CT124-3-2-MAE

MOBILE APP ENGINEERING

GROUP ASSIGNMENT

PROJECT REPORT

HAND OUT DATE: WEEK 3

HAND IN DATE: 13 November 2024

WEIGHTAGE: 75%

GROUP: Group 12

Name	TP Number
CHING JIA ZHONG	TP074569
THAM JIE WEI	TP074224
MOHAMMED FADHI PUZHANGARAYILLATH SAFEER	TP073289

Table of Contents

1.0 Introduction

The Elderly Medication and Health Tracker app, "Jaga4U," is designed to support elderly individuals, caregivers, and healthcare providers in managing medication adherence, monitoring health data, and enhancing communication. This app addresses the challenges often faced by elderly individuals in remembering medication schedules and by caregivers in ensuring consistent health monitoring. It is structured to provide key functionalities for each user role such as medication reminders, health data logging, monitoring tools, and real-time alerts. The functionalities are delivered through a user-friendly interface.

Developed using Flutter, Firebase Cloud Firestore, Firebase Cloud Messaging, Firebase Authentication, and a structured multi-user interface, Jaga4U seeks to simplify health management for all stakeholders. This report documents the development and design process for Jaga4U, covering problem-solving strategies, system architecture, CRUD operations, and user permission protocols. It includes key diagrams, such as the use case and system architecture, to illustrate the app’s structure and data flow. The report also details the app’s functionalities, validation processes, and dynamic data management for seamless, responsive user interaction. Additionally, a user manual and automated system testing results are provided to ensure the app's reliability and ease of use. The conclusion summarizes project outcomes and highlights areas for future improvement.

2.0 Problem Solving and Design

2.1 Functionalities

Universal Functionalities

- Users view their profile and make changes to their details.
- Users can change setting details such as vibrations, dark mode, light mode and so on.
- Users can click on sign out button to log out.
- Users can register new account or log in.
- Users can request for reset password email if they forgot their password.

Caregivers

- Caregivers can add elderly user in their list.
- Caregivers can select associated elderly user and view the elderly's medication schedule with details and can schedule a new medication schedule for them.
- Caregivers can select their associated elderly user and view the health metrics of the elderly user such as blood pressure and pulse rate over time
- Caregivers can add healthcare providers to their list, select on which healthcare providers they want to chat with and chat with them.
- Caregivers can receive notifications and see a list of alerts notifications where elderly users missed their scheduled dose for 5 minutes or when elderly user logs critical health data.
- Caregivers can receive notification of new chat message from healthcare provider if they are not active in the app at that moment.
- Caregivers can select associated elderly user and view the elderly's location.
- Caregivers can delete associated elderly users or healthcare providers, delete medication schedule and delete alert notifications.

Elderly Users

- Elderly users can log their blood pressure and pulse rate data over time and view the health metrics of themselves over time.
- Elderly users will receive reminders as notification when they miss medication time.
- Elderly users will receive reminders as notification when their medication schedule time is reached.
- Elderly users can press on a big red button to emergency call their caregiver.
- Elderly users can share their location so that their caregivers can track their location.
- Elderly users can make appointment to all healthcare providers.

Healthcare Providers

- Healthcare providers can add elderly users to their list.
- Healthcare providers can select associated elderly user to view their blood pressure and pulse rate data over time.

- Healthcare providers can update the medication schedule of elderly users or change the medication type.
- Healthcare providers can add caregivers to their list and chat message with the list of caregivers in their list.
- Healthcare providers can also receive missed dose alert and notification when elderly users didn't confirm that they took their medication within 5 minutes after reminder.
- Healthcare providers can view the appointments that their associated elderly users made and update the status of the appointment.
- Healthcare providers can receive notification when elderly users log critical health data.

2.2 Use Case Diagram

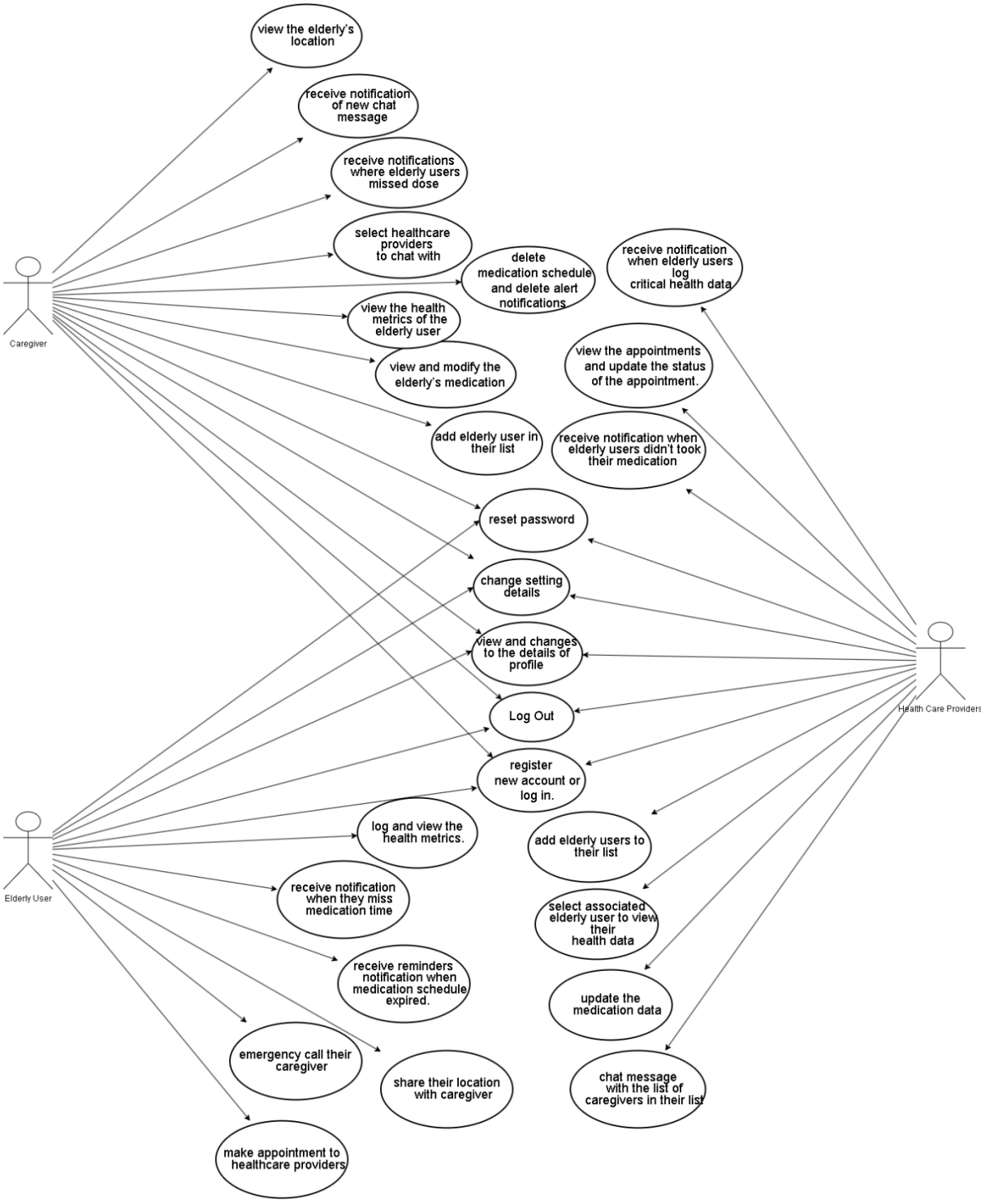


Figure 2.2.1: Use Case Diagram of Jaga4U

2.3 System Architecture Design

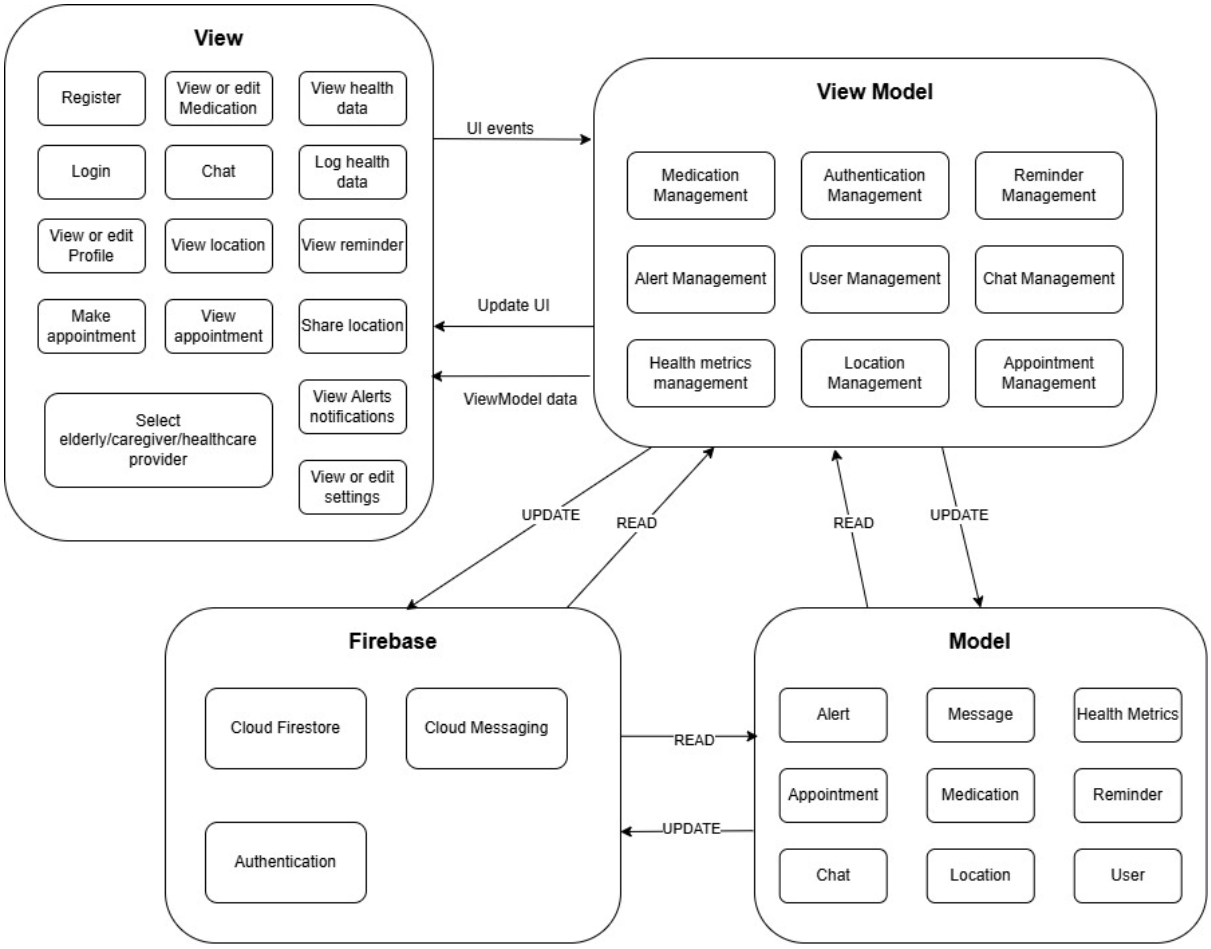


Figure 2.3.1: Model-View-ViewModel of JAGA4U.

MVVM (Model-View-ViewModel) is a software architectural pattern. The main goal of this architecture is to make the view completely independent from the application logic (Gallardo, 2023). The View represents the UI layer that users interact with, containing various screens like Register, Login, Chat, View Profile and so on. This layer sends UI events which are user interactions to the ViewModel and receives data from ViewModel to update the UI. Each view corresponds to a specific functionality, such as viewing medication, health data, location, appointments, and notifications, aligning with the needs of different user roles such as elderly, caregiver, healthcare provider.

The ViewModel serves as an intermediary between the View and the Model, managing the app's logic and handling data transformations. It includes various management modules such as Medication Management, Authentication Management, Alert Management and so on to handle specific aspects of the app, such as user authentication, medication schedules, reminders, and alerts. The ViewModel reads data from Firebase or the Model and updates the UI as needed.

It also sends update commands to Firebase or Model for actions like saving health metrics or appointments or saving new user.

Firebase serves as our backend service, handling data storage using Cloud Firestore, authentication using Firebase Authentication, and messaging using Cloud Messaging. Firebase interacts directly with the ViewModel to provide real-time data and updates, enabling functions like user authentication and message notifications. Data is either read from or updated in Firebase, depending on user actions such as updating user profiles and receiving new messages.

2.4 Dynamic Data Management

Dynamic data management refers to handling data that can change in real time or based on user interactions within the app. This includes fetching, updating, and displaying data dynamically, often from sources like databases or APIs. Some common approaches managing dynamic data in Flutter include state management, streams and streambuilder, futurebuilder, firebase integration, local data management and changenotifier. They ensure that the app's UI responds immediately to data changes, providing a seamless user experience.

```
runApp(
  MultiProvider(
    providers: [
      ChangeNotifierProvider(create: (_) => AuthProvider()),
      ChangeNotifierProvider(
          create: (_) => ReminderHandler(flutterLocalNotificationsPlugin)), // ChangeNotifierProvider
      ChangeNotifierProvider(create: (_) => LocationService()),
      ChangeNotifierProvider(create: (_) => ThemeProvider()),
      ChangeNotifierProvider(
          create: (context) => SelectElderlyPageProvider()), // ChangeNotifierProvider
      ChangeNotifierProvider(create: (_) => HealthMetricsProvider()),
      ChangeNotifierProvider(create: (context) => MedicationProvider()),
      ChangeNotifierProvider(
          create: (context) => SelectHealthcareProviderPageProvider()), // ChangeNotifierProvider
      ChangeNotifierProvider(create: (context) => HealthDataProvider()),
      ChangeNotifierProvider(create: (_) => CaregiverAlertProvider()),
      ChangeNotifierProvider(create: (_) => ProviderAlertProvider()),
      ChangeNotifierProvider(
          create: (context) => SelectElderlyPatientPageProvider()), // ChangeNotifierProvider
      ChangeNotifierProvider(
          create: (_) =>
              ReminderNotificationHandler(flutterLocalNotificationsPlugin)), // ChangeNotifierProvider
      ChangeNotifierProvider(create: (context) => PatientDataProvider()),
      ChangeNotifierProvider(
          create: (context) => SelectCaregiverProviderPageProvider()), // ChangeNotifierProvider
      ChangeNotifierProvider(
          create: (context) => AssociatedMedicationProvider()), // ChangeNotifierProvider
      ChangeNotifierProvider(create: (_) => AppointmentRepository()),
      ChangeNotifierProvider(
          create: (context) => AppointmentSchedulerLogic(
              Provider.of<AppointmentRepository>(context, listen: false))), // AppointmentSchedulerLogic // ChangeNotifierProvider
    ],
    child: MyApp(flutterLocalNotificationsPlugin),
  ), // MultiProvider
```

Figure 2.4.1: Code snippets in main.dart

This code demonstrates dynamic data management through dependency injection and state management. MultiProvider is used to register multiple providers, each of which manages a specific piece of app state or functionality. By injecting these providers at the top level of the app, they can be accessed throughout the widget tree without directly passing instances, making data management dynamic and centralized. Each provider extends ChangeNotifier, allowing them to notify listeners when there is a change in data. When notifyListeners() is called within any of these providers, it triggers a rebuild of any widgets that are listening, ensuring that the UI reflects the latest data dynamically. Several examples of dynamic data management will be shown.

2.4.1 Login

```
Future<void> signIn(BuildContext context, String username, String password) async {
  try {
    // Find user by username in Firestore
    DocumentSnapshot? userDoc = await _userRepository.getUserByUsername(username);

    String userEmail = userDoc['email'];
    String userID = userDoc.id;

    // Authenticate with Firebase using the retrieved email and entered password
    UserCredential userCredential = await _auth.signInWithEmailAndPassword(
      email: userEmail,
      password: password,
    );

    if (userCredential.user != null) {
      // After successful login or registration
      final authService = AuthService();
      await authService.saveUserToken(userID);
      authService.setupTokenRefresh(userID);

      // Get user role
      String userRole = _userRepository.getUserRole(userDoc);

      // Navigate to the correct dashboard based on role
      if (userRole == 'Caregiver') {
        Navigator.pushReplacement(
          context,
          MaterialPageRoute(
            builder: (context) => CaregiverDashboardPage(userID: userID),
          ), // MaterialPageRoute
        );
      } else if (userRole == 'Elderly') {
        Navigator.pushReplacement(
```

Figure 2.4.1.1: Sign in function in login provider

This function allows the app to fetch and verify user data in real time rather than relying on a static dataset. Once a user is logged in, the AuthService handles the FCM token management by saving the token and setting up a token refresh so that chat notification or health data alert notification can be sent to the correct user device later on. This dynamic approach ensures that user's FCM token remain valid and up to date. The role-based navigation is a dynamic way to adjust the app experience based on each user's specific role and context, supporting a personalized user experience.

2.4.2 Elderly Location

```dart
@override
void initState() {
  super.initState();
  _getElderlyLocationStream();
}

// Fetch live location updates from Firestore
void _getElderlyLocationStream() {
  FirebaseFirestore.instance
      .collection('locations')
      .doc(widget.elderlyID)
      .snapshots()
      .listen((snapshot) {
    if (snapshot.exists) {
      final data = snapshot.data()!;
      setState(() {
        elderlyLocation = LatLng(data['latitude'], data['longitude']);
        locationMessage = 'Latitude: ${data['latitude']}, Longitude: ${data['longitude']}';

        // Move the map camera to the new location
        if (mapController != null) {
          mapController!.animateCamera(
            CameraUpdate.newCameraPosition(
              CameraPosition(
                target: elderlyLocation!,
                zoom: 17, // Adjust zoom to better fit the elderly location
              ),
            ),
          );
        }
      });
    } else {
      setState(() {
        locationMessage = 'Location data not available.';
      });
    }
```

Figure 2.4.2.1: Function to get elderly location real time

This function connects to Firestore and listens to changes in the locations collection for a document with elderlyID. Whenever Firestore detects a change (such as new coordinates), the app automatically receives this update, ensuring that it always has the latest information without needing to poll or refresh manually. The data stream triggers setState() whenever a new location update is received, updating the elderlyLocation variable and the locationMessage. This, in turn, re-renders the UI with the updated coordinates, ensuring that users see the latest location in real time. This dynamic reactivity is crucial for tracking live data like location, where updates need to be instantly visible to users. If a new location update is received and a mapController is available, the code calls mapController!.animateCamera() to move the map's camera view to the elderly user's updated location.

```dart
// Get live location stream and save location to Firestore
Stream<LatLng> getLiveLocationStream({required String userID}) {
  LocationSettings locationSettings = LocationSettings(
    accuracy: LocationAccuracy.high,
    distanceFilter: 10,
  );

  return Geolocator.getPositionStream(locationSettings: locationSettings).map((Position position) {
    print("Received position: ${position.latitude}, ${position.longitude}"); // Debugging line
    _saveLocationToFirestore(userID, position.latitude, position.longitude);
    return LatLng(position.latitude, position.longitude);
  });
}

// Save the location to Firestore
Future<void> _saveLocationToFirestore(String userID, double latitude, double longitude) async {
  try {
    await FirebaseFirestore.instance.collection('locations').doc(userID).set(
      {
        'latitude': latitude,
        'longitude': longitude,
        'timestamp': DateTime.now(),
        'elderlyID': userID,
      },
      SetOptions(merge: true),
    );
    print("Location saved to Firestore");
  } catch (e) {
    print("Error saving location: $e");
  }
}
```

Figure 2.4.2.2: Function in Location provider

The getLiveLocationStream function continuously listens to the device's location updates via Geolocator.getPositionStream. It requests high-accuracy location data with a distance filter (10 meters) to only receive significant movements, reducing redundant updates and preserving resources. The stream is dynamic as it emits latitude and longitude pairs each time there is a position update, meaning that any listener subscribing to this stream will automatically receive and react to each new location update in real time. Each time a new location is received in the stream, _saveLocationToFirestore is called to save the updated location data to the firestore.

```dart
// Get the current location of the user
Future<LatLng> getCurrentLocation() async {
  Position position = await Geolocator.getCurrentPosition(desiredAccuracy: LocationAccuracy.high);
  return LatLng(position.latitude, position.longitude);
}

// Public method to update location in Firestore
Future<void> updateLocationInDatabase({required String userID, required LatLng position}) async {
  await _saveLocationToFirestore(userID, position.latitude, position.longitude);
}
```

Figure 2.4.2.3: Functions in Location Provider

getCurrentLocation retrieves the current location on demand, providing flexibility if real-time streaming is not required. The position is fetched with high accuracy and returned as a LatLng

object. updateLocationInDatabase allows for manual updates to Firestore based on the current location or an external trigger. This is useful if location data needs to be updated at specific times instead of continuously.

2.4.3 Select Elderly

```
Stream<List<Map<String, String>>> streamAssociatedElderlyUsers(
    String caregiverID) {
  return usersRef
      .doc(caregiverID)
      .collection('associatedElderlyUsers')
      .snapshots()
      .map((snapshot) {
    return snapshot.docs.map((doc) {
      return {
        'userID': doc['userID'] as String,
        'username': doc['username'] as String,
      };
    }).toList();
  });
}
```

Figure 2.4.3.1: Function of Select Elderly Screen Provider

```
final UserRepository _userRepository = UserRepository();

Stream<List<Map<String, String>>> streamElderlyUsers(String caregiverID) {
  return _userRepository.streamAssociatedElderlyUsers(caregiverID);
}
```

Figure 2.4.3.2: Function of User repository

```
Expanded(
  child: StreamBuilder<List<Map<String, String>>>(
    stream: Provider.of<SelectElderlyPageProvider>(context)
        .streamElderlyUsers(widget.caregiverID),
    builder: (context, snapshot) {
      if (snapshot.connectionState == ConnectionState.waiting) {
        return Center(child: CircularProgressIndicator());
      }
      if (snapshot.hasError) {
        return Center(child: Text("Error loading elderly users."));
      }
      final elderlyUsers = snapshot.data ?? [];
      if (elderlyUsers.isEmpty) {
        return Center(
          child: Text(
            "No elderly users added.",
            style: GoogleFonts.lato(
                color: Colors.black, fontSize: screenWidth * 0.045),
          ), // Text
        ); // Center
      }
      return ListView.builder(
        itemCount: elderlyUsers.length,
        itemBuilder: (context, index) {
          final elderly = elderlyUsers[index];
          return _buildElderlyButton(context, elderly);
        },
      ); // ListView.builder
```

Figure 2.4.3.3: Expanded widget code in the Select Elderly screen.

In the Expanded widget, a StreamBuilder listens to the streamElderlyUsers stream, which itself is a wrapper around the streamAssociatedElderlyUsers function. StreamBuilder manages real-time data from Firestore and rebuilds the UI whenever the data changes when the caregiver adds elderly to their list. When the stream emits a new list of elderly users due to changes in Firestore, StreamBuilder triggers a rebuild of the list view. This dynamic management ensures that the displayed list of elderly users is always up to date without manual data fetching or refresh operations. The ListView.builder constructs a list of buttons for each elderly user based on the real-time data from the StreamBuilder. Whenever the list of elderly users changes, ListView.builder dynamically rebuilds to show the updated list, making the UI interactive and responsive to data updates.

2.4.4 Chat

```
), // AppBar
body: Column(
  children: [
    Expanded(
      child: StreamBuilder<QuerySnapshot>(
        stream: FirebaseFirestore.instance
            .collection('chats')
            .doc(widget.chatID)
            .collection('messages')
            .orderBy('time', descending: true)
            .snapshots(),
        builder: (context, snapshot) {
          if (snapshot.connectionState == ConnectionState.waiting) {
            return Center(child: CircularProgressIndicator());
          }
          if (!snapshot.hasData || snapshot.data!.docs.isEmpty) {
            return Center(child: Text("No messages yet", style: TextStyle(color: textColor)));
          }

          // Mark messages as read whenever new messages are fetched
          _markMessagesAsRead();

          final messages = snapshot.data!.docs.map((doc) => Message.fromFirestore(doc)).toList();

          return ListView.builder(
            reverse: true,
            padding: EdgeInsets.all(16),
            itemCount: messages.length,
            itemBuilder: (context, index) {
              final message = messages[index];
              final isSender = message.senderID == widget.senderID;
```

Figure 2.4.4.1

The StreamBuilder widget listens to Firestore's messages collection for the specific chat identified by chatID. The chatID is the combination of the user ID of both sender and recipient. This dynamic streaming approach ensures that any new message sent or received in real-time is immediately reflected on the screen without needing a manual refresh.

```dart
@override
void initState() {
  super.initState();
  _markMessagesAsRead(); // Mark messages as read on opening the chat screen
  // Set a timer to mark messages as read every 10 seconds, for example
  _timer = Timer.periodic(Duration(seconds: 10), (timer) {
    _markMessagesAsRead();
  }); // Timer.periodic
}

Future<void> _markMessagesAsRead() async {
  final messagesQuery = await FirebaseFirestore.instance
      .collection('chats')
      .doc(widget.chatID)
      .collection('messages')
      .where('senderID', isEqualTo: widget.recipientID) // Only messages from the other user
      .where('isRead', isEqualTo: false) // Only unread messages
      .get();

  WriteBatch batch = FirebaseFirestore.instance.batch();

  for (var doc in messagesQuery.docs) {
    batch.update(doc.reference, {'isRead': true});
  }

  await batch.commit();
}
```

Figure 2.4.4.2

Using the _markMessagesAsRead function, Messages from the other participant of the chat room are marked as read as soon as the chat screen is opened by the current participant. A periodic timer also calls _markMessagesAsRead every 10 seconds to check for and update unread messages to a "read" state, providing dynamic feedback on message status. The WriteBatch operation for updating multiple documents ensures that all unread messages are marked as read in a single transaction, which improves efficiency and consistency.

```dart
if (isSender)
  Icon(
    message.isRead
        ? Icons.done_all  // Two ticks for read
        : Icons.done,     // One tick for sent but unread
    size: 16,
    color: message.isRead ? Colors.blue : Colors.grey,
  ), // Icon
],
```

Figure 2.4.4.3

Each message is displayed with a visual status icon indicating whether it is "sent" (one tick) or "read" (two blue ticks). This dynamic feedback depends on the isRead field, which updates in real time from Firestore, giving users instant status information about message delivery and reading status as this code is in the scope of the StreamBuilder widget.

```dart
Future<void> _sendMessage() async {
  if (_messageController.text.trim().isEmpty) return;

  final message = Message(
    senderID: widget.senderID,
    recipientID: widget.chatID,
    messageText: _messageController.text.trim(),
    time: Timestamp.now(),
    isRead: false,
  ); // Message

  FirebaseFirestore.instance
      .collection('chats')
      .doc(widget.chatID)
      .collection('messages')
      .add(message.toFirestore());

  _messageController.clear();

  // Retrieve the recipient's FCM token
  final recipientFcmToken = await _fcmService.getRecipientFcmToken(widget.recipientID);

  // Send a push notification if the FCM token is available
  if (recipientFcmToken != null) {
    await _fcmService.sendPushNotification(
      recipientFcmToken,
      message.messageText,
      "New Message"
    );
  }
}
```

Figure 2.4.4.4

The _sendMessage function dynamically manages sending messages by creating a Message object with relevant metadata, such as senderID, recipientID, messageText, time, and isRead. The message is then added to Firestore in the chats collection, where it becomes immediately available in the StreamBuilder, dynamically updating the chat UI. Push notifications are also managed here. After sending the message, the function retrieves the recipient's FCM token using the self-defined FCMService functions provider and triggers a notification to be sent to the message recipient's device, which enhance the dynamic responsiveness of the chat.

2.4.5 Setting

```dart
@override
void initState() {
  super.initState();
  _loadSettings(); // Load settings when the page is opened
}

@override
void dispose() {
  _audioPlayer.dispose();
  super.dispose();
}

// Load settings from SharedPreferences
Future<void> _loadSettings() async {
  final prefs = await SharedPreferences.getInstance();
  setState(() {
    darkmode = prefs.getBool('darkmode') ?? false;
    audioNotification = prefs.getBool('audioNotification') ?? true;
    vibrationNotification = prefs.getBool('vibrationNotification') ?? true;

    // Apply dark mode if enabled
    Provider.of<ThemeProvider>(context, listen: false).toggleTheme(darkmode);
  });
}

// Save a specific setting to SharedPreferences
Future<void> _saveSetting(String key, bool value) async {
  final prefs = await SharedPreferences.getInstance();
  await prefs.setBool(key, value);
}
```

Figure 2.4.5.1

When the setting page initializes, _loadSettings is called to fetch the stored values from SharedPreferences. If a setting hasn't been saved yet, a default value is used. The _saveSetting function takes a key and value to store a specific setting in SharedPreferences, so any changes made by the user are retained even if the app restarts.

ThemeProvider allows toggling between dark and light themes based on the darkmode setting. Changes are reflected across the app. The font size is adjustable with a slider that dynamically updates the fontSize property in ThemeProvider, allowing for a responsive font experience across UI elements.

```
SwitchListTile(
  title: Text("Audio", style: TextStyle(fontSize: fontSize)),
  value: audioNotification,
  onChanged: (value) {
    setState(() {
      audioNotification = value;
      if (audioNotification) playSound();
    });
    _saveSetting('audioNotification', audioNotification); // Save audio setting
  },
), // SwitchListTile
SwitchListTile(
  title: Text("Vibration", style: TextStyle(fontSize: fontSize)),
  value: vibrationNotification,
  onChanged: (value) {
    setState(() {
      vibrationNotification = value;
      if (vibrationNotification) triggerVibration();
    });
    _saveSetting('vibrationNotification', vibrationNotification); // Save vibration setting
  },
), // SwitchListTile
],
```

Figure 2.4.5.2

The audio setting controls whether a sound should be played for notifications. If enabled, the playSound function plays an audio file using the AudioPlayer instance. The vibration setting controls whether the device vibrates for notifications. If enabled, triggerVibration checks if the device has a vibrator and initiates vibration. Each setting uses widgets like SwitchListTile for toggles and Slider for font size. When users interact with these widgets, setState is called to update the UI, ensuring an immediate visual response to their actions.

2.4.6 Health Data Alert

```
Future<void> logHealthData(String userID, String bloodPressure, String pulseRate) async {
  double bpValue = double.parse(bloodPressure);
  double prValue = double.parse(pulseRate);
  bool isEmergency = checkEmergencyStatus(bpValue, prValue);

  try {
    // Log health data in repository
    await healthMetricsRepository.logHealthData(userID, bloodPressure, pulseRate, isEmergency);

    // Check if an emergency condition was detected
    if (isEmergency) {
      print("Emergency detected! Proceeding with alert and push notification.");

      // Get caregiver and healthcare provider IDs
      final ids = await alertService.getCaregiverAndHealthcareProviderIDs(userID);
      List<String> medicationIDs = await alertService.getMedicationIDs(userID);

      // Create an alert if emergency status is true
      await alertService.createAlert(
        elderlyUserID: userID,
        caregiverIDs: ids['caregiverID'] ?? [],
        healthcareProviderIDs: ids['healthcareProviderIDs'] ?? [],
        medicationID: medicationIDs,
      );

      // Loop through each caregiver ID and send push notification
      List<String> caregiverIds = List<String>.from(ids['caregiverID'] ?? []);  // Ensure the caregiver IDs are in a list

      for (String caregiverId in caregiverIds) {
```

Figure 2.4.6.1

The function first dynamically parses bloodPressure and pulseRate, evaluates these metrics to determine isEmergency, and creates a log entry that includes userID, health metrics, and the emergency status. If an emergency condition is identified (isEmergency is true), the function retrieves the user's caregiver and healthcare provider IDs, as well as any medication IDs associated with the user, using the alertService. This ensures that the alert is customized and includes all relevant information specific to the current emergency.The alert is then created in the system, dynamically associating it with the elderly user's ID, caregiver and provider IDs, and relevant medications. This alert becomes instantly available for real-time viewing in the UI, allowing for immediate response.

```
      // Loop through each caregiver ID and send push notification
      List<String> caregiverIds = List<String>.from(ids['caregiverID'] ?? []);  // Ensure the caregiver IDs are in a list

      for (String caregiverId in caregiverIds) {
        // Fetch the caregiver's FCM token
        String? caregiverFcmToken = await fcmService.getRecipientFcmToken(caregiverId);

        if (caregiverFcmToken != null) {
          print("Caregiver's FCM token: $caregiverFcmToken");

          // Send a push notification to the caregiver
          await fcmService.sendPushNotification(
            caregiverFcmToken,
            "Emergency alert! Please check the elderly person's health metrics.",
            "Health data alert"
          );
        } else {
          print("Caregiver's FCM token is null. Cannot send push notification.");
        }
      }
    }
  } catch (e) {
    print("Error logging health data: $e");
  }
}
```

Figure 2.4.6.2

After creating the alert, the function initiates a loop to dynamically retrieve FCM tokens for each caregiver associated with the user. It uses the fcmService to fetch the FCM token for each caregiver based on their ID. If a valid FCM token is retrieved, the function triggers a push notification to alert caregivers about the emergency, using a message that prompts immediate action. This ensures that caregivers are notified as soon as an emergency is detected, enhancing the system's responsiveness. logHealthData adapts to real-time health metrics and emergency statuses. It logs health data, creates alerts, and sends notifications instantly, updating the UI and notifying caregivers in real time.

