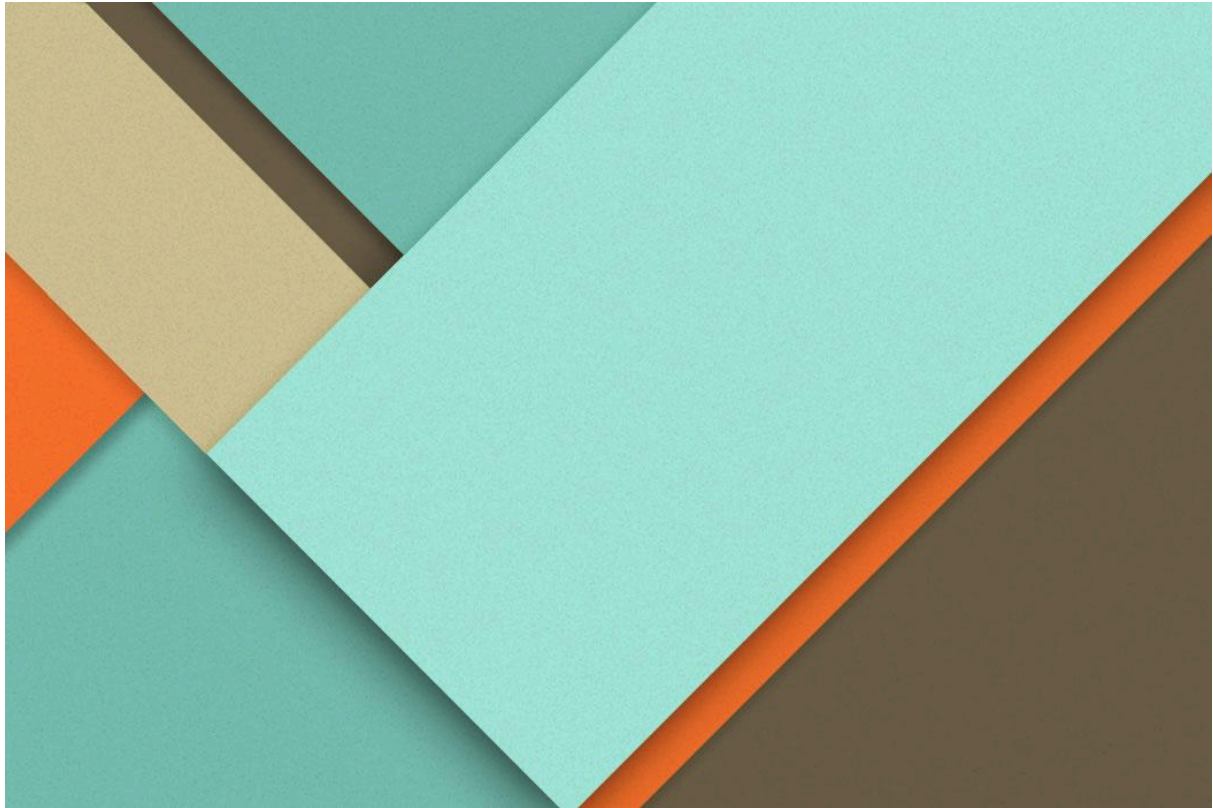
The messaging application thus addresses the common pitfalls that most modern communication platforms face: scalability, security, and usability. It provides a robust base for any improvements later on so that it will be able to serve its users by their dynamic demands. Security to data validation is highly observed in the application to ensure the privacy and integrity of the user's information are kept intact.

The overall success of this very project is an encouragement, then, to provide the landscape of messaging applications with complex yet reliable communication tools. This application will surely withstand further evolution in technology and scale up to the challenge, providing a dependable and people-oriented, clear communication solution for both personal and professional applications.

## 4.0 Appendices

### Project Proposal





# StealthComms

06.04.2024

---

Fagner Nunes

Computing Project (BSCCYBE4)

x19216718

## Objectives

The principal objective of the project is to implement a secret messaging application that provides high security for inter-user communication. In view of this, since it is able to masquerade as other apparently innocuous applications, such as a calculator, a tetris game, or even a snake game, this tool will enable the sending and receiving of messages without raising eyebrows or some other undesirable interest. Thus, this prototype is envisioned to yield a discreet communication tool while ensuring privacy and confidentiality for its users.

The protection of user confidentiality is one of the major objectives, which shall be attained by robust security features. This will include end-to-end encryption, two-factor authentication, and secure data storage and transmission protocols. It applies encryption methods that make conversations private and out of reach for any unwanted entity, hence reducing surveillance, interception, or even breaches in data.

## Functionalities

- End-to-end encryption for messages.
- Two-factor authentication for user login.
- Secure password management.
- Safe file sharing with encryption.
- Self-destruct message self-destruction option.
- Secure data storage and transmission.
- Regular security audits and updates applied.
- Secure login sessions and session management



## Goals

I decided to take up this project in consideration of the dire need for encrypted and confidential communication channels in today's digital environment. The world really needs innovative solutions that ensure people and groups communicate safely without putting their privacy into jeopardy or, at worst, sensitive information into jeopardy at a time when there are rising concerns about privacy breaches or surveillance. I will create a hidden messaging application, all self-contained within something as mundane as a calculator or even a game, and give it to the people as a discreet and safe platform for messaging.

Looking forward to achieving the objectives prescribed in Section 1.0, I intend to develop strong features of security and usability. Implementing technologies like React Native and Node.js, along with a host of libraries in both, for security measures, I will ensure that there is a smooth user experience but at the same time not turn a blind eye towards the erstwhile prerequisite of privacy and confidentiality guarantee.

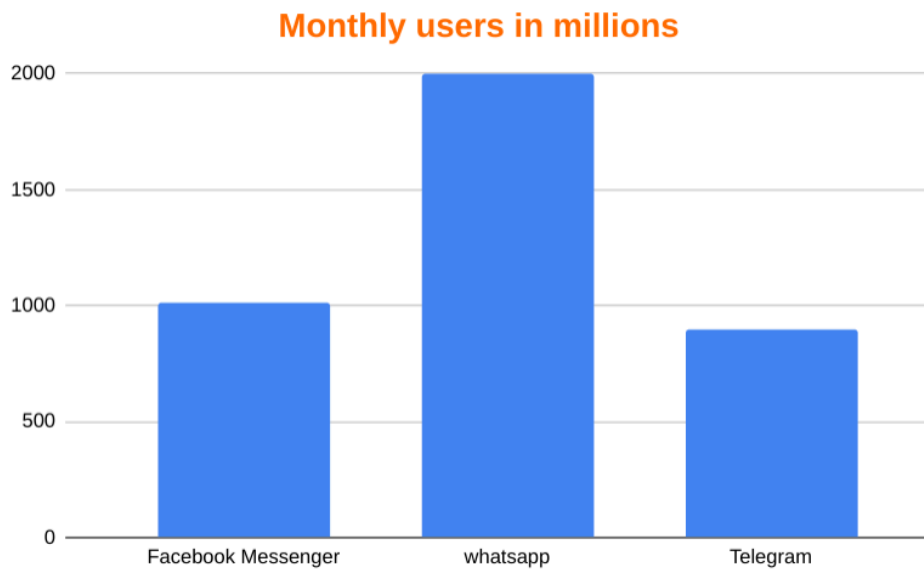
Key highlights include end-to-end message encryption, two-factor authentication, secure management for passwords, and encrypted file sharing that ensure the protection of user data. Moreover, add-ons such as self-destroyed messages and the possibility of verifying the identity of contacts aim to further improve security and reliability.

率 Such a secretive messaging app will provide reliable and secure communication across many contexts by catering to journalists, activists, government officials, law enforcement agencies, and sensitive citizens. In the process of implementing safety measures and updating them from time to time, I see myself instilling confidence in my users to communicate as they want in an increasingly digital world.

## Market Research

Therefore, conducting market research was one of the steps in deciding what features and functions would be required for an chatting app to potentially gain space in the market. Up to now, WhatsApp, Facebook Messenger, and Telegram have been the most frequently used apps. There can be plenty of reasons: one might choose a chat app over the other. Nevertheless, some users will go for Telegram or WhatsApp due to their safety and privacy aspects, whereas others may want integration with social media,

including Facebook Messenger. In developing my application, my focus and priority were on security and privacy.



## State Of The Art

What it has differs from WhatsApp, Facebook Messenger, and even Telegram with its standout feature of concealed messaging. While such mainstream messaging apps focus on the convenience and social connectivity dimensions, my app looks toward privacy and confidentiality above all.

### Concealed Messaging:

- Unlike WhatsApp and other messaging apps, be it Facebook Messenger or Telegram, my app disguises itself as innocuous everyday applications like a calculator, Tetris, or even a snake game. A feature that would make it rather especial to the user in a way that they were able to converse without giving away what actually the conversation is about.

### Discreet Communication:

- WhatsApp, Facebook Messenger, and even Telegram are rather well-known services for messaging. All of their interfaces are so familiar. My app provides private lanes of communication that don't look like anything identifiably special at a glance, therefore creating a further layer of privacy and security.

## Targeted User Base:

- While WhatsApp, Facebook Messenger, and Telegram cater to a broad user base for general messaging and social networking, my app targets specific user groups who prioritize privacy and confidentiality in their communication. This includes journalists, activists, government officials, law enforcement agencies, and privacy-conscious individuals.

## Emphasis on Privacy:

- While WhatsApp, Facebook Messenger, and Telegram have faced scrutiny over privacy concerns and data sharing practices, my app prioritizes user privacy above all else.

## Whatsapp:

- **The encryption** takes place at the front-end side of the application or on the user's device. The application makes use of the signal protocol so as to be able to encrypt the messages. The signal protocol is one of the popular protocols used by WhatsApp, Facebook, Skype, and at one stage, Google, which also comes with a good level of trust.
  - Signal protocol The Signal Protocol depends on three mechanisms to achieve this secure messaging. A theorem, the Double Ratchet Algorithm, continuously generates new encryption keys for each message; this ensures that if an attacker steals an old key, it will not be able to decrypt any conversation from before. Prekeys, stored on your device, enable the establishment of a secure connection while the user's device is offline. The triple Diffie-Hellman handshake keeps your communication private by protecting the exchange of keys between devices without any central authority.
- **FrontEnd:** The frontend of the app is developed in React Native. It is a framework developed by Facebook, whose main goal is to let developers who know and love React to become native app developers for mobile applications using JavaScript. Websocket was used for state management. It enables bi-directional communication channel between a client and a server so that either party can send data to the other at any time. SQLite is locally installed on the device so that it can store data locally on the user's device.
  - React Native eases mobile app development by taking leverage from JavaScript. Once written, the code can be launched on both Android and iOS. This framework builds apps with the look and feel of being genuinely

native. Since it bridges the gap between web development and native mobile app creation, you are at liberty to leverage your knowledge of JavaScript in developing mobile apps that don themselves in a familiar look and feel.

