



MOBILE APP ENGINEERING
(CT124-3-2MAE)
GROUP ASSIGNMENT
REPORT

INTAKE : APU2F2402CS(AI)
LECTURER : AMAD ARSHAD
HAND-IN DATE : 13/11/2024
HAND-OUT DATE : 13/11/2024
GROUP :




NAME	TP NUMBER	SIGNATURE
Angelina Leanore	TP072929	
Chua Jun Yi	TP065882	
Chang Zheng Fang	TP066899	

Table of Contents

1.0	Introduction	3
2.0	Problem Solving and Design	4
2.1	Functionalities	4
2.2	Use Case Diagram.....	5
2.3	System Architecture Design.....	6
2.4	Dynamic Data Management	7
2.5	CRUD	52
2.6	Validation	60
2.7	User Permission	61
2.8	Wireframe	62
3.0	User Manual.....	69
4.0	Automated System Testing	83
5.0	Conclusion	83
6.0	References.....	84
7.0	Appendix.....	Error! Bookmark not defined.
8.0	Workload Matrix	84

1.0 Introduction

Today's society is fast-paced and multifaceted, meaning that people are subjected to very high levels of stress and pressure due to the need to balance both work and family obligations, as well as having to deal with constant other demand on their time. In Gallup report indicated that in the year 2021, out of all the respondents from across the globe, 41% of them claimed to be stressed whereas 42% admitted to being worried, which means that the rise in adverse feelings experienced by people all over the world has substantially increased (Ray, 2024). This indicates that there is a dire need for remedies, which will assist people to balance out their lives, whilst enhancing their productivity and health. ZenAssist was created in order to cater to these growing concerns by being full management of day-to-day activities coupled with a care approach to mental and physical health.

ZenAssist is a one-stop application for individuals and families and combines a suite of productivity and wellness applications in one application. It contains functions such as a to-do list, tips on daily planning, and a meal planner aimed at helping the user achieve their target without endangering their health. App wellness reminders and meal planning tools are also included in the system to help health outcome expectancies in addition to task performance.

ZenAssits, on the other hand, provides a solution to the stress and mental health problems brought about by modern society which the American Psychological Association among other organizations has reported an increase in (Betune, 2023). It provides a modern solution to the problem of balancing the demands of contemporary society and helps in promoting a productive, satisfying, and most importantly healthy lifestyle to its users. The marriage of productivity and well aspects in ZenAssist gave its users the power to excel, both in personal and professional aspects of their lives.

2.0 Problem Solving and Design

2.1 Functionalities

ZenAssist offers the following features to ensure the user satisfaction:

- Users, families, and admin can establish new accounts by giving information such as email addresses and passwords.
- Users, families, and admin can log in to already created accounts using their emails and passwords.
- Users, families, and admin can log out of the account.
- Users and families members can browse the calendar on the main page to easily view the task dates.
- Users and families can make a to-do list.
- Users and families can clear their to-do lists by completing the activity.
- Users and families may plan their weekly meals.
- Users and families may see the food suggested in by the algorithm.
- Users and families may see the recipes supplied in the app.
- Users and families members may browse their inboxes, reminder, and assigned chores.
- Users and families can look for certain chores.
- Users and families can provide comments to the administrator.
- Users and families can delegate tasks to individuals.
- Family admin can invite users to contribute on the family to-do lists.
- Family admin can remove people from collaborating on the family to-do list.
- Family admin can transfer the admin to a user.
- Admin may access the mailbox, which shows the update whether someone register for the account of fresh feedback.
- Admin can examine the feedback supplied by users.
- Admin can choose their option of tracking by days, months, or years.
- Admins can monitor system performance.
- Admins may examine the number of new users that have registered in the system.
- Admin may examine the total number of tasks completed by users.
- Admin can see how many of features utilized from the app by the users.
- Admin may examine the user interaction statistics.
- Admin can examine the feedback sent by the users.
- Admin can order the feedback list ascending and descending.
- Admin can look for feedback depending on the ID number.