2.4.7 Emergency Call

```
Future<void> handleEmergency(String elderlyUserID) async {
  try {
    DocumentSnapshot elderlyUserDoc = await FirebaseFirestore.instance
        .collection('users')
        .doc(elderlyUserID)
        .get();

    if (!elderlyUserDoc.exists) {
      print("No elderly user found with ID: $elderlyUserID");
      return;
    }

    // Get the list of caregiver IDs (now it's an array)
    List<dynamic> caregiverIDs = elderlyUserDoc['caregiverID'] ?? [];

    if (caregiverIDs.isEmpty) {
      print("No caregivers associated with elderly user ID: $elderlyUserID");
      return;
    }

    for (var caregiverID in caregiverIDs) {
      // Fetch each caregiver's document
      DocumentSnapshot caregiverDoc = await FirebaseFirestore.instance
          .collection('users')
          .doc(caregiverID)
          .get();

      if (caregiverDoc.exists) {
        String? contactInfo = caregiverDoc['contactInfo'];
        if (contactInfo != null && contactInfo.isNotEmpty) {
          await callCaregiver(contactInfo);
        } else {
          print("No valid contact info found for caregiver with ID: $caregiverID.");
        }
```

Figure 2.4.7.1

Rather than hardcoding a specific number of caregivers, this function retrieves an array of caregiverIDs and processes each one individually. This makes it flexible and scalable, capable of handling any number of caregivers. The function iterates through each caregiverID dynamically, retrieving the caregiver's document from Firestore in real time. By processing caregivers one by one, it allows each caregiver's information such as contact info to be checked and acted upon individually. The function checks if contactInfo exists and is non-empty for each caregiver, only attempting a call if this data is valid. This conditional approach adapts to incomplete or missing data, preventing unnecessary errors and ensuring only valid caregivers are contacted.

2.4.8 View Alerts

```dart
// caregiver_alert_provider.dart
import 'package:flutter/material.dart';
import 'package:mae_assignment/models/alert.dart';
import 'package:mae_assignment/repositories/alert_repository.dart';

class CaregiverAlertProvider extends ChangeNotifier {
  List<Alert> _alerts = [];
  List<Alert> get alerts => _alerts;
  final AlertRepository _alertRepository = AlertRepository();

  Future<void> fetchAlertsCaregiver(String caregiverID) async {
    try {
      _alerts = await _alertRepository.fetchAlertsForCaregiver(caregiverID);
      notifyListeners();
    } catch (e) {
      print('Error retrieving alerts for caregiver: $e');
    }
  }

  Future<void> deleteAlert(int index) async {
    try {
      String alertID = _alerts[index].alertID;

      await _alertRepository.deleteAlert(alertID);

      _alerts.removeAt(index);
      notifyListeners();
    } catch (e) {
      print('Error deleting alert: $e');
    }
  }
}
```

Figure 2.4.8.1

The fetchAlertsCaregiver function retrieves alert data specifically for a given caregiverID. This is a dynamic approach because the function fetches alerts based on the current caregiver's ID, allowing it to adapt to any caregiver in the system. By passing a different caregiverID each time, this provider can handle data retrieval for various caregivers, adjusting dynamically based on who is logged in or interacting with the app. Once alerts are fetched, notifyListeners() is called to update any UI elements listening to this provider, ensuring real-time data display in the app.

The deleteAlert function removes an alert from the list based on its index. This allows alerts to be deleted dynamically based on user actions, rather than predefined data. First, it retrieves the alertID of the alert to be deleted. The alertID is then passed to _alertRepository.deleteAlert to remove it from the database. After the alert is removed, _alerts.removeAt(index) updates the in-memory list to remove the alert locally, and notifyListeners() is called to update the UI immediately. This dynamic deletion allows the caregiver's alert list to adjust in real-time, reflecting the deletion action instantly in the UI.

```
Widget _buildAlertList(CaregiverAlertProvider alertsProvider) {
  return ListView.builder(
    itemCount: alertsProvider.alerts.length,
    itemBuilder: (context, index) {
      final alert = alertsProvider.alerts[index];
      return FutureBuilder<String?>(
        future: UserRepository().getElderlyUserNameByID(alert.elderlyID),
        builder: (context, snapshot) {
          if (snapshot.connectionState == ConnectionState.waiting) {
            return CircularProgressIndicator();
          }
          if (!snapshot.hasData) {
            return Text("User not found");
          }

          final elderlyUserName = snapshot.data!;

          return Column(
            children: [
              Padding(
                padding:
                    const EdgeInsets.symmetric(horizontal: 10, vertical: 6),
                child: Dismissible(
                  key: Key(alert.alertID),
                  background: Container(
                    color: Colors.red,
                    alignment: Alignment.centerLeft,
                    padding: EdgeInsets.only(left: 20),
                    child: Icon(Icons.delete, color: Colors.white),
                  ), // Container
                  onDismissed: (_) {
                    alertsProvider.deleteAlert(index);
                    ScaffoldMessenger.of(context).showSnackBar(
                      SnackBar(content: Text('Alert deleted')),
                    );
```

Figure 2.4.8.2

The widget dynamically adjusts the number of items displayed. If the alert list changes like an alert is added or deleted, the UI automatically updates to reflect the current state of the list. For each alert, a FutureBuilder retrieves the name of the elderly user associated with that alert. This

process is dynamic because it fetches data asynchronously for each specific alert, adapting to the specific elderlyID associated with each alert.

```
if (index < alertsProvider.alerts.length - 1)
  Divider(
    thickness: 1,
    color: Colors.grey[300],
    indent: 20,
    endIndent: 20,
  ), // Divider
```

Figure 2.4.8.3

The conditional rendering of the divider adapts based on the alert list's length and index, demonstrating dynamic management of UI components.

2.5 CRUD

2.5.1 Create

2.5.1.1 Add Alert

```dart
// Add a new alert entry
Future<void> addAlert(Alert alert) async {
  try {
    if (alert.alertID.isEmpty) {
      throw ArgumentError('Alert ID cannot be empty.');
    }
    await _alertCollection.doc(alert.alertID).set(alert.toJson());
  } catch (e) {
    print('Failed to add alert: $e');
    throw Exception('Error adding alert.');
  }
}
```

Figure 2.5.1.1.1: Add Alert

The function show is the repository function to create a new alert document in firebase cloud firestore database.

2.5.1.2 Add appointment

```
Future<void> scheduleAppointment(
    String elderlyID, String providerID, DateTime dateTime) async {
  try {
    if (providerID.isEmpty || elderlyID.isEmpty) {
      throw ArgumentError('Elderly ID and Provider ID cannot be empty.');
    }

    String appointmentID = _appointmentCollection.doc().id;

    await _appointmentCollection.doc(appointmentID).set({
      'appointmentID': appointmentID,
      'elderlyID': elderlyID,
      'healthCareID': providerID,
      'appointmentDateTime': Timestamp.fromDate(dateTime),
      'status': 'scheduled',
    });
  } catch (e) {
    print('Failed to schedule appointment: $e');
    throw Exception('Error scheduling appointment.');
  }
}
```

Figure 2.5.1.2.1: add appointment

The add appointment function show that is to check is there the provider ID or elderly ID is empty, if not then add the appointment that add by the elderly user by the field of appointment ID, elderly user ID, healthcare provider ID, appointment date time and its status.

2.5.1.3 add chat

```dart
// Add a new chat entry
Future<void> addChat(Chat chat) async {
  try {
    if (chat.chatID.isEmpty) {
      throw ArgumentError('Chat ID cannot be empty.');
    }
    await _chatCollection.doc(chat.chatID).set(chat.toJson());
  } catch (e) {
    print('Failed to add chat: $e');
    throw Exception('Error adding chat.');
  }
}
```

Figure 2.5.1.3.1: add chat repository

The add chat repository is to verify is there any chatID exist, if True it will create new document in the chat collection.

2.5.1.4 add message

```dart
Future<void> addMessage(String chatID, String senderID, String recipientID, String messageText) async {
  try {
    if (chatID.isEmpty || senderID.isEmpty || recipientID.isEmpty || messageText.isEmpty) {
      throw ArgumentError('Chat ID, Sender ID, Recipient ID, and Message Text cannot be empty.');
    }

    final messageID = _chatCollection.doc(chatID).collection('messages').doc().id;
    final messageRef = _chatCollection.doc(chatID).collection('messages').doc(messageID);

    await messageRef.set({
      'senderID': senderID,
      'recipientID': recipientID,
      'messageText': messageText,
      'time': Timestamp.now(),
      'isRead': false,
    });

    // Update the last message and time in the Chat document
    await _chatCollection.doc(chatID).update({
      'lastMessage': messageText,
      'lastMessageTime': Timestamp.now(),
    });
  } catch (e) {
    print('Failed to add message: $e');
    throw Exception('Error adding message.');
  }
}
```

Figure 2.5.1.4.1: add message repository

In this code the add message function it will check is that the chatID, senderID, recipientID and messageText is it any one empty. If not, it will add the new message in the firestore database's sub collection that call "message" and inside it have the senderID, recipientID, time and is it read.

2.5.1.5 log health data

```
// Log new health data entry
Future<void> logHealthData(String userID, String bloodPressure,
    String pulseRate, bool isEmergency) async {
  try {
    String healthMetricsID = _healthMetricsCollection.doc().id;

    await _healthMetricsCollection.doc(healthMetricsID).set({
      'healthMetricsID': healthMetricsID,
      'elderlyUserID': userID,
      'dateTime': DateTime.now(),
      'bloodPressure': double.parse(bloodPressure),
      'pulseRate': double.parse(pulseRate),
      'emergencyStatus': isEmergency,
    });
  } catch (e) {
    print("Error logging health data: $e");
    throw Exception("Error logging health data.");
  }
}
```

Figure 2.5.1.5.1: add health metrics repository

This rerpository create function is about to verify the metric ID, then if it is verified then it will add the create a new health metric document in the collection called “HealthMetrics”.

2.5.1.6 add medication

```
// Add a new medication to Firestore
Future<DocumentReference> addMedication(Medication medication) async {
  return await medicationsRef.add(medication.toJson());
}
```

Figure 2.5.1.6.1: add medication

In the medication part when the caregiver add a new medication to elderly user then it will create a new document into the firestore database collection named medication.

2.5.1.7 add reminder

```dart
final CollectionReference remindersRef = FirebaseFirestore.instance.collection('reminders');
// Add a new reminder to Firestore
Future<DocumentReference> addReminder(Reminder reminder) async {
  try {
    return await remindersRef.add(reminder.toJson());
  } catch (e) {
    print("Error adding reminder: $e");
    rethrow;
  }
}
```

Figure 2.5.1.7.1: add reminder repository

The add reminder is to function when the caregiver add a medication for elderly user, at the same time it will generate a create a new document into the reminder collection.

2.5.1.8 add associated healthcare provider

```
Future<void> addAssociatedHealthcareProvider(
    String caregiverID, String healthcareProviderID, String username) async {
  await usersRef
      .doc(caregiverID)
      .collection('associatedHealthcareProviders')
      .doc(healthcareProviderID)
      .set({
    'username': username,
    'userID': healthcareProviderID,
  });
}
```

Figure 2.5.1.8.1: add associated healthcare provider repository

The function add associated healthcare provider is a function that to create a sub collection in the "users" collection and username and the healthcare provider ID.

2.5.1.9 add associated elderly user

```
Future<void> addAssociatedElderlyUser(
    String caregiverID, String elderlyUserID, String username) async {
  await usersRef
      .doc(caregiverID)
      .collection('associatedElderlyUsers')
      .doc(elderlyUserID)
      .set({
    'username': username,
    'userID': elderlyUserID,
  });
}
```

Figure 2.5.1.9.1: add associated elderly user repository

This function is similar with Figure 2.5.8.1 but in this it is just changing the subcollection to associated elderly users and the document in sub collection is the elderly user's username and his userID.

2.5.2 Read

2.5.2.1 get alert for health care provider

```
// Retrieve all alerts for a specific healthcare provider
Stream<List<Alert>> getAlertsForHealthcareProvider(String providerID) {
  try {
    if (providerID.isEmpty) {
      throw ArgumentError('Provider ID cannot be empty.');
    }
    return _alertCollection
        .where('healthcareProviderIDs', arrayContains: providerID)
        .snapshots()
        .map((snapshot) => snapshot.docs.map((doc) => Alert.fromFirestore(doc)).toList());
  } catch (e) {
    print('Failed to retrieve alerts for healthcare provider: $e');
    return Stream.error('Error retrieving alerts for healthcare provider.');
  }
}
```

Figure 2.5.2.1.1: get alert for health care provider

The function getAlertsForHealthcareProvider is the Read operation in CRUD. It fetches and listens to real-time updates for alerts associated with a specified providerID.

2.5.2.2 fetch alert for elderly

```dart
// Retrieve alerts for a specific elderly user
Stream<List<Alert>> fetchAlertsForElderly(String elderlyID) {
  try {
    if (elderlyID.isEmpty) {
      throw ArgumentError('Elderly ID cannot be empty.');
    }
    return _alertCollection
        .where('elderlyID', isEqualTo: elderlyID)
        .snapshots()
        .map((snapshot) => snapshot.docs.map((doc) => Alert.fromFirestore(doc)).toList());
  } catch (e) {
    print('Failed to retrieve alerts for elderly: $e');
    return Stream.error('Error retrieving alerts for elderly.');
  }
}
```

Figure 2.5.2.2.1: fetch alert for elderly repository

fetchAlertsForElderly retrieves (Read) a stream of Alert objects from Firestore where the elderlyID matches the specified ID. It uses where to filter the alerts and snapshots () to listen to real-time updates. Then, it maps the Firestore documents into Alert objects using Alert.fromFirestore.

2.5.2.3 fetch alerts for caregiver and fetch alerts for provider

```
Future<List<Alert>> fetchAlertsForCaregiver(String caregiverID) async {
  try {
    QuerySnapshot alertSnapshot = await _alertCollection
        .where('caregiverIDs', arrayContains: caregiverID)
        .get();
    return alertSnapshot.docs.map((doc) => Alert.fromFirestore(doc)).toList();
  } catch (e) {
    print('Error retrieving alerts for caregiver: $e');
    return [];
  }
}

Future<List<Alert>> fetchAlertsForProvider(String ProviderID) async {
  try {
    QuerySnapshot alertSnapshot = await _alertCollection
        .where('healthcareProviderIDs', arrayContains: ProviderID)
        .get();
    return alertSnapshot.docs.map((doc) => Alert.fromFirestore(doc)).toList();
  } catch (e) {
    print('Error retrieving alerts for healthcare provider: $e');
    return [];
  }
}
```

Figure 2.5.2.3.1: fetch alert for caregiver and fetch alerts for provider

The two function shows in the figure is fetch alert for caregiver and fetch alerts for provider. It both are the same function to check is there the userID is equal then fetch alerts from it.

2.5.2.4 get Appointments with Elderly Username

```dart
// Stream to get appointments along with elderly usernames
Stream<List<Map<String, dynamic>>> getAppointmentsWithElderlyUsername(String userID) async* {
  await for (final snapshot in _appointmentCollection
      .where('healthCareID', isEqualTo: userID)
      .snapshots()) {
    List<Map<String, dynamic>> appointmentsWithUsername = [];

    for (final doc in snapshot.docs) {
      final appointment = Appointment.fromFirestore(doc);

      // Fetch elderly username from users collection using elderlyID
      final elderlyUsername = await _userRepository.getElderlyUserNameByID(appointment.elderlyID);

      appointmentsWithUsername.add({
        'appointment': appointment,
        'elderlyUsername': elderlyUsername ?? 'Unknown',
      });
    }
    yield appointmentsWithUsername;
  }
}
```

Figure 2.5.2.4.1: get appointments with elderly Username repository

The code uses the Retrieve operation to fetch a continuous stream of appointments associated with a healthcare provider. It creates a stream of appointments, filtering by healthCareID to only fetch relevant ones. For each appointment, it retrieves the elderly user's name by calling _userRepository.getElderlyUserNameByID(appointment.elderlyID), augmenting each appointment with the elderly user's username. The method creates a list of maps, each containing an appointment and the corresponding elderly user's name, which is then streamed to subscribers, allowing live updates on the appointments.

2.5.2.5 get chat by ID

```dart
// Retrieve a single chat by ID
Future<Chat?> getChatById(String chatID) async {
  try {
    if (chatID.isEmpty) {
      throw ArgumentError('Chat ID cannot be empty.');
    }
    DocumentSnapshot doc = await _chatCollection.doc(chatID).get();
    if (doc.exists) {
      return Chat.fromFirestore(doc);
    } else {
      print('Chat not found for ID: $chatID');
      return null;
    }
  } catch (e) {
    print('Failed to retrieve chat: $e');
    throw Exception('Error retrieving chat.');
  }
}
```

Figure 2.5.2.5.1: get chat by ID repository

The code defines a method to retrieve a chat document by its chatID, checking if it's empty, retrieving the document from Firestore's _chatCollection, checking if it exists, handling errors, and throwing an exception if necessary.

2.5.2.6 fetch health data

```
Future<List<HealthMetrics>> fetchHealthData(String userID) async {
  try {
    final snapshot = await _healthMetricsCollection
        .where('elderlyUserID', isEqualTo: userID)
        .get();

    return snapshot.docs.map((doc) {
      final data = doc.data() as Map<String, dynamic>;
      return HealthMetrics(
        dateTime:
            data['dateTime'] ?? DateTime.now(), // Provide a default value
        bloodPressure:
            data['bloodPressure'] ?? 0.0, // Provide a default value
        pulseRate: data['pulseRate'] ?? 0.0, // Provide a default value
        metricID: doc.id,
        elderlyUserID: userID,
        emergencyStatus:
            data['emergencyStatus'] ?? false, // Default to false if null
      ); // HealthMetrics
    }).toList();
  } catch (e) {
    print("Error fetching health data: $e");
    throw Exception("Error fetching health data.");
  }
}
```

Figure 2.5.2.6.1: fetch health data repository

The code retrieves health metrics for an elderly user based on their userID, filtering by userID, mapping data to HealthMetrics collection, providing default values for missing data, and handling errors to ensure robust data handling.

2.5.2.7 fetch Medications Stream

```
Stream<List<Medication>> fetchMedicationsStream(String elderlyUserID) {
  return medicationsRef
      .where('elderlyUserID', isEqualTo: elderlyUserID)
      .where('medicationStatus', isEqualTo: 'Scheduled')
      .snapshots()
      .map((querySnapshot) {
        return querySnapshot.docs.map((doc) {
          DateTime medicationDateTime = (doc['dateTime'] as Timestamp).toDate();
          return Medication(
            medicationID: doc.id,
            medicationType: doc['medicationType'] ?? 'Unknown',
            dosage: doc['dosage'] ?? 'N/A',
            medicationStatus: doc['medicationStatus'] ?? 'Scheduled',
            dateTime: medicationDateTime,
            elderlyUserID: doc['elderlyUserID'],
          );
        }).toList();
      });
}
```

Figure 2.5.2.7.1: fetch medications stream

The code creates a real-time stream for a specific elderly user, filtering medications based on their status. It maps the data into a list of Medication objects, including the elderlyUserID, medicationType, dosage, medicationStatus, and elderlyUserID. The stream emits the latest list of medications, ensuring up-to-date information without explicitly re-fetching data.

2.5.2.8 fetch medication for user today

```
Future<List<Medication>> fetchMedicationsForUserToday(String userID) async {
  try {
    DateTime today = DateTime.now();
    QuerySnapshot querySnapshot = await medicationsRef
        .where('elderlyUserID', isEqualTo: userID)
        .get();

    return querySnapshot.docs.map((doc) {
      if (doc['dateTime'] != null) {
        DateTime medicationDateTime = (doc['dateTime'] as Timestamp).toDate();
        DateTime medicationDate = DateTime(medicationDateTime.year, medicationDateTime.month, medicationDateTime.day);

        if (medicationDate.isAtSameMomentAs(DateTime(today.year, today.month, today.day))) {
          return Medication(
            medicationID: doc.id,
            medicationType: doc['medicationType'] ?? 'Unknown',
            dosage: doc['dosage'] ?? 'N/A',
            medicationStatus: doc['medicationStatus'] ?? 'pending',
            dateTime: medicationDateTime,
            elderlyUserID: doc['elderlyUserID'],
          );
        }
      }
      return null;
    }).whereType<Medication>().toList();
  } catch (e) {
    print("Error fetching medications: $e");
    return [];
  }
}
```

Figure 2.5.2.8.1: fetch medication for user today

The code retrieves all medications scheduled for a specific user on the current day, using DateTime.now () to compare the dateTime field in each document. It then queries MedicationsRef, maps each document, creates a new DateTime object, compares it with the current date, returns a list of scheduled medications, and filters out null values.

2.5.2.9 get medication by ID

```dart
// Retrieve a medication by ID
Future<Medication?> getMedicationByID(String medicationID) async {
  try {
    if (medicationID.isEmpty) {
      throw ArgumentError("Medication ID cannot be empty.");
    }
    final doc = await medicationsRef.doc(medicationID).get();
    if (doc.exists) {
      return Medication.fromFirestore(doc);
    } else {
      print("Medication not found for ID: $medicationID");
      return null;
    }
  } catch (e) {
    print("Failed to retrieve medication by ID: $e");
    throw Exception("Error retrieving medication.");
  }
}
```

Figure 2.5.2.9.1: get medication by ID repository

The code retrieves a medication document from Firestore by its medicationID, performs basic validation, checks if the document exists, and handles errors gracefully. If the document exists, it maps the Firestore document to a Medication object, and if not, it returns null.

2.5.2.10 get medication by elderly user and medication ID

```
Future<List<Medication>> getMedicationsByElderlyUserAndMedicationID(
    String elderlyUserID, String medicationID) async {
  try {
    if (elderlyUserID.isEmpty || medicationID.isEmpty) {
      throw ArgumentError(
          "Both Elderly User ID and Medication ID cannot be empty.");
    }
    final snapshot = await medicationsRef
        .where('elderlyUserID', isEqualTo: elderlyUserID)
        .where('medicationID', isEqualTo: medicationID)
        .get();
    return snapshot.docs.map((doc) => Medication.fromFirestore(doc)).toList();
  } catch (e) {
    print(
        "Failed to retrieve medications for elderly user and medication ID: $e");
    throw Exception(
        "Error retrieving medications for elderly user and medication ID.");
  }
}
```

Figure 2.5.2.10.1: get medication by elderly user and medicationID

The code retrieves medications for an elderly user based on their ID and medication ID from the Firestore database. It validates the IDs, filters documents, maps data to Medication objects, and returns the list of medications. If an error occurs, it throws an exception.

2.5.2.11 get reminder by elderly user

```dart
// Retrieve reminders for a specific elderly user
Future<List<Reminder>> getRemindersByElderlyUser(String elderlyUserID) async {
  try {
    if (elderlyUserID.isEmpty) {
      throw ArgumentError("Elderly User ID cannot be empty.");
    }
    final snapshot = await remindersRef.where('elderlyUserID', isEqualTo: elderlyUserID).get();
    return snapshot.docs.map((doc) => Reminder.fromFirestore(doc)).toList();
  } catch (e) {
    print("Failed to retrieve reminders for elderly user: $e");
    throw Exception("Error retrieving reminders for elderly user.");
  }
}
```

Figure 2.5.2.11.1: get reminder by elderly user

The code retrieves reminders for an elderly user based on their elderlyUserID by executing a query on the Firestore database, mapping the documents to Reminder objects, and returning a list of Reminder objects. If an error occurs, an exception is thrown, ensuring all reminders are retrieved.

2.5.2.12 get user by username

```
Future<DocumentSnapshot> getUserByUsername(String username) async {
  final userSnapshot =
      await usersRef.where('username', isEqualTo: username).limit(1).get();

  if (userSnapshot.docs.isNotEmpty) {
    return userSnapshot.docs.first;
  } else {
    throw Exception("User  not found");
  }
}
```

Figure 2.5.2.12.1: get user by username repository

The code retrieves a user document from Firestore based on their username using a query with a limit of 1, checking for results, and returning a DocumentSnapshot containing all user data fields. If no results are found, an exception is thrown.

2.5.13 stream associated elderly users

```
Stream<List<Map<String, String>>> streamAssociatedElderlyUsers(
    String caregiverID) {
  return usersRef
      .doc(caregiverID)
      .collection('associatedElderlyUsers')
      .snapshots()
      .map((snapshot) {
    return snapshot.docs.map((doc) {
      return {
        'userID': doc['userID'] as String,
        'username': doc['username'] as String,
      };
    }).toList();
  });
}
```

Figure 2.5.2.13.1: stream associated elderly users' repository

The code defines a Read process that streams data from Firestore, fetching a list of elderly users associated with a caregiver in real-time. It sets up a stream, maps data to maps, and emits updates to the stream. This method is useful for live displays of elderly users.

2.5.2.14 stream healthcare Providers

```
Stream<List<Map<String, String>>> streamHealthcareProviders(
    String caregiverID) {
  return usersRef
      .doc(caregiverID)
      .collection('associatedHealthcareProviders')
      .snapshots()
      .map((snapshot) {
    return snapshot.docs.map((doc) {
      return {
        'userID': doc['userID'] as String,
        'username': doc['username'] as String,
      };
    }).toList();
  });
}
```

Figure 2.5.2.14.1: stream healthcare providers repository

The code creates a Read operation from Firestore to fetch healthcare providers associated with a specific caregiver. It sets up a stream, maps data to a list of maps, and emits updates automatically. This method ensures live tracking of caregivers and maintains data sync with database changes.

2.5.2.15 get user by ID

```dart
// Retrieve a user by ID
Future<UserData?> getUserByID(String userID) async {
  try {
    if (userID.isEmpty) {
      throw ArgumentError("User ID cannot be empty.");
    }
    final doc = await usersRef.doc(userID).get();
    if (doc.exists) {
      return UserData.fromFirestore(doc);
    } else {
      print("User not found for ID: $userID");
      return null;
    }
  } catch (e) {
    print("Failed to retrieve user by ID: $e");
    throw Exception("Error retrieving user.");
  }
}
```

Figure 2.5.2.12.1: get user by ID repository

The function is to get the user data by its userID in the collection. If it is match it will fetch the data from the firestore.

2.5.2.16 fetch healthcare provider

```
Future<List<Map<String, dynamic>>> fetchHealthcareProviders() async {
  try {
    QuerySnapshot querySnapshot =
        await usersRef.where('role', isEqualTo: 'Healthcare Provider').get();

    return querySnapshot.docs.map((doc) {
      return {
        'id': doc.id,
        ...doc.data() as Map<String, dynamic>,
      };
    }).toList();
  } catch (e) {
    print("Failed to fetch healthcare providers: $e");
    throw Exception("Failed to fetch healthcare providers.");
  }
}
```

Figure 2.5.2.16.1: fetch healthcare provider repository

this function is about to fetch the healthcare provider by validate the user's role and then get the data of healthcare providers.

2.5.2.17 get elderly username by ID

```
Future<String?> getElderlyUserNameByID(String elderlyUserID) async {
  try {
    final doc = await usersRef.doc(elderlyUserID).get();
    if (doc.exists) {
      return doc['username'] as String?;
    } else {
      print("Elderly user not found for ID: $elderlyUserID");
      return null;
    }
  } catch (e) {
    print("Error fetching elderly user name: $e");
    return null;
  }
}
```

Figure 2.5.2.17.1: get elderly username by ID

This function is to return the username in the firestore database by using the parameter called “elderlyUserID” to read the data in the firestore database.

2.5.2.18 fetch Associated Elderly User IDs

```
// Fetch associated elderly user IDs for a given caregiver
Future<List<String>> fetchAssociatedElderlyUserIDs(String caregiverID) async {
  try {
    QuerySnapshot associatedUsersSnapshot = await usersRef
        .doc(caregiverID)
        .collection('associatedElderlyUsers')
        .get();

    List<String> elderlyUserIDs =
        associatedUsersSnapshot.docs.map((doc) => doc.id).toList();

    return elderlyUserIDs;
  } catch (e) {
    print("Error fetching associated elderly users: $e");
    return [];
  }
}
```

Figure 2.5.2.18.1: fetch Associated Elderly User IDs

The code Read from Firestore to fetch elderlyuserIDs associated with a specific caregiver. Then it get the data map it into a list.

2.5.2.19 Stream Caregivers

```dart
Stream<List<Map<String, String>>> streamCaregivers(String providerID) {
  final usersRef = FirebaseFirestore.instance.collection('users');

  return usersRef.doc(providerID).snapshots().asyncMap((providerDoc) async {
    // Check if the document exists and has the caregiverID field
    if (!providerDoc.exists || !providerDoc.data()!.containsKey('caregiverID')) {
      return [];
    }

    List<String> caregiverIDs = List<String>.from(providerDoc['caregiverID']);
    List<Map<String, String>> caregivers = [];

    for (String caregiverID in caregiverIDs) {
      final caregiverDoc = await usersRef.doc(caregiverID).get();
      if (caregiverDoc.exists) {
        caregivers.add({
          'userID': caregiverDoc.id,
          'username': caregiverDoc['username'] as String,
        });
      }
    }

    return caregivers;
  });
}
```

Figure 2.5.2.19.1: Stream Caregivers repository

The code creates a Read operation from Firestore to fetch caregiverID associated with a specific caregiver. It sets up a stream, maps data to a list of maps, and emits updates automatically. This method ensures live tracking of caregivers and maintains data sync with database changes.

2.5.2.20 fetch notification

```
Future<void> _fetchNotifications() async {
  List<Alert> fetchedAlerts = await AlertService.fetchAlerts(widget.userID);
  setState(() {
    alerts = fetchedAlerts;
  });
}
```

Figure 2.5.2.20.1: fetch Notification

The code asynchronously fetches user-specific notifications and updates the state with the retrieved data. It calls the Alert Fetching Service, awaits data, and sets the state with fetchedAlerts, triggering a re-render of the widget and allowing UI components to display new notifications.

2.5.2.21 fetch Health data

```
Future<void> fetchHealthData() async {
  setState(() {
    isLoading = true;
  });

  try {
    final snapshot = await FirebaseFirestore.instance
        .collection('HealthMetrics')
        .where('elderlyUserID', isEqualTo: widget.elderlyID)
        .get();

    bloodPressureData = snapshot.docs.map((doc) {
      final healthMetric = HealthMetrics.fromFirestore(doc);
      double bloodPressureValue = (healthMetric.bloodPressure is int)
          ? (healthMetric.bloodPressure as int).toDouble()
          : healthMetric.bloodPressure;

      return HealthData(
        time: DateFormat('yyyy-MM-dd').format(healthMetric.dateTime.toDate()),
        value: bloodPressureValue,
      ); // HealthData
    }).toList();

    pulseRateData = snapshot.docs.map((doc) {
      final healthMetric = HealthMetrics.fromFirestore(doc);
      double pulseRateValue = (healthMetric.pulseRate is int)
          ? (healthMetric.pulseRate as int).toDouble()
          : healthMetric.pulseRate;

      return HealthData(
        time: DateFormat('yyyy-MM-dd').format(healthMetric.dateTime.toDate()),
        value: pulseRateValue,
      ); // HealthData
    }).toList();
```

Figure 2.5.2.21.1: fetch health data repository

The code retrieves health data metrics for an elderly user from Firestore and updates the state to display the data. It sets a loading state, queries Firestore for health metrics, maps data to blood pressure and pulse rate, and updates the state. The code uses isLoading to manage visual feedback during data fetching, ensuring a smooth user experience.

2.5.3 Update

2.5.3.1 update alert status

```dart
// Update an alert's status
Future<void> updateAlertStatus(String alertID, String newStatus) async {
  try {
    if (alertID.isEmpty) {
      throw ArgumentError('Alert ID cannot be empty.');
    }
    if (newStatus.isEmpty) {
      throw ArgumentError('Status cannot be empty.');
    }
    await _alertCollection.doc(alertID).update({'alertStatus': newStatus});
  } catch (e) {
    print('Failed to update alert status: $e');
    throw Exception('Error updating alert status.');
  }
}
```

Figure 2.5.3.1.1: update alert status repository

The code is an Update process to modify the status of an alert in Firestore. It validates input, updates the status in Firestore, and handles errors. The update operation allows real-time adjustments and provides feedback if necessary.