- SQLite is a small, fast, and self-contained database system that runs without a separate server; thus, it is embedded directly into an application. This implies that it helps in managing databases from applications without any resource constraints, like in the case of mobile applications.
- WebSockets enable a full-duplex, real-time communication channel between the browser and the server. That is, unlike classical web interaction, in which a browser requests something from the server and then waits for the server's response, WebSockets hold a persistent, open connection. Information can flow constantly in both directions, making it the perfect solution for applications that need constant updates, such as a chat feature, multi-player games, or live dashboards.
- **Backend** The Whatsapp back-end uses the Erlang programming language on its servers. It is a language that supports real-time communication and hence serves well the building of messaging apps. Cassandra NoSQL database is used for storage, thus offering incredible scalability.
  - Erlang is a really powerful language for software development of large, complex, and fault-tolerant systems. The language was conceptualized to allow many executions to take place simultaneously. That basically means that Erlang has grown very adept at concurrency, making it just perfect for applications that need continuous availability, such as messaging services or online banking.

## Telegram

- **Encryption** and decryption take place on the user's device. Telegram makes use of MTProto, otherwise known as the Telegram protocol; this is a custom-developed protocol for end-to-end encryption, very efficient and highly secure.
- **Front-end:** Telegram uses Java for their android app, Swift for the IOs version of their app, and C++ for their desktop application.
- **Back-end:** Very little information is shared by Telegram about their back-end stack. Not much information is found.

## Technical Approach:

- I. **Use Case Analysis:** Identify key use cases for the concealed chatting app, such as sending encrypted messages, verifying contact identities, and securely sharing files.
- II. **Design Phase:** Create wireframes and prototypes to visualize the app's user interface and functionality, ensuring it effectively disguises as a calculator, tetris game, or snake game while providing seamless access to the hidden messaging features.
- III. **Development:** Utilize React Native for frontend development, implementing features such as end-to-end encryption, two-factor authentication, secure password management, self-destructing messages, functional calculator, functional tetris game, functional snake game, an option for the user to select how they would like to conceal the messaging app.
- IV. **Backend Development:** Build the backend server using Node.js and Express.js, integrating Socket.io for real-time communication and MongoDB or PostgreSQL for secure data storage. Implement JWT for authentication and session management, and bcrypt for password hashing. Implement 2 factor authentication using google authenticator.
- V. **Testing:** Perform extensive testing, including unit testing, integration testing, and security testing, to ensure the app functions correctly and securely across different platforms and devices.
- VI. **Deployment:** Deploy the app to app stores while ensuring compliance with their guidelines and regulations. Implement regular security audits and updates to address any vulnerabilities and ensure ongoing protection of user data

## Milestones:

- Requirement Gathering and Analysis
- Design and Prototyping
- Frontend and Backend Development
- Integration and Testing

- Deployment
- Security Audits and Updates

## Project Plan

### Week 1-2: Requirement Gathering and Design

- Analyze use cases and prioritize features.
- Create wireframes and prototypes for the app's UI and functionality.

### Week 3-4: Frontend Development

- Set up the project environment using React Native.
- Implement the disguised UI elements (calculator, tetris game, snake game).
- Develop the messaging interface with end-to-end encryption and self-destructing messages.
- Integrate React Navigation for seamless navigation within the app.

### Week 5-6: Backend Development and Integration

- Build the backend server using Node.js and Express.js.
- Implement user authentication with two-factor authentication and JWT.
- Set up secure data storage with MongoDB or PostgreSQL.
- Integrate Socket.io for real-time communication between clients and the server.

### Week 7: Testing and Refinement

- Conduct unit testing and integration testing to ensure functionality.
- Perform security testing to identify and address vulnerabilities.
- Gather feedback from users for further refinement.
- Address any bugs or issues discovered during testing.

### Week 8: Deployment and Finalization

- Prepare the app for deployment to app stores.
- Create documentation and training materials for users.
- Deploy the app to app stores while ensuring compliance with guidelines.
- Perform a final security audit and implement any necessary updates.
- Provide ongoing support and maintenance as needed.

## Objective

Use Case Analysis: The main use cases of the hidden chatting application will deal with sending encrypted messages, checking contacts' identity, and sharing files over a secure network.

Design Phase: Wireframe and prototype the user interface for this app, which should act like a working calculator, Tetris game, or snake game but give access through seamless interactions to the hidden messaging features.

Development will be done in React Native, adding the next set of features: end-to-end encryption, two-factor authentication, secure password management, self-destructing messages, a functional calculator, a functional Tetris game, a snake game that works, and an option for the user to select how they would like to conceal the messaging app.

Backend Development: Create a backend server using Node.js and Express.js. Add Socket.io for real-time communication and MongoDB or PostgreSQL for secure data storage. Further, include JWT since authentication is going to be required for session management and bcrypt for password hashing. 2-factor authentication will be implemented using Google Authenticator.

Testing involves running thorough tests for appropriateness, integration, and security to make sure the app performs its functions properly and with utter security on different platforms and devices.

Deploy: Make the app ready after compliance guidelines and regulations of app stores, then make a publication. Run security audits every now and then for fixing vulnerabilities so that users' data may remain safe.

## Functionalities

### Must have

- Authentication using 2-factor authentication Must have ▾ In progress ▾
  - The purpose of this functionality is to add another security layer to the application. The 2-factor-authentication process requires the user to provide something he knows (Password) and something he processes (his phone). The app is going to a service provided by google which sends the user a code in order to verify the user is logging from his own device.
- Registration (ideally should be done via SMS message, but it will be done using email) Must have ▾ Completed ▾

- This functionality allows users to register for an account. While the ideal method is through SMS messages for convenience and immediacy, registration will be processed by identifying the user via email instead.
- Secure local data storage Must have ▾ In progress ▾
  - Secure local data storage refers to the practice of storing sensitive or private data on a device or system in a manner that protects it from unauthorized access or tampering. The objective is to store the users private and public keys safely on the users device.
- Secure session management Must have ▾ Completed ▾
  - The purpose of this feature is to prevent unauthorized users from sending requests pretending to be an authentic user. Key aspects include implementing strong authentication methods, encrypting session data, regularly expiring sessions, and guarding against common threats such as session hijacking and fixation.
- End-to-end encryption Must have ▾ Completed ▾
  - End-to-end encryption is the key aspect of this application. The messages are encrypted in the sender's device and decrypted in the receiver's device. The public and private keys or the decrypted messages are not stored in the servers. Thus, if the servers are compromised, no sensitive information will be lost.
- Secure notifications (notify new messages without creating suspicion) Must have ▾ Completed ▾
  - This feature is important as the purpose of the application is to be disguised as a regular app. By sending disguised notifications and not as regular as a messaging app, only the device owner should understand they have a new message.
- real time messaging Must have ▾ Completed ▾
- Secure Image sharing, taking photo Must have ▾ Not started ▾
  - "Image sharing, taking photo" functionality enables users to capture photos using their device's camera and share them seamlessly with others.
- Adding new contacts Must have ▾ Completed ▾
  - The user should be able to find a user by their email or username and add them to their contacts.
- Deleting messages Must have ▾ Obsoleted ▾
  - A user should be able to delete a specific message or several messages from his device.
- Blocking user Must have ▾ Obsoleted ▾



- This feature allows a user to block unwanted users from messaging them or finding their profile.
- Option for user to select which app should be used as disguise **Must have** **Completed**
  - The messaging app can disguise itself as several other apps. The user can select in the settings screen which disguise they would prefer.
- Secret Mechanism for users to switch into messaging apps. **Must have** **Not started**
  - Once the user opens the primary app i.e. Calculator, the user should use a secret mechanism to open the messaging app i.e. pressing the screen with three fingers for 3 seconds.
- Profile editing (Name, description, Picture) **Must have** **Completed**
- Functionalities of a calculator (disguise) **Must have** **Completed**
- Users should be able to see if message was delivered and/or read **Must have** **Completed**
- Functionalities of a tetris game (Disguise) **Must have** **Not started**

## Should Have

- Functionalities of a Reminder app for drinking water (Disguise) **Should have** **Obsolete**

## Could Have

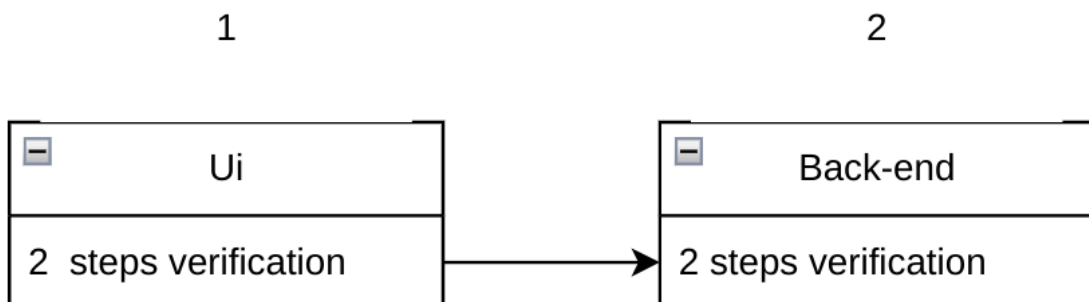
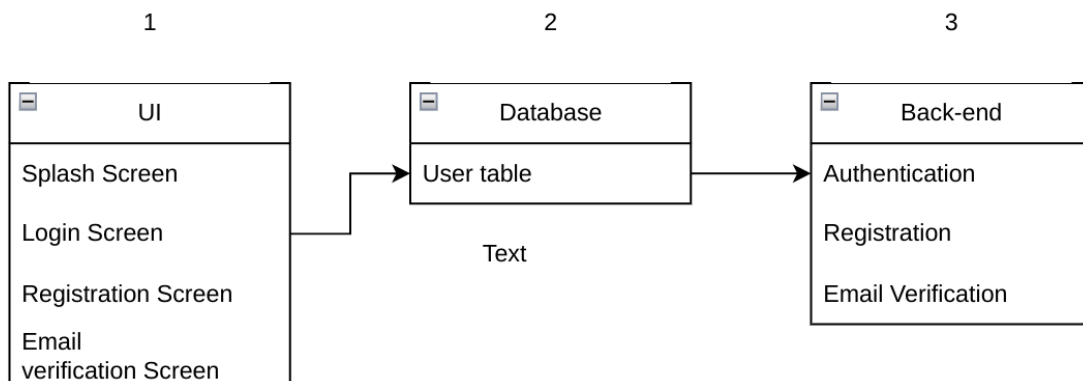
- Users should be able react to messages (emoji) **Could have** **Obsolete**
  - This feature should allow one user to react to another user's message using an emoji.
- Secure File sharing **Could have** **Obsolete**
  - This feature allows a user to send files such as .pdf, .docx, .mp3, .mp4, .wav.
  - Every file will be encrypted and decrypted on users devices.
- Option for self destructing messages **Could have** **Completed**
  - This feature allows for messages to be deleted after they have been read by the receiver.
- Giphy sharing via (giphy api) **Could have** **Obsolete**
  - This feature allows users to send animated images to each other.