2.2 Use Case Diagram



Figure 1: Use Case Diagram

2.3 System Architecture Design

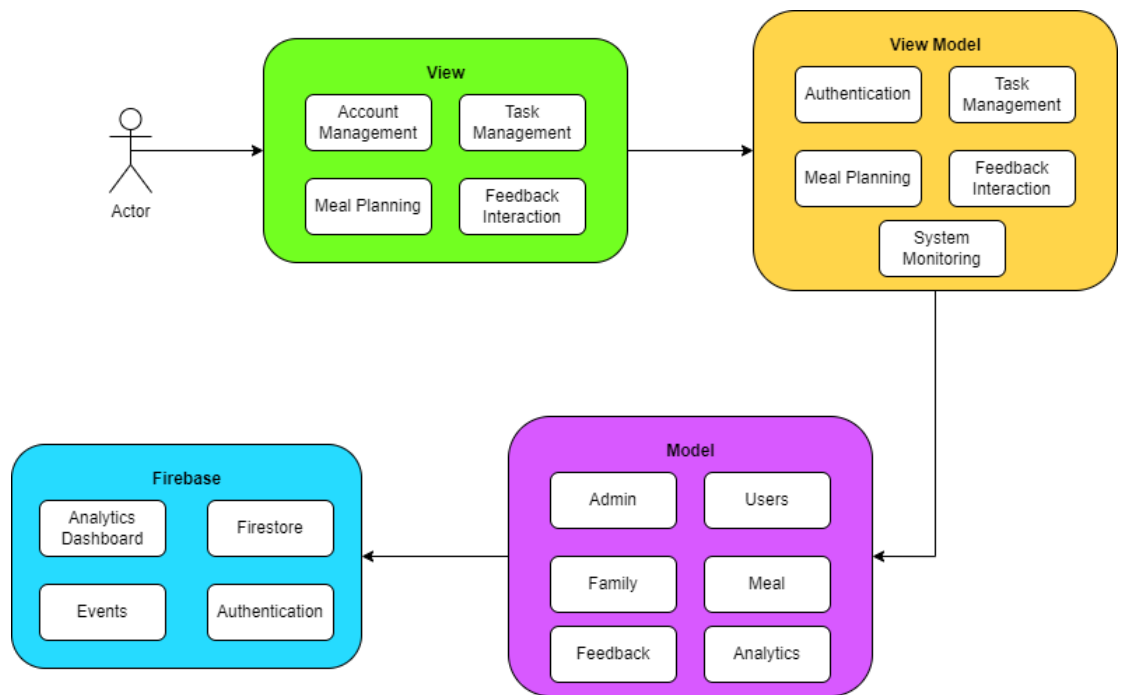


Figure 2: ZenAssist System Architecture Design

The design system architecture of ZenAssist revolves around the Model View ViewModel design pattern with firebase for all its backend services. The view segment of the application which is open to users, members of their families, and admin contains parts such as Account Management, Task Management, Meal Management, and Feedback. This enables them to create accounts, give tasks, and plan for meals as well as give feedback without hitches. ViewModel is also the architect for the encompassed logic behind the core user interface known as the Model and even the centre of focus in this application which is the Model. It includes components for Authentication, Task Management, Meal Planning, Feedback Interaction, and System Monitoring. This layer ensures that data is processed and provided to the view in a user-friendly format.

The Model represents data structures such as Admin, Users, Family, Meal, Feedback, and Analytics, maintaining the app's essential information. Firebase integrates as the backend, offering Authentication, Firestore database, an Analytics Dashboard, and Events tracking. This setup real-time data synchronization, user authentication, and activity tracking, enabling ZenAssist to provide a responsive and engaging user experience.

2.4 Dynamic Data Management

```
// File generated by FlutterFire CLI.
// ignore_for_file: type=lint
import 'package:firebase_core/firebase_core.dart' show FirebaseOptions;
import 'package:flutter/foundation.dart'
    show defaultTargetPlatform, kIsWeb, TargetPlatform;

static const FirebaseOptions web = FirebaseOptions(
  apiKey: 'AIzaSyCrs14xaT4JMj0e9x9jZ2-I0MpRAFSbQhw',
  appId: '1:650216056799:web:bc59b1d745e9ad1a02866a',
  messagingSenderId: '650216056799',
  projectId: 'zen-assist-app-ab614',
  authDomain: 'zen-assist-app-ab614.firebaseio.com',
  storageBucket: 'zen-assist-app-ab614.firebaseio.com',
  measurementId: 'G-KYRXYRY6RX',
);
```

Figure 3 Dynamic Data Management

In the process explaining a constant "FirebaseOptions" object called "web," this code configures Firebase for a Flutter web application. This object contains the configuration information required to link the application to Firebase services, such as parameters like "apiKey," "appId," "messagingSenderId," "projectId," "authDomain," "storageBucket," and "measurementId." Interaction with Firebase services like authentication, storage, and messaging is made possible via these values, which are unique to the Firebase project. When dealing with Firebase initialization in the application, the code also imports the "firebase_core" package. It also imports the "flutter/foundation.dart" package to access platform-specific tools like "defaultTargetPlatform," which indicates the current platform (iOS, Android, or Web), and "kIsWeb," a boolean that indicates whether the application is running on the web. In essence, this configuration takes platform-specific parameters into account and guarantees that the Flutter web application is properly connected to Firebase with the right configuration.

Tracking Dashboard Engagement with Firebase Analytics [_logAppOpenEvent() and _logAppCloseEvent()]

```
void _logAppOpenEvent() async {  
  await analytics.logEvent(  
    name: 'app_open',  
    parameters: {'screen': 'dashboard_screen'},  
  );  
}  
  
void _logAppCloseEvent() async {  
  await analytics.logEvent(  
    name: 'app_close',  
    parameters: {'screen': 'dashboard_screen'},  
  );  
}
```

Figure 4 Admin tracking dashboard engagement

The `_logAppOpenEvent` and `_logAppCloseEvent` functions track user interaction with the dashboard screen using Firebase Analytics. `_logAppOpenEvent` logs an `app_open` event when the app opens, while `_logAppCloseEvent` logs an `app_close` event when the app closes. Together, these functions provide insights into user engagement and session duration on the dashboard.