2.5.3.2 update appointment status

```dart
// Update an appointment's status
Future<void> updateAppointmentStatus(
    String appointmentID, String newStatus) async {
  try {
    if (appointmentID.isEmpty) {
      throw ArgumentError('Appointment ID cannot be empty.');
    }
    if (newStatus.isEmpty) {
      throw ArgumentError('Status cannot be empty.');
    }
    await _appointmentCollection
        .doc(appointmentID)
        .update({'status': newStatus});
  } catch (e) {
    print('Failed to update appointment status: $e');
    throw Exception('Error updating appointment status.');
  }
}
```

Figure 2.5.3.2.1: update appointment status repository

The update appointment status it will first check the appointment ID and if the status is not selected by the user it will show error message to user elderly user.

2.5.3.3 update Medication ID

```
// Update the medicationID field after adding a medication
Future<void> updateMedicationID(String medicationID) async {
  await medicationsRef.doc(medicationID).update({'medicationID': medicationID});
}
```

Figure 2.5.3.3.1: update Medication ID repositories

In this function it will update the medication ID after adding a medication.

2.5.3.4 update reminder ID

```
// Update the reminderID field after adding a reminder
Future<void> updateReminderID(String reminderID) async {
  await remindersRef.doc(reminderID).update({'reminderID': reminderID});
}
```

Figure2.5.3.4.1: update reminder ID repository

In this function it is just a easy understanding update function it will update the reminder after adding a reminder.

2.5.3.5 update elderly user with caregiver ID

```
Future<void> updateElderlyUserWithCaregiverID(
    String elderlyUserID, String caregiverID) async {
  await usersRef.doc(elderlyUserID).update({
    'caregiverID': FieldValue.arrayUnion([caregiverID])
  });
}
```

Figure 2.5.3.5.1: update elderly user with caregiver ID repository

The code defines an Update operation that associates a caregiver with a healthcare provider by updating the caregiverID field in the provider's document. It uses FieldValue.arrayUnion to maintain a unique list of caregivers, ensuring no duplicate entries.

2.5.3.6 update healthcare provider with caregiverID

```
Future<void> updateHealthcareProviderWithCaregiverID(
    String healthcareProviderID, String caregiverID) async {
  await usersRef.doc(healthcareProviderID).update({
    'caregiverID': FieldValue.arrayUnion([caregiverID])
  });
}
```

Figure 2.5.3.6.1: update healthcare provider with caregiver ID

This function adds a caregiver's ID to a healthcare provider's document in Firestore, ensuring unique entries. It uses an atomic Firestore update operation and arrayUnion to prevent duplicates. However, it lacks explicit error handling.

2.5.3.7 update user

```
// Update a user in Firestore
Future<void> updateUser(UserData user) async {
  try {
    if (user.userID.isEmpty) {
      throw ArgumentError("User ID cannot be empty.");
    }
    await usersRef.doc(user.userID).update(user.toJson());
  } catch (e) {
    print("Failed to update user: $e");
    throw Exception("Error updating user.");
  }
}
```

Figure 2.5.3.7.1: update user repository

The code defines an Update operation that updates a user's data in Firestore using a UserData object. It validates the user's ID, updates the user's document, and handles errors using a try-catch block. This efficient and reliable operation ensures non-empty userIDs.

2.5.4 Delete

2.5.4.1 delete alert

```dart
// Delete an alert
Future<void> deleteAlert(String alertID) async {
  try {
    if (alertID.isEmpty) {
      throw ArgumentError('Alert ID cannot be empty.');
    }
    await _alertCollection.doc(alertID).delete();
  } catch (e) {
    print('Failed to delete alert: $e');
    throw Exception('Error deleting alert.');
  }
}
```

Figure 2.5.4.1.1: delete alert repository

The delete alert is to check the alert ID and it will delete it if the alertID is found in the firestore database. Then it will removed it from the firestore database.

2.5.4.2 Delete alert for caregiver

```dart
Future<void> deleteAlertForCaregiver(String alertID) async {
  try {
    await _alertCollection.doc(alertID).delete();
  } catch (e) {
    print('Error deleting alert: $e');
    throw e;
  }
}
```

Figure 2.5.4.2.1: Delete alert for caregiver repository

This delete function is to be implemented when the caregiver deletes the alert of elderly user.

2.5.4.3 delete appointment

```dart
// Delete an appointment
Future<void> deleteAppointment(String appointmentID) async {
  try {
    if (appointmentID.isEmpty) {
      throw ArgumentError('Appointment ID cannot be empty.');
    }
    await _appointmentCollection.doc(appointmentID).delete();
  } catch (e) {
    print('Failed to delete appointment: $e');
    throw Exception('Error deleting appointment.');
  }
}
```

Figure 2.5.4.3.1: delete appointment repository

This function is to delete the appointment that booked by elderly user and the action is doing by the healthcare providers. Hence, the data that created before in the firestore database will be deleted.

2.5.4.4 delete chat

```dart
// Delete a chat
Future<void> deleteChat(String chatID) async {
  try {
    if (chatID.isEmpty) {
      throw ArgumentError('Chat ID cannot be empty.');
    }
    await _chatCollection.doc(chatID).delete();
  } catch (e) {
    print('Failed to delete chat: $e');
    throw Exception('Error deleting chat.');
  }
}
```

Figure 2.5.4.4.1: delete chat repository

The delete function of this shows is the user can delete their chat history in the chat screen then the firestore database will be removed.

2.5.4.5 delete reminder by medication ID

```
Future<void> deleteReminderByMedicationID(String medicationID) async {
  try {
    final reminderSnapshot = await remindersRef
        .where('medicationID', isEqualTo: medicationID)
        .get();

    for (var doc in reminderSnapshot.docs) {
      await doc.reference.delete();
    }
  } catch (e) {
    print("Error deleting reminder: $e");
  }
}
```

Figure 2.5.4.5.1: delete reminder by medication ID repository

In this function, it will use the medicationID that get from the collection that called reminder to verify the reminder been deleted is that one needed to be delete.

2.5.4.6 delete associated elderly user and healthcare provider

```
Future<void> deleteAssociatedElderlyUser(
    String caregiverID, String elderlyUserID) async {
  await usersRef
      .doc(caregiverID)
      .collection('associatedElderlyUsers')
      .doc(elderlyUserID)
      .delete();
}

Future<void> deleteAssociatedHealthcareProviders(
    String caregiverID, String HealthcareProviderID) async {
  await usersRef
      .doc(caregiverID)
      .collection('associatedHealthcareProviders')
      .doc(HealthcareProviderID)
      .delete();
}
```

Figure 2.5.4.6.1: delete associated elderly user and healthcare provider *repository*

In these two delete function are similar both functions work to check the associated collection and delete the associated with the elderly user and caregiver.

2.5.4.7 remove caregiver from elderly users or healthcare providers

```
Future<void> removeCaregiverFromElderlyUser(
    String elderlyUserID, String caregiverID) async {
  await usersRef.doc(elderlyUserID).update({
    'caregiverID': FieldValue.arrayRemove([caregiverID])
  });
}

Future<void> removeCaregiverFromHealthcareProvider(
    String HealthcareProviderID, String caregiverID) async {
  await usersRef.doc(HealthcareProviderID).update({
    'caregiverID': FieldValue.arrayRemove([caregiverID])
  });
}
```

Figure 2.5.4.7.1: remove caregiver from elderly user

This function is work to remove the caregiver associated in the collection of elderly user and healthcare providers.

2.6 Validation

```
Future<void> addElderlyUser(
    String providerID, String username, BuildContext context) async {
  try {
    final elderlySnapshot = await _userRepository.getUserByUsername(username);
    final elderlyUserID = elderlySnapshot.id;
    final elderlyUsername = elderlySnapshot['username'];
    final role = elderlySnapshot['role'];

    // Check if the role is 'Elderly User'
    if (role == 'Elderly') {
      await _userRepository.addAssociatedElderlyUser(
          providerID, elderlyUserID, elderlyUsername);

      // Update the elderly user's document with the caregiver ID
      await _userRepository.updateElderlyUserWithCaregiverID(
          elderlyUserID, providerID);

      // Add to the local list and notify listeners
      elderlyUsers
          .add({'username': elderlyUsername, 'userID': elderlyUserID});
      notifyListeners(); // This updates the UI immediately
    } else {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text("User  is not an elderly user.")),
      );
    }
  } catch (e) {
    print("Error adding elderly user: $e");
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text("An error occurred: $e")),
    );
  }
}
```

Figure 2.6.1: add elderly function validation

There are several validations have been implemented, the first validation is to validate the elderly user. The validation is when the caregiver adds an elderly user to associated with him if the firestore fetch that the role is equal to “Elderly” then it will proceed to associate the elderly user with the caregiver, else it will inform that the user is not an elderly user.

```dart
Future<void> callCaregiver(String caregiverContact) async {
  // Ensure that the phone number is valid and properly formatted
  caregiverContact = caregiverContact.replaceAll(RegExp(r'\D'), ''); // Remove non-numeric characters

  final String telUri = 'tel:$caregiverContact';

  try {
    // Check if the device can launch the phone call
    bool canLaunchCall = await canLaunch(telUri);

    if (canLaunchCall) {
      // Launch the phone call
      await launch(telUri);
      print("Call launched successfully.");
    } else {
      print("Cannot launch call to $caregiverContact. Invalid phone number format or device issue.");
    }
  } catch (e) {
    print("Error launching call: $e");
  }
}
```

Figure 2.6.2: emergency call validates the number format

The second one validation is about that the function call caregiver first it will use the phone number that fetch from the firestore and it will remove all non-numeric characters in the contact. Last, it will call the number if the number is error, it will show a message invalid number format or device issue.

```dart
class AuthProvider with ChangeNotifier {
  final FirebaseAuth _auth = FirebaseAuth.instance;
  final UserRepository _userRepository = UserRepository(); // Initialize UserRepository

  Future<void> signIn(BuildContext context, String username, String password) async {
    try {
      // Find user by username in Firestore
      DocumentSnapshot? userDoc = await _userRepository.getUserByUsername(username);

      String userEmail = userDoc['email'];
      String userID = userDoc.id;

      // Authenticate with Firebase using the retrieved email and entered password
      UserCredential userCredential = await _auth.signInWithEmailAndPassword(
        email: userEmail,
        password: password,
      );

      if (userCredential.user != null) {
        // After successful login or registration
        final authService = AuthService();
        await authService.saveUserToken(userID);
        authService.setupTokenRefresh(userID);
      } else {
        _showError(context, "Authentication failed. Please try again.");
      }
    } catch (e) {
      _showError(context, "Failed to sign in. Please check your credentials.");
      print(e); // For debugging
    }
  }
```

Figure 2.6.3: login validation

This is the validation of login function first after the user input their username and password it will fetch user by the username that input by user. Then, it will authenticate the password input by user, if the password is incorrect then it will inform the user that “Failed to sign in. Please check your credentials.”

```dart
if (_validateInputs(username, password, email, selectedRole, contactInfo)) {
  try {
    // Check if the username is unique
    final isUnique = await userService.isUsernameUnique(username);
    if (!isUnique) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('Username is already taken. Please choose another one.')),
      );
      return;
    }
    } catch (e) {
      // Handle error with a message
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('Error: ${e.toString()}')),
      );
    }
  } else {
    // Show message if validation fails
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('Please fill out all fields')),
    );
  }
}
```

Figure 2.6.5: validate while registering

This is the validation of check is that the username not been taken by another user. First it will check is there the username unique mean that is it not been taken, if it is then it will show a message to user “Username is already taken. Please choose another one.” After that, it will also validate the user to fulfil all the field in the register form if it is not fulfilled it will leave a message to user too “Please fill out all fields”.

```
        await userRepository.addAssociatedHealthcareProvider(
            caregiverID, providerID, caregiverUsername);

        // Update the Healthcare Provider document with the caregiver ID
        await userRepository.updateHealthcareProviderWithCaregiverID(
            providerID, caregiverID);

        // Add to the local list and notify listeners
        HealthcareProvider.add(
            {'username': caregiverUsername, 'userID': providerID});
        notifyListeners(); // This updates the UI immediately
      } else {
        ScaffoldMessenger.of(context).showSnackBar(
          SnackBar(content: Text("User  is not a Caregiver")),
        );
      }
    } catch (e) {
      print("Error adding Caregiver: $e");
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text("An error occurred: $e")),
      );
    }
  }
```

Figure 2.6.6: Validate the caregiver user

In this part when the caregiver input the caregiver user's username if it is valid it will add into the firestore database, else it will notify user "User is not a Caregiver".

```
Future<void> addElderlyUser (
    String caregiverID, String username, BuildContext context) async {
  try {
    final elderlySnapshot = await _userRepository.getUserByUsername(username);
    final elderlyUserID = elderlySnapshot.id;
    final elderlyUsername = elderlySnapshot['username'];
    final role = elderlySnapshot['role'];

    // Check if the role is 'Elderly User'
    if (role == 'Elderly') {
      await _userRepository.addAssociatedElderlyUser (caregiverID, elderlyUserID, elderlyUsername);

      // Update the elderly user's document with the caregiver ID
      await _userRepository.updateElderlyUserWithCaregiverID(elderlyUserID, caregiverID);

      // Add to the local list and notify listeners
      elderlyUsers.add({'username': elderlyUsername, 'userID': elderlyUserID});
      notifyListeners(); // This updates the UI immediately
    } else {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text("User  is not an elderly user.")),
      );
    }
  } catch (e) {
    print("Error adding elderly user: $e");
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text("An error occurred: $e")),
    );
  }
}
```

Figure 2.6.7: Validate the elderly username

Similar with last part in Figure 2.6.6 shows that if the role in the firestore database is equal to Elderly then it will associate the elderly with caregiver and update the elderly user's collection field, else it will warning that the user is not an elderly user.

```
if (selectedDate != null && selectedTime != null) {
  final appointmentDateTime = DateTime(
    selectedDate!.year,
    selectedDate!.month,
    selectedDate!.day,
    selectedTime!.hour,
    selectedTime!.minute,
  );

  try {
    await _appointmentRepository.scheduleAppointment(
      userID,
      providerID,
      appointmentDateTime,
    );
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text("Appointment scheduled successfully!")),
    );
    Navigator.pop(context);
  } catch (e) {
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text("Failed to schedule appointment: $e")),
    );
  }
} else {
  ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(content: Text("Please select a date and time.")),
  );
}
}
```

Figure 2.6.8: Validate the appointment added

Move to the appointment it will validate is the elderly user select the date and time, if the elderly user is not selecting either the date or time then it will notify the user “Please select a date and time.”

```
    // Check if the role is 'Healthcare Provider'
    if (role == 'Healthcare Provider') {
      // Add the healthcare provider to the caregiver's associated providers
      await userRepository.addAssociatedHealthcareProvider(
          caregiverID, healthcareProviderID, healthcareProviderUsername);

      // Update the Healthcare Provider document with the caregiver ID
      await userRepository.updateHealthcareProviderWithCaregiverID(
          healthcareProviderID, caregiverID);

      // Add to the local list and notify listeners
      HealthcareProvider.add({
        'username': healthcareProviderUsername,
        'userID': healthcareProviderID
      });
      notifyListeners(); // This updates the UI immediately
    } else {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text("User  is not a Healthcare Provider.")),
      );
    }
  } catch (e) {
    print("Error adding Healthcare Provider: $e");
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text("An error occurred: $e")),
    );
  }
}
```

Figure 2.6.9: validate the healthcare provider

Similar with last part in Figure 2.6.6 and 2.6.7 shows that if the role in the firestore database is equal to Healthcare Provider then it will associate the healthcare provider with caregiver and update the healthcare provider's collection field, else it will be warning that the user is not an healthcare provider.

```dart
// Reset password functionality
String email = emailController.text.trim();
if (email.isNotEmpty) {
  try {
    await FirebaseAuth.instance.sendPasswordResetEmail(email: email);
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text("Password reset email sent!")),
    );
  } catch (e) {
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text("Failed to send reset email. Try again.")),
    );
  }
} else {
  ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(content: Text("Please enter an email")),
  );
```

Figure 2.6.10: Validate the forgot password screen to input email

In this part it will validate the text field make sure that the user is input their email address correctly, if it isn't done it will show a message "Failed to send reset email. Try again."

```dart
Future<Uint8List?> pickImage(ImageSource source) async {
  final ImagePicker _imagePicker = ImagePicker();
  XFile? _file = await _imagePicker.pickImage(source: source);
  if (_file != null) {
    return await _file.readAsBytes();
  } else {
    print("No Image Selected");
    return null;
  }
}
```

Figure 2.6.11: Validation of image select

In this validation it shows that the image upload of changing the profile picture if it is not selected it will show a message “No Image selected”.

```dart
TextFormField(
  controller: usernameController,
  decoration: InputDecoration(
    labelText: 'Username',
    labelStyle: AppTextStyles.placeholderText,
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(10),
    ), // OutlineInputBorder
    filled: true,
    fillColor: Colors.white,
  ), // InputDecoration
  validator: (value) {
    if (value == null || value.isEmpty) {
      return 'Username is required';
    }
    return null;
  },
), // TextFormField
```

```dart
TextFormField(
  controller: passwordController,
  obscureText: true,
  decoration: InputDecoration(
    labelText: 'Password',
    labelStyle: AppTextStyles.placeholderText,
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(10),
    ), // OutlineInputBorder
    filled: true,
    fillColor: Colors.white,
  ), // InputDecoration
  validator: (value) {
    if (value == null || value.length < 6) {
      return 'Password must be at least 6 characters';
    }
    return null;
  },
```

```dart
TextFormField(
  controller: emailController,
  decoration: InputDecoration(
    labelText: 'Email',
    labelStyle: AppTextStyles.placeholderText,
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(10),
    ), // OutlineInputBorder
    filled: true,
    fillColor: Colors.white,
  ), // InputDecoration
  validator: (value) {
    if (value == null || value.isEmpty) {
      return 'Email is required';
    } else if (!RegExp(r'^[^@]+@[^@]+\.[^@]+').hasMatch(value)) {
      return 'Please enter a valid email';
    }
    return null;
  },
```

```
  decoration: InputDecoration(
    labelText: 'Role',
    labelStyle: AppTextStyles.placeholderText,
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(10),
    ), // OutlineInputBorder
    filled: true,
    fillColor: Colors.white,
  ), // InputDecoration
  onChanged: (value) {
    setState(() {
      selectedRole = value;
    });
  },
  validator: (value) =>
      value == null ? 'Please select a role' : null,
), // DropdownButtonFormField
```

```
TextFormField(
  controller: contactInfoController,
  decoration: InputDecoration(
    labelText: 'Phone number',
    labelStyle: TextStyle(color: Colors.grey), // Replace with your
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(10),
    ), // OutlineInputBorder
    filled: true,
    fillColor: Colors.white,
  ), // InputDecoration
  validator: (value) {
    if (value == null || value.isEmpty) {
      return 'Phone number is required';
    } else if (!phoneRegex.hasMatch(value)) {
      return 'Please enter a valid phone number (e.g. +60111088888)';
    }
    return null;
  },
```

```
      // Ensure that the first character is always '+'
      if (!value.startsWith('+60')) {
        contactInfoController.text = '+60';
        contactInfoController.selection = TextSelection.fromPosition(
          TextPosition(offset: contactInfoController.text.length),
        ); // TextSelection.fromPosition
      }
    },
  ), // TextFormField
```

Figure 2.6.12: Validate text field while register

In this part while the user registering as new user it will require the user to input their username or else it will show an error message “Username is required”, password is also required and need to be more than 6 words if it is less than 6 or empty field it will show an error message “Password must at least 6 characters.”. After that, email of user input will also check is that the email address is a valid email or not. Other than that, it will also require user to select their role. Finally, the phone number will auto be generated the (+60) which is the country code phone number then the user only input their phone number after the 0 else it will show message, Please enter a valid phone number (e.g. +601110888888).

2.7 Users Permission

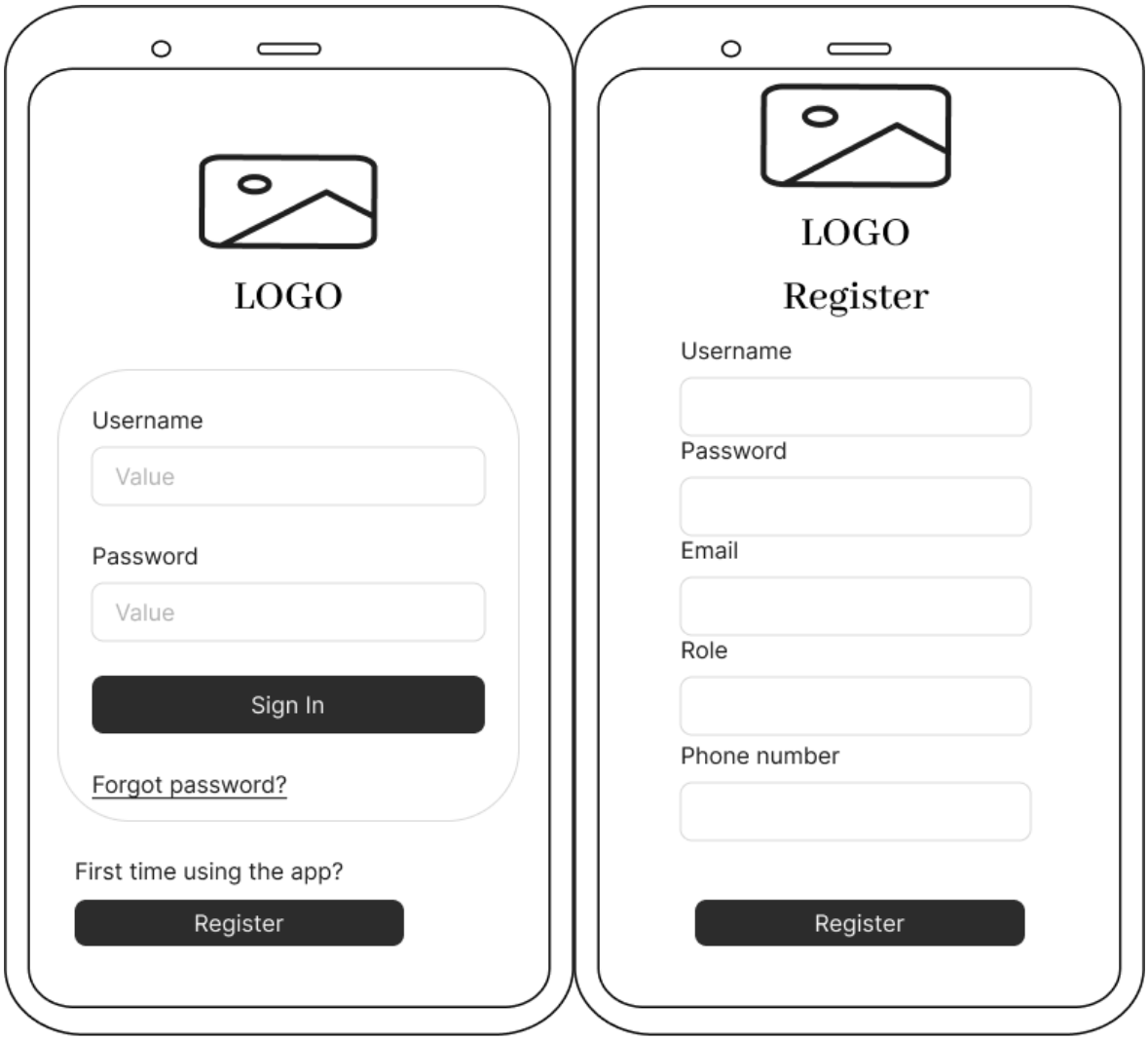
```
// Get user role
String userRole = _userRepository.getUserRole(userDoc);

// Navigate to the correct dashboard based on role
if (userRole == 'Caregiver') {
  Navigator.pushReplacement(
    context,
    MaterialPageRoute(
      builder: (context) => CaregiverDashboardPage(userID: userID),
    ), // MaterialPageRoute
  );
} else if (userRole == 'Elderly') {
  Navigator.pushReplacement(
    context,
    MaterialPageRoute(
      builder: (context) => ElderlyDashboardPage(userID: userID),
    ), // MaterialPageRoute
  );
} else if (userRole == 'Healthcare Provider') {
  Navigator.pushReplacement(
    context,
    MaterialPageRoute(
      builder: (context) => HealthcareProviderDashboardPage(userID: userID),
    ), // MaterialPageRoute
  );
}
```

Figure 3.7.1: Part of code snippet of Sign In function.

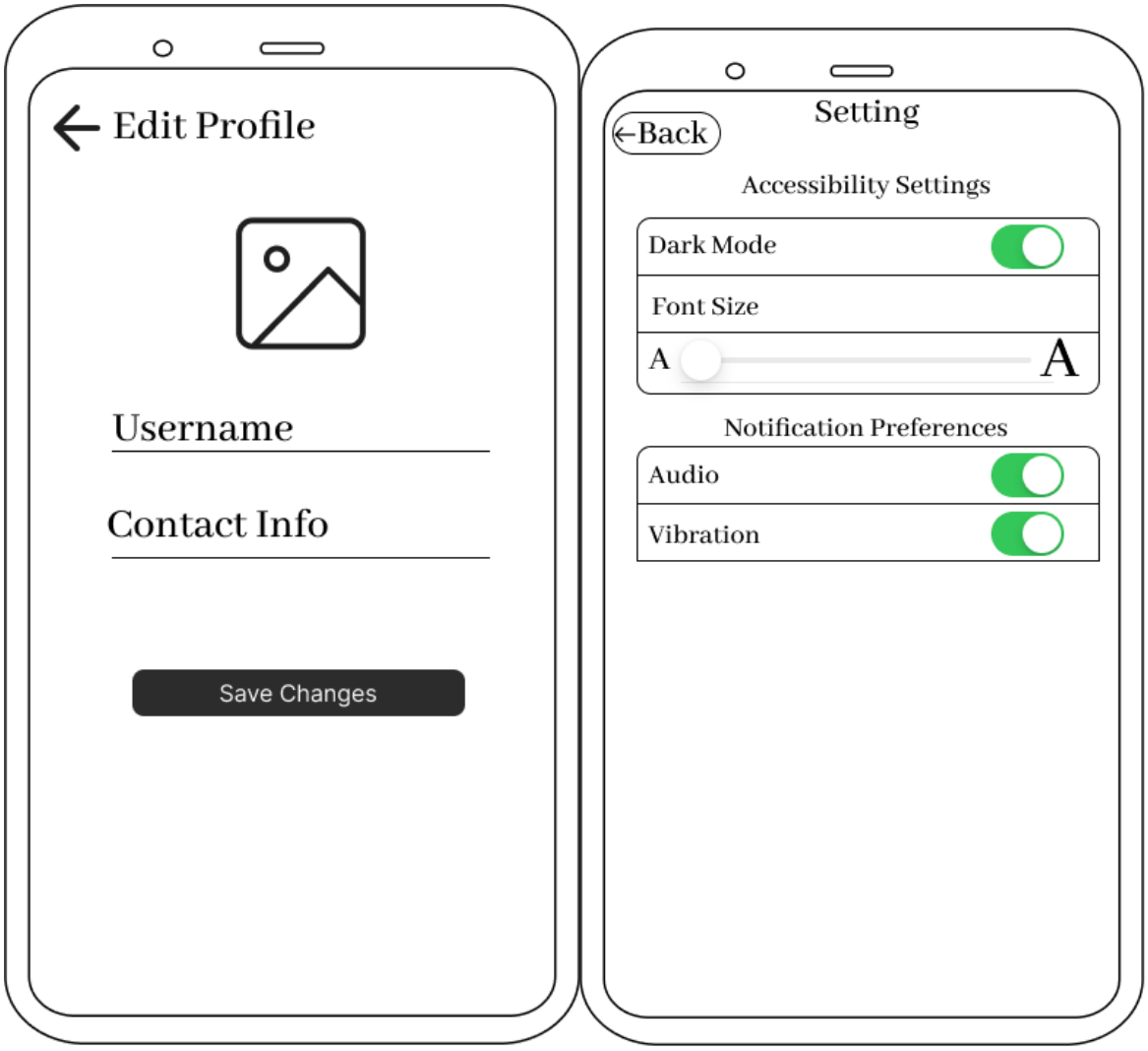
The “signIn” function retrieves the user’s role, which are caregiver, elderly and healthcare provider through “getUserRole” function from user repository class. Based on the retrieved role, the app navigates the user to the corresponding dashboard. This navigation setup is indirectly enforcing permissions because users are taken to different interfaces with unique features based on their roles. Each dashboard page would typically present or restrict access to certain functions, providing a level of role-based permissions within the UI.