## Implementation Order

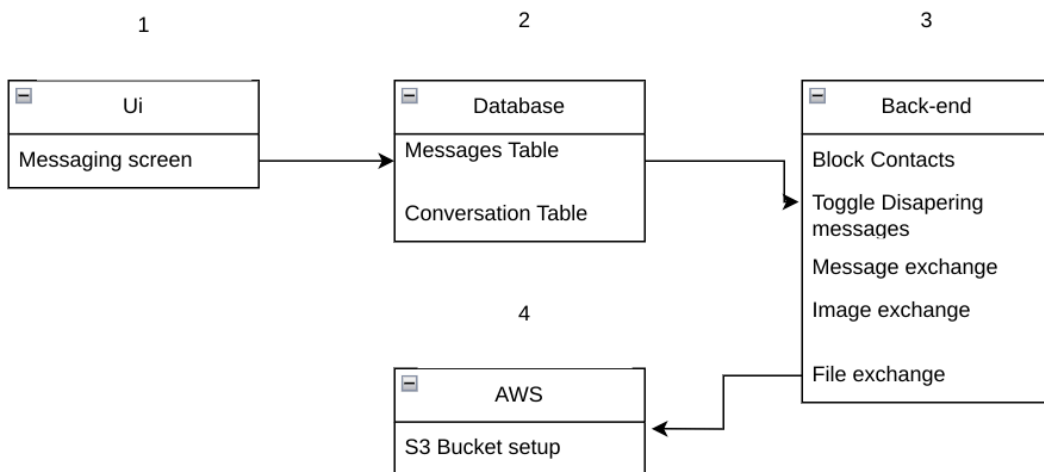
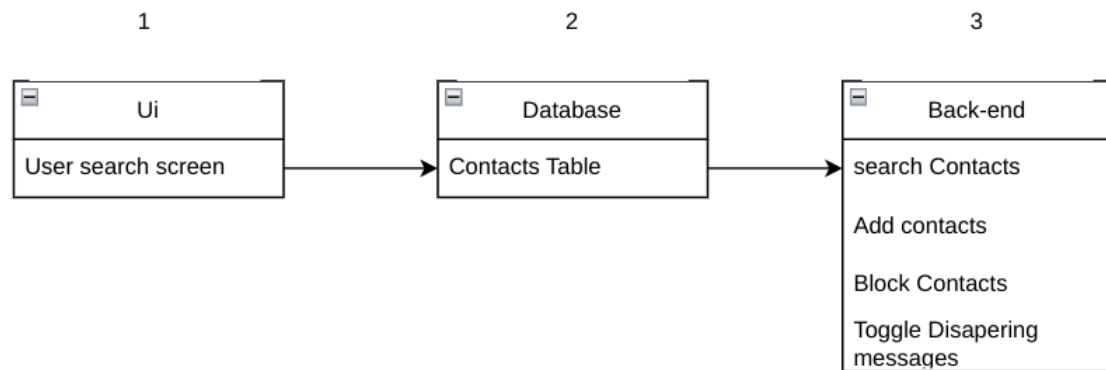
In the implementation process, I am going to start by implementing the screens and functionalities on the "**before authentication**" Stage. The stage entails registration, login, 2 steps verification, setting up email verification database and its appropriate tables, dummy apps. After the UI and the back end for these features have been properly implemented, we are going to move to the next stage, the "**post authentication**" stage.

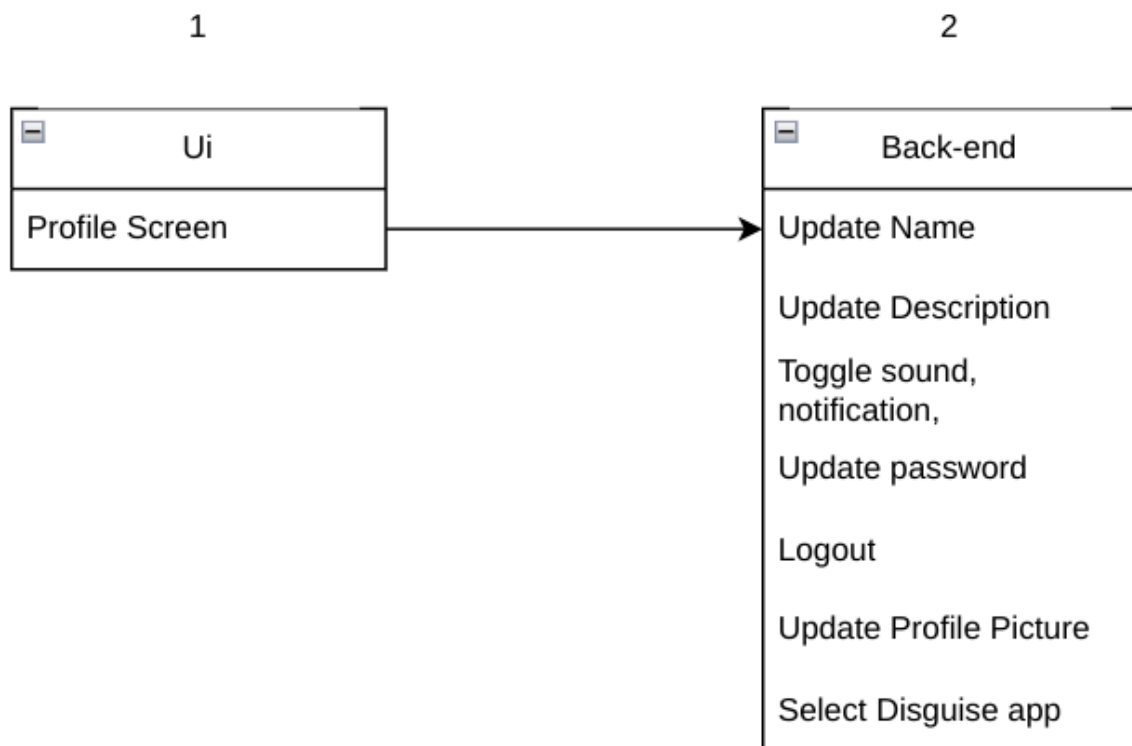
It is during this process that I am going to follow the interactive incremental development model. In stubbing out a feature, testing it, fixing any bugs related to it, and re-testing is how I would implement following this model. We follow this workflow until the feature works as per plan.