Accurate Session Tracking for User Engagement Analysis [_startSession() and _endSession]

```
void _startSession() {  
  sessionStartTime = DateTime.now();  
}  
  
Future<void> _endSession() async {  
  if (sessionStartTime != null) {  
    DateTime endTime = DateTime.now();  
    await FirebaseFirestore.instance.collection('sessions').add({  
      'startTime': sessionStartTime,  
      'endTime': endTime,  
    });  
  }  
}
```

Figure 5 Admin Accurate session tracking

The `_startSession` function logs the start time when a user opens the app, marking the session's beginning. This start time, saved as a `DateTime` object, is used to track user activity duration. The `_endSession` function records the end time, calculates the session duration, and saves this data to Firebase Firestore. Together, these functions enable accurate tracking and analysis of user session lengths.

Calculate Average Session Time Function [calculateAverageSessionTime()]

```
Future<double> calculateAverageSessionTime() async {
  QuerySnapshot sessionsSnapshot =
    await FirebaseFirestore.instance.collection('sessions').get();

  if (sessionsSnapshot.docs.isEmpty) return 0;

  double totalDuration = 0;
  int sessionCount = sessionsSnapshot.docs.length;

  for (var session in sessionsSnapshot.docs) {
    DateTime? startTime = (session['startTime'] as Timestamp)?.toDate();
    DateTime? endTime = (session['endTime'] as Timestamp)?.toDate();

    if (startTime != null && endTime != null) {
      double duration = endTime.difference(startTime).inSeconds.toDouble();
      totalDuration += duration;
    }
  }

  return totalDuration / sessionCount;
}
```

Figure 6 Admin calculate average session time function

This function calculates the average session duration by querying all session records from Firestore. It retrieves the start and end times for each session, calculating the duration in seconds. After summing all session durations, the total is divided by the number of sessions to find the average. The result is a value that helps the app monitor user engagement and app usage trends over time.

Calculate Hourly Average Session Time Function [calculateHourlyAverageSessionTime()]

```
Future<Map<int, double>> calculateHourlyAverageSessionTime() async {
  QuerySnapshot sessionsSnapshot =
    await FirebaseFirestore.instance.collection('sessions').get();

  if (sessionsSnapshot.docs.isEmpty) return {};

  Map<int, List<double>> sessionDurationsByHour = {};

  for (var session in sessionsSnapshot.docs) {
    DateTime? startTime = (session['startTime'] as Timestamp)?.toDate();
    DateTime? endTime = (session['endTime'] as Timestamp)?.toDate();

    if (startTime != null && endTime != null) {
      int hour = startTime.hour;
      double duration = endTime.difference(startTime).inSeconds.toDouble();

      if (!sessionDurationsByHour.containsKey(hour)) {
        sessionDurationsByHour[hour] = [];
      }
      sessionDurationsByHour[hour]!.add(duration);
    }
  }
}
```

Figure 7 Admin Calculate hourly average session time function

This function calculates the average session time for each hour of the day by grouping sessions based on the hour they started. It calculates the duration of each session and groups these durations by the hour they occurred. Once the data is grouped, it computes the average session duration for each hour of the day, providing insights into user behavior throughout the day. This helps identify peak engagement times and user activity patterns during specific hours.

Show Setting Dialog [_showSettingsDialog ()]

```
void _showSettingsDialog() {
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: Text('Settings'),
        content: Column(
          mainAxisAlignment: MainAxisAlignment.min,
          children: [
            Divider(),
            ListTile(
              leading: Icon(Icons.logout),
              title: Text('Log Out'),
              onTap: () {
                Navigator.of(context).pop();
                Navigator.pushReplacement(
                  context,
                  MaterialPageRoute(builder: (context) => ZenAssistApp()),
                );
              },
            ),
          ],
        ),
        actions: [
          TextButton(
            child: Text('Close'),
            onPressed: () {
              Navigator.of(context).pop();
            },
          ),
        ],
      );
    },
  );
}
```

Figure 8 Admin Show Setting Dialog

This function presents a dialog box containing app settings options, such as the ability to log out. When the "Log Out" option is selected, the user is redirected to the login screen, allowing for a clean session termination. The dialog also has a "Close" button to dismiss the settings without making changes. This function is essential for providing users with a way to manage their session settings easily.