2.8 Wireframe

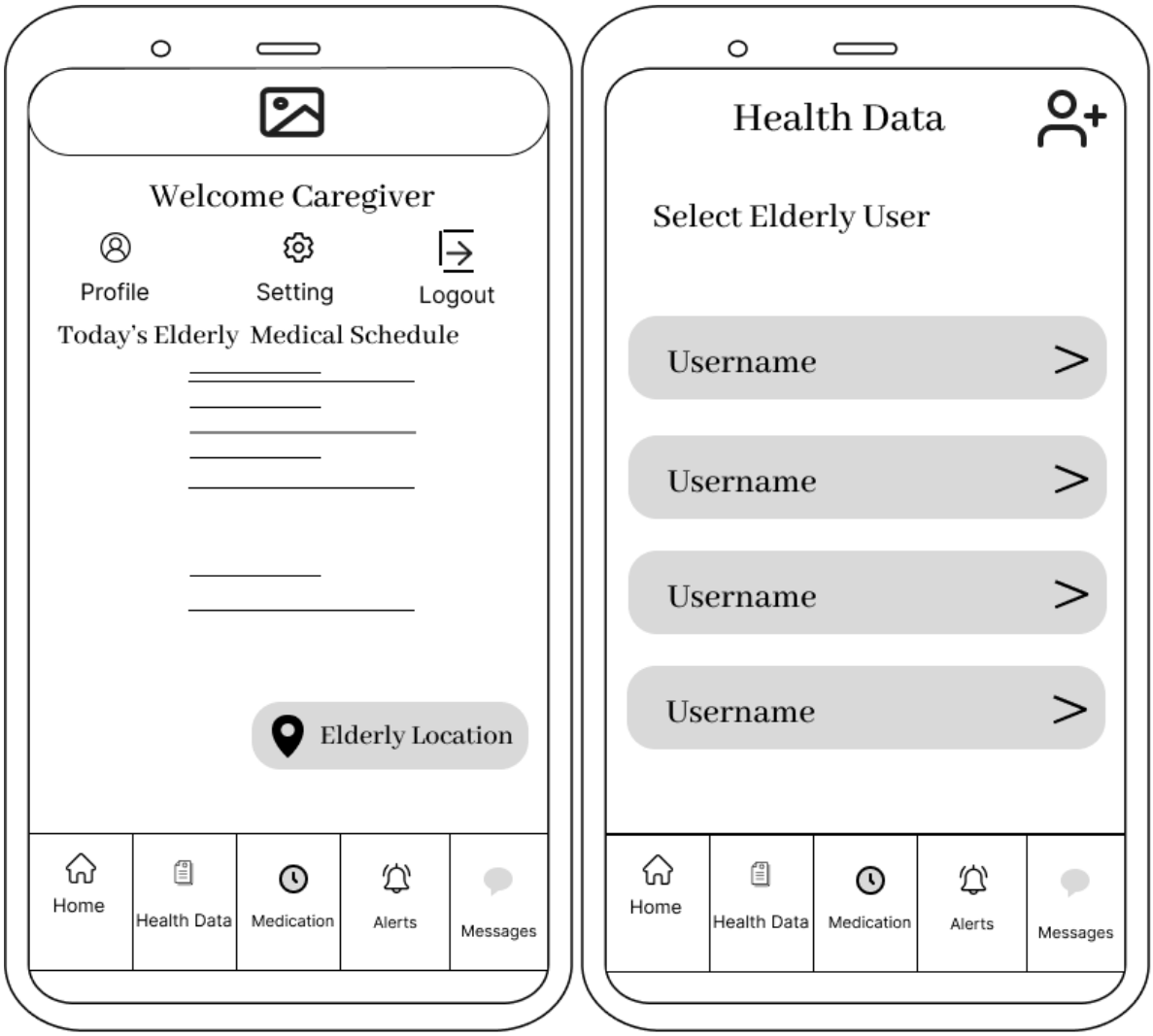
Universal

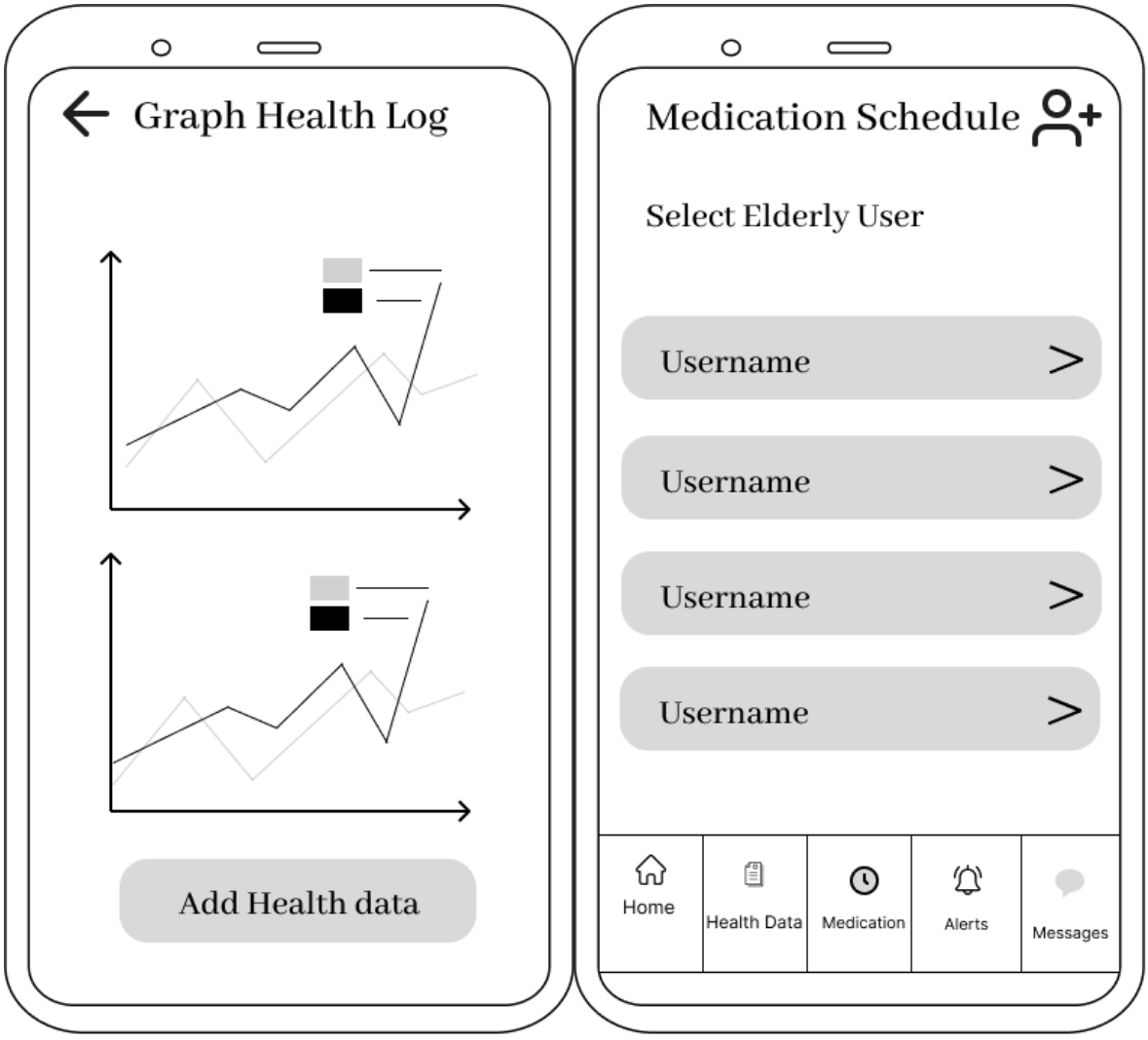


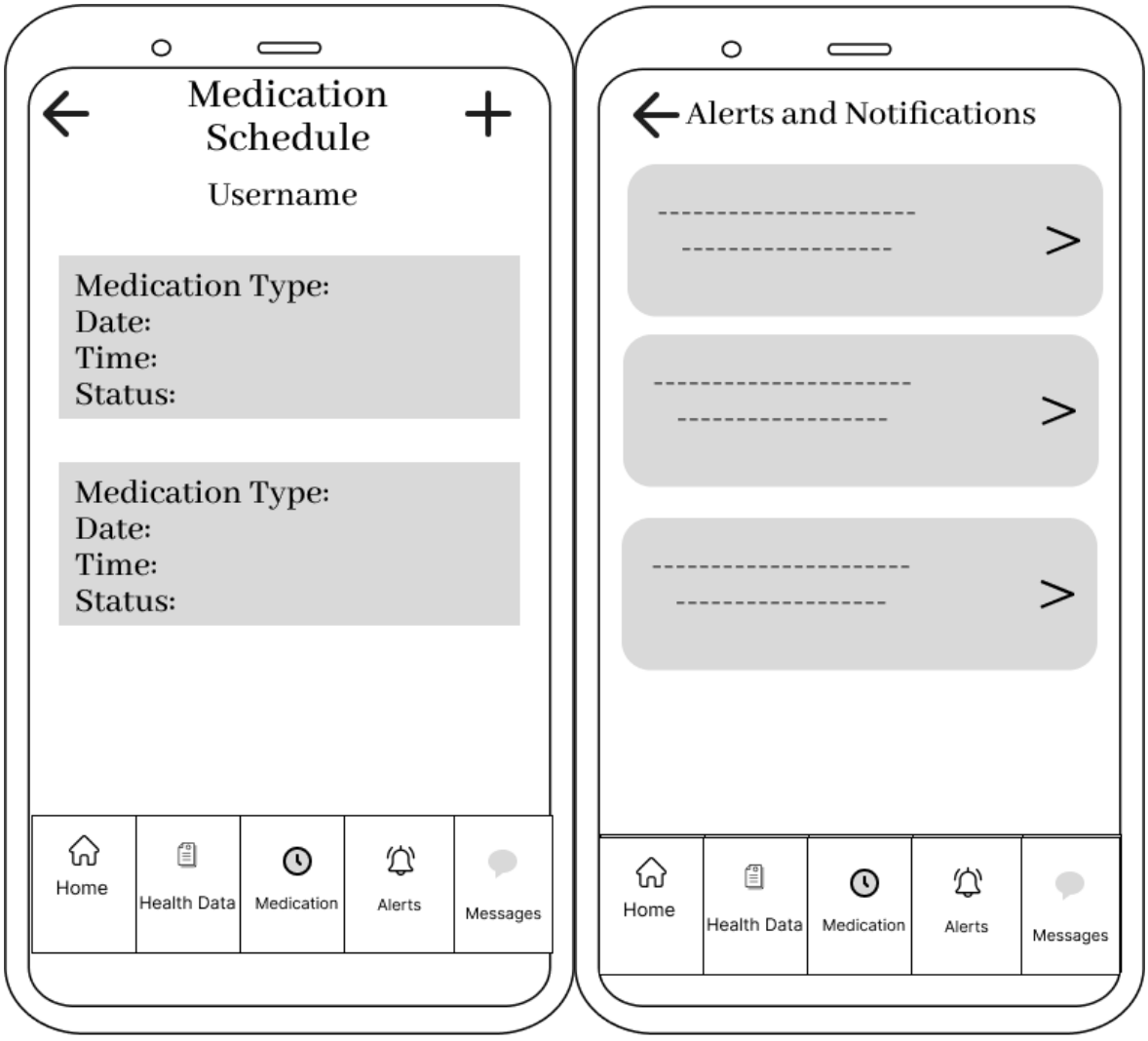

LOGO
Forgot Password
Email
Reset Password
User Profile
Username
Email
Edit Profile

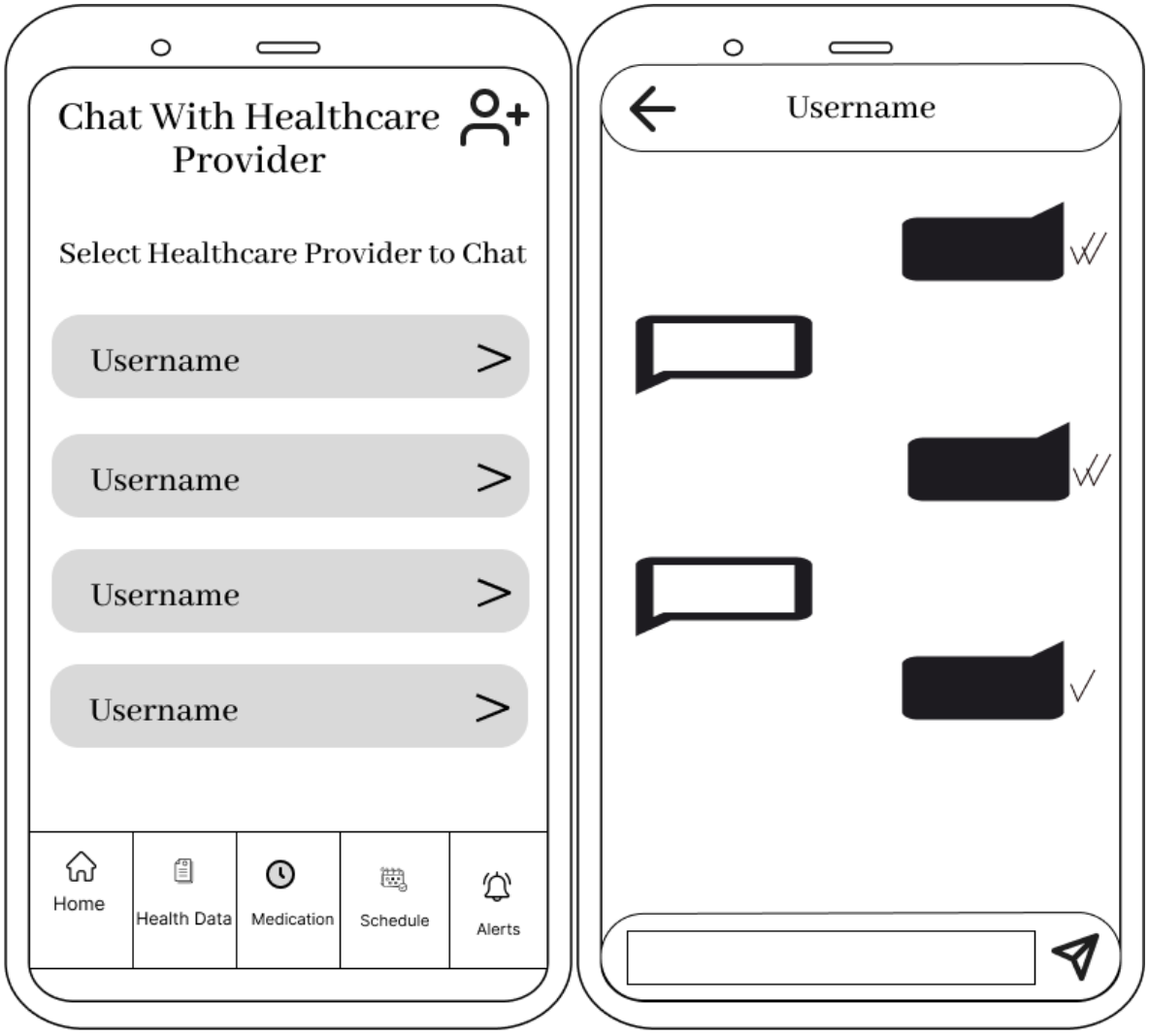

Edit Profile
Username
Contact Info
Save Changes
Setting
Back
Accessibility Settings
Dark Mode
Font Size
A
A
Notification Preferences
Audio
Vibration

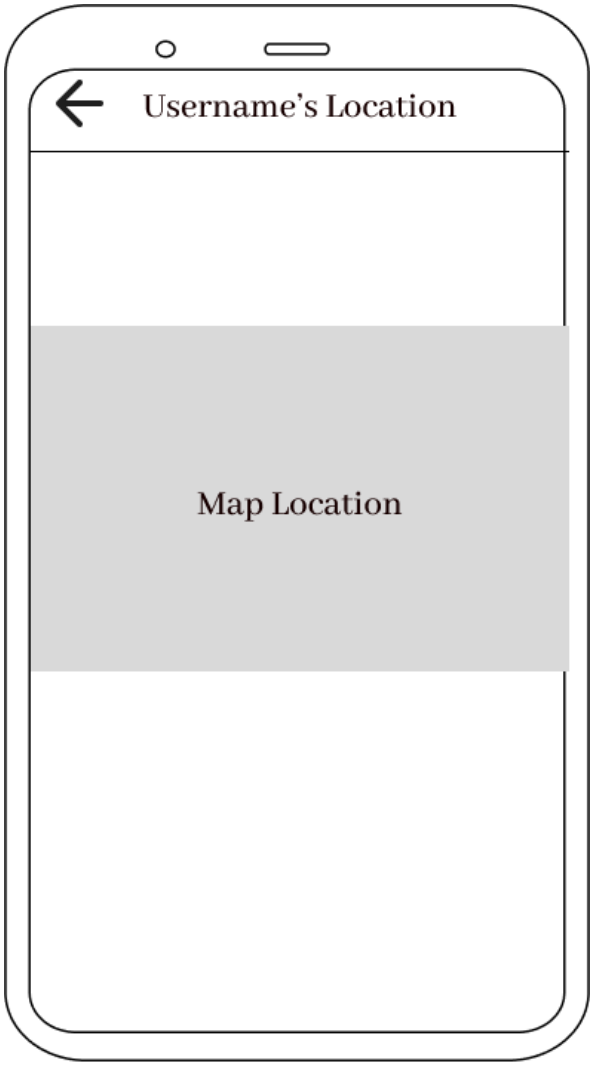
Caregivers


Welcome Caregiver
Profile
Setting
Logout
Today's Elderly Medical Schedule
Elderly Location
Home
Health Data
Medication
Alerts
Messages
Health Data
Select Elderly User
Username
Username
Username
Username
Home
Health Data
Medication
Alerts
Messages

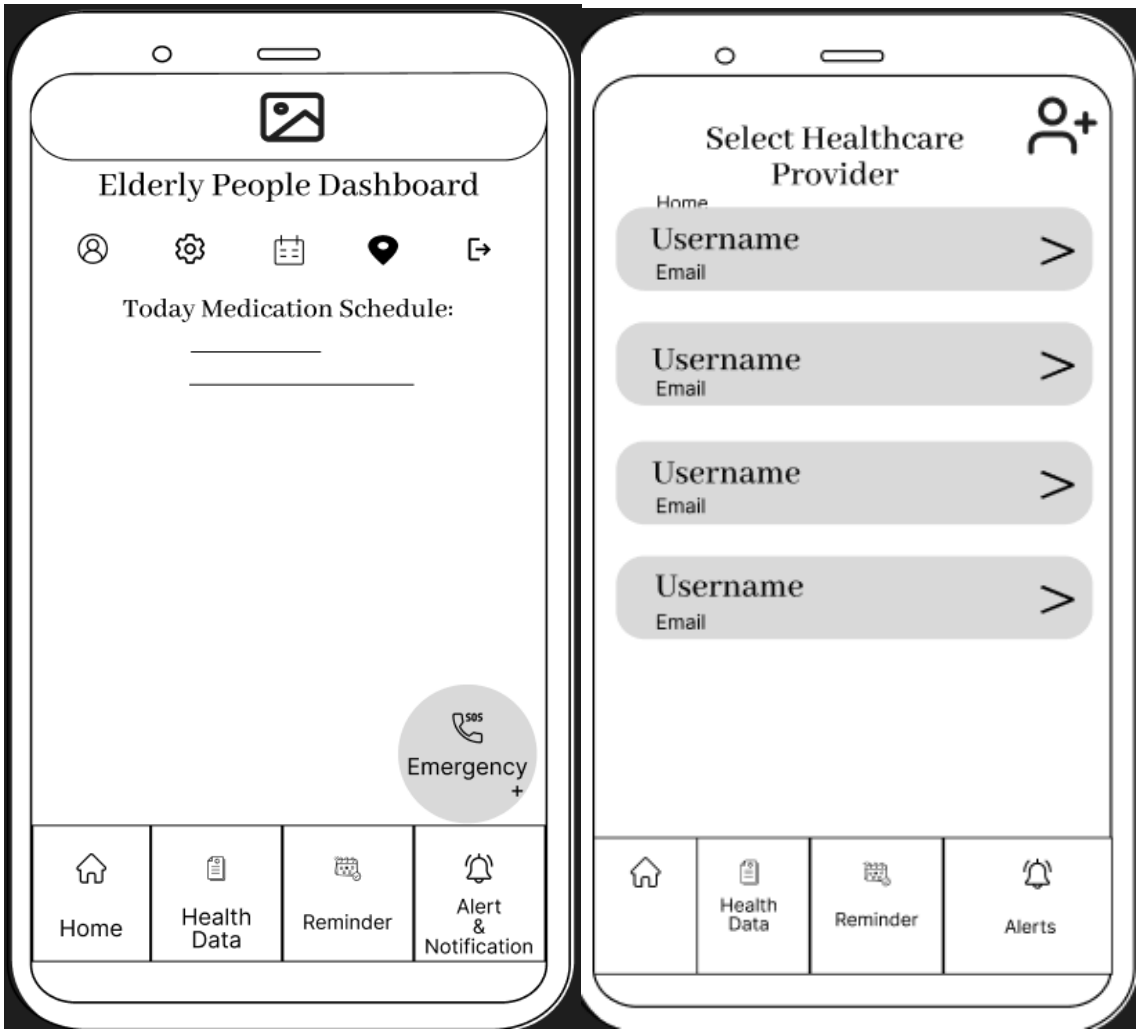

Graph Health Log
Add Health data
Medication Schedule
Select Elderly User
Username
Username
Username
Username
Home
Health Data
Medication
Alerts
Messages

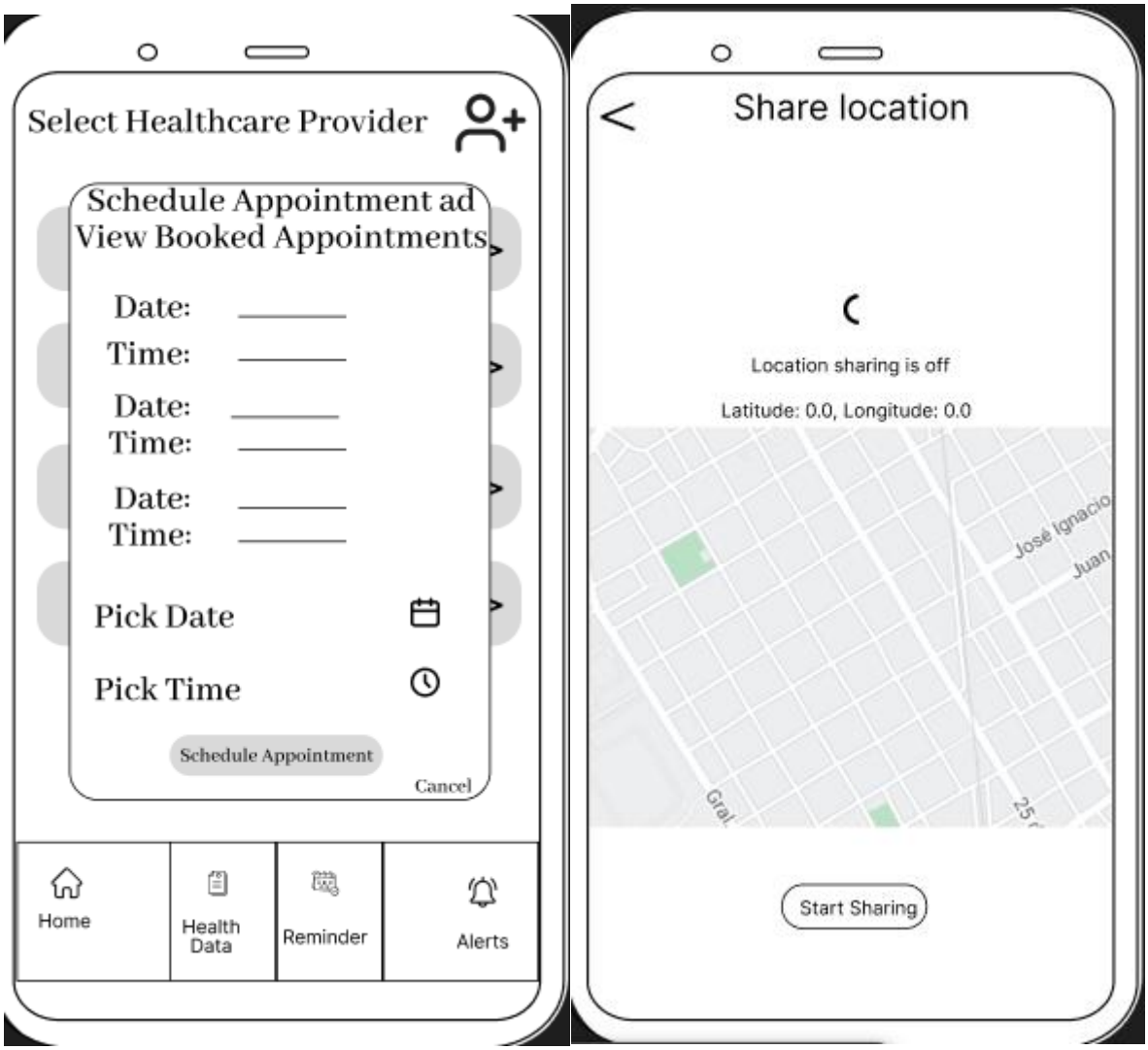

Medication Schedule
Username
Medication Type:
Date:
Time:
Status:
Medication Type:
Date:
Time:
Status:
Home
Health Data
Medication
Alerts
Messages
Alerts and Notifications
Home
Health Data
Medication
Alerts
Messages

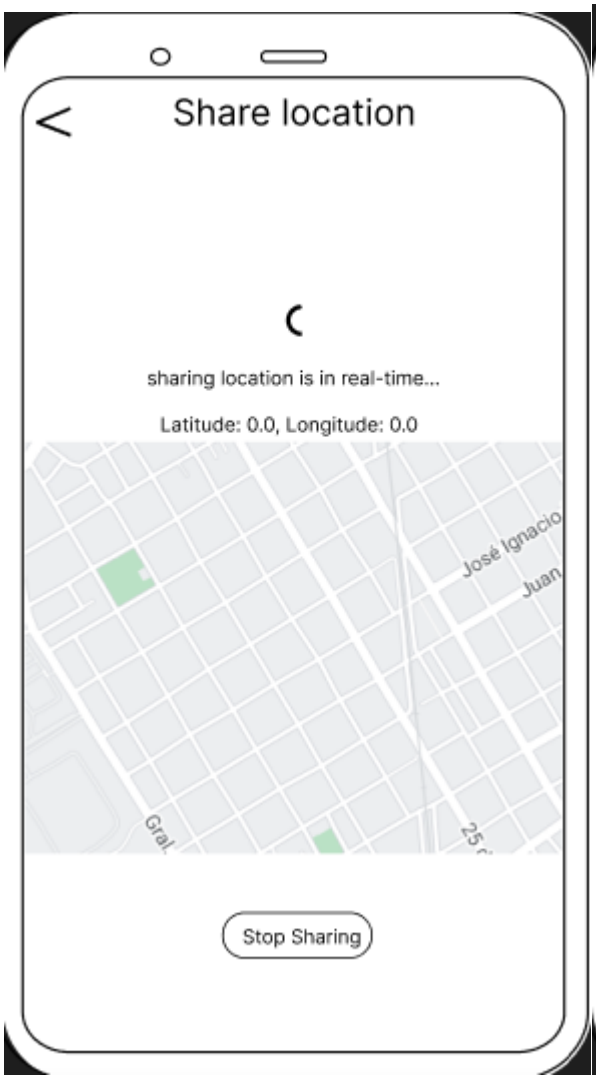

Chat With Healthcare Provider
Select Healthcare Provider to Chat
Username
Username
Username
Username
Home
Health Data
Medication
Schedule
Alerts
Username

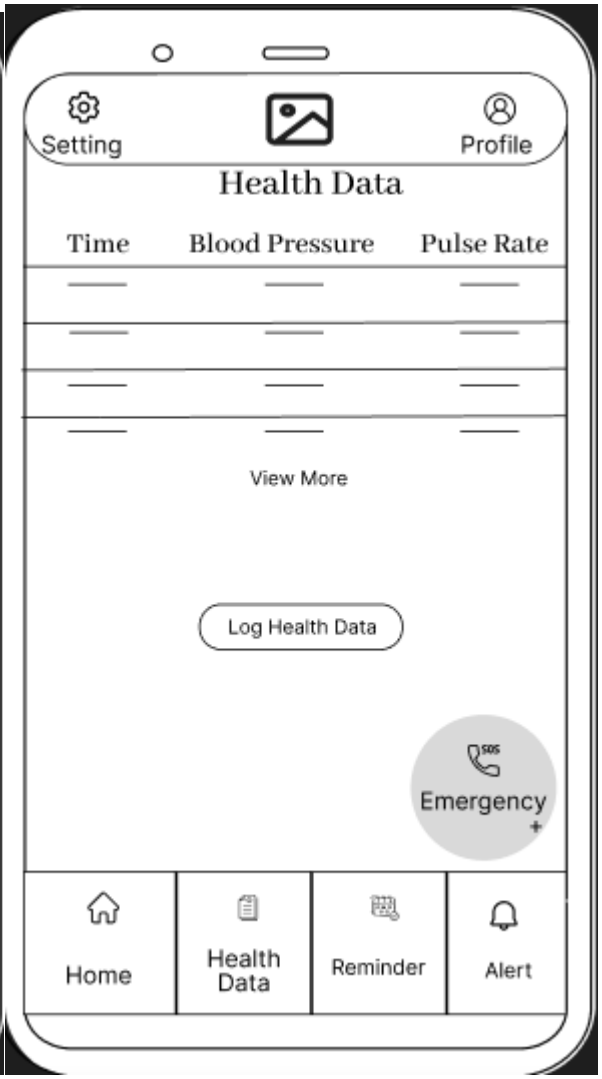

Username's Location
Map Location

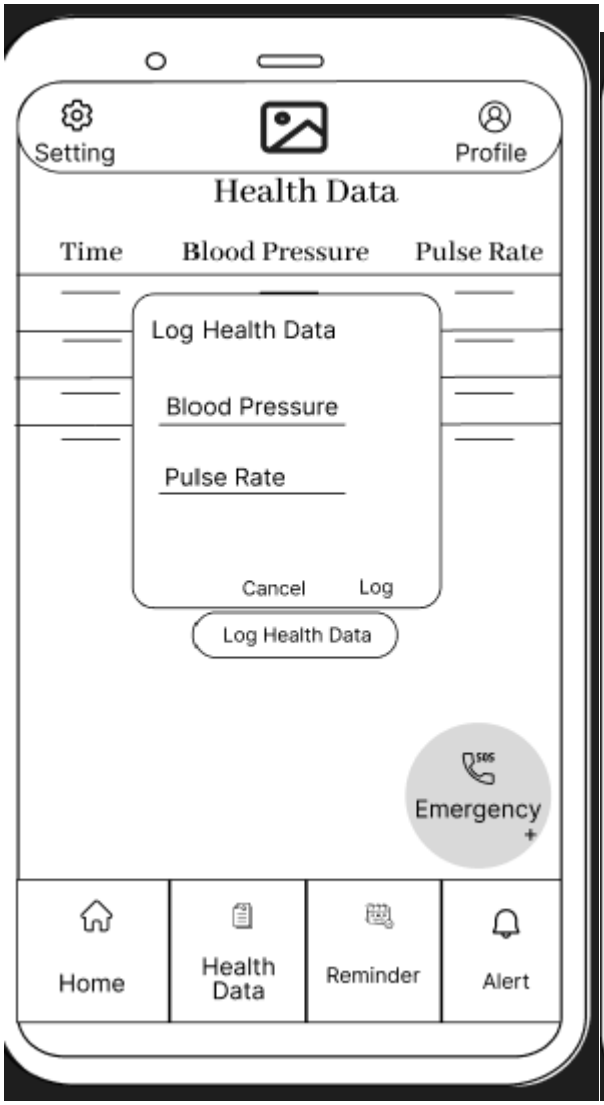
Elderly Users

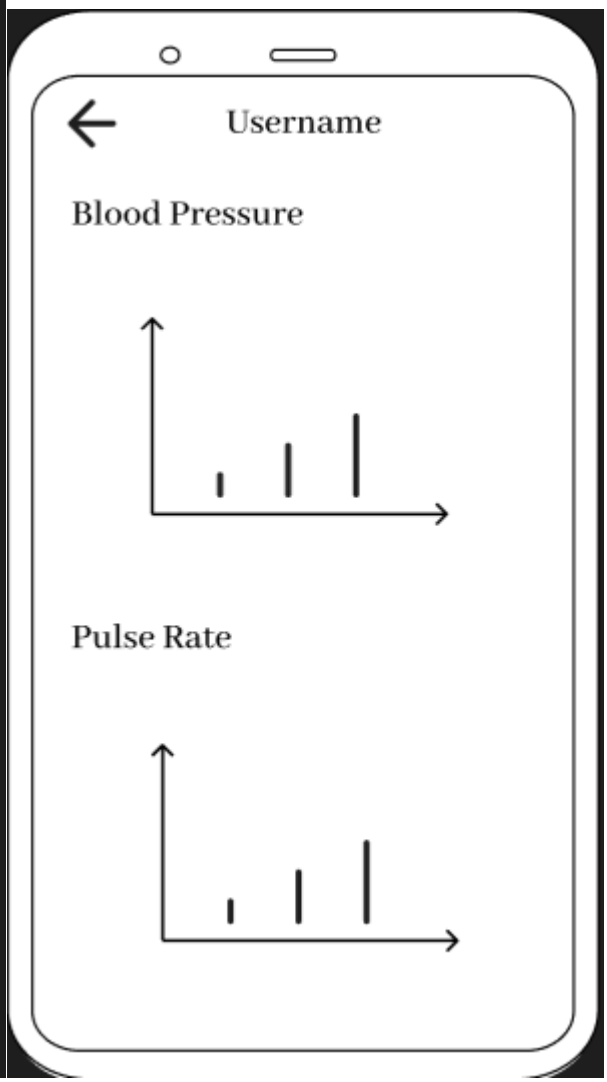


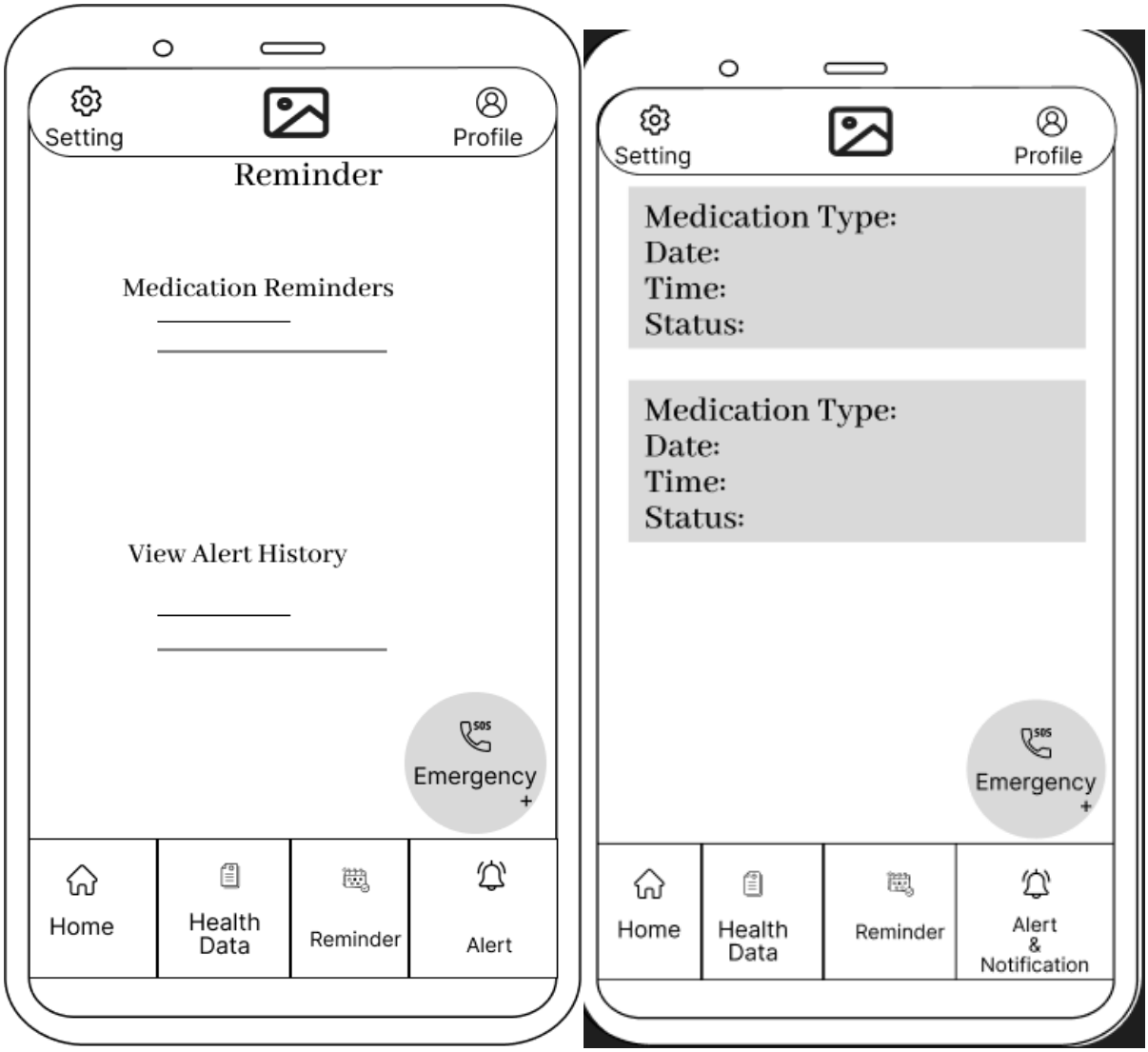

Select Healthcare Provider
Schedule Appointment ad
View Booked Appointments
Date:
Time:
Date:
Time:
Date:
Time:
Pick Date
Pick Time
Schedule Appointment
Cancel
Home
Health Data
Reminder
Alerts
Share location
Location sharing is off
Latitude: 0.0, Longitude: 0.0
Start Sharing