### Before authentication



## Post authentication



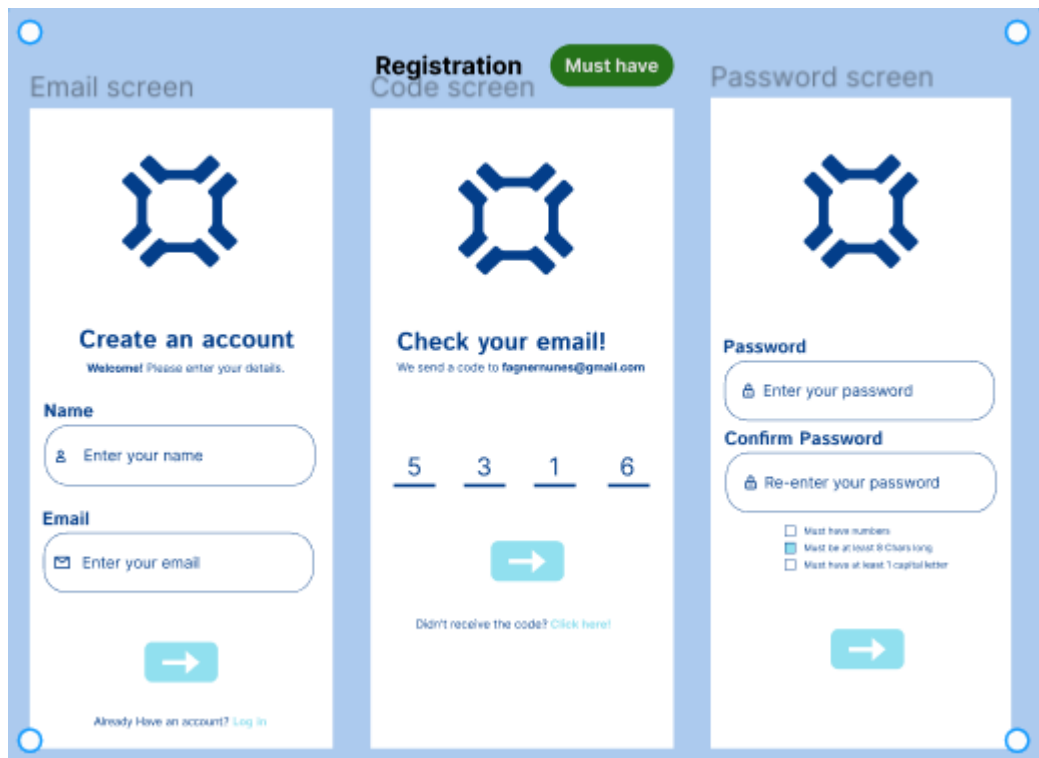


## Design

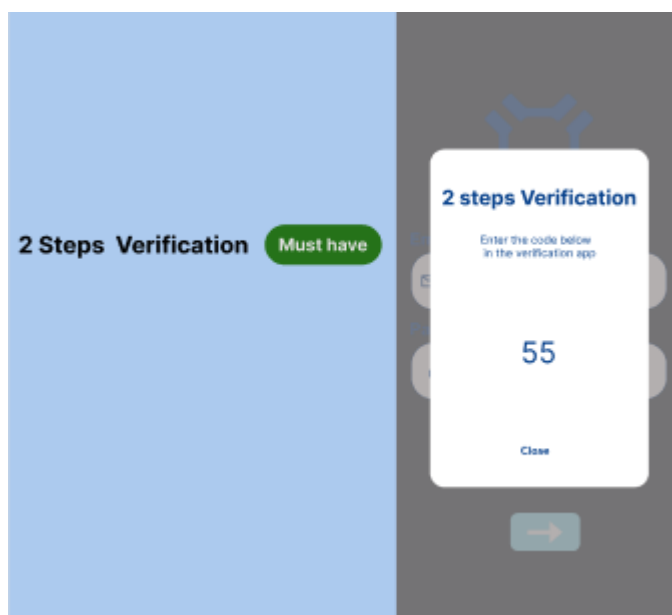
### Wireframe for the dummy apps



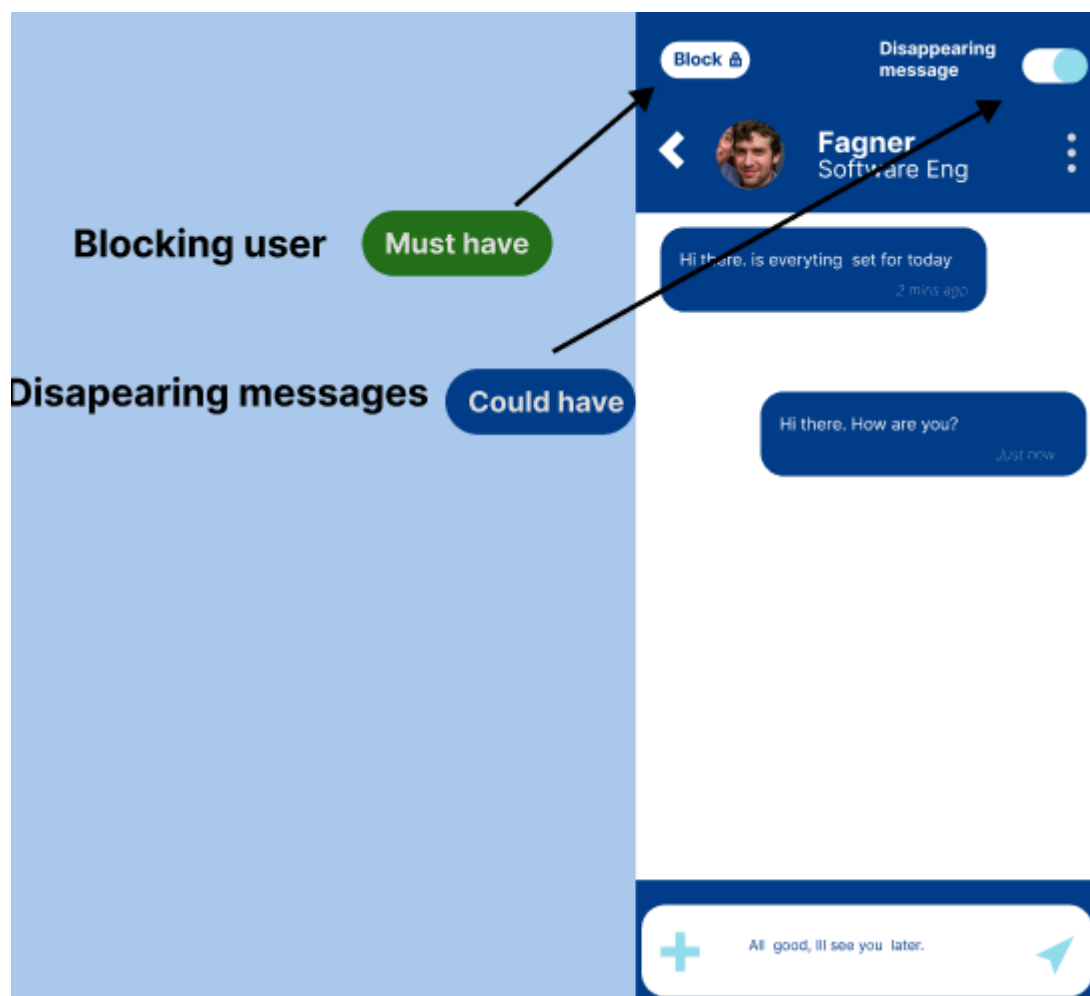
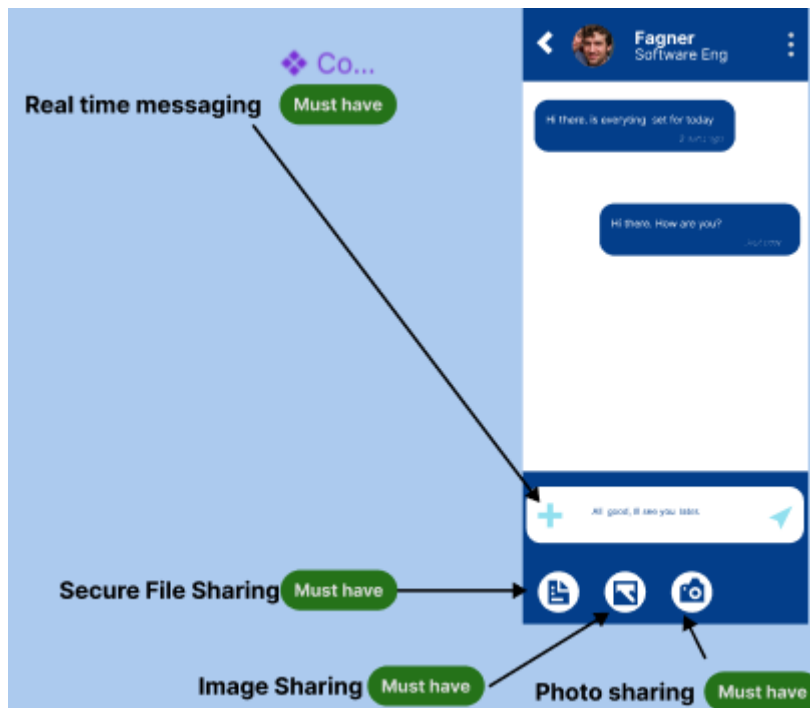
### Registration screens