Customizable Stat Card Widget Function [_buildStatCard()]

```
Widget _buildStatCard(
  BuildContext context,
  String title,
  String count,
  String subtitle,
  Color color,
  IconData icon, {
  double titleFontSize = 14,
  double countFontSize = 24,
  double subtitleFontSize = 12,
}) {
  return Container(
    padding: EdgeInsets.all(16),
    decoration: BoxDecoration(
      color: color,
      borderRadius: BorderRadius.circular(8),
    ),
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
        Text(
          title,
          style:
            TextStyle(fontSize: titleFontSize, fontWeight: FontWeight.bold),
        ),
        SizedBox(height: 8),
        Row(
          mainAxisAlignment: MainAxisAlignment.spaceBetween,
          children: [
            Text(
              count,
              style: TextStyle(
                fontSize: countFontSize, fontWeight: FontWeight.bold),
            ),
            Icon(icon, size: 48),
          ],
        ),
        SizedBox(height: 8),
        Text(
          subtitle,
          style:
            TextStyle(fontSize: subtitleFontSize, color: Colors.grey[600]),
        ),
      ],
    ),
  );
}
```

Figure 9 Customizable stat card widget function

This function creates a customizable stat card widget that displays key metrics such as titles, counts, and icons. It takes parameters for the title, count, subtitle, color, and icon, allowing for flexible customization of the card's appearance. The widget is used throughout the dashboard to display important statistics like user count or task completion rates. This ensures that data is presented in a visually appealing and easy-to-read format for the user.

Log Feature Usage Function [_logFeatureUsage (String featureName)]

```
// Function to log feature usage in Firestore
void _logFeatureUsage(String featureName) {
    FirebaseFirestore.instance
        .collection('stats')
        .doc('featureUtilization')
        .update({
            featureName: FieldValue.increment(1),
        }).catchError((error) {
            print("Failed to log feature usage: $error");
        });
}
```

Figure 10 Log feature usage function

The _logFeatureUsage function logs the usage of a specified feature by updating a Firestore document in the stats collection. It increments the count of a field named after featureName by 1 to track how often the feature is used. If the update fails, the catchError block prints an error message indicating the failure. This function help for monitor feature usage for analytics purposes within the Firestore database.

Get Time Ago Function [_getTimeAgo(Timestamp timestamp)]

```
// Helper function to convert Firestore timestamp to days ago
String _getTimeAgo(Timestamp timestamp) {
    final now = DateTime.now();
    final feedbackDate = timestamp.toDate();
    final difference = now.difference(feedbackDate).inDays;
    return difference == 0 ? 'Today' : '$difference days ago';
}
```

Figure 11 Get time ago function

The _getTimeAgo function calculates how long ago a given Timestamp was from the current date. It first converts the Timestamp to a DateTime object using toDate(). Then, it calculates the difference in days between the current date (now) and the feedback date (feedbackDate). If the difference is 0, it returns 'Today'; otherwise, it returns the number of days ago in the format 'X days ago'.

Fetching Feedback Details [_getFeedbackDetails()]

```
class FeedbackDetailPage extends StatelessWidget {
  final String id;

  FeedbackDetailPage({required this.id});

  Future<DocumentSnapshot> _getFeedbackDetails() async {
    try {
      return await FirebaseFirestore.instance
        .collection('feedback')
        .doc(id)
        .get();
    } catch (e) {
      print('Error fetching feedback details: $e');
      rethrow;
    }
  }
}
```

Figure 12 Fetching feedback details

This function retrieves detailed information for a single feedback item when the user navigates to the feedback's detail page. It performs a Firestore read on the document with the specified ID in the "feedback" collection. If the document exists, it returns the document snapshot for displaying details like email, title, and message. If an error occurs during retrieval, it catches and rethrows the error, while also printing an error message for debugging purposes.

Displaying Message Details [_showMessageDetail()]

```
void _showMessageDetail(BuildContext context, Map<String, String> message) {
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: Text(message['title'] ?? ''),
        content: Text(message['content'] ?? ''),
        actions: [
          TextButton(
            onPressed: () {
              Navigator.of(context).pop();
            },
            child: Text('Close'),
          ),
        ],
      );
    },
  );
}
```

Figure 13 Display message details

This function displays a dialog with the details of a selected message when a user taps on a message card. It takes the message data and opens an AlertDialog showing the message title and content. The dialog has a "Close" button that allows the user to dismiss the message details and return to the inbox. It provides an interactive way to view more information about each message.

Building Message Cards [_buildMessageCard()]

```
Widget _buildMessageCard(BuildContext context, Map<String, String> message) {  
  return Card(  
    shape: RoundedRectangleBorder(  
      borderRadius: BorderRadius.circular(8),  
    ),  
    elevation: 2,  
    margin: EdgeInsets.symmetric(vertical: 8),  
    child: ListTile(  
      contentPadding: EdgeInsets.all(16),  
      title: Text(  
        message['title'] ?? '',  
        style: TextStyle(fontSize: 16, fontWeight: FontWeight.bold),  
      ),  
      subtitle: Column(  
        crossAxisAlignment: CrossAxisAlignment.start,  
        children: [  
          Text(  
            message['subtitle'] ?? '',  
            style: TextStyle(color: Colors.grey[600]),  
            maxLines: 1,  
            overflow: TextOverflow.ellipsis,  
          ),  
          SizedBox(height: 8),  
          Text(  
            message['date'] ?? '',  
            style: TextStyle(color: Colors.grey, fontSize: 12),  
          ),  
        ],  
      ),  
      trailing: Icon(Icons.chevron_right),  
      onTap: () => _showMessageDetail(context, message),  
    ),  
  );  
}
```

Figure 14 Building message cards

This function constructs each message card for the inbox screen. It takes the message data and uses a ListTile widget to display the message's title, subtitle, and date in a structured way. The card has a rounded border, and tapping it navigates the user to a detailed view of the message. It handles layout and design for each message in the inbox list.