Share location
sharing location is in real-time...
Latitude: 0.0, Longitude: 0.0
Stop Sharing

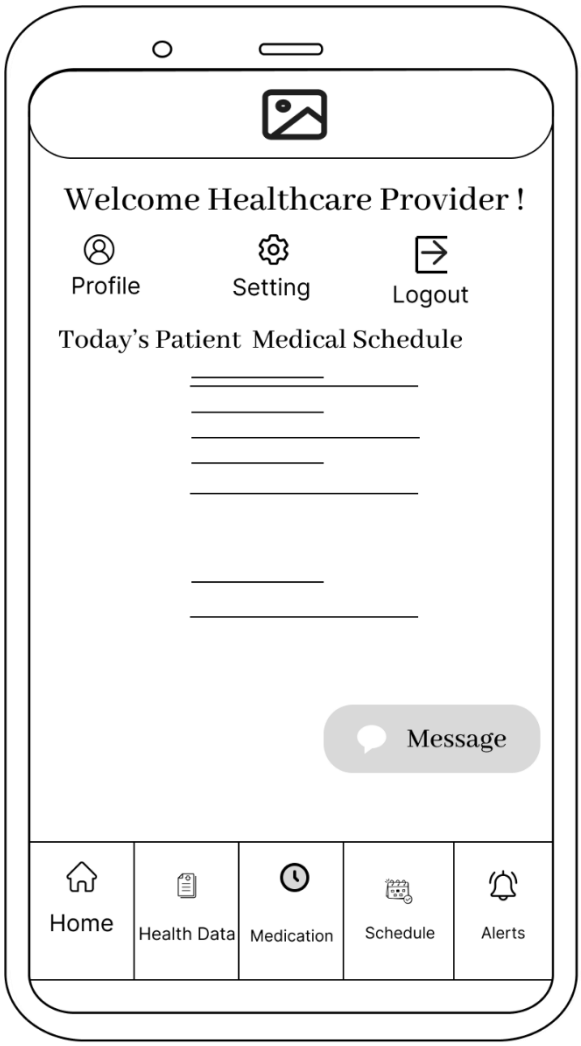

Setting
Profile
Health Data
Time
Blood Pressure
Pulse Rate
View More
Log Health Data
Emergency
Home
Health Data
Reminder
Alert

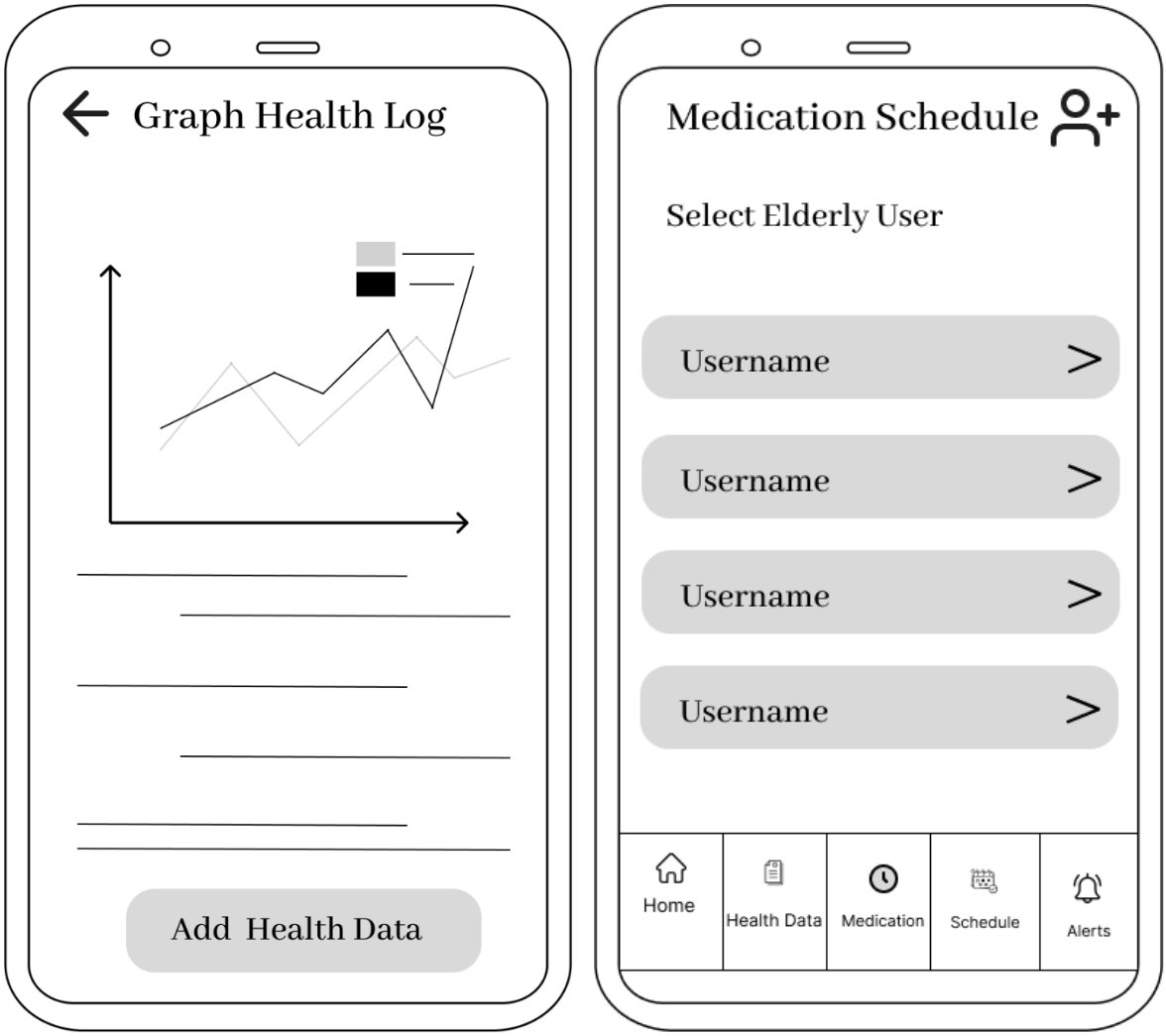

Setting
Profile
Health Data
Time
Blood Pressure
Pulse Rate
Log Health Data
Blood Pressure
Pulse Rate
Cancel
Log
Log Health Data
Emergency
Home
Health Data
Reminder
Alert

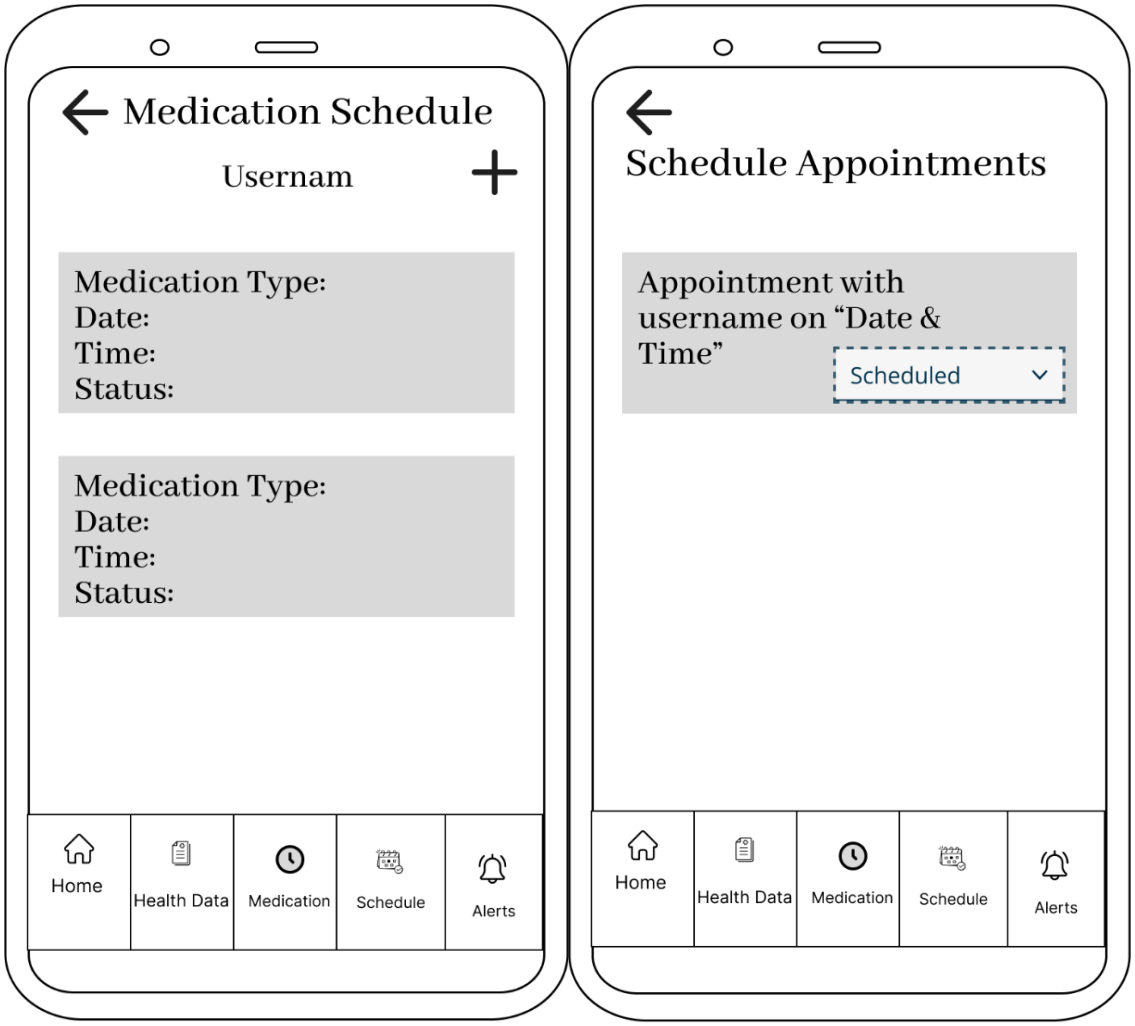

Username
Blood Pressure
Pulse Rate

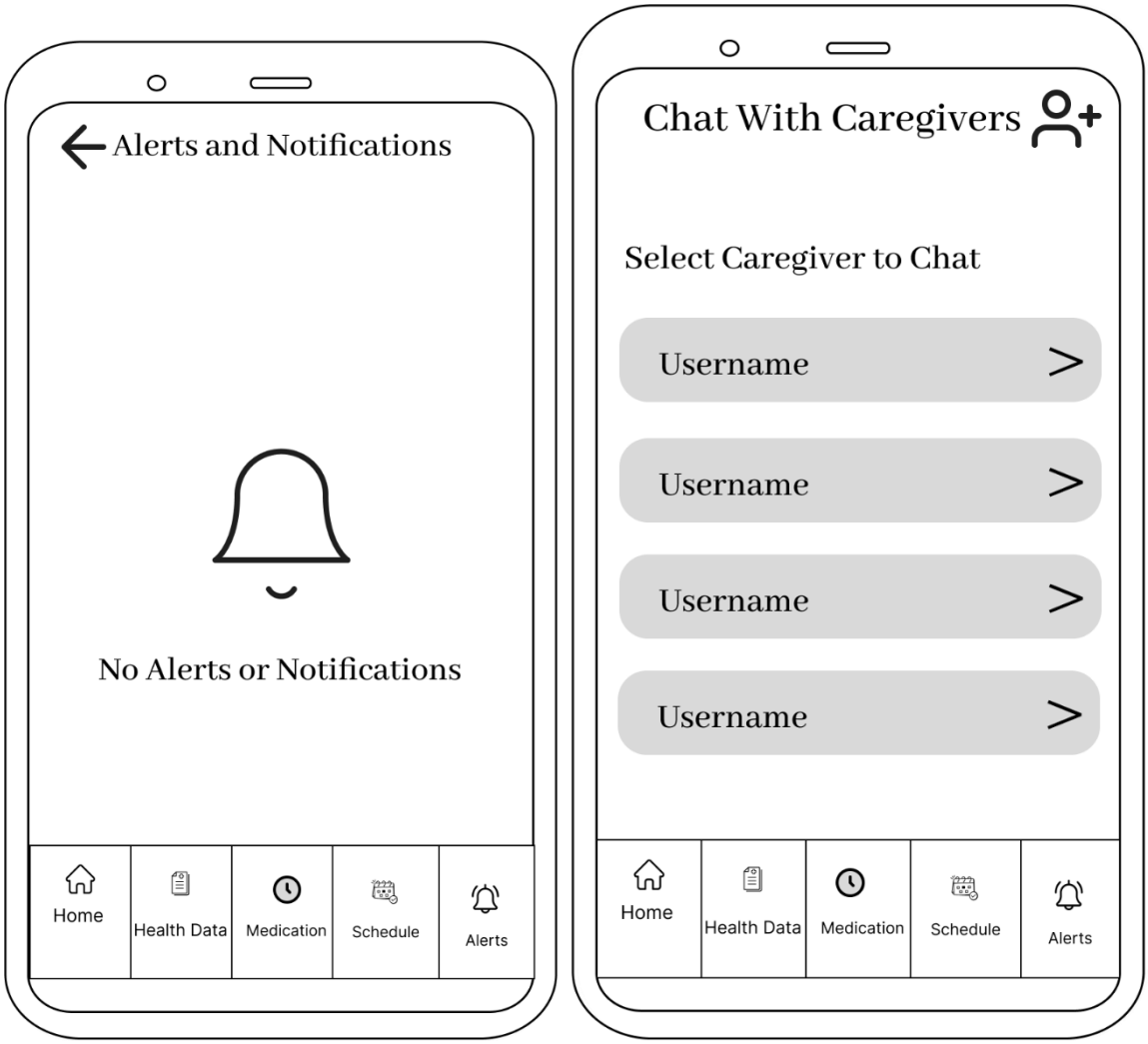

Setting
Profile
Reminder
Medication Reminders
View Alert History
Emergency
+
Home
Health Data
Reminder
Alert
Setting
Profile
Medication Type:
Date:
Time:
Status:
Medication Type:
Date:
Time:
Status:
Emergency
+
Home
Health Data
Reminder
Alert & Notification

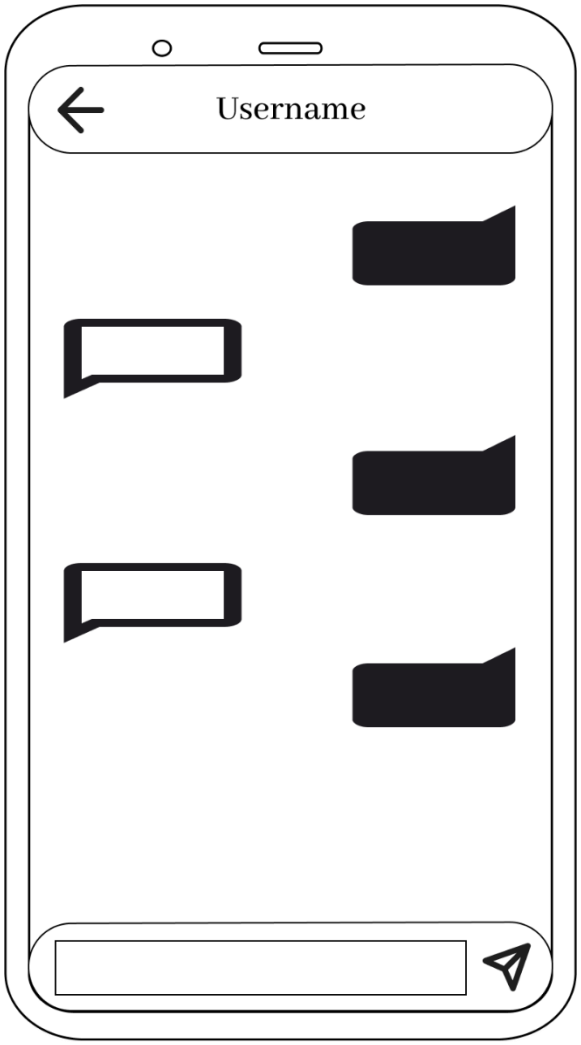
Healthcare Providers




Graph Health Log
Add Health Data
Medication Schedule
Select Elderly User
Username
Username
Username
Username
Home
Health Data
Medication
Schedule
Alerts


Medication Schedule
Usernam
Medication Type:
Date:
Time:
Status:
Medication Type:
Date:
Time:
Status:
Home
Health Data
Medication
Schedule
Alerts
Schedule Appointments
Appointment with username on “Date & Time”
Scheduled
Home
Health Data
Medication
Schedule
Alerts


Alerts and Notifications
No Alerts or Notifications
Home
Health Data
Medication
Schedule
Alerts
Chat With Caregivers
Select Caregiver to Chat
Username
Username
Username
Username
Home
Health Data
Medication
Schedule
Alerts


Username

3.0 User Manual

3.1 Universal Screens

3.1.1 Login Screen

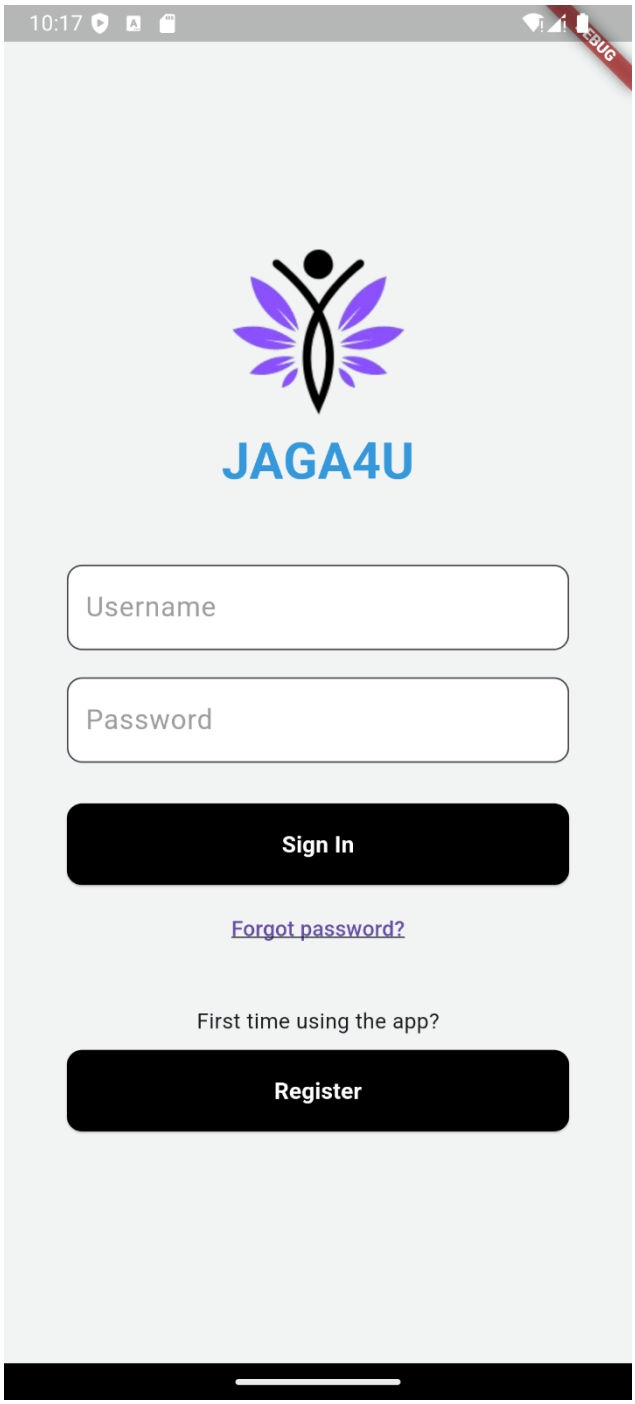


Figure 3.1.1.1: Login Screen

When users open the app, login screen is shown as the home screen. Make sure to enter valid username and password in order to log in to the app successfully. Click on "Sign In" button to log in.

If user is a first-time user, click on "Register" button to register an account and then proceed to login after successfully registered an account.

If users forgot about their password, click on the underlined "Forgot password" text. Users will be directed to a screen to enter their email. The reset password email will be sent to the email users provided.

3.1.2 Register Screen

Figure 3.1.2.1: Register Screen

In register screen, users have to fill up all the information. Missing one would cause registration failure. The role field will allow users to select from 3 fixed value, which are caregiver, elderly and healthcare provider.

For the phone number field, the "+60" is the default value in the field.

For users' information, example of valid phone number here is as following:

- +60126661112
- +60146678346
- +60123456789

3.1.3 Profile Screen

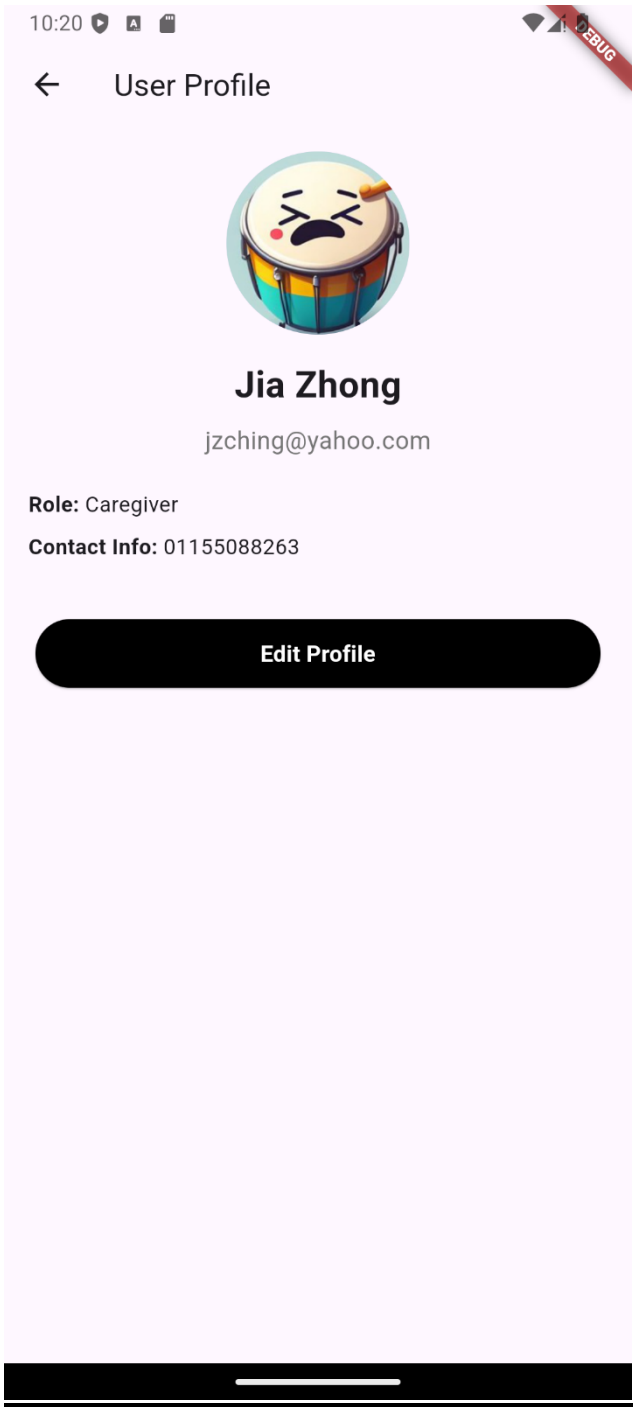


Figure 3.1.3.1: Profile Screen

When users click on the profile icon button on their respective dashboard, the profile screen will be displayed.

If users wish to edit profile, click on the black “Edit Profile” button.

3.1.4 Edit Profile Screen

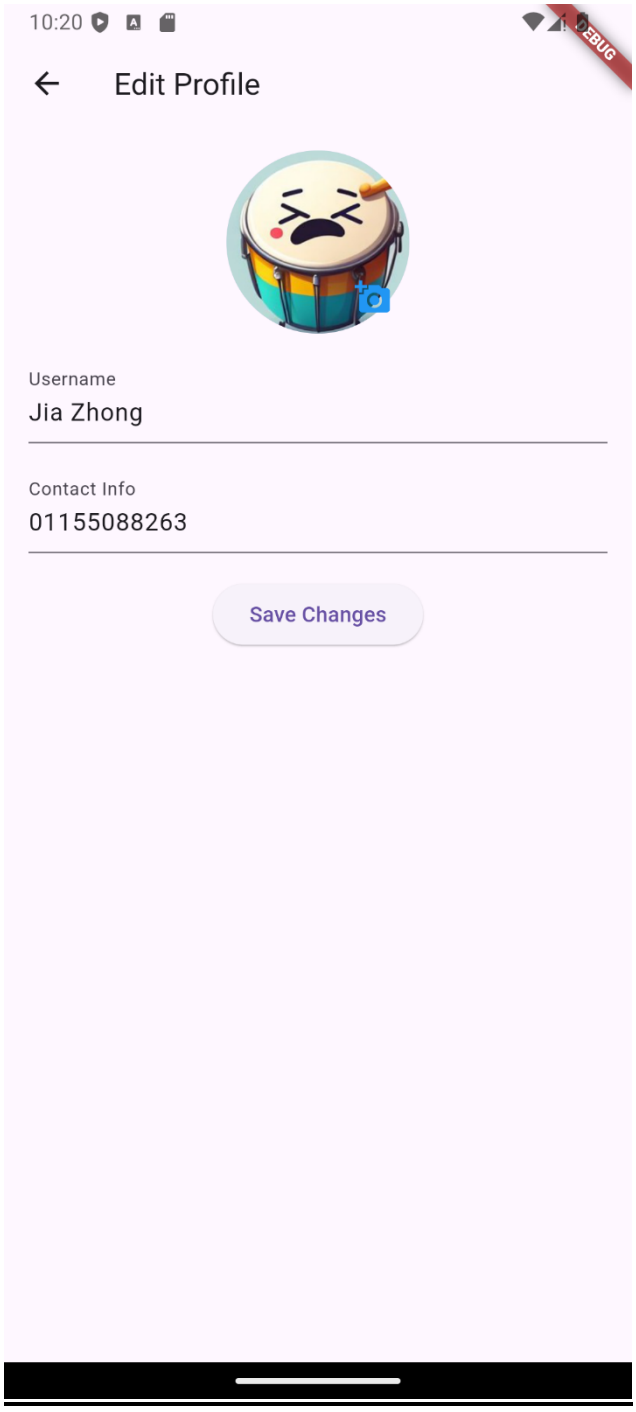


Figure 3.1.4.1: Edit Profile Screen

Users can click on the line fields to edit their details such as username and contact info. Once changes completed, click on the “Save Changes” button. This will bring users back to profile screen and the user profile details should be updated.

3.1.5 Setting Screen

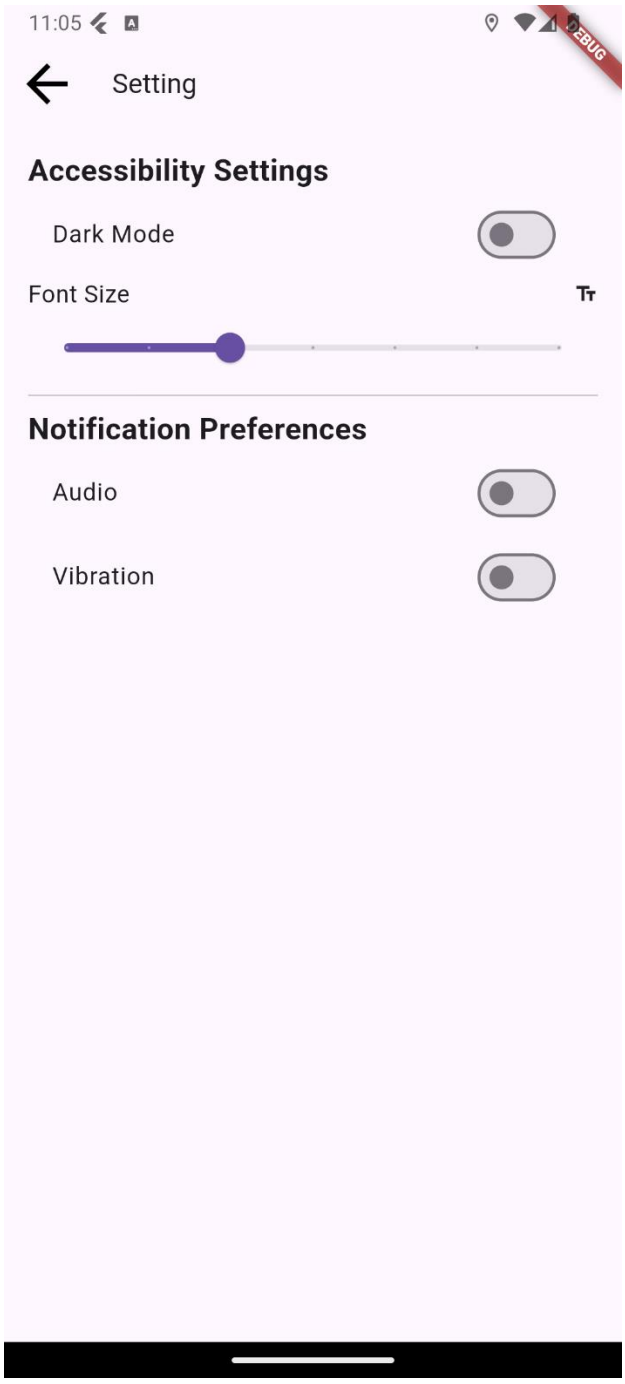


Figure 3.1.5.1: Setting Screen

When user click on the setting it will navigate to the setting page that provide user to toggle dark mode, change to font size, turn off the audio of JAGA4U and Vibration.