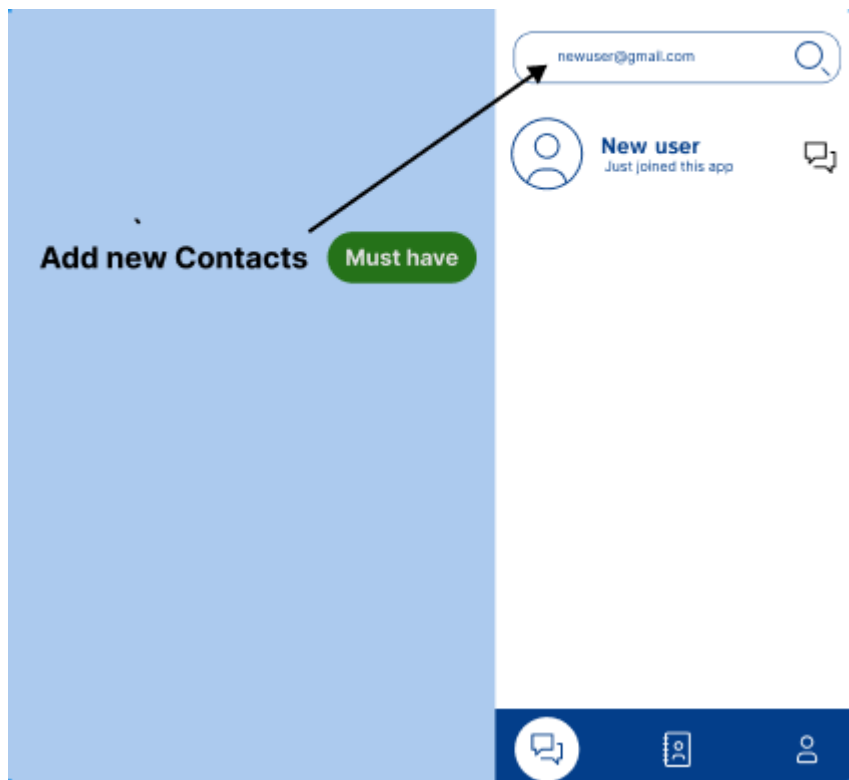
## 2 steps verification screen



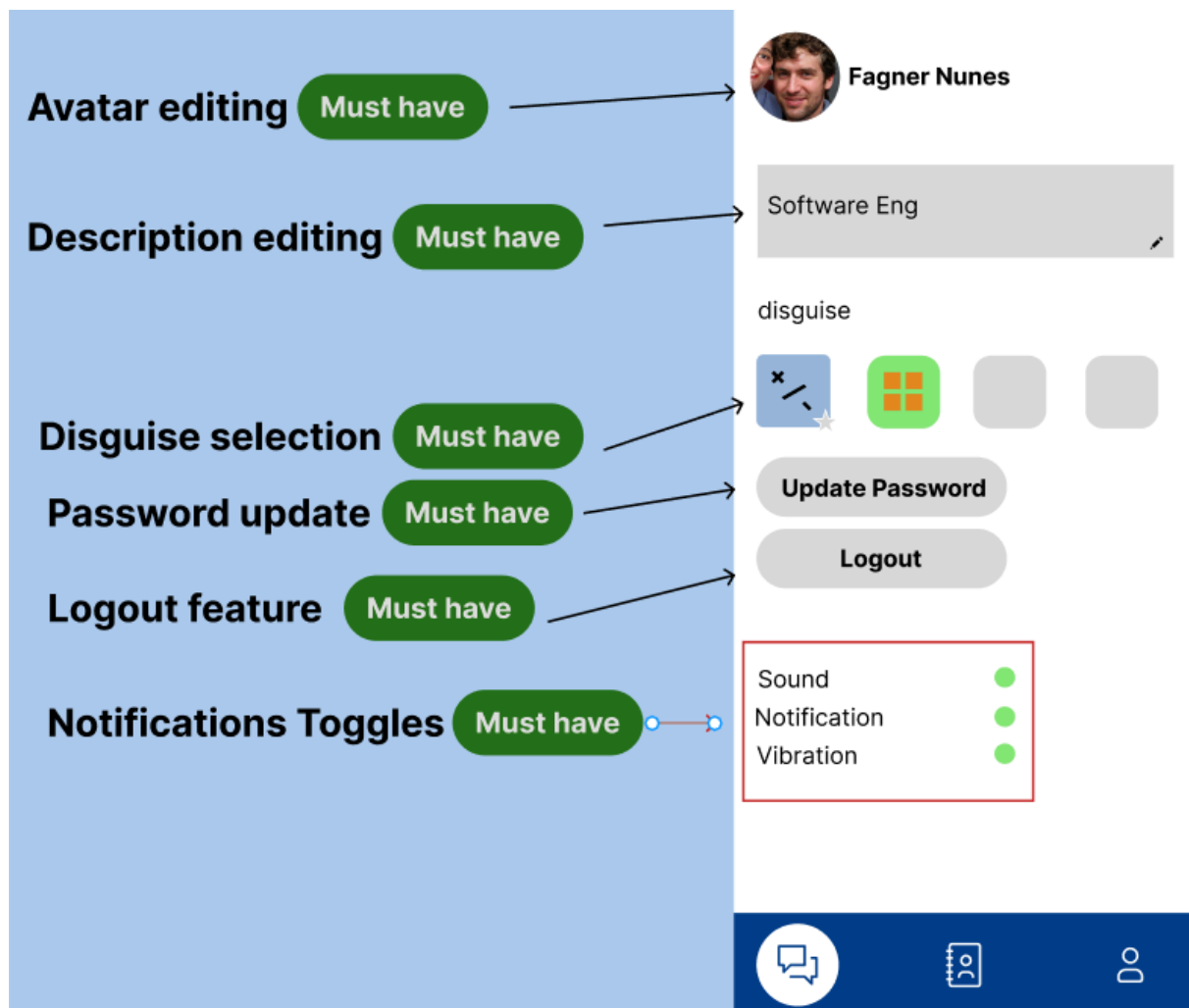
## Chat screen



## User search Screen

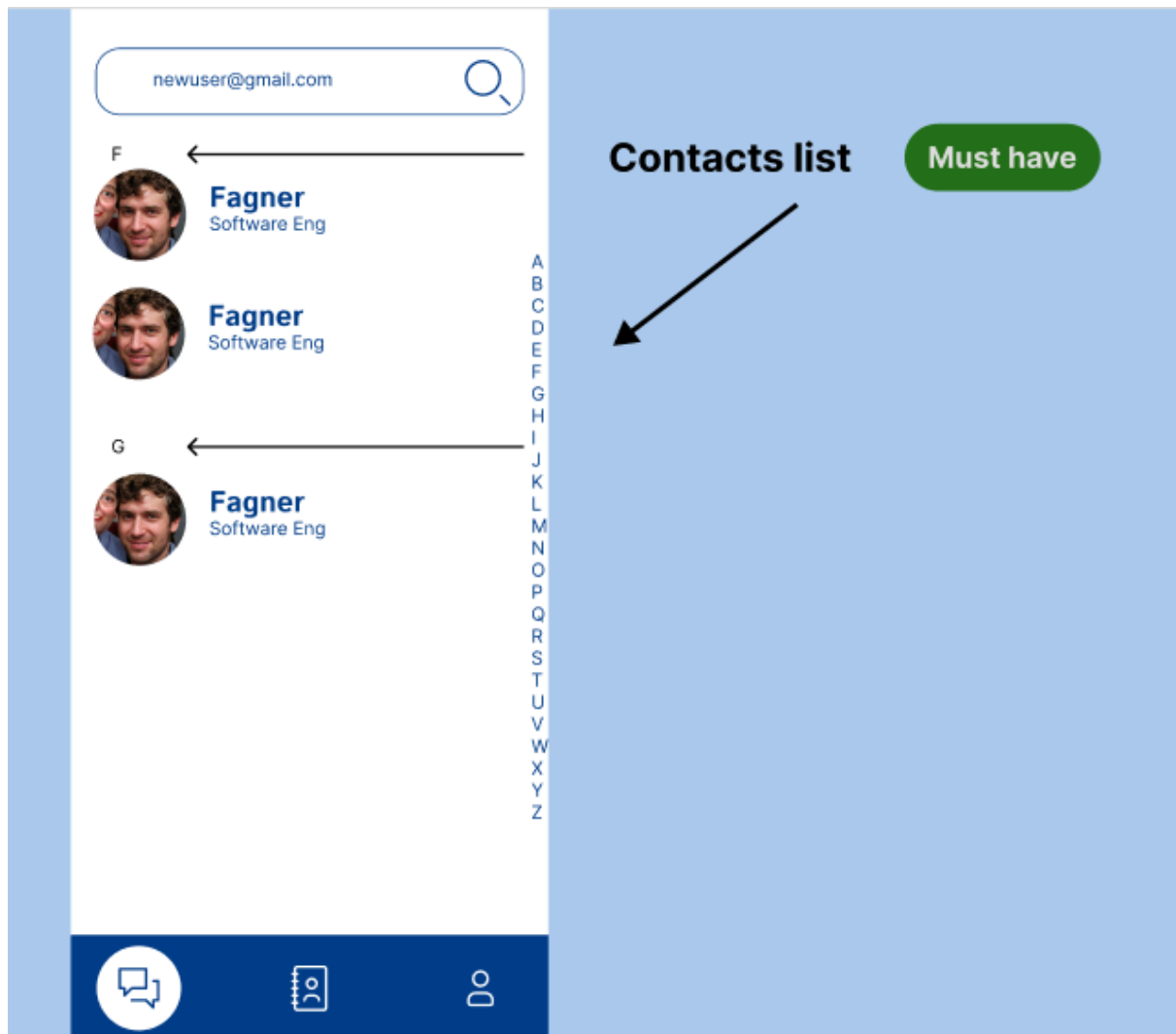


## Profile Screen





## Contact list screen



### 1.1. Documentation

```
(inner) InsideApp() → {React.Element}
```

Description:

This is the inside app component. It is the main component of the app. It contains the navigation container and the main navigation component.

Source:

App.js, line 125

Returns:

Rendered component.

Type `React.Element`

```
(inner) MainNavigation() → {React.Element}
```

Description:

This is the main navigation component. It contains the tabs for the app. It is only rendered when the user is logged in.

Source:

[App.js, line 43](#)

Returns:

Rendered component.

Type `React.Element`

```
(inner) ProfileRoute() → {React.Element}
```

Description:

This is the route component for the Profile screen.

Source:

[App.js, line 101](#)

Returns:

Rendered component.

Type `React.Element`

```
(inner) StackTest() → {React.Element}
```

Description:

This is the stack navigator component for the app.

Source:

[App.js, line 108](#)

Returns:

Rendered component.

Type `React.Element`

```
(inner) checkLoggedInStatus() → {void}
```

Description:

This function checks if the user is logged in by fetching the value from AsyncStorage. It also sets the loading state to false after checking.

Source:

[App.js, line 136](#)

Throws:

If there is an error fetching the logged-in status.

Type `Error`

Returns:

Type `void`

# AuthContext

Provides authentication context and WebSocket connection management.

Description:

Provides authentication context and WebSocket connection management.

Source:

[AuthContext.js, line 6](#)

## Methods

```
(inner) AuthProvider(children) → {JSX.Element}
```

Description:

Provides authentication context to its children.

Source:

[AuthContext.js, line 17](#)

Parameters:

Name	Type	Description
------	------	-------------

---

`children`      `Object`      The child components that will consume the context.

## Returns:

The AuthContext provider with its value.

Type `JSX.Element`

```
(inner) connectWebSocket ()
```

### Description:

Establishes a WebSocket connection and handles events.

### Source:

`AuthContext.js`, line 87

```
(inner) getChatsAndMessages ()
```

### Description:

Fetches chats and messages from the backend.

### Source:

`AuthContext.js`, line 122

## Throws:

Will throw an error if the network request fails.

```
(inner) getUserData ()
```

### Description:

Fetches user data from the backend.

### Source:

`AuthContext.js`, line 33

## Throws:

Will throw an error if the network request fails.

# Calculator

A simple calculator component for React Native.

Description:

A simple calculator component for React Native.

Source:

[screens/beforeLogin/Calculator.js, line 6](#)

## Methods

```
(inner) handleCalculate() → {void}
```

Description:

Calculates the result of the input expression.

Source:

[screens/beforeLogin/Calculator.js, line 59](#)

Throws:

Will throw an error if the input expression is invalid.

Returns:

Type `void`

```
(inner) handleClear() → {void}
```

Description:

Clears the input and result

Source:

[screens/beforeLogin/Calculator.js, line 38](#)

Returns:

Type `void`

```
(inner) handleDelete() → {void}
```

Description:

Deletes the last character from the input.

Source:

[screens/beforeLogin/Calculator.js, line 49](#)

Returns:

Type `void`

```
(inner) handlePress(value) → {void}
```

Description:

Handles button press events.

Source:

[screens/beforeLogin/Calculator.js, line 18](#)

Parameters:

Name	Type	Description
<code>value</code>	<code>string</code>	The value of the button pressed.

Returns:

Type `void`

# Camera

Camera component for taking and uploading photos.

Description:

Camera component for taking and uploading photos.

Source:

[screens/Camera.js, line 15](#)

## Methods

```
(inner) pickImage()
```

Description:

Pick an image from the gallery.

Source:

[screens/Camera.js, line 156](#)

Throws:

Will throw an error if the image picking fails.

```
(inner) sendMessage(imgUrl)
```

Description:

Send a message with an image link.

Source:

[screens/Camera.js, line 90](#)

Parameters:

Name	Type	Description
<code>imgUrl</code>	<code>string</code>	The URL of the image to send.

Throws:

Will throw an error if the message fails to send.

```
(inner) takePhoto()
```

Description:

Take a photo using the camera.

Source:

[screens/Camera.js, line 59](#)

Throws:

Will throw an error if the camera fails to take a photo.

```
(inner) toggleCameraFacing()
```

Description:

Toggle the camera facing direction.

Source:

screens/Camera.js, line 51

```
(inner) uploadImage(uri)
```

Description:

Upload the image to the server.

Source:

screens/Camera.js, line 116

Parameters:

Name	Type	Description
uri	string	The URI of the image to upload.

Throws:

Will throw an error if the image upload fails.

# ChatComponent

This module handles the chat functionality including sending, receiving, and displaying messages.

Description:

This module handles the chat functionality including sending, receiving, and displaying messages.

Source:

screens/afterLogin/ChatStack/ChatComponent.js, line 13

## Methods

```
(inner) blockUser()
```

Description:



Blocks the user by making an API call.

Source:

[screens/afterLogin/ChatStack/ChatComponent.js](#), line 113

Throws:

Will throw an error if the API call fails.

```
(inner) findChatByOtherUserId() → {Object|null}
```

Description:

Finds the chat by the other user's ID.

Source:

[screens/afterLogin/ChatStack/ChatComponent.js](#), line 188

Returns:

The chat object if found, otherwise null.

Type `Object | null`

```
(inner) sendMessage(imgUlr)
```

Description:

Sends a message through the socket.

Source:

[screens/afterLogin/ChatStack/ChatComponent.js](#), line 161

Parameters:

Name	Type	Description
<code>imgUlr</code>	<code>string</code>	The URL of the image to send.

Throws:

Will throw an error if the socket is not available.

```
(inner) unblockUser()
```

Description:

Unblocks the user by making an API call.

Source:

[screens/afterLogin/ChatStack/ChatComponent.js, line 137](#)

Throws:

Will throw an error if the API call fails.

# ChatItem

A component that displays a chat item with user information, last message, and unread message count.

Description:

A component that displays a chat item with user information, last message, and unread message count.

Source:

[screens/afterLogin/ChatStack/ChatItem.js, line 5](#)

Author:

Fagner Nunes

## Methods

```
(inner) convertTimeStamp(timestamp) → {string}
```

Description:

Converts a timestamp to a human-readable format. The function calculates the difference between the current date and the message date and returns a string with the time difference.

Source:

[screens/afterLogin/ChatStack/ChatItem.js, line 14](#)

Parameters:

Name	Type	Description
------	------	-------------

---

<code>timestamp</code>	<code>number</code>	The timestamp to convert.
------------------------	---------------------	---------------------------

### Throws:

- Throws an error if the timestamp is invalid.

Type `Error`

### Returns:

- The formatted time difference.

Type `string`

```
(inner) countUnreadMessages(messages) → {number}
```

### Description:

Counts the number of unread messages. The function iterates over the messages array and counts the number of messages that are unread.

### Source:

[screens/afterLogin/ChatStack/ChatItem.js](#), line 40

### Parameters:

Name	Type	Description
------	------	-------------

---

<code>messages</code>	<code>Array</code>	The array of message objects.
-----------------------	--------------------	-------------------------------

### Throws:

- Throws an error if the messages array is invalid.

Type `Error`

### Returns:

- The count of unread messages.

Type `number`

```
(inner) workWithMsg(msg) → {string}
```

Description:

Processes the message to ensure it is displayed correctly. The function checks the length of the message and truncates it if it is too long.