Toggle Task Completion Function [_toggleTask ()]

```
void _toggleTask(String id, bool currentStatus) {  
  tasksCollection.doc(id).update({'completed': !currentStatus});  
  
  // Update the completion count in Firestore when a task is completed  
  if (!currentStatus) {  
    FirebaseFirestore.instance  
      .collection('stats')  
      .doc('taskCompletion')  
      .update({'count': FieldValue.increment(1)});  
  } else {  
    FirebaseFirestore.instance  
      .collection('stats')  
      .doc('taskCompletion')  
      .update({'count': FieldValue.increment(-1)});  
  }  
}
```

Figure 15 Toggling task completion function

This function toggles the completion status of a task when the user interacts with the checkbox. If a task is marked as completed, it updates the task's "completed" field in Firestore. Additionally, it increments or decrements the task completion count in the Firestore "stats" collection. This provides a dynamic way to track the completion status of tasks and update the UI accordingly.

Show Family Option Function [_showFamilyOptionsDialog()]

```
// Show dialog for choosing family creation or joining an existing one
void _showFamilyOptionsDialog() {
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: const Text('Family Setup'),
        content: Column(
          mainAxisAlignment: MainAxisAlignment.min,
          children: [
            ListTile(
              title: const Text('Create a new family'),
              leading: Radio<bool>{
                value: true,
                groupValue: _isCreatingFamily,
                onChanged: (value) {
                  setState(() {
                    _isCreatingFamily = value!;
                    _familyId = null; // Reset familyId if creating new family.
                  });
                  Navigator.pop(context);
                },
              ),
            ),
            ListTile(
              title: const Text('Join an existing family'),
              leading: Radio<bool>{
                value: false,
                groupValue: _isCreatingFamily,
                onChanged: (value) {
                  setState(() {
                    _isCreatingFamily = value!;
                  });
                  Navigator.pop(context);
                  _showFamilyIdDialog();
                },
              ),
            ),
          ],
        ),
      );
    },
  );
}
```

Figure 16 Show family option function

This function shows a dialog with options for either creating a new family or joining an existing one. It uses a Radio button to allow the user to choose between the two options. When the user selects "Create a new family," it resets the family ID, while selecting "Join an existing family" prompts the user to enter a family ID. After the choice is made, the dialog closes. The function updates the `_isCreatingFamily` state based on the user's selection.

Show Family ID Function [_showFamilyIdDialog ()]

```
// Show dialog for entering familyId when joining an existing family
void _showFamilyIdDialog() {
  String familyIdInput = '';

  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: const Text('Enter Family ID'),
        content: TextField(
          onChanged: (value) {
            familyIdInput = value;
          },
          decoration: const InputDecoration(hintText: 'Enter Family ID'),
        ),
        actions: [
          TextButton(
            onPressed: () {
              Navigator.pop(context);
            },
            child: const Text('Cancel'),
          ),
          TextButton(
            onPressed: () async {
              // Check if the family exists
              var familyDoc = await _firestore.collection('families').doc(familyIdInput).get();
              if (familyDoc.exists) {
                setState(() {
                  _familyId = familyIdInput;
                });
                ScaffoldMessenger.of(context).showSnackBar(
                  const SnackBar(content: Text('Joined the family successfully!')),
                );
              } else {
                ScaffoldMessenger.of(context).showSnackBar(
                  const SnackBar(content: Text('Family ID not found.')),
                );
              }
              Navigator.pop(context);
            },
            child: const Text('Join'),
          ),
        ],
      );
    },
  );
}
```

Figure 17 Show family ID function

This function displays a dialog for the user to input a family ID if they are choosing to join an existing family. It waits for the user to enter the ID and then checks the Firestore database to see if the family exists. If the family ID is valid, the user is successfully joined to the family and the family ID is saved. If the family does not exist, a message is displayed indicating the failure. The dialog is then closed after the operation is completed.

Homepage Widget Function

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Container(
      decoration: const BoxDecoration(
        gradient: LinearGradient(
          begin: Alignment.topCenter,
          end: Alignment.bottomCenter,
          colors: [
            Color(0xFF689080),
            Color(0xFFA4C382),
          ],
        ),
      ),
    ),
    child: SafeArea(
      child: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            // Logo
            Image.asset(
              'assets/images/ZenAssistLogo.png',
              height: 120,
              width: 120,
            ),
            const SizedBox(height: 32),
            // Welcome Text
            const Text(
              'Welcome to ZenAssist',
              style: TextStyle(
                fontSize: 28,
                fontWeight: FontWeight.bold,
                color: Colors.white,
              ),
            ),
            const SizedBox(height: 16),
            const Padding(
              padding: EdgeInsets.symmetric(horizontal: 32),
              child: Text(
                'Your personal assistant for a balanced and productive life',
                textAlign: TextAlign.center,
                style: TextStyle(
                  fontSize: 16,
                  color: Colors.white70,
                ),
              ),
            ),
          ],
        ),
      ),
    ),
  );
}
```

Figure 18 Homepage widget function

The Homepage class is a StatelessWidget that defines the main screen of the app, which includes a gradient background, a logo, welcome text, a description, and a button to navigate to the login page. The build function returns a Scaffold with a Container containing the UI layout, centered using a SafeArea and Column. The Image.asset widget displays the logo from the app's assets at a fixed size of 120x120 pixels. The ElevatedButton widget creates a styled button that, when pressed, triggers the Navigator.push function to navigate to the LoginPage. The button's style is customized with a white background, padding, and rounded corners. Navigator.push uses the MaterialPageRoute to transition smoothly to the login screen.

Login Function [_login()]