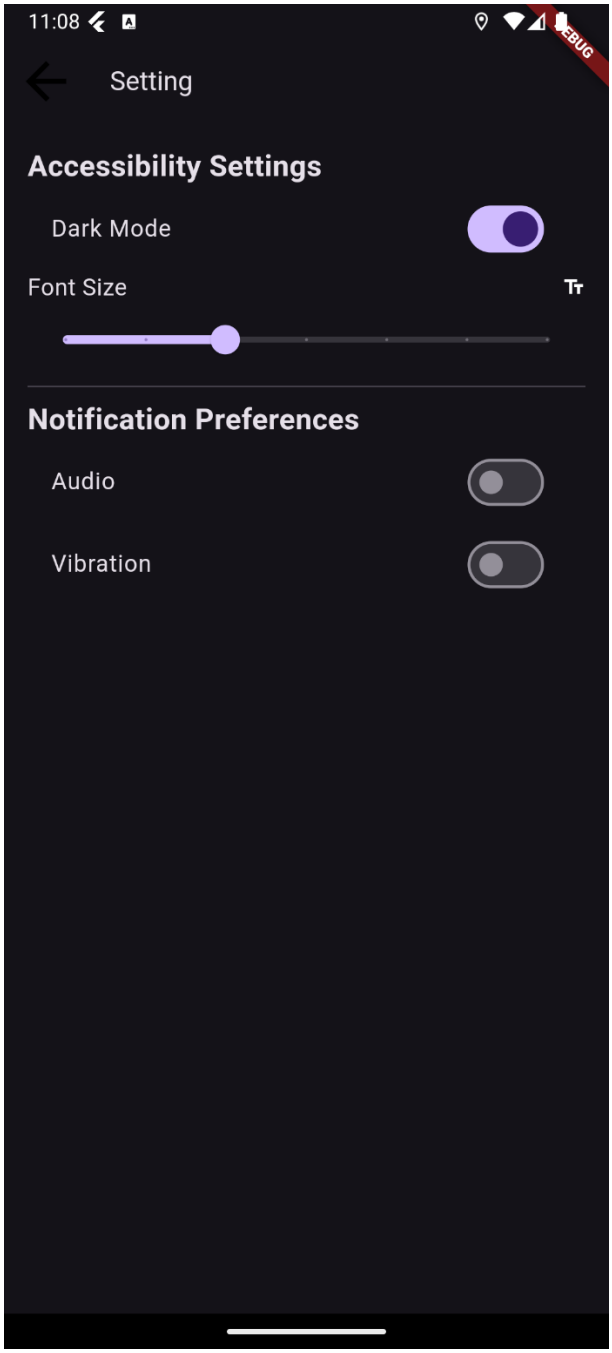


Figure 3.1.5.2: Dark Mode Activate

When the user activates the dark mode toggle, the entire app will undergo a smooth transition, transforming its colour theme to embrace a rich, dark aesthetic. This shift will be consistently applied across all pages, ensuring a cohesive and visually comfortable experience throughout the app. Every screen will instantly reflect the dark mode, creating a unified, immersive environment that aligns with the user's preferences and enhances usability in low-light settings.

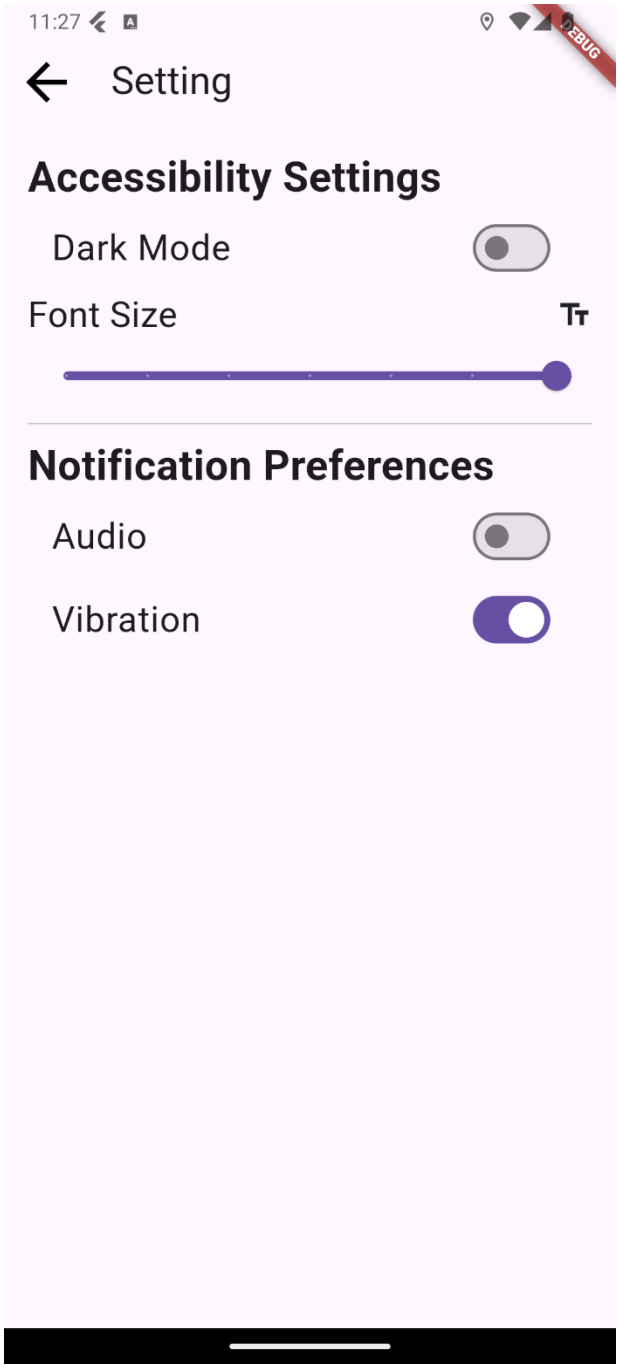


Figure 3.1.5.3: Font size changing in setting screen

When the user changes the slider of font size, the entire app will undergo a smooth transition, transforming its font size. This shift will be consistently applied across all pages, ensuring a cohesive and visually comfortable experience throughout the app. Every screen will instantly change the font size. The purpose is to let the elderly user have a better experience while using the mobile application.

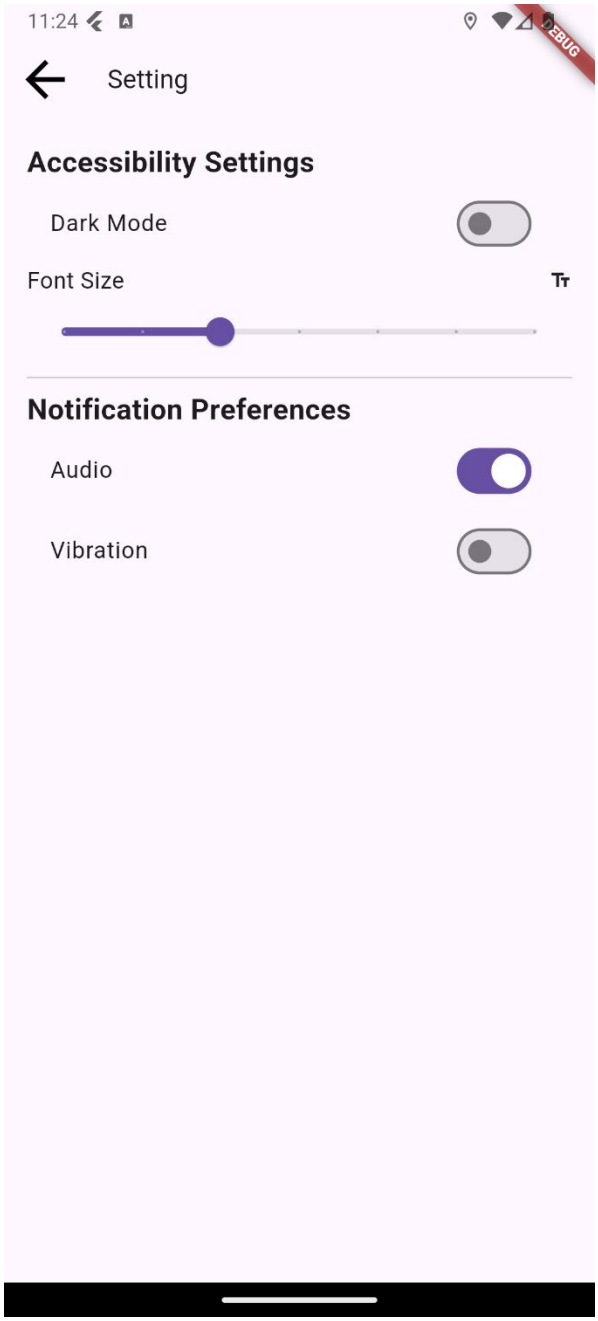


Figure 3.1.5.4: Audio Button Toggle in setting screen

When the user toggles the audio button on it will keep all the audio sound available. Otherwise, it will keep all silent.

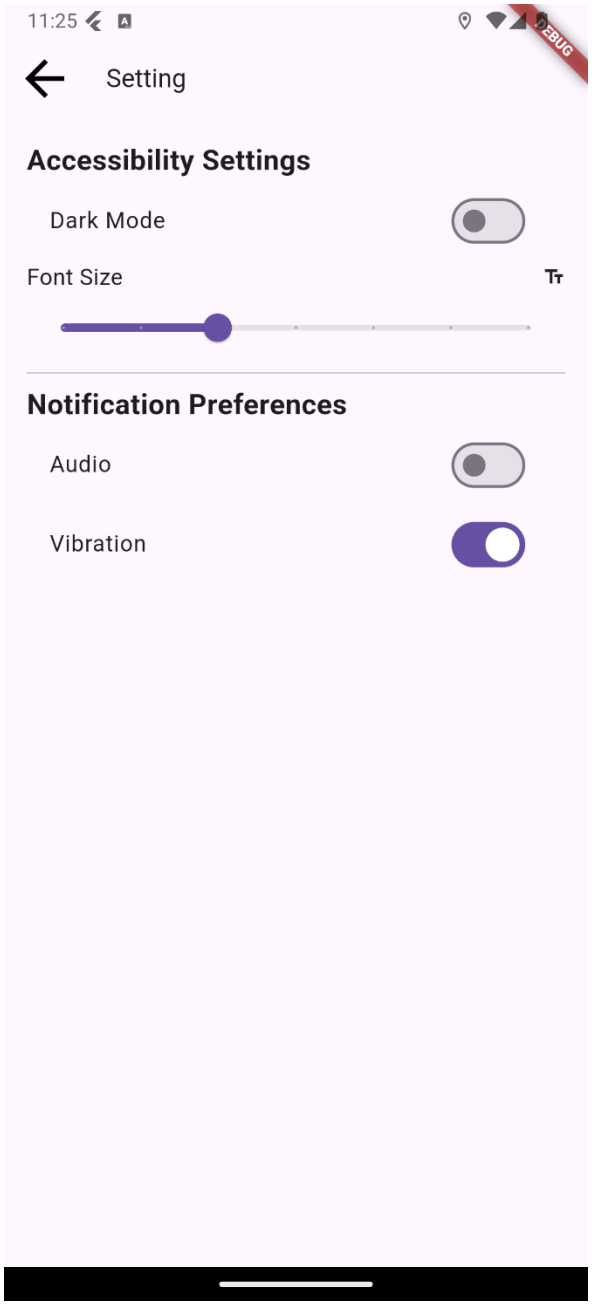


Figure 3.1.5.5: Vibration Button Toggle in setting screen

When the user toggles the vibration button on it will have a vibration feedback to user then all the app vibration will keep vibration.

3.2 Caregivers

3.2.1 Caregivers Dashboard

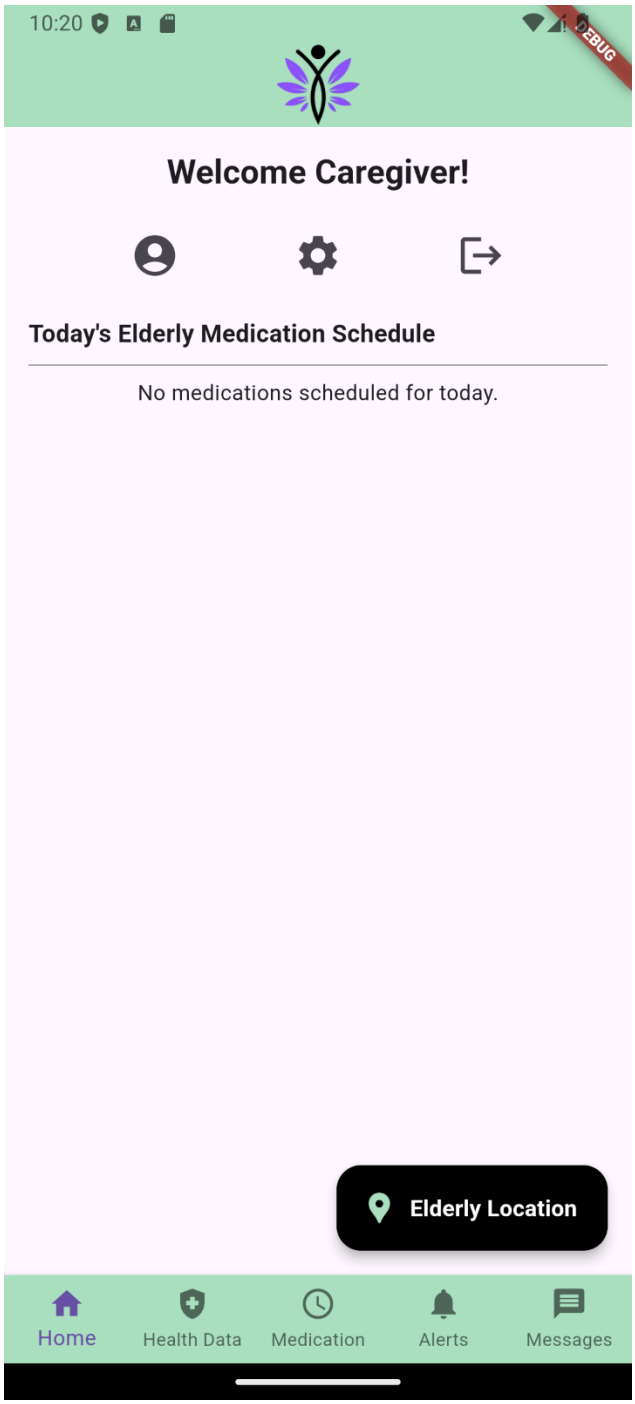


Figure 3.2.1.1: Caregiver Dashboard Screen

If user logged in as caregiver, this caregiver dashboard page will be displayed, it is the home screen for caregivers. Users can click on the profile icon button to view their profile. The setting icon button allows user to view and adjust the settings. The rightmost icon button on the upper part of the screen allows users to log out of their account and return to login screen. The "Elderly Location" button allows caregivers will allows caregivers to select their associated elderly who they want to view the location.

Bottom Navigation Bar:

Home: Caregiver Dashboard Page

Health Data: View Health Data of elderly in table or chart.

Medication: View Medication Schedule of Elderly or Add Medication Schedule.

Alerts: View Alerts and Notifications History

Messages: Message Healthcare Providers

3.2.2 Select Elderly Screen

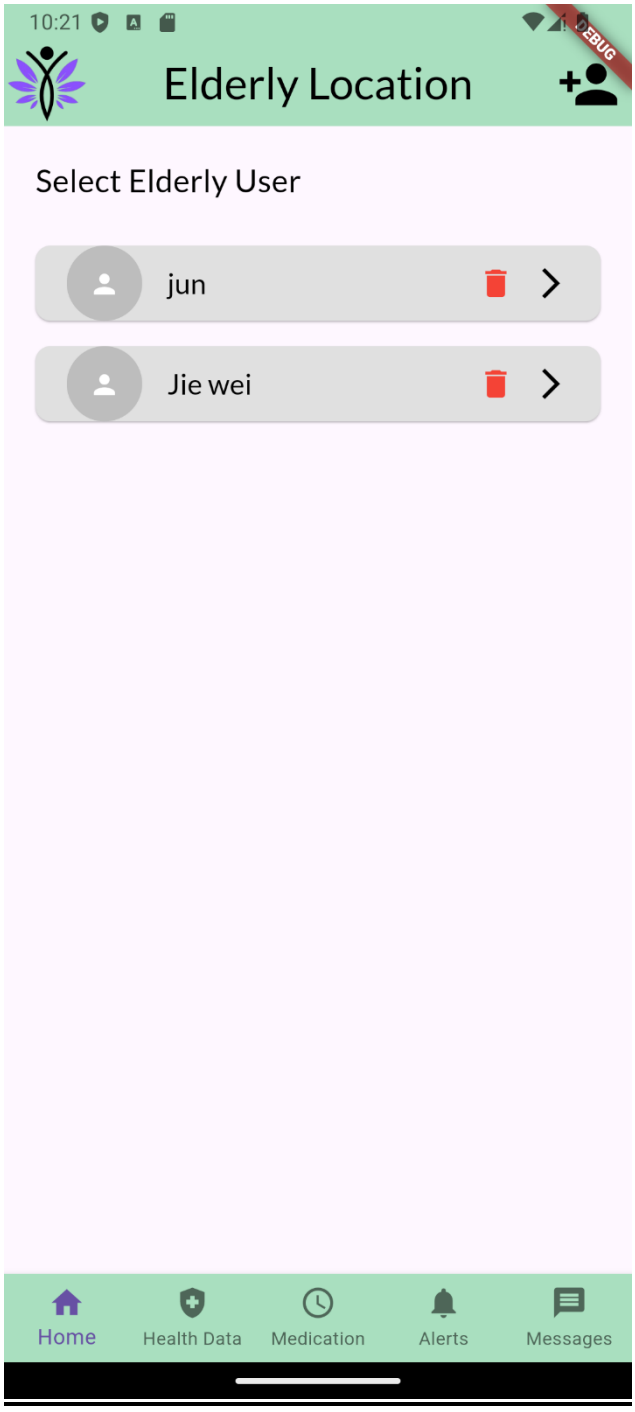


Figure 3.2.2.1: Select Elderly Screen

This select elderly screen will be displayed on 3 occasions. This screen will show before caregiver view elderly health data, view and add medication schedule and view elderly location. The list is displaying the list of elderly users associated to the caregiver. Caregivers can select on particular elderly user by clicking on the elderly name tab and view or perform action on their data.

On the top right corner, there is a icon button with an add sign together with person sign. Clicking this will allow caregivers to add their elderly to their list by inputting their elderly username. Username is unique for every user. The red trash bin icon button allows caregivers to delete the elderly user from their list.

Caregivers can navigate through the bottom navigation bar to navigate to other screens.

3.2.3 View Elderly Location

Figure 3.2.3.1: View Elderly Location

This screen caregivers can look at the selected elderly's location to track them and ensure their safety.

Caregivers can swipe around the map and also click on the add sign to zoom in and click on the subtract sign to zoom out.

3.2.4 View Health Data

Figure 3.2.5.1: View Health Data Screen

This screen allows caregivers to view the selected elderly's health data in table form.

By "View More", caregivers can view the health data in graph form.

3.2.5 Medication Schedule Screen

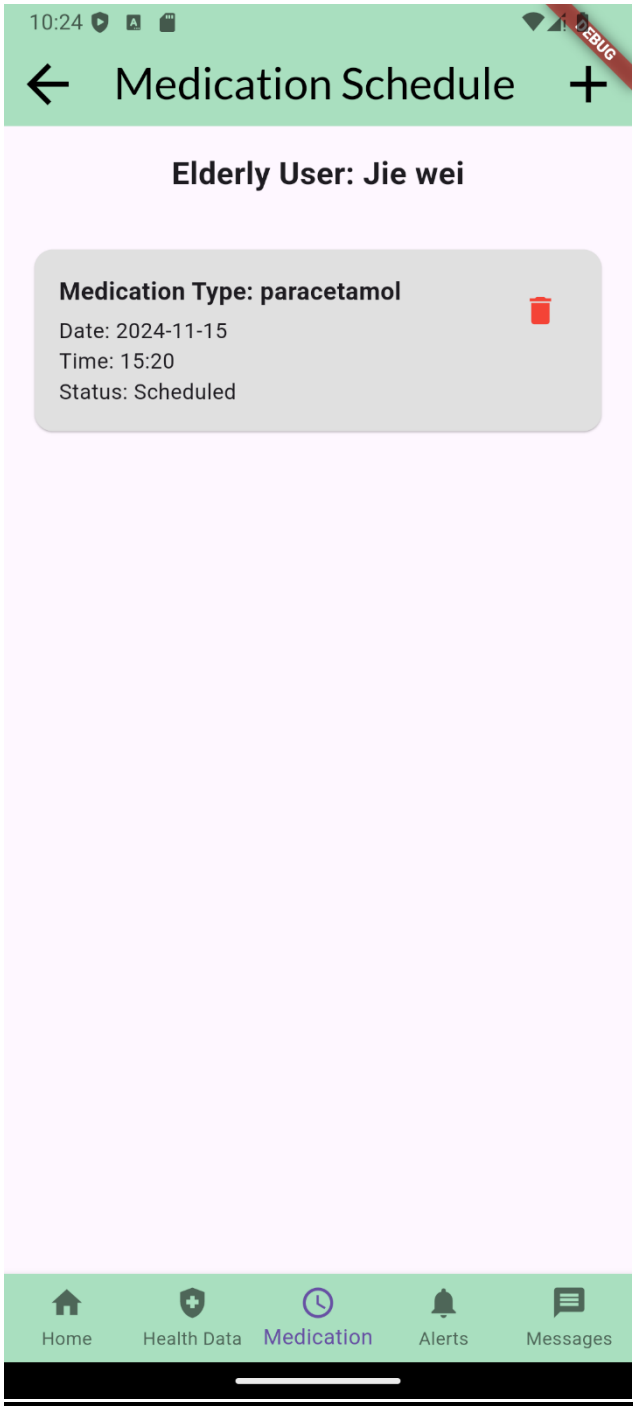


Figure 3.2.5.1: Medication Schedule Screen

In this screen, caregivers can view the medication schedule of selected elderly. Caregivers could click on the trash icon button to delete the medication schedule if they added the schedule wrongly.

By clicking on the add icon button on top right corner, caregivers can add new medication to the elderly's medication schedule.

3.2.6 Add Medication Screen

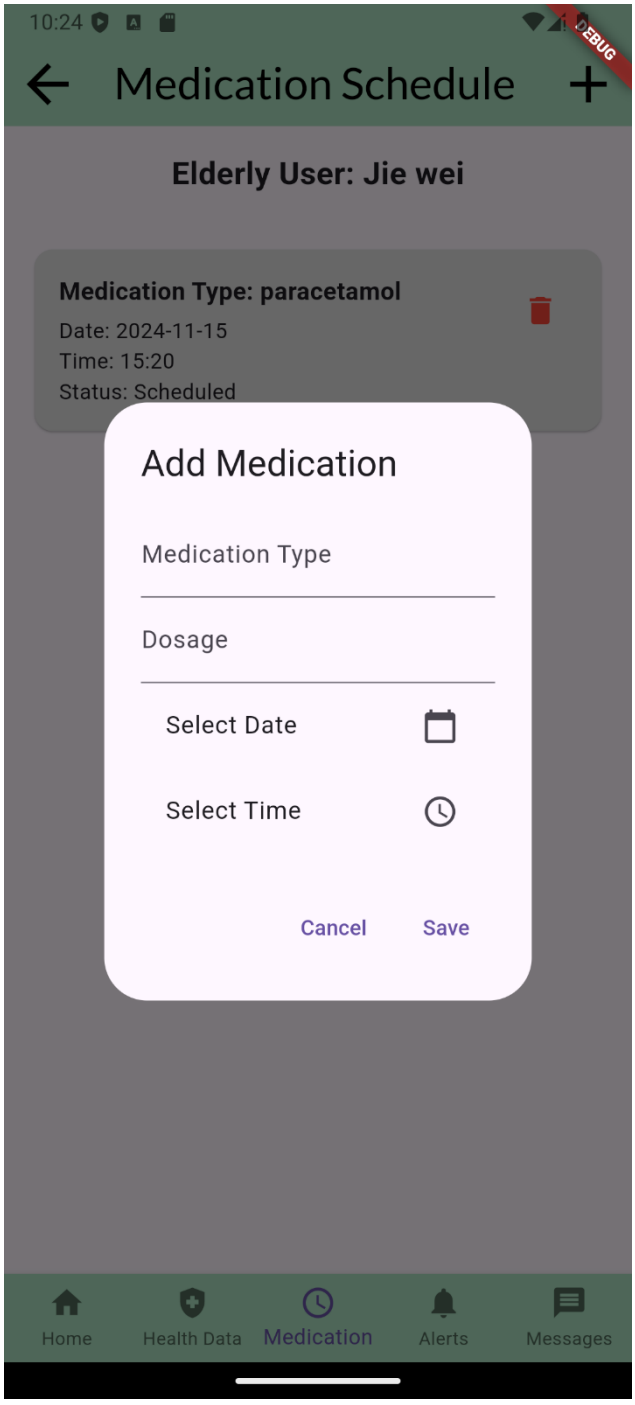
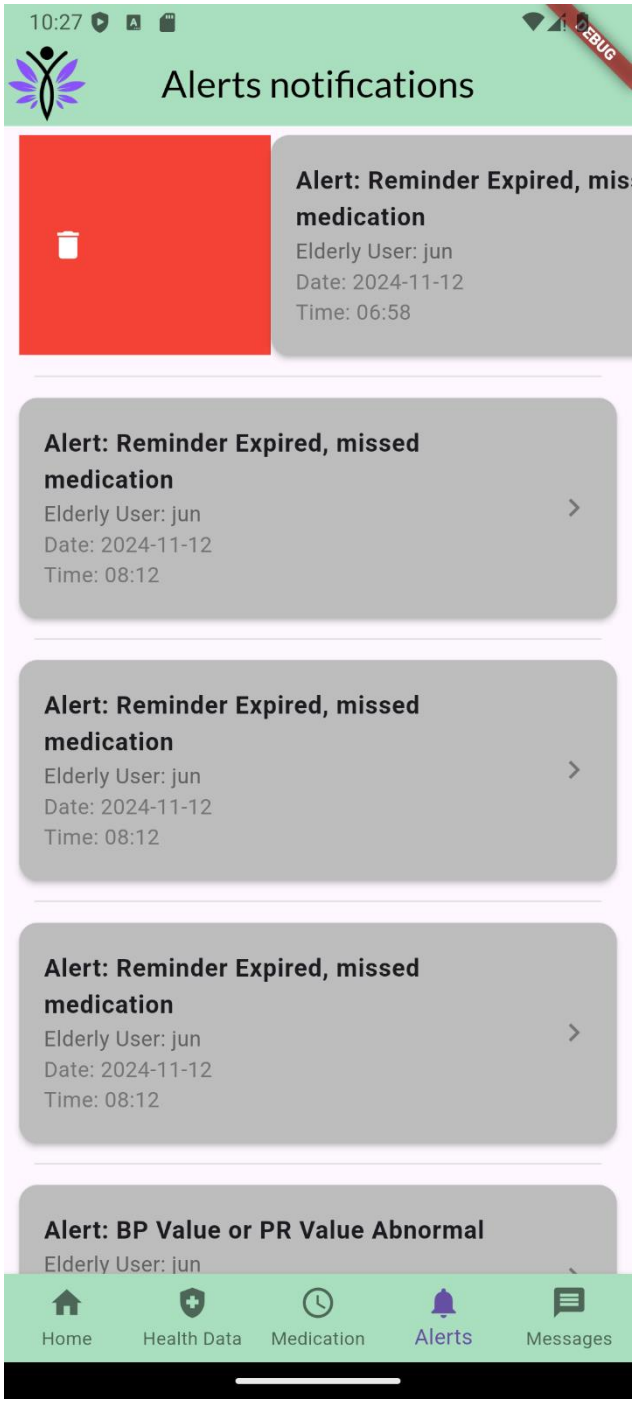


Figure 3.2.6.1: Add Medication Screen.

To add medication schedule to the selected elderly, fill up the text fields. Then, select dates on calendar and select time by typing or clicking on parts of clock.

After done filling up the details, press save button to add the medication schedule to the selected elderly.

3.2.7 Alerts and Notifications Screen



Caregivers can view all the alerts and notifications in this screen. Swiping the alert tab to the right to delete it.

Figure 3.2.7.1: Alerts and Notifications Screen

3.2.8 Select Healthcare Provider Screen

This screen is similar to select elderly screen. Caregivers can view the list of associated healthcare provider. Caregivers can delete healthcare provider by clicking on the red trash icon button and they can add healthcare provider by clicking on the add person button on top right corner.

Figure 3.2.8.1: Select Healthcare Provider Screen

3.2.9 Chat Screen

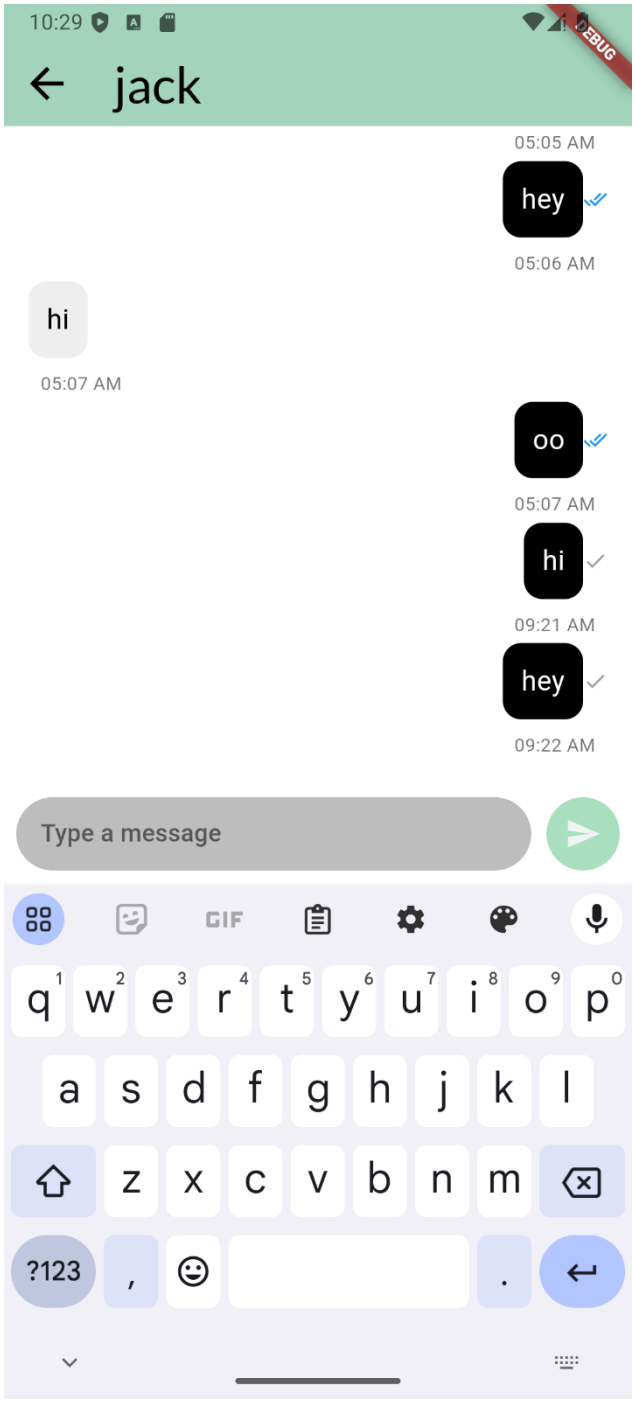


Figure 3.2.9.1: Chat Screen

After caregivers selected the healthcare provider they want to chat with, this chat screen will display. Click on the "Type a message" text field and type the message to be sent, then click on the send button on the right-hand side of the text field to send the message to the healthcare provider.

2 ticks indicate that the message is already read and 1 tick indicates that the message is sent but haven't read by the healthcare provider.