Source:

[screens/afterLogin/ChatStack/ChatItem.js, line 57](#)

Parameters:

Name	Type	Description
<code>msg</code>	<code>string</code>	The message to process.

Throws:

- Throws an error if the message is invalid.

Type `Error`

Returns:

- The processed message.

Type `string`

# ChatScreen

This module represents the chat screen of the application. It displays the chat messages and allows the user to search for messages.

Description:

This module represents the chat screen of the application. It displays the chat messages and allows the user to search for messages.

Source:

[screens/afterLogin/ChatStack/ChatScreen.js, line 20](#)

Author:

Fagner Nunes

## Methods

```
(inner) getUserData()
```

Description:

Fetches user data from the backend and sets it in the context. It sends a request to the backend to get the user data.

Source:

[screens/afterLogin/ChatStack/ChatScreen.js, line 36](#)

Throws:

Will throw an error if the network request fails.

```
(inner) search(name)
```

Description:

Filters the messages based on the search query. It filters the messages based on the name of the contact.

Source:

[screens/afterLogin/ChatStack/ChatScreen.js, line 90](#)

Parameters:

Name	Type	Description
<code>name</code>	<code>string</code>	The name to search for.

# ContactItem

Component to display a contact item with options to add or remove a friend.

Description:

Component to display a contact item with options to add or remove a friend.

Source:

[screens/afterLogin/ContactsStack/ContactItem.js, line 1](#)

Author:

## Methods

```
(inner) addFriend() → {Promise.<void>}
```

Description:

Adds a friend to the contact list. It sends a request to the backend to add the contact to the user's contact list.

Source:

[screens/afterLogin/ContactsStack/ContactItem.js](#), line 54

Throws:

Will throw an error if the contact cannot be added.

Returns:

Type `Promise.<void>`

```
(inner) removeFriend() → {Promise.<void>}
```

Description:

Removes a friend from the contact list. It sends a request to the backend to remove the contact from the user's contact list.

Source:

[screens/afterLogin/ContactsStack/ContactItem.js](#), line 23

Throws:

Will throw an error if the contact cannot be removed.

Returns:

Type `Promise.<void>`

# ContactsScreen

Screen component for displaying and searching contacts. The user can search for contacts by name or email.

Description:

Screen component for displaying and searching contacts. The user can search for contacts by name or email.

Source:

[screens/afterLogin/ContactsStack/ContactsScreen.js, line 1](#)

Author:

Fagner Nunes

# ProfileScreen

Screen component for displaying and updating user profile information.

Description:

Screen component for displaying and updating user profile information.

Source:

[screens/afterLogin/ProfileStack/ProfileScreen.js, line 1](#)

Author:

Fagner Nunes

# UpdatePassword

This module handles the password update functionality. It validates the password based on certain criteria and updates the user's password.

Description:

This module handles the password update functionality. It validates the password based on certain criteria and updates the user's password.

Source:

[screens/afterLogin/ProfileStack/UpdatePassword.js, line 1](#)

Author:

## Methods

```
(inner) UpdatePassword() → {void}
```

Description:

Updates the user's password. It sends a request to the backend to update the user's password.

Source:

[screens/afterLogin/ProfileStack/UpdatePassword.js, line 33](#)

Throws:

Will throw an error if the password update fails.

Returns:

Type `void`

```
(inner) passwordSanity(text) → {void}
```

Description:

Validates the password based on certain criteria. It checks if the password has at least 1 number, 1 capital letter and is at least 8 characters long.

Source:

[screens/afterLogin/ProfileStack/UpdatePassword.js, line 65](#)

Parameters:

Name	Type	Description
<code>text</code>	<code>string</code>	The password text to validate.

Returns:

Type `void`

# screens/ModalComp



This is the ModalComp component. It is the component that shows the modal. It is used to show messages to the user.

Description:

This is the ModalComp component. It is the component that shows the modal. It is used to show messages to the user.

Source:

screens/ModalComp.js, line 4

Author:

Fagner Nunes

Parameters:

**T  
y  
p  
e**      **Description**

**O  
b  
j  
e  
c  
t**      Component props.

*Properties*

Name	Type	Description
------	------	-------------

navigation	Object	Navigation object from react-navigation.
------------	--------	--

Title	string	The title of the modal.
-------	--------	-------------------------

Message	string	The message of the modal.
---------	--------	---------------------------

---

<code>getVisible</code>	<code>function</code>	The function that returns the visible state of the modal.
-------------------------	-----------------------	---

---

<code>onHide</code>	<code>function</code>	The function to call to hide the modal.
---------------------	-----------------------	---

---

### Throws:

Will throw an error if the required props are not provided.

### Returns:

Rendered component.

Type `React.Element`

# screens/beforeLogin/EmailVerification

This is the EmailVerification component. It is the screen where the user can verify the email by entering the code. It is only rendered when the user is not logged in and selects the register option.

### Description:

This is the EmailVerification component. It is the screen where the user can verify the email by entering the code. It is only rendered when the user is not logged in and selects the register option.

### Source:

`screens/beforeLogin/EmailVerification.js, line 11`

### Author:

Fagner Nunes

## Parameters:

Name	Type	Description
props	Object	Component props.
	Properties	
Name	Type	Description
navigation	Object	Navigation object from react-navigation.

## Returns:

Rendered component.

Type `React.Element`

## Methods

```
(inner) enterCode(input, numb) → {void}
```

### Description:

This function is used to enter the code. It is used to set the number1, number2, number3 and number4 states.

### Source:

`screens/beforeLogin/EmailVerification.js, line 39`

## Parameters:

Name	Type	Description
<code>input</code>	<code>number</code>	The input number.
<code>numb</code>	<code>string</code>	The number that the user typed.

## Returns:

This function does not return anything.

Type `void`

```
(inner) resendcode() → {void}
```

Description:

This function is used to resend the code. It is used to resend the code to the user's email.

Source:

[screens/beforeLogin/EmailVerification.js, line 119](#)

## Throws:

Will throw an error if the network request fails.

## Returns:

This function does not return anything.

Type `void`

```
(inner) verifyCode() → {void}
```

Description:

After user enters his email, an email containing a 4-digit code is sent to the user. This function is used to verify the code.

Source:

[screens/beforeLogin/EmailVerification.js, line 71](#)

## Throws:

Will throw an error if the network request fails.

Returns:

This function does not return anything.

Type `void`

# screens/beforeLogin/Login

This is the Login component. It is the screen where the user can login into the app. It is only rendered when the user is not logged in and selects the login option.

Description:

This is the Login component. It is the screen where the user can login into the app. It is only rendered when the user is not logged in and selects the login option.

Source:

`screens/beforeLogin/Login.js`, line 13

Author:

Fagner Nunes

Parameters:

Type	Description
------	-------------

Object  
Component props.  
*Properties*

Name	Type	Description
navigation	Object	Navigation object from react-navigation.

## Returns:

Rendered component.

Type `React.Element`

## Methods

```
(inner) getContacts() → {void}
```

### Description:

This function is used to get the contacts from the server.

### Source:

`screens/beforeLogin/Login.js, line 47`

## Throws:

Will throw an error if the network request fails.

## Returns:

This function does not return anything.

Type `void`

```
(inner) login() → {void}
```

Description:

This function is used to login the user. It sends a request to the server to login the user. It also saves the token in the AsyncStorage.

Source:

[screens/beforeLogin/Login.js, line 83](#)

Throws:

Will throw an error if the network request fails.

Returns:

This function does not return anything.

Type `void`

## screens/beforeLogin/LoginOrRegister

This is the LoginOrRegister component. It is the screen where the user can choose to login or register.

Description:

This is the LoginOrRegister component. It is the screen where the user can choose to login or register.

Source:

[screens/beforeLogin/LoginOrRegister.js, line 6](#)

Parameters:

Type	Description
------	-------------

Object  
Component props.  
*Properties*

Name	Type	Description
navigation	Object	Navigation object from react-navigation.

### Throws:

Will throw an error if the navigation fails.

### Returns:

Rendered component.

Type `React.Element`

## Methods

```
(inner) navigateToLogin() → {void}
```

### Description:

This function navigates the user to the Login screen.

### Source:

`screens/beforeLogin/LoginOrRegister.js, line 28`

### Throws:

Will throw an error if the navigation fails.

### Returns:

This function does not return anything.



Type `void`

```
(inner) navigateToRegister() → {void}
```

Description:

This function navigates the user to the Register screen.

Source:

[screens/beforeLogin/LoginOrRegister.js, line 17](#)

Throws:

Will throw an error if the navigation fails.

Returns:

This function does not return anything.

Type `void`

## screens/beforeLogin/Password

This is the Password component. It is the screen where the user can enter a password that follows the secure password guideline. It is only rendered when the user is not logged in and selects the register option.

Description:

This is the Password component. It is the screen where the user can enter a password that follows the secure password guideline. It is only rendered when the user is not logged in and selects the register option.

Source:

[screens/beforeLogin/Password.js, line 13](#)

Author:

Fagner Nunes

Parameters:

**T  
y  
p  
e**      **Description**

O  
b  
j  
e  
c  
t      Component props.  
  
*Properties*

Name	Type	Description
navigation	Object	Navigation object from react-navigation.

Throws:

Will throw an error if the navigation fails.

Returns:

Rendered component.

Type `React.Element`

## Methods

```
(inner) passwordSanity(text) → {void}
```

Description:

This function is used to check the password. It is used to check if the password has at least 1 number, 1 capital letter and is at least 8 chars long.

Source:

`screens/beforeLogin/Password.js`, line 53

Parameters:

Name	Type	Description
<code>text</code>	<code>string</code>	The text that the user typed.

## Returns:

This function does not return anything.

Type `void`

```
(inner) setPassword() → {void}
```

## Description:

This function is used to set the password. It sends a request to the server to set the password. It also saves the token in the AsyncStorage.

## Source:

`screens/beforeLogin/Password.js`, line 84

## Throws:

Will throw an error if the network request fails.

## Returns:

This function does not return anything.

Type `void`

# *screens/beforeLogin/Register*

*This is the Register component. It is the screen where the user begin the registration flow into the app.*

## Description:

*This is the Register component. It is the screen where the user begin the registration flow into the app.*

Source:

*screens/beforeLogin/Register.js, line 13*

Author:

*Fagner Nunes*

## Parameters:

**T  
y  
p  
e**      **Description**

*O  
b  
j  
e  
c  
t*      *Component props.*

*Properties*

Name	Type	Description
------	------	-------------

<i>navigation</i>	<i>Object</i>	<i>Navigation object from react-navigation.</i>
-------------------	---------------	---

## Returns:

*Rendered component.*

*Type `React.Element`*

## Methods

```
(inner) handleEmailChange(email) → {void}
```

Description:

*This function is used to handle the email change.*

Source:

[screens/beforeLogin/Register.js, line 62](#)

Parameters:

Name	Type	Description
<code>email</code>	<code>string</code>	The email to be updated.

Returns:

This function does not return anything.