```
Future<void> _login() async {
  setState(() {
    _isLoading = true; // Start loading
  });

  try {
    // Use Firebase Authentication to sign in
    UserCredential userCredential =
      await FirebaseAuth.instance.signInWithEmailAndPassword(
        email: _emailController.text.trim(),
        password: _passwordController.text.trim(),
      );

    // If successful, navigate to the main page
    if (userCredential.user != null) {
      Navigator.pushReplacement(
        context,
        MaterialPageRoute(builder: (context) => const MainPage()),
      );
    }
  } catch (e) {
    // Handle errors (e.g., invalid credentials)
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('Failed to sign in: ${e.toString()}')),
    );
  } finally {
    setState(() {
      _isLoading = false; // Stop loading
    });
  }
}
```

Figure 19 Login function

The `_login` function is responsible for handling the user authentication process. It starts by setting the `_isLoading` state to true, which triggers a loading indicator on the UI. The function then attempts to sign in the user using Firebase Authentication's `signInWithEmailAndPassword` method with the email and password entered by the user. If the login is successful, the user is navigated to the `MainPage`, otherwise, an error message is shown using a `SnackBar`. After the process, the loading indicator is hidden by setting `_isLoading` to false.

ElevatedButton

```
ElevatedButton(
  onPressed: _isLoading
    ? null // Disable the button while loading
    : () {
        if (_formKey.currentState!.validate()) {
          _login(); // Call the login function
        }
      },
  style: ElevatedButton.styleFrom(
    backgroundColor: Theme.of(context).primaryColor,
    minimumSize: const Size(double.infinity, 50),
  ),
  child: _isLoading
    ? const CircularProgressIndicator(
        color: Colors.white,
      ) // Show loading indicator when signing in
    : const Text(
        'Login',
        style: TextStyle(fontSize: 18),
      ),
),
```

Figure 20 ElevatedButton widget

The `ElevatedButton` widget represents the login button in the UI. When clicked, it triggers the `_login` function if the form validation passes. If the app is in the loading state (`_isLoading`), the button is disabled to prevent multiple clicks. While loading, a `CircularProgressIndicator` is displayed on the button to indicate that the login process is ongoing. After the user successfully logs in, the button behavior allows for seamless navigation to the next page without any interruptions.

Text Editing Function

```
TextFormField(
  controller: _emailController,
  decoration: const InputDecoration(
    labelText: 'Email',
    border: OutlineInputBorder(),
  ),
  validator: (value) {
    if (value == null || value.isEmpty) {
      return 'Please enter your email';
    }
    return null;
  },
),
const SizedBox(height: 20),
TextFormField(
  controller: _passwordController,
  obscureText: true,
  decoration: const InputDecoration(
    labelText: 'Password',
    border: OutlineInputBorder(),
  ),
  validator: (value) {
    if (value == null || value.isEmpty) {
      return 'Please enter your password';
    }
    return null;
  },
),
),
```

Figure 21 Text editing function

These two controllers are used to manage the state of the email and password input fields. They are attached to the respective `TextFormField` widgets, allowing you to retrieve or modify the input text. The `TextEditingController` listens to changes in the input field and updates the text, which is useful when you need to access the user's email and password for authentication. These controllers also help clear the text fields programmatically when needed. By using these controllers, the app can easily handle user input and maintain a clean UI state.

Load Events Function [_loadCalendarEvents()]

```
Future<void> _loadCalendarEvents() async {
  if (_auth.currentUser == null) return;

  setState(() => _isLoading = true);

  try {
    final startOfMonth = DateTime(_selectedDate.year, _selectedDate.month, 1);
    final endOfMonth =
      DateTime(_selectedDate.year, _selectedDate.month + 1, 0);

    final snapshot = await _firestore
      .collection('events')
      .where('userId', isEqualTo: _auth.currentUser!.uid)
      .where('dateTime',
        isGreaterThanOrEqualTo: Timestamp.fromDate(startOfMonth))
      .where('dateTime',
        isLessThanOrEqualTo: Timestamp.fromDate(endOfMonth))
      .orderBy('dateTime')
      .get();