3.3 Elderly Users

3.3.1 Elderly Dashboard

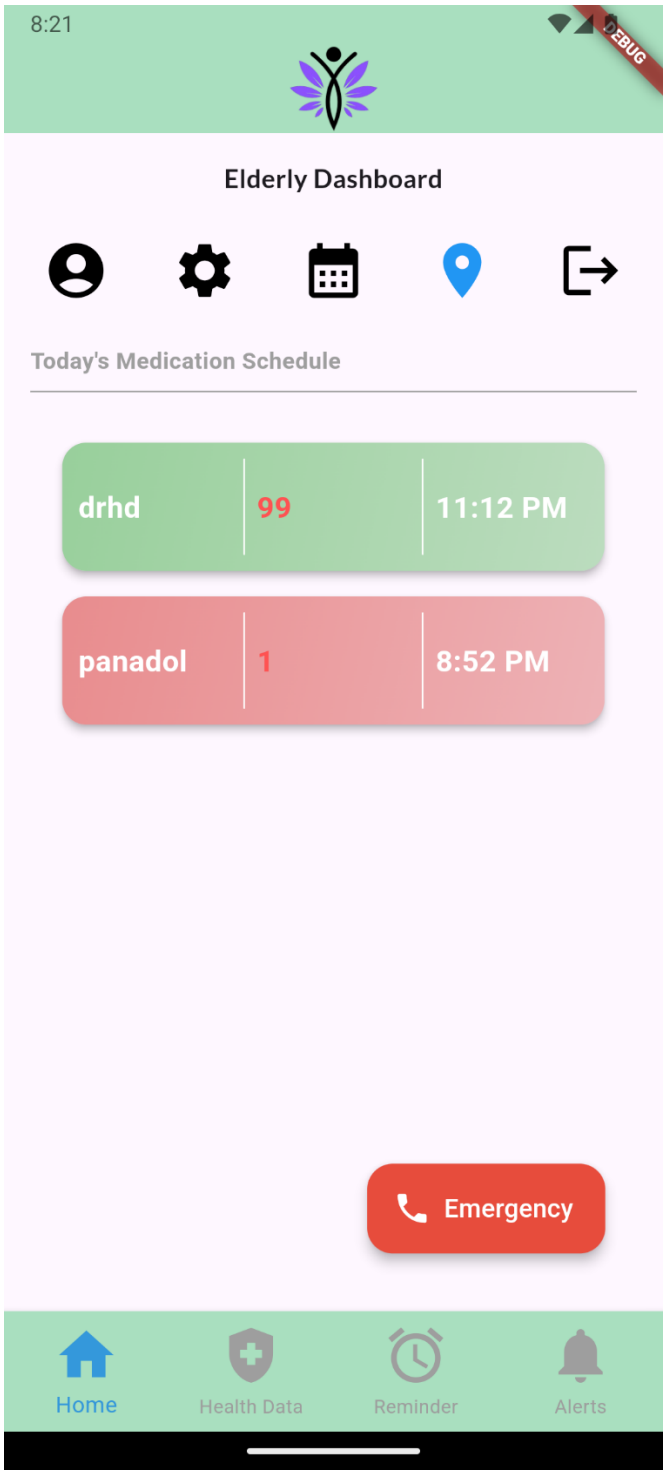


Figure 3.3.1.1: Elderly Dashboard

If the user login as Elderly user, the screens will show a dashboard of elderly user. This is the page of elderly user, under the logo show a row of button which are profile, setting, appointment, share location and logout. The profile button will navigate the user to their profile screens, setting page will let user to make some change of the app settings. Then, the appointment will let user to make an appointment from healthcare providers. Lastly, the location button will navigate user for share their location dynamically with their caregiver. After that, it shows a table with some data of today medication schedule in the table if show red mean that the elderly user hasn't complete to take their medicine else green mean completed. Then move to the floating button is to let elderly user make the emergency call to their caregivers. Then if the elderly user swipe left it will mark the medicine as complete/

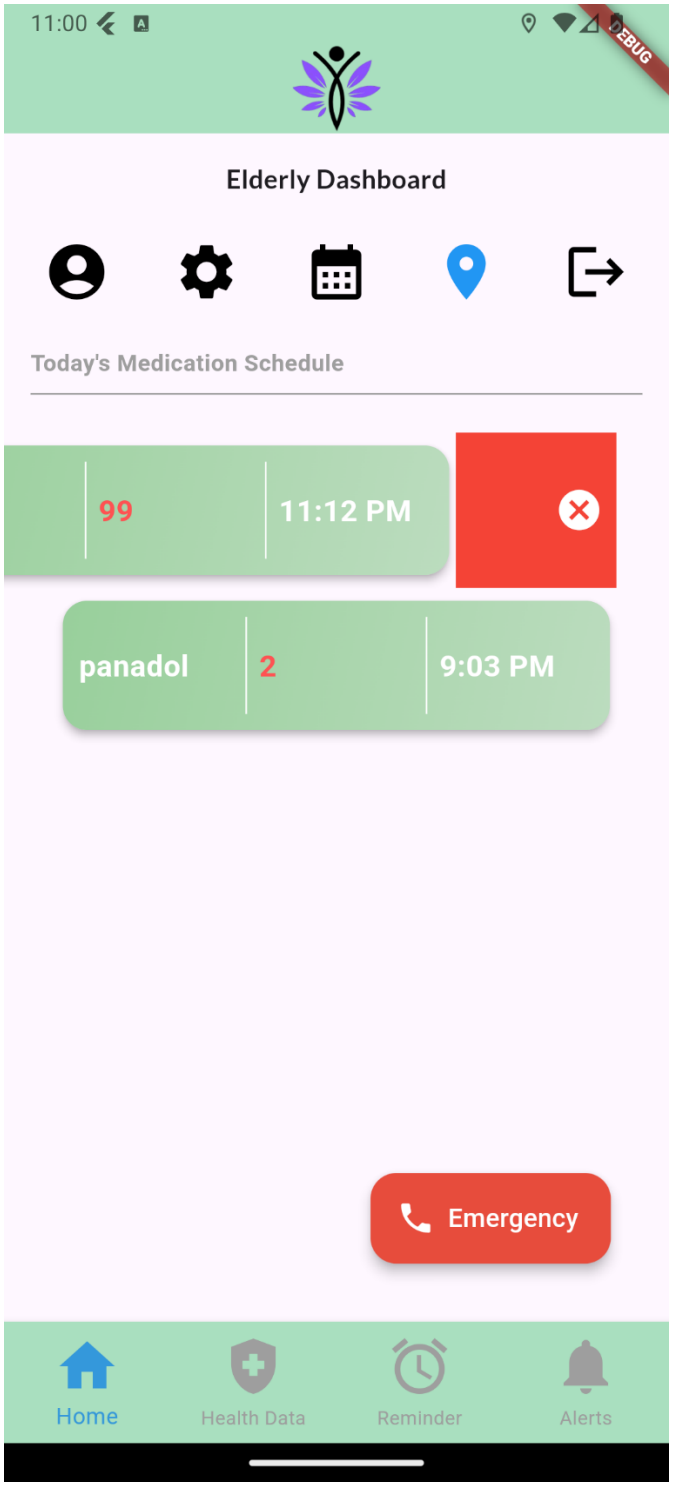
Floating navigation bar:

Home: Elderly Dashboard

Health Data: Elderly users log their health data

Reminder: Elderly users check their reminder of medication and alert history

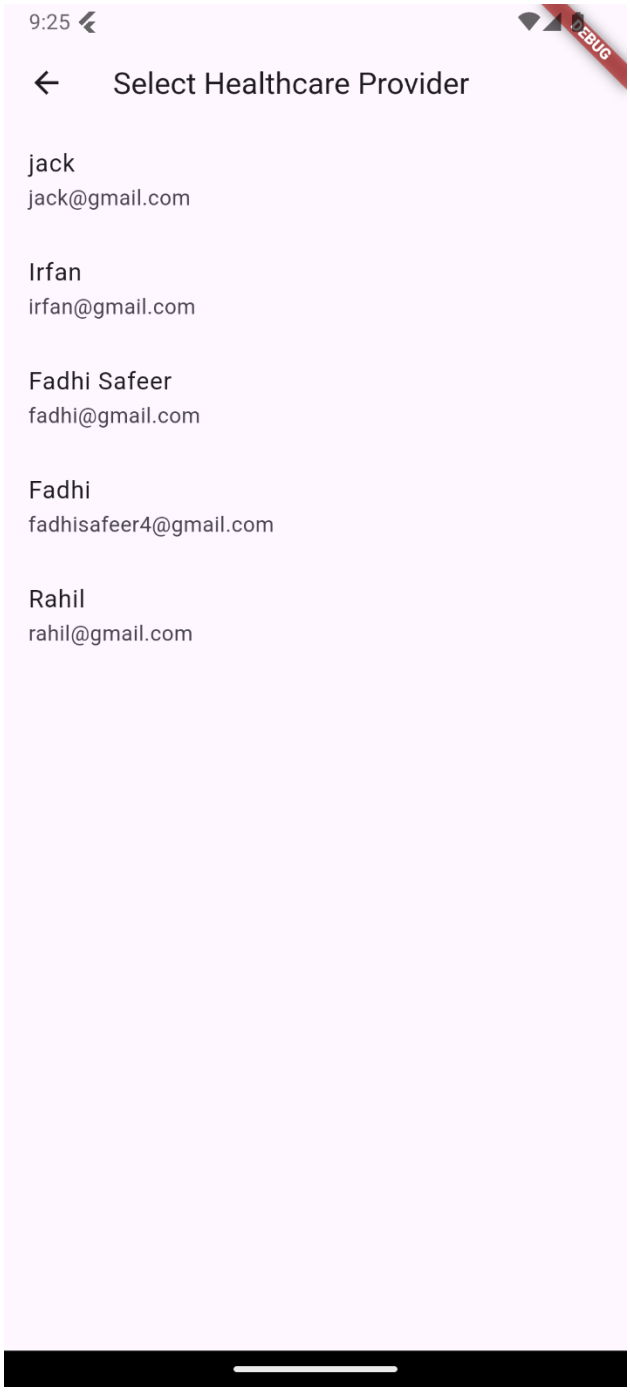
Alerts: The new alert that add in.



The example of that elderly user can swipe the card left to mark the today medication as completed.

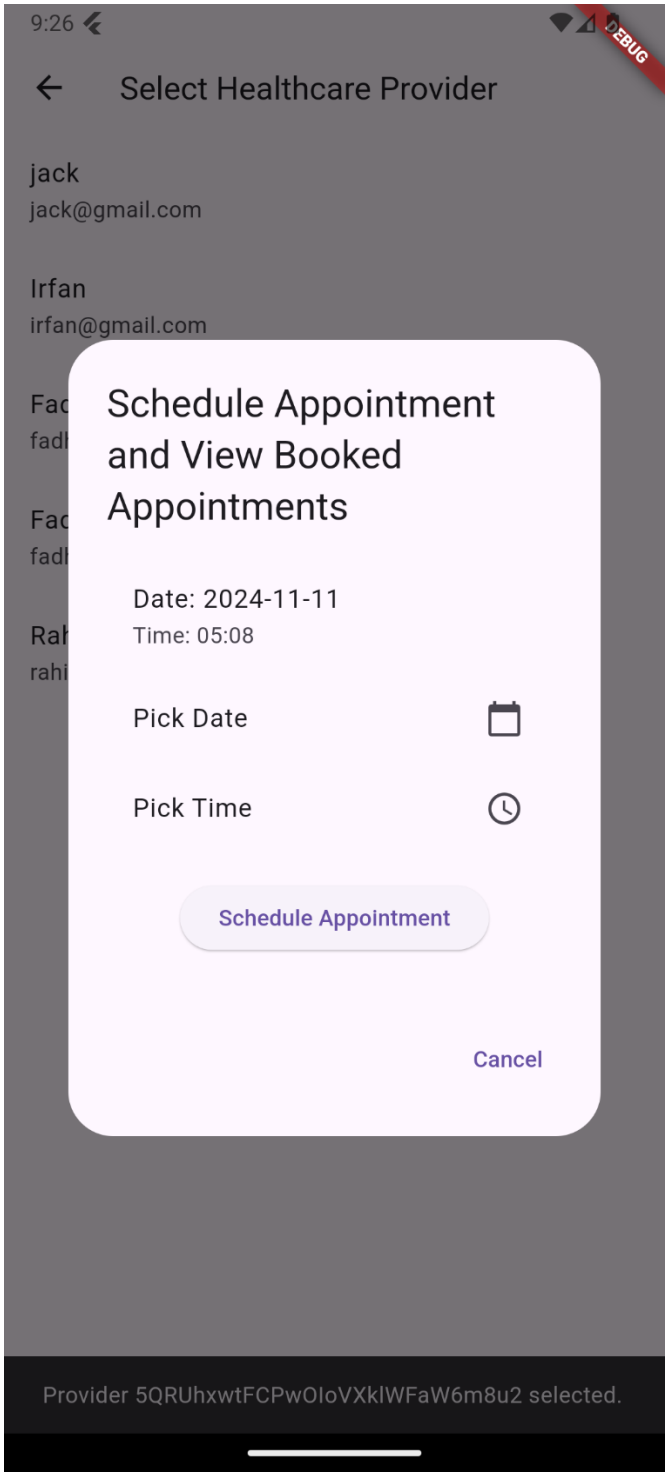
Figure 3.3.1.2: Elderly Dashboard Today medication care swipe left

3.3.2 Appointment Screen



The appointment page show that a list of elderly users with their name and email that to let the elderly user to make a medication consultation appointment or other purpose.

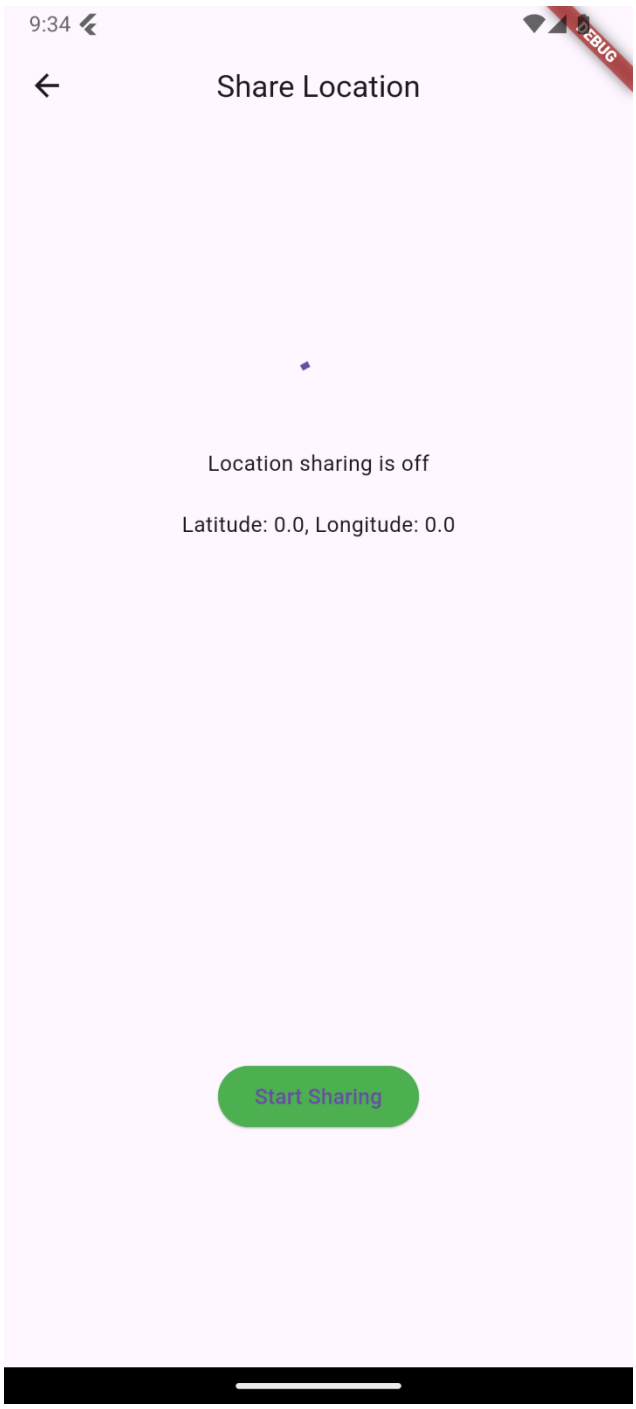
Figure 3.3.2.1: Appointment Select Healthcare Provider Screen



If the user selects any of the healthcare provider, it will show the appointment history and let elderly user to pick the time and date to schedule appointment.

Figure 3.3.2.2: Appointment screens choose date & time and check history

3.3.3 Share Location Screen



The share location screen will first show the status is off. Then the latitude and longitude. The bottom shows a start sharing button.

Figure 3.3.3.1: Location Sharing close status

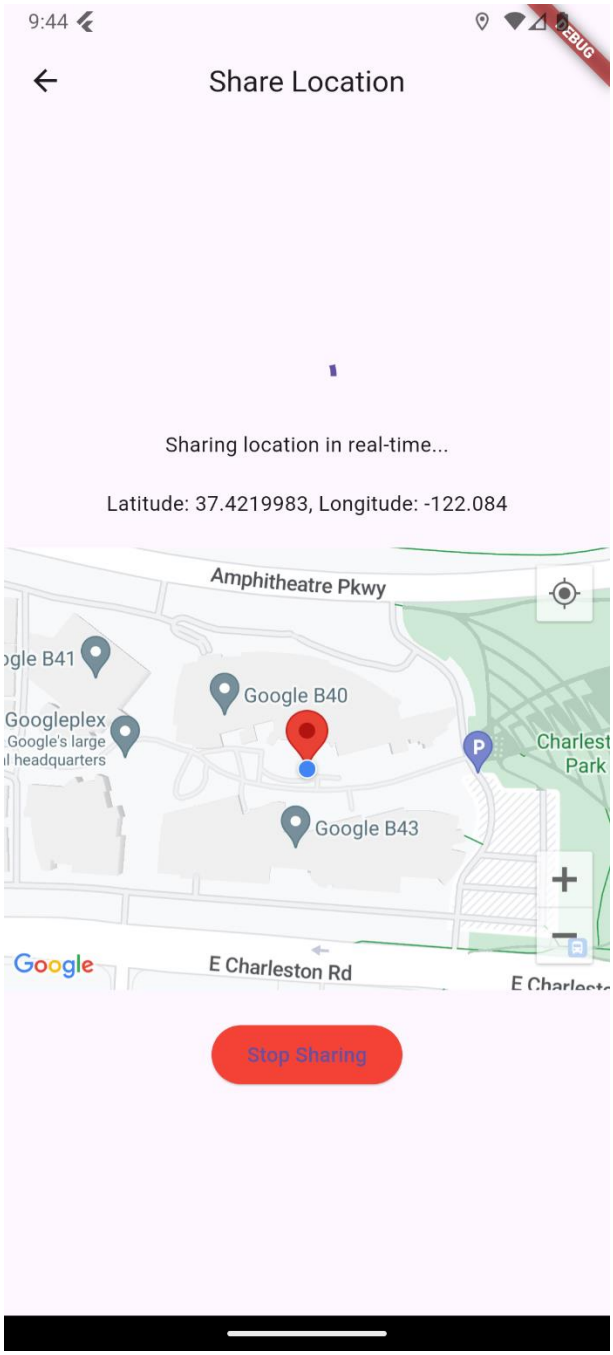


Figure 3.3.3.2: Location Sharing start status

After the elderly toggle the button start sharing it will show the current location of elderly user now and its latitude and longitude under it also show the google maps preview to let user know where specific there are now. The system will update the elderly user's location dynamically even they leave the screens or close the app, I they don't toggle the stop sharing button.

3.3.4 Emergency call button

If the user toggles the emergency button it will directly navigate the user to the phone built in phone call application and directly key in the phone number of caregiver's contact number.

Figure 3.3.4.1: Emergencies navigate to phone call

3.3.5 Health Data Screen

Figure 3.3.5.1: Health data screen

The elderly user health data screen will show elderly user his health metric data history and under the history data will have a view more data to show the graph of health metric data and a log health data to let elderly user log their health data.

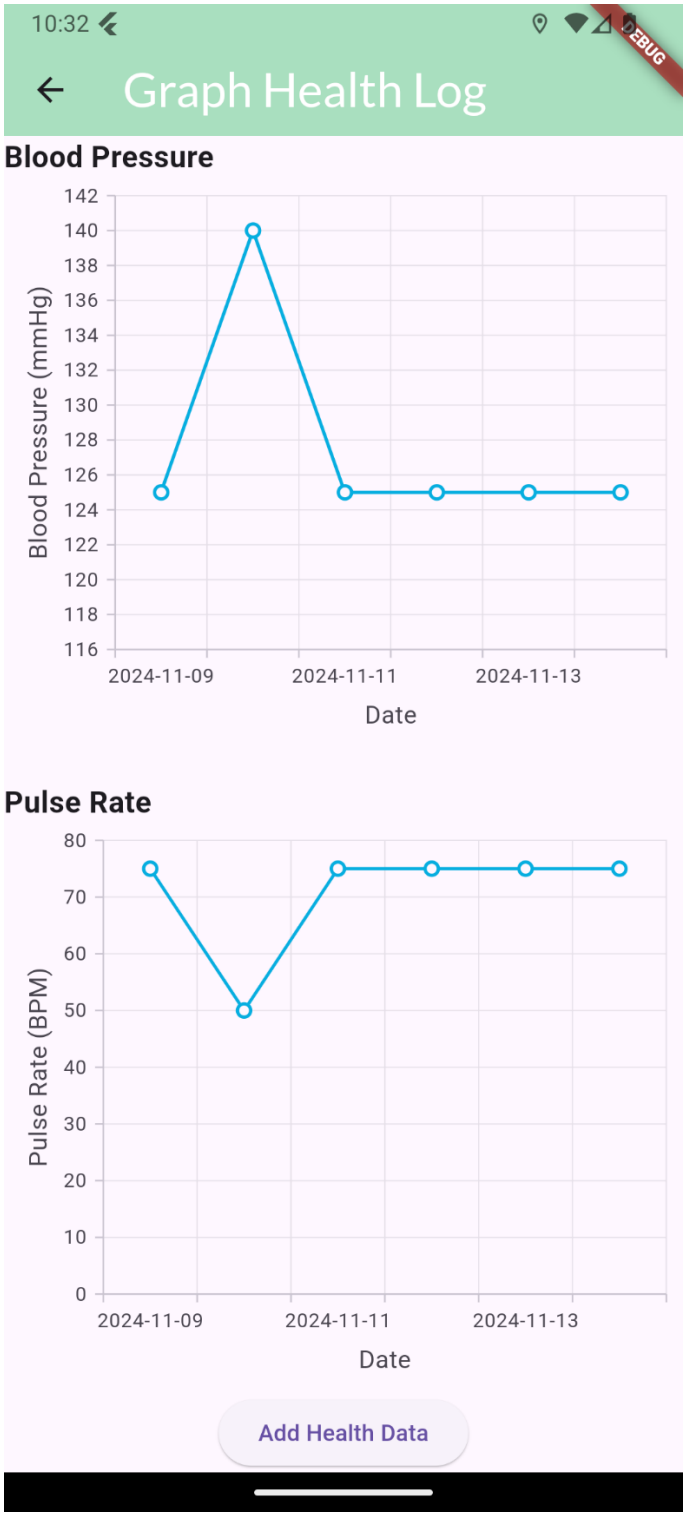
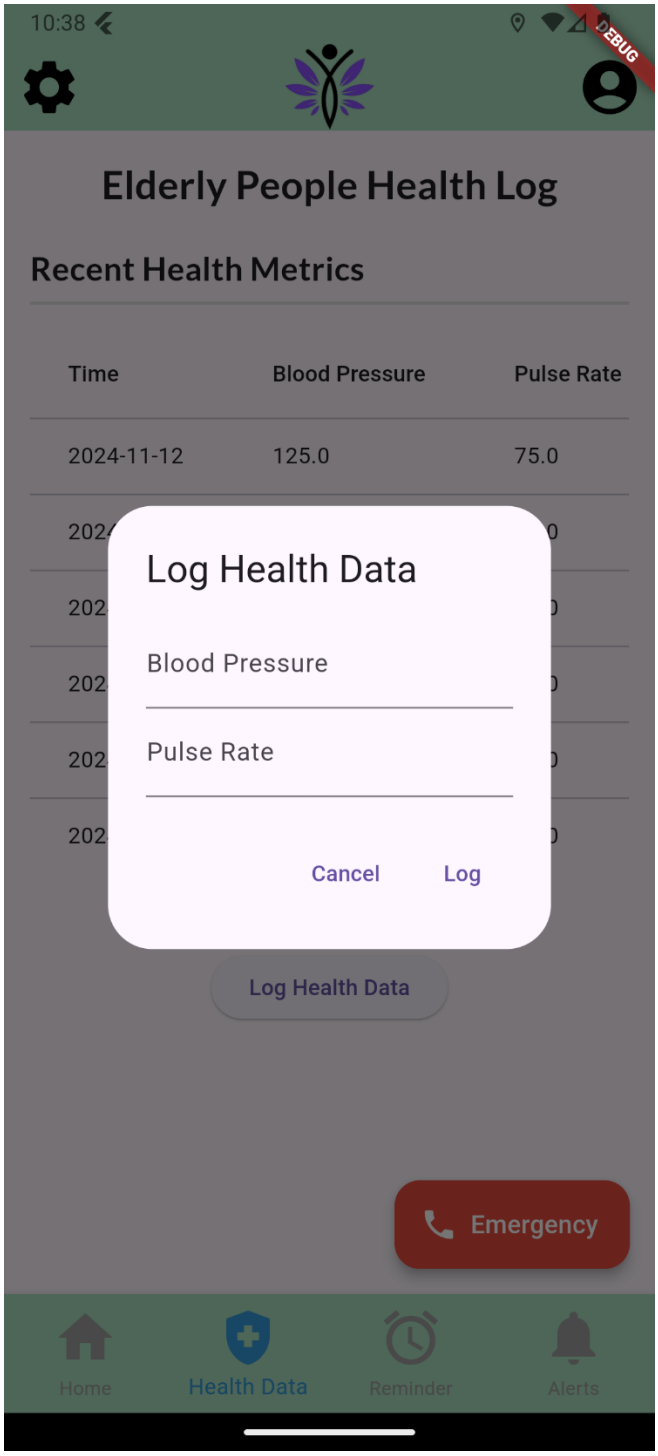


Figure 3.3.5.2: Health data graph screen

If the user clicks the view more button it will show two graph of elderly user's health metrics data graph which are blood pressure data and pulse rate data. It will follow the date order to let both elderly user, caregiver and healthcare provider to observe the health data of elderly user more easily.



If the user clicks the log health data, it will let elderly user to input their blood pressure and pulse rate today.

Figure 3.3.5.3: log health data screen

3.3.6 Reminder Page

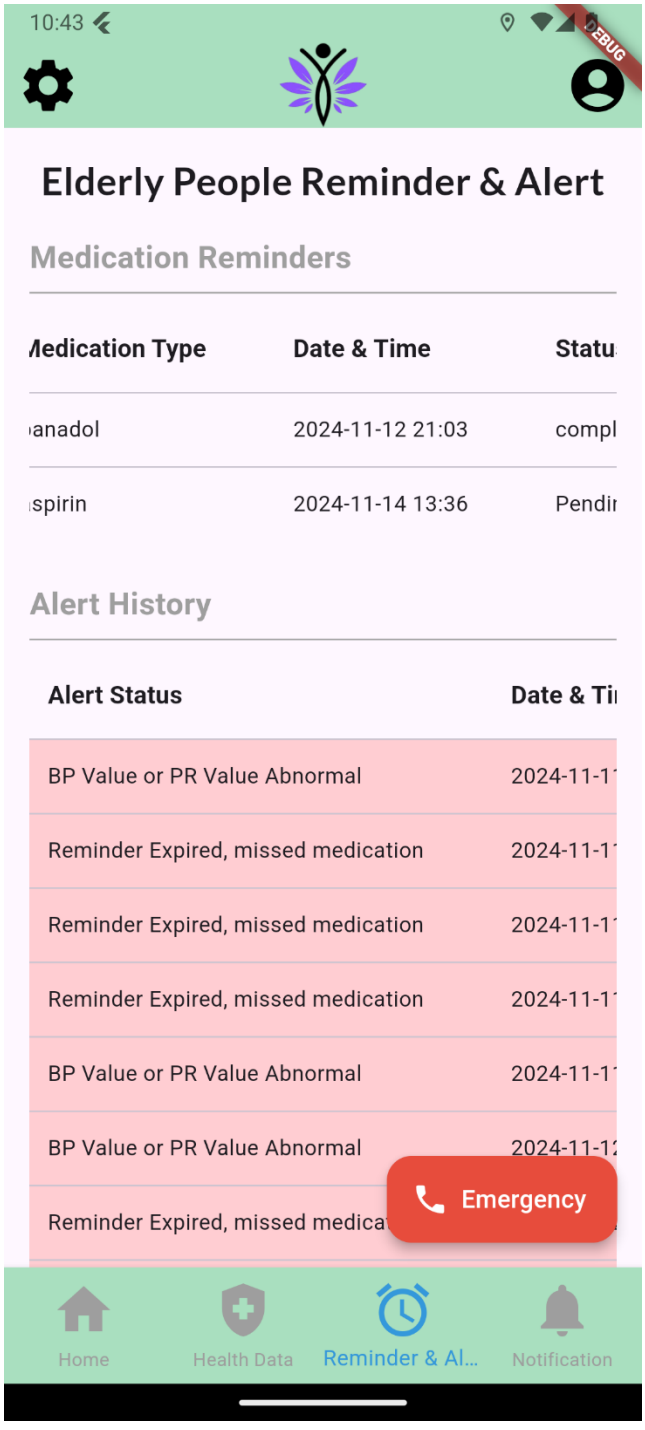


Figure 3.3.6.1: Reminder page screen

Reminder page of elderly user show that the medication reminder that is coming up and the alert history that haven't deleted yet by the caregiver. Inside the reminder table it shows that the medication type, date & time and status (pending or completed). Then the alert history table will show the alert status which mean why will the alert come up and the date & time.

3.3.7 Alert screen

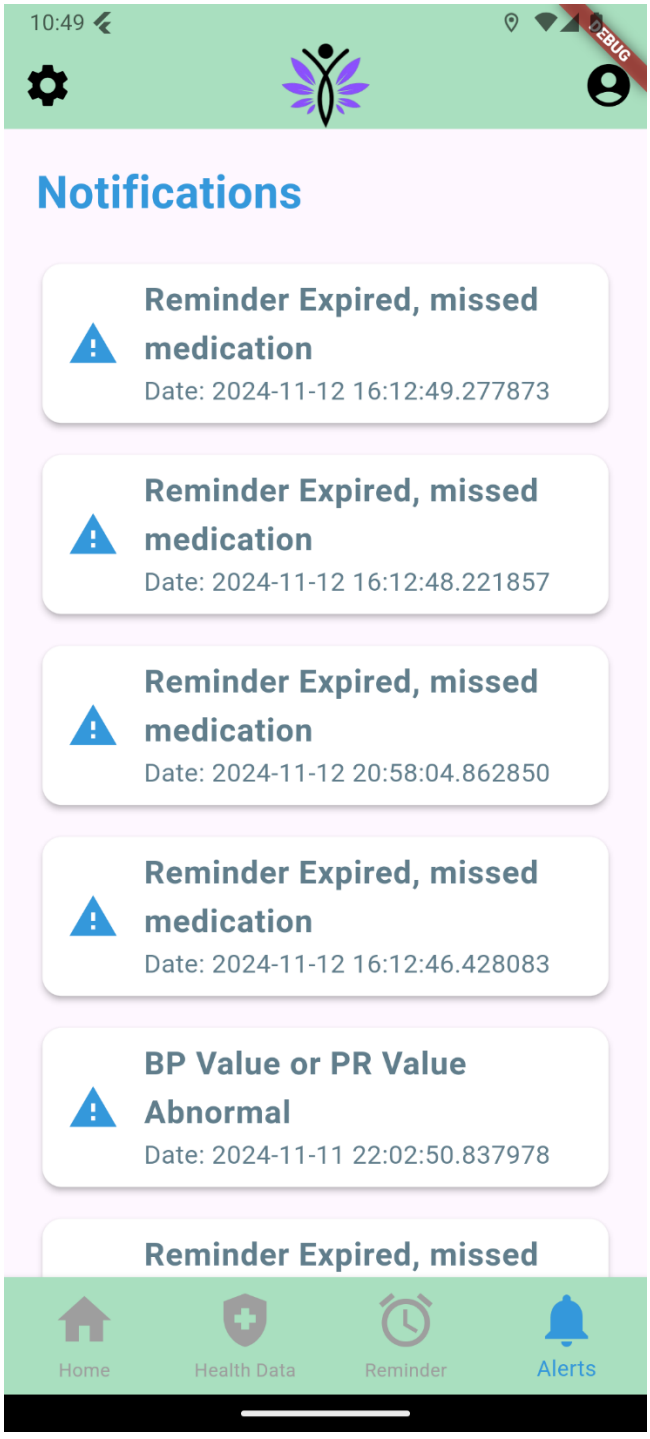


Figure 3.3.7.1: Alert Screen

In alert screen the alert cards feature a blue exclamation mark on the left side, indicating an urgent alert. Each card has a bolded title message explaining the type of alert, such as "Reminder Expired, missed medication" or "BP Value or PR Value Abnormal" for abnormal readings in health data. The card also displays a timestamp, allowing users to track the date and time of the alert. These cards provide a quick summary of missed actions or abnormal health data, helping elderly users or caregivers stay informed about potential issues.