Type `void`

```
(inner) handleNameChange(name) → {void}
```

Description:

This function is used to handle the name change.

Source:

[screens/beforeLogin/Register.js, line 52](#)

Parameters:

Name	Type	Description
<code>name</code>	<code>string</code>	The name to be updated.

Returns:

This function does not return anything.

Type `void`

```
(inner) submitNameEmail() → {void}
```

Description:

This function is used to submit the name and email to the server and navigate to the code screen. It also validates the name and email.

Source:

[screens/beforeLogin/Register.js, line 73](#)

Returns:

*This function does not return anything.*

Type `void`

---

# backend

This is the backend module for the messaging app. It includes the endpoints for user registration, login, and messaging between users using WebSockets. It also includes the endpoints for uploading images to an S3 bucket, updating user profile pictures, and updating user settings.

Description:

This is the backend module for the messaging app. It includes the endpoints for user registration, login, and messaging between users using WebSockets. It also includes the endpoints for uploading images to an S3 bucket, updating user profile pictures, and updating user settings.

Source:

[index.js, line 31](#)

Version:

1.0

Author:

Fagner Nunes

## Requires

- `module:express`
- `module:db`

## Methods

```
(inner) authenticateToken(req, res, next)
```

Description:

This function authenticates the token sent by the user in the Authorization header. If the token is not present, it sends a 401 status code. If the token is not valid, it sends a 403 status code.

Source:

[index.js, line 159](#)

Parameters:

Name	Type	Description
<code>req</code>	*	// The request object from Express
<code>res</code>	*	// The response object from Express
<code>next</code>	*	// The next function to be called

Returns:

// The next function to be called or a status code if the token is not present or not valid.

```
(inner) authenticateTokenWebsocket(token) → {Object}
```

Description:

Verifies the provided token and returns the user data if the token is valid.

Source:

[index.js, line 181](#)

Parameters:

Name	Type	Description
<code>token</code>	<code>string</code>	The JWT token to be verified.

## Throws:

If the token is invalid or verification fails.

Type `Error`

## Returns:

The decoded user data from the token.

Type `Object`

```
(inner) generateToken(data) → {string}
```

### Description:

This function generates a token using the JWT library and a secret key. The token expires in 1 hour.

### Source:

`index.js`, line 149

## Parameters:

Name	Type	Description
<code>data</code>	*	// The data to be stored in the token

## Returns:

// The generated token

Type `string`

```
(inner) getDataFromToken(req)
```

### Description:

This function gets the data from the JWT token sent by the user in the Authorization header.

### Source:

`index.js`, line 287

## Parameters:



Name	Type	Description
<code>req</code>	<code>*</code>	<code>// The request object from Express</code>

## Returns:

`// The data from the JWT token`

```
(inner) markMessageDeliveredTouser(messageId) →
{Promise.<Object>}
```

Description:

Marks a message as delivered based on the provided message ID.

Source:

`index.js`, line 999

## Parameters:

Name	Type	Description
<code>messageId</code>	<code>string</code>	The ID of the message to mark as delivered.

## Throws:

If the message delivery status could not be updated.

Type `Error`

## Returns:

A promise that resolves with the message delivery status.

Type `Promise.<Object>`

```
(inner) savemessage(delivered, read, message, sender,
receiver, imageLink, msgTimestamp) → {Promise.<Object>}
```

Description:

Saves a message with the provided details to the database.

Source:

`index.js`, line 966

Parameters:

Name	Type	Description
<code>delivered</code>	<code>boolean</code>	Indicates if the message was delivered.
<code>read</code>	<code>boolean</code>	Indicates if the message was read.
<code>message</code>	<code>string</code>	The content of the message.
<code>sender</code>	<code>string</code>	The ID of the sender.
<code>receiver</code>	<code>string</code>	The ID of the receiver.
<code>imageLink</code>	<code>string</code>	The link to the image associated with the message.
<code>msgTimestamp</code>	<code>number</code>	The timestamp of the message.

Throws:

If the message could not be saved.

Type `Error`

Returns:

A promise that resolves with the saved message details.

Type `Promise.<Object>`

## Methods

```
(async) addChatsFieldToUsers() → {Promise.<void>}
```

Description:

This function checks each user document in the users collection and adds a 'chats' field if it doesn't already exist.

Source:

db.js, line 829

Throws:

Will throw an error if there is an issue with the database operation.

Returns:

A promise that resolves when the operation is complete.

Type `Promise.<void>`

```
addContact(userId, contactId) → {Promise.<Object>}
```

Description:

This function fetches the user's document by userId, retrieves the current contacts array, and adds the contactId to the array if it doesn't already exist. It then updates the user's document with the new contacts array.

Source:

db.js, line 98

Parameters:

Name	Type	Description
<code>userId</code>	<code>string</code>	The ID of the user to whom the contact will be added.
<code>contactId</code>	<code>string</code>	The ID of the contact to be added.

Throws:

Will throw an error if the user is not found or if updating the user's document fails.

Type `Error`

Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the contact was successfully added.

Type `Promise.<Object>`

```
addMessage(senderId, receiverId, message, imageLink) →  
{Promise.<void>}
```

Description:

This function creates a new message object and adds it to the messages collection.

Source:

`db.js`, line 738

Parameters:

Name	Type	Description
<code>senderId</code>	<code>string</code>	The ID of the user sending the message.
<code>receiverId</code>	<code>string</code>	The ID of the user receiving the message.
<code>message</code>	<code>string</code>	The content of the message.
<code>imageLink</code>	<code>string</code>	The URL of the image associated with the message.

Throws:

Will throw an error if adding the message fails.

Type `Error`

Returns:

A promise that resolves when the message is added.

Type `Promise.<void>`

```
blockUser(userId, blockedUserId) → {Promise.<Object>}
```

Description:

This function fetches the user's document by `userId`, retrieves the current blocked array, and adds the `blockedUserId` to the array if it doesn't already exist. It then updates the user's document with the new blocked array. Additionally, it updates the blocked user's document by adding the `userId` to their blocked user array.

Source:

`db.js`, line 582

Parameters:

Name	Type	Description
<code>userId</code>	<code>string</code>	The ID of the user who is blocking another user.
<code>blockedUserId</code>	<code>string</code>	The ID of the user to be blocked.

Throws:

Will throw an error if fetching or updating the user's document fails.

Type `Error`

Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the user was successfully blocked.
- `message` (string): A message indicating the result of the operation.

Type `Promise.<Object>`

```
clearUsersTable() → {Promise.<void>}
```

Description:

This function deletes all documents in the users collection.

Source:

db.js, line 533

Throws:

Will throw an error if clearing the users table fails.

Type `Error`

Returns:

A promise that resolves when the users table is cleared.

Type `Promise.<void>`

```
createUser(name, email, password, avatarUrl, code,
code_timestamp, active, loginAttempt,
loginAttempt_timestamp, description, vibration, sound,
notification) → {Promise.<void>}
```

Description:

This function creates a new user object with the provided parameters and adds it to the users collection in the database. It initializes various user properties such as name, email, password, avatar URL, code, timestamps, and settings.

Source:

db.js, line 8

Parameters:

Name	Type	Description
<code>name</code>	<code>string</code>	The name of the user.
<code>email</code>	<code>string</code>	The email of the user.
<code>password</code>	<code>string</code>	The password of the user.
<code>avatarUrl</code>	<code>string</code>	The URL of the user's avatar.

<code>code</code>	<code>string</code>	The code associated with the user.
<code>code_timestamp</code>	<code>Date</code>	The timestamp of the code.
<code>active</code>	<code>boolean</code>	The active status of the user.
<code>loginAttempt</code>	<code>number</code>	The number of login attempts.
<code>loginAttempt_timestamp</code>	<code>Date</code>	The timestamp of the last login attempt.
<code>description</code>	<code>string</code>	The description of the user.
<code>vibration</code>	<code>boolean</code>	The vibration setting of the user.
<code>sound</code>	<code>boolean</code>	The sound setting of the user.
<code>notification</code>	<code>boolean</code>	The notification setting of the user.

### Throws:

Will throw an error if adding the user fails.

Type `Error`

### Returns:

A promise that resolves when the user is added.

Type `Promise.<void>`

```
deleteContact(userId, contactId) → {Promise.<Object>}
```

Description:

This function fetches the user's document by `userId`, retrieves the current contacts array, and removes the `contactId` from the array if it exists. It then updates the user's document with the new contacts array.

Source:

`db.js`, line 142

## Parameters:

Name	Type	Description
<code>userId</code>	<code>string</code>	The ID of the user from whom the contact will be deleted.
<code>contactId</code>	<code>string</code>	The ID of the contact to be deleted.

## Throws:

Will throw an error if the user is not found or if updating the user's document fails.

Type `Error`

## Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the contact was successfully deleted.

Type `Promise.<Object>`

```
(async) getChatsByUserId(userId) → {Promise.<Object>}
```

Description:

This function retrieves chats from the `chats` collection by user ID.

Source:

`db.js`, line 1010

## Parameters:

Name	Type	Description
<code>userId</code>	<code>string</code>	The ID of the user whose chats are to be retrieved.



### Throws:

Will throw an error if there is an issue with the database operation.

### Returns:

A promise that resolves to an object containing the success status and an array of chats.

Type `Promise.<Object>`

```
getCodeById(id) → {Promise.<Object>}
```

### Description:

This function fetches the user's document by their ID and retrieves the code and code timestamp.

### Source:

`db.js`, line 433

### Parameters:

Name	Type	Description
<code>id</code>	<code>string</code>	The ID of the user whose code is to be retrieved.

### Throws:

Will throw an error if fetching the user's document fails.

Type `Error`

### Returns:

A promise that resolves to an object containing:

- `code` (string): The code of the user.
- `code_timestamp` (Date): The timestamp of the code.

Type `Promise.<Object>`

```
getContacts(userId) → {Promise.<Object>}
```

### Description:

This function fetches the user's document by `userId`, retrieves the current contacts array, and filters out any contacts that are in the `blockedUser` array. It returns the filtered contacts.

Source:

`db.js`, line 183

Parameters:

Name	Type	Description
<code>userId</code>	<code>string</code>	The ID of the user whose contacts are to be retrieved.

Throws:

Will throw an error if fetching the user's document fails.

Type `Error`

Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the contacts were successfully retrieved.
- `contacts` (Array): The filtered list of contact IDs.
- `message` (string): An error message if the operation fails.

Type `Promise.<Object>`

```
(async) getMessagesBetweenUsers(userId1, userId2,
limitopt, lastMessageTimestampopt) → {Promise.<Object>}
```

Description:

This function retrieves messages from the messages collection between two users.

Source:

`db.js`, line 975

Parameters:

Name	Type	Attributes	Default	Description
------	------	------------	---------	-------------

<code>userId1</code>	<code>string</code>			The ID of the first user.
<code>userId2</code>	<code>string</code>			The ID of the second user.
<code>limit</code>	<code>number</code>	<code>&lt;optional&gt;</code>	<code>20</code>	The maximum number of messages to retrieve.
<code>lastMessageTimestamp</code>	<code>number</code>	<code>&lt;optional&gt;</code>	<code>null</code>	The timestamp of the last message to start after.