    _calendarEvents = snapshot.docs
      .map((doc) => CalendarEvent.fromMap(doc.data()))
      .toList();

    setState(() {});
  } catch (e) {
    _showErrorSnackBar('Error loading events: $e');
  } finally {
    setState(() => _isLoading = false);
  }
}
```

Figure 22 Load Events function

This method fetches calendar events for the current user from Firestore based on the selected month. It uses a Firestore query to filter events by userId and date range for the selected month. It updates the _calendarEvents list with the fetched events. The setState() method is called to trigger a rebuild of the widget after the events are loaded. If there is an error during the fetch process, an error message is shown to the user.

Add Events Function [_addCalendarEvent ()]

```
Future<void> _addCalendarEvent() async {
  if (_eventTitle == null || _eventDateTime == null) {
    _showErrorSnackBar('Please fill in all required fields');
    return;
  }

  try {
    final newEvent = CalendarEvent(
      id: DateTime.now().millisecondsSinceEpoch.toString(),
      title: _eventTitle!,
      dateTime: _eventDateTime!,
      userId: _auth.currentUser!.uid,
    );

    await _firestore
      .collection('events')
      .doc(newEvent.id)
      .set(newEvent.toMap());

    // Clear input fields and reload events
    _eventTitle = null;
    _eventDateTime = null;
    await _loadCalendarEvents();
    _showSuccessSnackBar('Event added successfully');
  } catch (e) {
    _showErrorSnackBar('Error saving event: $e');
  }
}
```

Figure 23 Add Events function

This function is responsible for creating a new calendar event and saving it to Firestore. It checks that both the event title and date are provided before proceeding. If the fields are valid, a new `CalendarEvent` object is created and added to the Firestore database. Once the event is successfully added, the event list is reloaded, and the input fields are cleared. If an error occurs, an error message is shown to the user.

Display Error Message [_showErrorSnackBar ()]

```
void _showErrorSnackBar(String message) {  
  ScaffoldMessenger.of(context).showSnackBar(  
    SnackBar(  
      content: Text(message),  
      backgroundColor: Colors.red,  
    ),  
  );  
}
```

Figure 24 Display error message

This function shows an error message in the form of a SnackBar at the bottom of the screen. The SnackBar has a red background and displays the message passed to it as a parameter. It is typically used to alert the user when something goes wrong, like an error in loading or saving data. The ScaffoldMessenger.of(context) is used to display the SnackBar. It helps improve the user experience by giving real-time feedback about errors.

Display Success Message [_showSuccessSnackBar ()]

```
void _showSuccessSnackBar(String message) {  
  ScaffoldMessenger.of(context).showSnackBar(  
    SnackBar(  
      content: Text(message),  
      backgroundColor: Colors.green,  
    ),  
  );  
}
```

Figure 25 Display success message

Similar to the error SnackBar, this function shows a success message when an operation is successful. It takes a message as a parameter and displays it using a green-colored SnackBar. This provides visual feedback to the user, indicating that the action (like adding or deleting an event) was successful. It uses the ScaffoldMessenger to display the SnackBar. This function helps keep the user informed about the result of their actions.

Build Calendar Header [_buildCalendarHeader()]

```
Widget _buildCalendarHeader() {
  return Padding(
    padding: const EdgeInsets.all(8.0),
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: [
        IconButton(
          icon: const Icon(Icons.chevron_left),
          onPressed: () {
            setState(() {
              _selectedDate = DateTime(
                _selectedDate.year,
                _selectedDate.month - 1,
                _selectedDate.day,
              );
            });
            _loadCalendarEvents();
          },
        ),
        Text(
          DateFormat('MMMM yyyy').format(_selectedDate),
          style: const TextStyle(fontSize: 18, fontWeight: FontWeight.bold),
        ),
        IconButton(
          icon: const Icon(Icons.chevron_right),
          onPressed: () {
            setState(() {
              _selectedDate = DateTime(
                _selectedDate.year,
                _selectedDate.month + 1,
                _selectedDate.day,
              );
            });
          },
        ),
      ],
    ),
  );
}
```

Figure 26 Build Calendar Header

This function creates the header of the calendar page, which includes buttons to navigate between months. It displays the current month and year in a Text widget and allows the user to move to the previous or next month using the left and right arrows. It uses the setState() method to update the selected month when the user presses the navigation buttons. The header is displayed at the top of the screen, providing context for the calendar view. This function helps manage month navigation.

Build Calendar Grid [_buildCalendarGrid()]

```
Widget _buildCalendarGrid() {  
  return GridView.builder(  
    gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(  
      crossAxisCount: 7,  
      crossAxisSpacing: 4.0,  
      mainAxisSpacing: 4.0,  
    ),  
    itemCount: 42,  
    itemBuilder: (context, index) {  
      final date = _getDateForIndex(index);  
      final events = _getEventsForDate(date);  
      final isToday = _isToday(date);  
      final isSelectedMonth = date.month == _selectedDate.month;  
  