3.4 Healthcare Providers

3.4.1 Healthcare Provider Dashboard Page

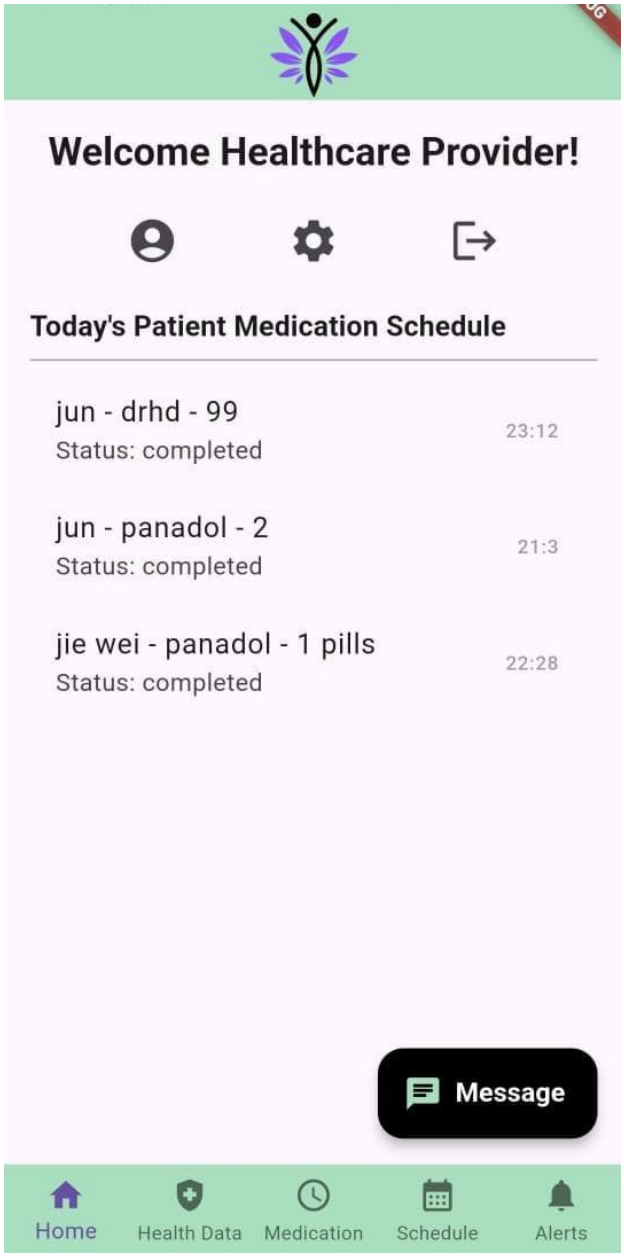


Figure 3.4.1: Healthcare Provider Dashboard Screen

If user logged in as healthcare provider, this healthcare provider dashboard page will be displayed, it is the home screen for healthcare provider. Users can click on the profile icon button to view their profile. The setting icon button allows user to view and adjust the settings. The rightmost icon button on the upper part of the screen allows users to log out of their account and return to login screen. The "Message" button allows healthcare providerwill allows them to message caregivers.

Bottom Navigation Bar:

Home: Healthcare Provider Dashboard Page

Health Data: View Health Data of elderly in table or chart.

Medication: View Medication Schedule of Elderly or Add Medication Schedule.

Schedule: Allows viewing and management of appointment **schedules with elderly patients.**

Alerts: View Alerts and Notifications History

3.4.2 Health Data Page

On this screen, healthcare providers can select an elderly user to view their health data on the "View Health Data" page. Next to each user, there is a red recycle bin icon that allows the provider to delete that user if needed. Additionally, a top-right option lets providers add new elderly users to the list.

Figure 3.4.2.1: Health Data Screen

3.4.2.1 View Health Log Page

This screen allows healthcare provider to view the selected elderly's health data in table form.

By "View More", healthcare provider can view the health data in graph form.

Figure 3.4.2.2: View Health Log Page Screen

3.4.2.2 View Graph Health Log Page

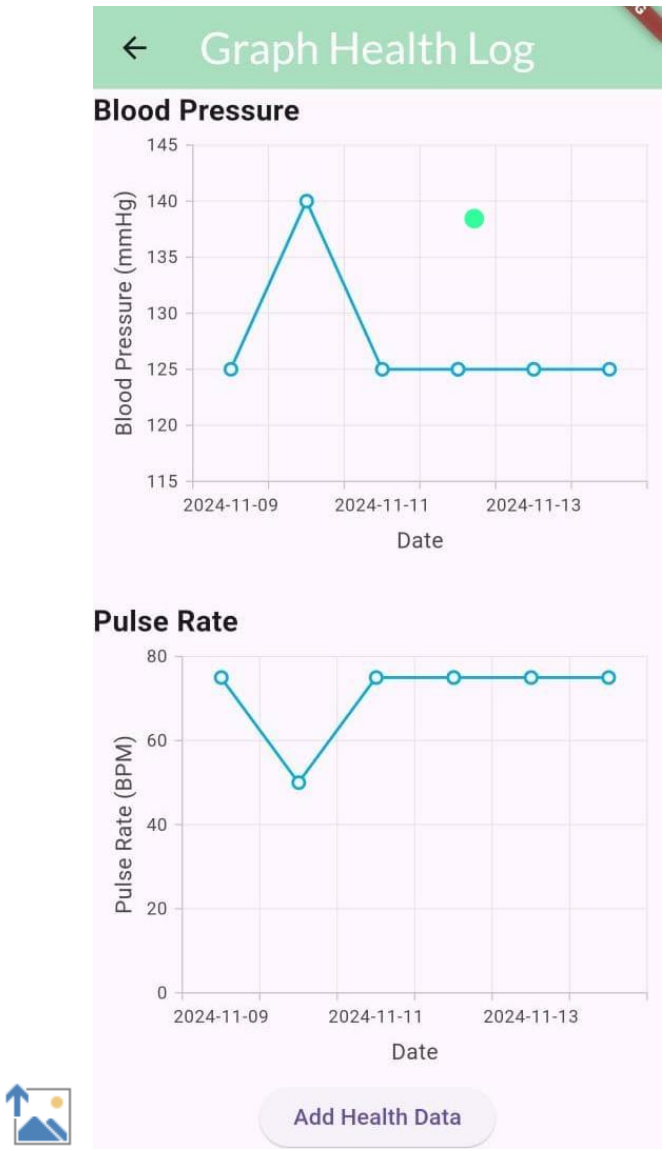


Figure 3.4.2.3: View Graph Health Log Page Screen

3.4.3 Medication Schedule Page

On this screen, healthcare providers can select an elderly user to view their health data on the "View Health Data" page. Next to each user, there is a red recycle bin icon that allows the provider to delete that user if needed. Additionally, a top-right option lets providers add new elderly users to the list.

Figure 3.4.3.1: Medication Schedule

3.4.3.1 View Medication Schedule Page

In this screen, healthcare providers can view the medication schedule of selected elderly. Healthcare Providers could click on the trash icon button to delete the medication schedule if they added the schedule wrongly.

By clicking on the add icon button on top right corner, healthcare Providers can add new medication to the elderly's medication schedule.

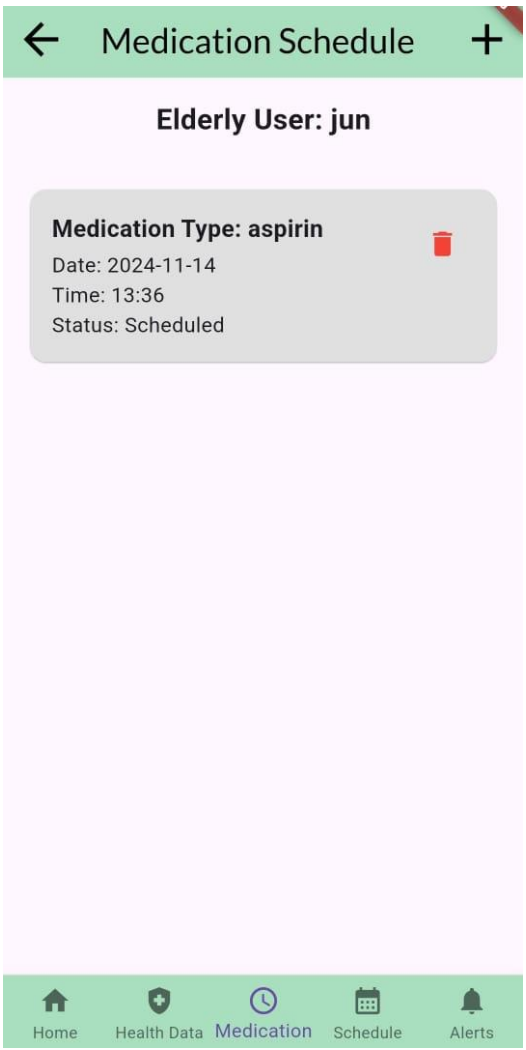


Figure 3.4.3.2: View Medication Schedule

3.4.4 Schedule Appointments Page

In this screen, healthcare providers can view the appointment schedule with elderly patients. Healthcare Providers could click on the listbox to change the appointment schedule status (Cancelled,Scheduled,Completed) .

Figure 3.4.4.1: Schedule Appointments Page

3.4.5 Alerts Notification Page

Healthcare providers can view all the alerts and notifications in this screen. Swiping the alert tab to the right to delete it.

Figure 3.4.5.1: Alerts Notification Page

3.4.6 Chat List Page:

In this screen, healthcare providers can select the associated caregivers to chat. Healthcare Providers could click on the icon on top right to add new chat with the required caregiver.

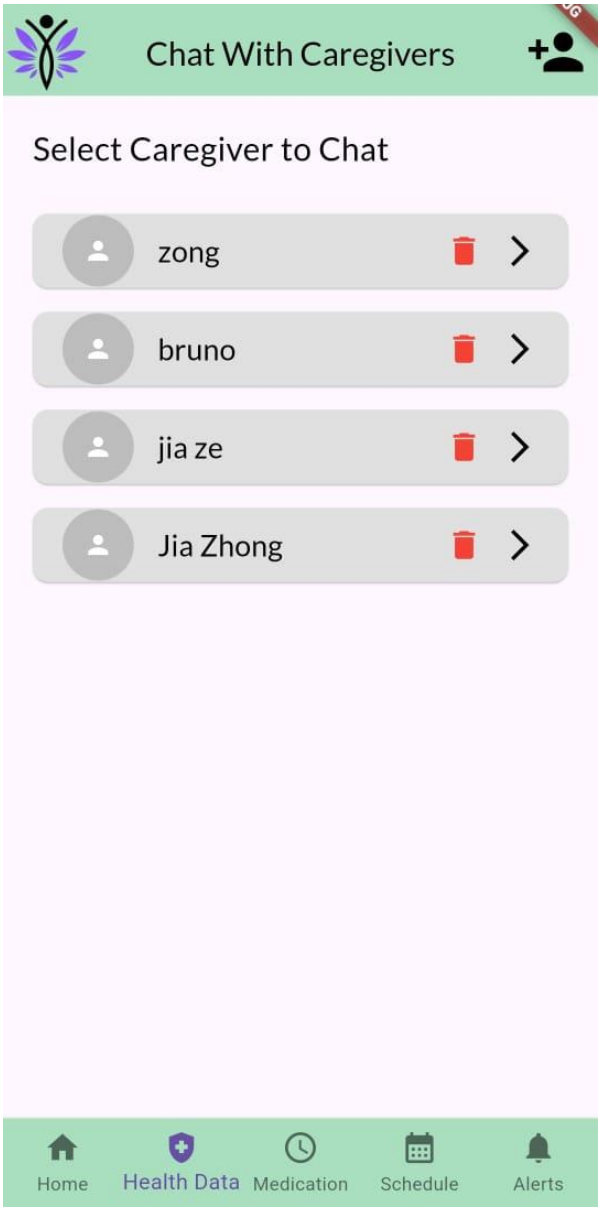


Figure 3.4.6.1: Chat List Page

3.4.6.1 Message Caregivers Page:

After healthcare providers selected the caregiver, they want to chat with, this chat screen will display. Click on the “Type a message” text field and type the message to be sent, then click on the send button on the right-hand side of the text field to send the message to the caregiver.

2 ticks indicate that the message is already read and 1 tick indicates that the message is sent but haven’t read by the caregiver.

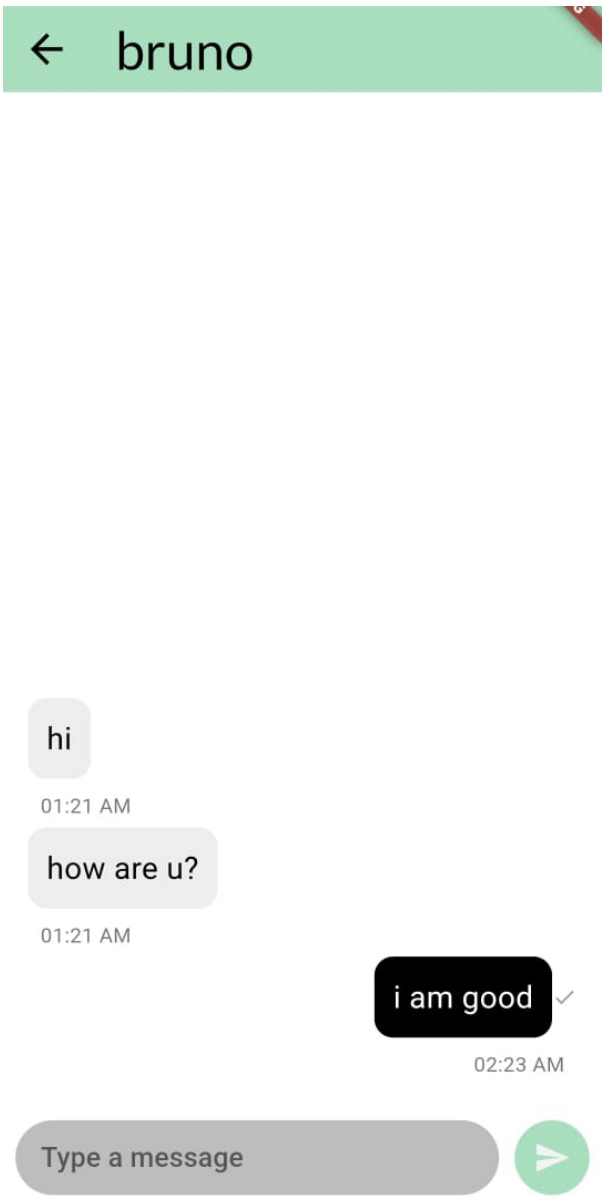


Figure 3.4.6.2: Message Caregivers Page

4.0 Automated System Testing

4.1 Robo Test

The type of automated testing that is used for this android application is Robo Testing. It is available on Firebase Test Lab that helps identify issues in Android applications by simulating user interactions (Sutanto, 2019). This tool systematically navigates through an app, exploring different screens, buttons, and paths without requiring any written test cases. Robo test is useful for finding crashes, layout issues, and performance bottlenecks.

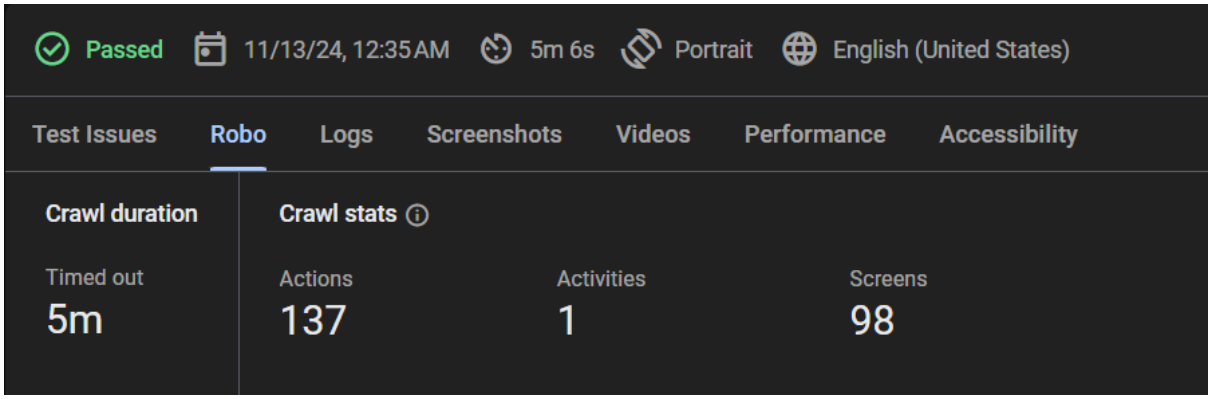


Figure 4.1

The result indicates that the test ran successfully without any major issues, such as crashes. The test duration was set to 5 minutes, which is the maximum time allowed for the Robo test to explore the app. It "timed out," meaning that it reached the time limit before covering every possible path in the app. The Robo test performed 137 actions. These actions include interactions with UI elements like taps, swipes, and input fields. The test explored 98 unique screens or states within the app's UI, which could include variations within the single activity, such as different dialog boxes or input states.

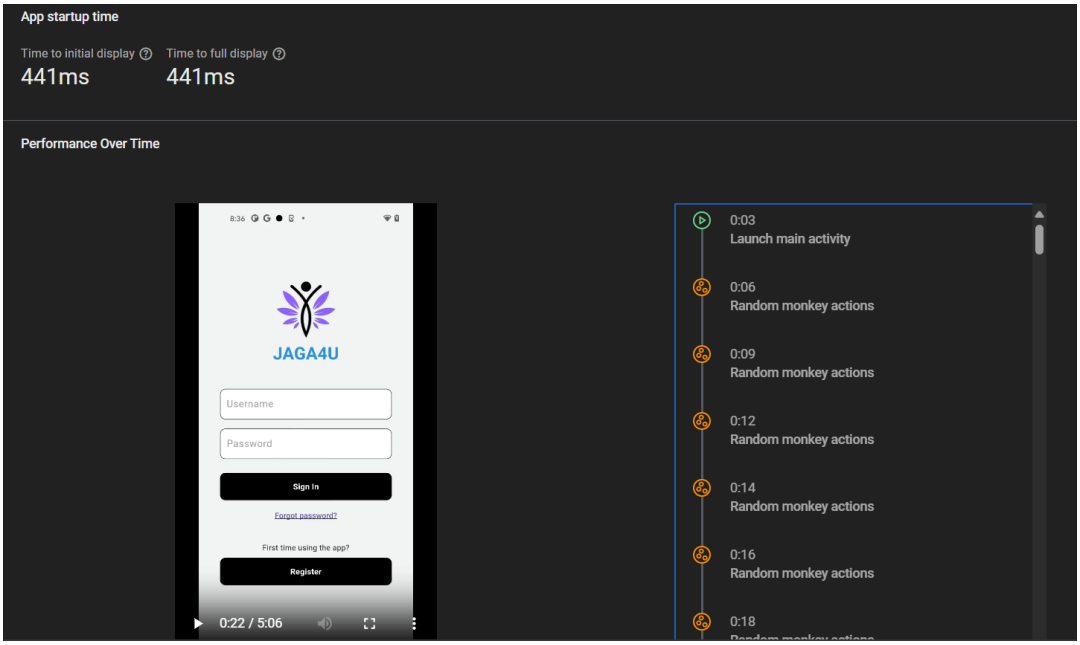


Figure 4.2

The time it took for the app to fully render the main UI, is 441 milliseconds, showing that the app was quick to load and render.

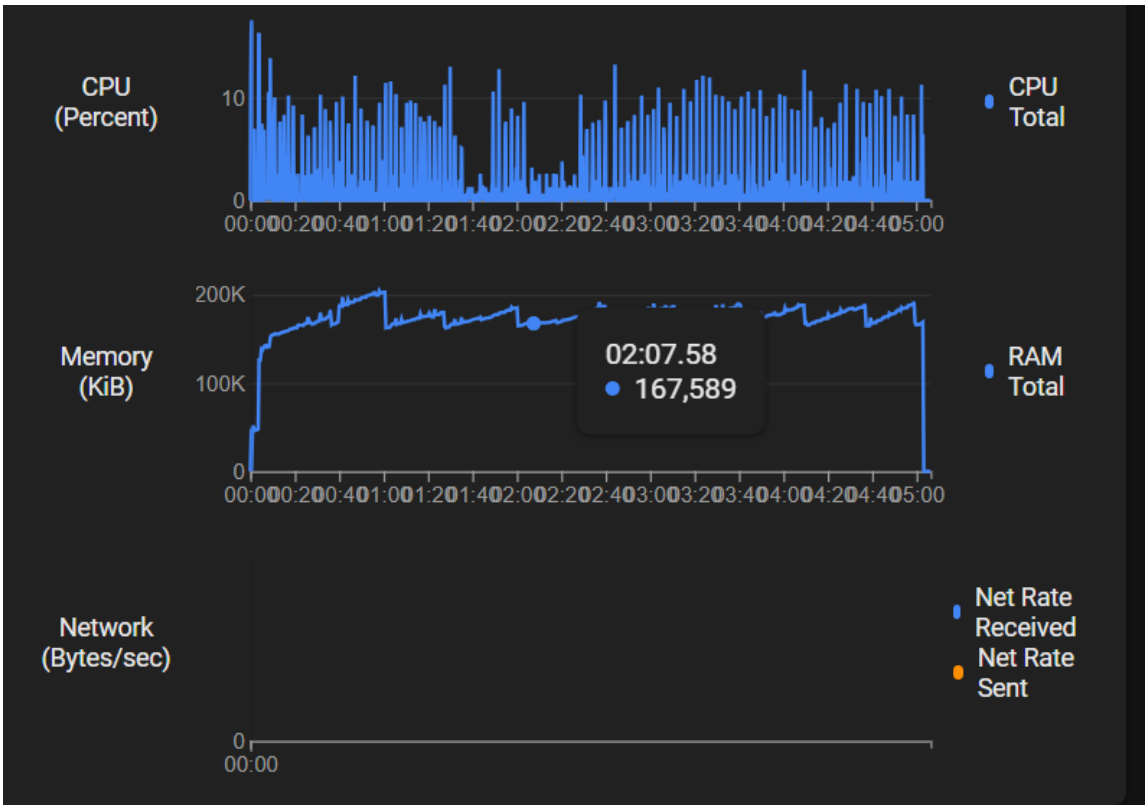


Figure 4.3

The CPU usage looks healthy as it's not consistently high. Memory usage appears stable. The sharp drop at the end indicates test completion and app termination.

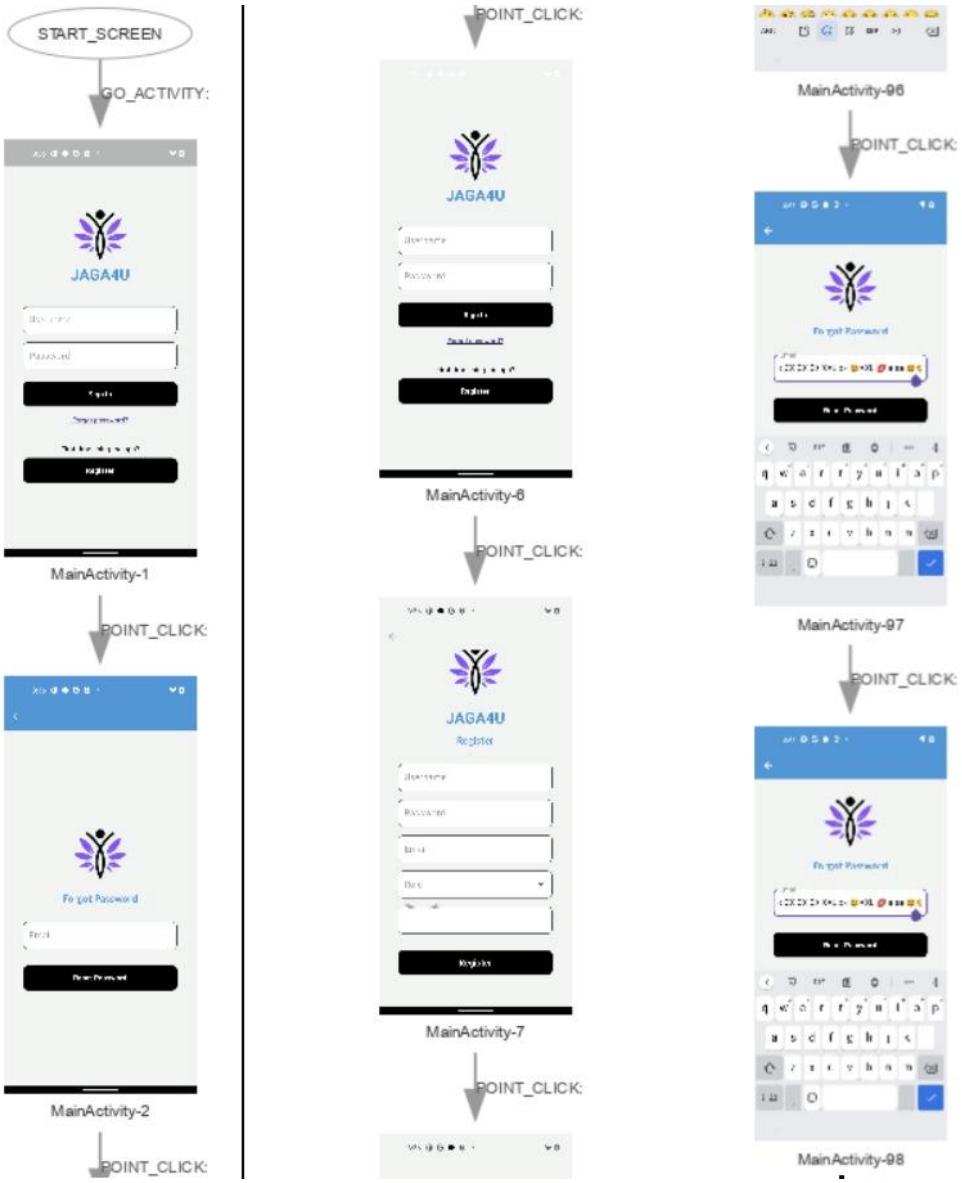


Figure 4.4

This diagram illustrates the user flow and interaction points for login, registration, and password recovery features in the JAGA4U app. It shows how users navigate between screens and the actions required to complete each task.

4.2 Unit Testing

```dart
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:mae_assignment/providers/appointment_scheduler.dart';  // Import the provider class

Run | Debug
void main() {
  Run | Debug
  group('AppointmentSchedulerLogic Tests', () {
    late AppointmentSchedulerLogic appointmentSchedulerLogic;

    setUp(() {
      // Initialize the AppointmentSchedulerLogic
      appointmentSchedulerLogic = AppointmentSchedulerLogic();  // Passing null as we don't need the repository here
    });

    Run | Debug
    test('setSelectedTime should update selectedTime correctly', () {
      // Arrange: Create a TimeOfDay object
      final time = TimeOfDay(hour: 10, minute: 30);

      // Act: Call setSelectedTime with the time object
      appointmentSchedulerLogic.setSelectedTime(time);

      // Assert: Verify that selectedTime is updated correctly
      expect(appointmentSchedulerLogic.selectedTime, equals(time));
    });

    Run | Debug
    test('setSelectedTime should set null when null is passed', () {
      // Act: Call setSelectedTime with null
      appointmentSchedulerLogic.setSelectedTime(null);

      // Assert: Verify that selectedTime is null
      expect(appointmentSchedulerLogic.selectedTime, isNull);
    });
  });
}
```

Figure 4.2.1

This Dart code is a Flutter test suite for testing the AppointmentSchedulerLogic class, specifically its setSelectedTime method. The test suite is structured with group and test functions, organizing the tests under a group labeled "AppointmentSchedulerLogic Tests". In setUp, it initializes an instance of AppointmentSchedulerLogic before each test. The first test verifies that calling setSelectedTime with a TimeOfDay object correctly updates the selectedTime property to the specified time. The second test checks that calling setSelectedTime with a null argument sets selectedTime to null, ensuring that the method handles null values correctly. Both tests use assertions to confirm that selectedTime behaves as expected after each method call.

```
PS C:\Users\HP\OneDrive\Desktop\MAE Backup\MAE-APP-JAGA4U\mae_assignment> flutter test
00:01 +2: All tests passed!
```

Figure 4.2.2

The Unit Testing shows result that passed.

4.3 Widget Testing

login_widget_test.dart

mae_assignment > lib > login_widget_test.dart > ...

```dart
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:mae_assignment/providers/login_provider.dart';
import 'package:mae_assignment/screens/login.dart';
import 'package:provider/provider.dart';
import 'package:mockito/mockito.dart';

class MockAuthProvider extends Mock implements AuthProvider {}

Run | Debug | Profile
void main() {
  late MockAuthProvider mockAuthProvider;

  setUp(() {
    mockAuthProvider = MockAuthProvider();
  });

  Widget createTestWidget() {
    return MaterialApp(
      home: ChangeNotifierProvider<AuthProvider>(
        create: (_) => mockAuthProvider,
        child: LoginPage(),
      ), // ChangeNotifierProvider
    ); // MaterialApp
  }

  testWidgets('displays logo, title, and input fields', (WidgetTester tester) async {
    await tester.pumpWidget(createTestWidget());

    // Verify the logo, title, and input fields are displayed
    expect(find.byType(Image), findsOneWidget);
    expect(find.text('JAGA4U'), findsOneWidget);
    expect(find.byType(TextField), findsNWidgets(2));
    expect(find.text('Username'), findsOneWidget);
    expect(find.text('Password'), findsOneWidget);
  });
```

Figure 5.3.1

The code is a Flutter widget test file for testing a LoginPage screen within a Flutter app. The tests ensure that the login screen is correctly displaying essential UI elements and that user interactions behave as expected.

MockAuthProvider is a mock class for AuthProvider, using mockito to simulate the behavior of the AuthProvider without relying on the actual implementation. setUp initializes a new MockAuthProvider instance before each test, allowing the tests to reset their state and avoid dependency on external resources. createTestWidget returns a MaterialApp widget wrapped with ChangeNotifierProvider, making mockAuthProvider available to LoginPage. This setup is essential for injecting dependencies into the widget tree during testing.

This test verifies a logo image widget is present, the text JAGA4U is displayed as the page title and two TextField widgets are shown for "Username" and "Password" inputs.

```
testWidgets('calls signIn on button press', (WidgetTester tester) async {
  await tester.pumpWidget(createTestWidget());

  // Enter text into username and password fields
  await tester.enterText(find.byType(TextField).at(0), 'testuser');
  await tester.enterText(find.byType(TextField).at(1), 'testpassword');



  // Verify signIn was called with correct arguments
});
}
```

Figure 4.3.2

The test verifies the interaction with signIn when a login button is pressed. To complete the test, find the login button. Simulate a button press and verify if mockAuthProvider.signIn is called with the expected arguments using verify from the mockito package.

```
PS C:\Users\CHING JIA ZHONG\Dropbox\PC\Downloads\MAE Testing\mae_assignment> flutter test lib/login_widget_test.dart
00:06 +2: All tests passed!
```

Figure 4.3.3

The result shows that all widget tests are passed.

5.0 Conclusion

"Jaga4U" offers a much-needed answer for elder care by improving the responsiveness, organized, and accessibility of health management. The software, which was created with simplicity and ease of use in mind, gives elderly users, caregivers, and healthcare providers the ability to interact, communicate, and track vital health data. The app's architecture, which was developed with Flutter and Firebase, guarantees that users will encounter a reliable, smooth interface that adjusts to the particular requirements of each role. Firebase powers the app's real-time features, such as live health updates and instant notifications, allowing caregivers and healthcare providers to respond quickly to any changes in an elderly user's health. Elderly users may log their health metrics, caregivers may monitor on and help numerous people, and healthcare providers can keep track of and manage appointments. While automated testing verifies the app's reliability and efficient operation of every function, the team's emphasis on role-specific permissions and secure data processing helps safeguard user privacy and critical health information. In order to make the app even more user-friendly for senior citizens, it is recommended to include future accessibility features like voice commands or easier navigation. "Jaga4U" is a dependable and useful instrument for enhancing senior care and health management because of its solid base and has the potential for additional customisation. Throughout out development, we found that some features that we are trying to implement needed the billing plan of Firebase, as we are using free plan. The features include scheduling a target reminder notification to only the target elderly user.

References

Gallardo, E. G. (2023, January 9). *What Is MVVM Architecture? (Definition, Advantages) | Built In*. Builtin.com. https://builtin.com/software-engineering-perspectives/mvvm-architecture

Sutanto, M. (2019, July 22). *Android Application Testing with Firebase Robo Test*. Wantedly Engineering. https://medium.com/wantedly-engineering/android-application-testing-with-firebase-robo-test-c674e1754298

Workload matrix

Name and TP number	User role	Features
Ching Jia Zhong TP074569	Caregiver	• Login • Register • Forgot Password • Caregiver Dashboard • View Profile • Select Elderly • Add Elderly • Delete Elderly from list • Select Healthcare Provider • Add Healthcare Provider • Delete healthcare provider from list • Chat • Chat notification • View Medication Schedule • Add Medication Schedule • Delete Medication Schedule for elderly • View alerts • Delete alerts • View elderly location
Tham Jie Wei TP074224	Elderly User	• Setting • Edit Profile • Sign out • Elderly Dashboard

		• View health data • Log health data • Appointment Booking Page • Share real-time location with caregivers • Notification to take medicine • Notification of missed dossed • Alert notification if the elderly health metric abnormal • Dark Mode • Font size adjustment • Emergency call function • Mark the reminder or medication as completed • Send Push Notification When the elderly user's health metrics in abnormal
Mohammed Fadhi P Safeer TP073289	Healthcare Provider	• Healthcare provider dashboard • View appointment • Update appointment • Delete appointment • View Alert

		• Select Caregiver • Add Caregiver • Delete caregiver from list • View Patient List • Add Patient • Delete patient from list