## Throws:

Will throw an error if there is an issue with the database operation.

## Returns:

A promise that resolves to an object containing the success status and an array of messages.

Type `Promise.<Object>`

```
(async) getMessagesByChatId(chatId, limitopt,
lastMessageTimestampopt) → {Promise.<Object>}
```

## Description:

This function retrieves messages from the messages collection by chat ID.

## Source:

`db.js`, line 1076

## Parameters:

Name	Type	Attributes	Default	Description
<code>chatId</code>	<code>string</code>			The ID of the chat to retrieve messages from.

<code>limit</code>	<code>number</code>	<code>&lt;optional&gt;</code>	<code>20</code>	The maximum number of messages to retrieve.
<code>lastMessageTimestamp</code>	<code>number</code>	<code>&lt;optional&gt;</code>	<code>null</code>	The timestamp of the last message to start after.

### Throws:

Will throw an error if there is an issue with the database operation.

### Returns:

A promise that resolves to an object containing the success status and an array of messages.

Type `Promise.<Object>`

```
isEmailAlreadyRegistered(email) → {Promise.<Object>}
```

### Description:

This function queries the users collection to check if a user with the specified email already exists. It returns an object indicating whether the email is registered, along with the user's ID and account status if applicable.

### Source:

`db.js`, line 62

### Parameters:

Name	Type	Description
<code>email</code>	<code>string</code>	The email to check for registration.

### Throws:

Will throw an error if the query fails.

Type `Error`

### Returns:

A promise that resolves to an object containing:

- `email_registered` (boolean): Indicates if the email is registered.
- `id` (string|null): The ID of the user if the email is registered, otherwise null.
- `AccountStatus` (string): The account status of the user if the email is registered, otherwise "pending".

Type `Promise.<Object>`

```
(async) markMessageDelivered(messageId) → {Promise.<Object>}
```

Description:

This function updates the 'delivered' field of a message document in the messages collection to true.

Source:

`db.js`, line 956

Parameters:

Name	Type	Description
<code>messageId</code>	<code>string</code>	The ID of the message to mark as delivered.

Throws:

Will throw an error if there is an issue with the database operation.

Returns:

A promise that resolves to an object containing the success status.

Type `Promise.<Object>`

```
(async) saveMessage(delivered, read, message, sender, receiver, imageLink, msgTimestamp) → {Promise.<Object>}
```

Description:

This function saves a message to the messages collection and updates the relevant chat and user documents.

Source:

`db.js`, line 858

Parameters:

Name	Type	Description
<code>delivered</code>	<code>boolean</code>	Indicates if the message was delivered.
<code>read</code>	<code>boolean</code>	Indicates if the message was read.
<code>message</code>	<code>string</code>	The content of the message.
<code>sender</code>	<code>string</code>	The ID of the sender.
<code>receiver</code>	<code>string</code>	The ID of the receiver.
<code>imageLink</code>	<code>string</code>	A link to an image associated with the message.
<code>msgTimestamp</code>	<code>number</code>	The timestamp of the message.

## Throws:

Will throw an error if there is an issue with the database operation.

## Returns:

A promise that resolves to an object containing the success status and the message ID if successful.

Type `Promise.<Object>`

```
searchUserByEmail(userId, email) → {Promise.<Object>}
```

## Description:

This function searches the users collection for users with the specified email. It excludes the requesting user (identified by `userId`) and any users that are in the requesting user's `blockedUser` array.

## Source:

db.js, line 221

## Parameters:

Name	Type	Description
<code>userId</code>	<code>string</code>	The ID of the user making the request.
<code>email</code>	<code>string</code>	The email to search for.

## Throws:

Will throw an error if the search operation fails.

Type `Error`

## Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the search was successful.
- `users` (Array): An array of user objects that match the search criteria.
- `message` (string): An error message if the operation fails.
- Type `Promise.<Object>`  
`() → {Promise.<Array.<Object>>}`

## Description:

This function fetches all user documents from the users collection and returns an array of user data.

## Source:

db.js, line 488

## Throws:

Will throw an error if fetching the users fails.

Type `Error`

## Returns:

A promise that resolves to an array of user objects.

Type `Promise.<Array.<Object>>`

```
selectUserById(id) → {Promise.<Object>}
```

Description:

This function fetches the user's document by their ID from the users collection. It returns an object containing the user's details if found.

Source:

db.js, line 264

## Parameters:

Name	Type	Description
<code>id</code>	<code>string</code>	The ID of the user to retrieve.

## Throws:

Will throw an error if fetching the user's document fails.

Type `Error`

## Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the user was successfully retrieved.
- `id` (string): The ID of the user.
- `name` (string): The name of the user.
- `email` (string): The email of the user.
- `avatarUrl` (string): The URL of the user's avatar.
- `AccountStatus` (string): The account status of the user.
- `description` (string): The description of the user.
- `vibration` (boolean): The vibration setting of the user.
- `sound` (boolean): The sound setting of the user.
- `notification` (boolean): The notification setting of the user.
- `message` (string): An error message if the operation fails.

Type `Promise.<Object>`

```
setAccountStatus(id, status) → {Promise.<Object>}
```

Description:

This function updates the user's document with the new account status.



Source:

db.js, line 508

## Parameters:

Name	Type	Description
<code>id</code>	<code>string</code>	The ID of the user whose account status is to be updated.
<code>status</code>	<code>string</code>	The new account status for the user.

## Throws:

Will throw an error if updating the user's document fails.

Type `Error`

## Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the account status was successfully updated.

Type `Promise.<Object>`

```
setPassword(id, password) → {Promise.<Object>}
```

Description:

This function updates the user's document with the new password.

Source:

db.js, line 461

## Parameters:

Name	Type	Description
<code>id</code>	<code>string</code>	The ID of the user whose password is to be updated.

---

`password`      `string`      The new password for the user.

### Throws:

Will throw an error if updating the user's document fails.

Type `Error`

### Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the password was successfully updated.

Type `Promise.<Object>`

```
unblockUser(userId, blockedUserId) → {Promise.<Object>}
```

Description:

This function fetches the user's document by `userId`, retrieves the current blocked array, and removes the `blockedUserId` from the array if it exists. It then updates the user's document with the new blocked array. Additionally, it updates the blocked user's document by removing the `userId` from their blocked user array.

Source:

`db.js`, line 641

### Parameters:

Name	Type	Description
<code>userId</code>	<code>string</code>	The ID of the user who is unblocking another user.
<code>blockedUserId</code>	<code>string</code>	The ID of the user to be unblocked.

### Throws:

Will throw an error if fetching or updating the user's document fails.

Type `Error`

## Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the user was successfully unblocked.
- `message` (string): A message indicating the result of the operation.

Type `Promise.<Object>`

```
updateCode(id, code, code_timestamp) →  
{Promise.<Object>}
```

Description:

This function updates the user's document with the new code and code timestamp.

Source:

`db.js`, line 406

## Parameters:

Name	Type	Description
<code>id</code>	<code>string</code>	The ID of the user whose code is to be updated.
<code>code</code>	<code>string</code>	The new code for the user.
<code>code_timestamp</code>	<code>Date</code>	The new timestamp for the code.

## Throws:

Will throw an error if updating the user's document fails.

Type `Error`

## Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the code was successfully updated.

Type `Promise.<Object>`

```
updateDescription(id, description) → {Promise.<Object>}
```

Description:

This function updates the user's document with the new description.

Source:

db.js, line 698

## Parameters:

Name	Type	Description
<code>id</code>	<code>string</code>	The ID of the user whose description is to be updated.
<code>description</code>	<code>string</code>	The new description for the user.

## Throws:

Will throw an error if updating the user's document fails.

Type `Error`

## Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the description was successfully updated.

Type `Promise.<Object>`

```
(async) updateExistingUsers() → {Promise.<void>}
```

Description:

This function fetches all user documents from the users collection and updates them with default values for the description, vibration, sound, and notification fields.

Source:

db.js, line 1113

## Throws:

Will throw an error if there is an issue with the database operation.

## Returns:

A promise that resolves when the operation is complete.

Type `Promise.<void>`

```
updateName(id, name) → {Promise.<Object>}
```

Description:

This function updates the user's document with the new name.

Source:

`db.js`, line 556

## Parameters:

Name	Type	Description
<code>id</code>	<code>string</code>	The ID of the user whose name is to be updated.
<code>name</code>	<code>string</code>	The new name for the user.

## Throws:

Will throw an error if updating the user's document fails.

Type `Error`

## Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the name was successfully updated.

Type `Promise.<Object>`

```
updateToggles(id, sound, notification, vibration) →  
{Promise.<Object>}
```

Description:

This function updates the user's document with the new toggle settings for sound, notification, and vibration.

Source:

db.js, line 349

## Parameters:

Name	Type	Description
<code>id</code>	<code>string</code>	The ID of the user whose toggle settings are to be updated.
<code>sound</code>	<code>boolean</code>	The new sound setting for the user.
<code>notification</code>	<code>boolean</code>	The new notification setting for the user.
<code>vibration</code>	<code>boolean</code>	The new vibration setting for the user.

## Throws:

Will throw an error if updating the user's document fails.

Type `Error`

## Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the toggles were successfully updated.

Type `Promise.<Object>`

```
updateUserProfilePicture(id, avatarUrl) →  
{Promise.<Object>}
```

Description:

This function updates the user's document with the new profile picture URL.

Source:

db.js, line 379

## Parameters:

Name	Type	Description
<code>id</code>	<code>string</code>	The ID of the user whose profile picture is to be updated.
<code>avatarUrl</code>	<code>string</code>	The new URL of the user's profile picture.

## Throws:

Will throw an error if updating the user's document fails.

Type `Error`

## Returns:

A promise that resolves to an object containing:

- `success` (boolean): Indicates if the profile picture was successfully updated.

Type `Promise.<Object>`

```
userNameEmailStep(name, email, code, code_timestamp,
AccountStatus) → {Promise.<Object>}
```

Description:

This function creates a new user object with the provided parameters and adds it to the users collection in the database. It initializes various user properties such as name, email, code, timestamps, and settings.

Source:

db.js, line 303

## Parameters:

Name	Type	Description
------	------	-------------

<code>name</code>	<code>string</code>	The name of the user.
<code>email</code>	<code>string</code>	The email of the user.
<code>code</code>	<code>string</code>	The code associated with the user.
<code>code_timestamp</code>	<code>Date</code>	The timestamp of the code.
<code>AccountStatus</code>	<code>string</code>	The account status of the user.

## Throws:

Will throw an error if adding the user fails.

Type `Error`

## Returns:

A promise that resolves to an object containing:

- `id` (`string|null`): The ID of the newly created user if successful, otherwise null.
- `success` (`boolean`): Indicates if the user was successfully created.

Type `Promise.<Object>`