      return GestureDetector(  
        onTap: () => _showAddEventDialog(date),  
        child: Container(  
          decoration: BoxDecoration(  
            border: Border.all(  
              color: isToday ? Colors.blue : Colors.grey[300]!,  
              width: isToday ? 2 : 1,  
            ),  
            color: isSelectedMonth ? Colors.white : Colors.grey[100],  
          ),  
          child: Stack(  
            children: [  
              Align(  
                alignment: Alignment.topLeft,  
                child: Padding(  
                  padding: const EdgeInsets.all(2.0),  
                  child: Text(  
                    '${date.day}'  
                  ),  
                ),  
            ],  
          ),  
        ),  
      );  
    },  
  );  
}
```

Figure 27 Build calendar grid

This function generates the calendar grid using a `GridView.builder`. It calculates the dates for each cell of the calendar and checks if there are events on those dates. It uses the `GestureDetector` widget to allow the user to tap on any date and add an event. The grid displays the days of the month, and events are indicated by colored containers. This function visually represents the calendar and provides interactivity.

Check Date [_isToday()]

```
bool _isToday(DateTime date) {  
  final now = DateTime.now();  
  return date.year == now.year &&  
    date.month == now.month &&  
    date.day == now.day;  
}
```

Figure 28 Check date

This function checks if a given date is today. It compares the year, month, and day of the input date with the current date (`DateTime.now()`). It returns a boolean value, true if the dates match, indicating that the date is today. The function helps highlight the current date in the calendar with special styling. This check is essential for user interaction, marking today's date as special.

Build Event Indicator [_buildEventIndicator ()]

```
Widget _buildEventIndicator(CalendarEvent event) {  
  return Container(  
    margin: const EdgeInsets.only(bottom: 2),  
    height: 12,  
    decoration: BoxDecoration(  
      color: TaskPriorityColors.getColor(event.priority).withOpacity(0.8),  
      borderRadius: BorderRadius.circular(6),  
    ),  
    child: Tooltip(  
      message: '${event.title} (${event.priority})',  
      child: const SizedBox.expand(),  
    ),  
  );  
}
```

Figure 29 Build Event indicator

This function displays a small colored indicator on top of the calendar day to show that there are events scheduled for that date. It takes a `CalendarEvent` object as a parameter and uses its color and title for the indicator. The event's title is shown in a `Tooltip` to help the user identify the event. The indicator appears in the calendar grid cell corresponding to the event's date. This function helps highlight dates with events.

Key Calendar Functions [_getDateForIndex() and _getEventsForDate()]

```
DateTime _getDateForIndex(int index) {  
    final firstDayOfMonth =  
        DateTime(_selectedDate.year, _selectedDate.month, 1);  
    final firstDayWeekday = firstDayOfMonth.weekday;  
    final firstDisplayedDate =  
        firstDayOfMonth.subtract(Duration(days: firstDayWeekday - 1));  
    return firstDisplayedDate.add(Duration(days: index));  
}  
  
List<CalendarEvent> _getEventsForDate(DateTime date) {  
    return _calendarEvents  
        .where((event) =>  
            event.dateTime.year == date.year &&  
            event.dateTime.month == date.month &&  
            event.dateTime.day == date.day)  
        .toList();  
}
```

Figure 30 Key calendar function

The functions `_getDateForIndex(int index)` and `_getEventsForDate(DateTime date)` are designed to enhance a calendar's functionality. `_getDateForIndex` calculates the specific date for a given index in the calendar grid, accounting for the first day of the month and adjusting for any offset due to the starting weekday, allowing the grid to display accurate dates, including those from adjacent months if needed. `_getEventsForDate` retrieves events for a specific date by matching the event's year, month, and day to the input date, returning a list of relevant `CalendarEvent` objects. Together, these functions ensure the calendar grid displays accurate dates and reflects the correct events for each day.

Delete Calendar Event [_deleteCalendarEvent()]

```
Future<void> _deleteCalendarEvent(CalendarEvent event) async {  
    try {  
        await _firestore.collection('events').doc(event.id).delete();  
        await _loadCalendarEvents();  
        _showSuccessSnackBar('Event deleted successfully');  
    } catch (e) {  
        _showErrorSnackBar('Error deleting event: $e');  
    }  
}
```

Figure 31 Delete calendar event

This function deletes a specific event from Firestore based on the event's ID. It removes the event from the database and then reloads the event list to reflect the change. If successful, a success message is shown; otherwise, an error message is displayed. The function helps maintain the event list by allowing users to remove unwanted or outdated events. This operation keeps the calendar data up to date.

Load User Family Data [_loadUserFamilyData()]

```
Future<void> _loadUserFamilyData() async {
  try {
    final user = _auth.currentUser;
    if (user != null) {
      final userData =
        await _firestore.collection('users').doc(user.uid).get();
      if (userData.exists) {
        setState(() {
          familyId = userData.data()?['familyId'];
          isCreator = userData.data()?['role'] == 'creator';
        });
      }
      _setupStreams();
      setState(() {
        _isLoading = false;
      });
    }
  } catch (e) {
    print('Error loading user data: $e');
    setState(() {
      _isLoading = false;
    });
  }
}
```

Figure 32 Load user family data

This function loads the user's data from Firebase when the screen is initialized. It checks if the user is logged in and then retrieves their familyId and role (creator or not) from the Firestore database. The function updates the state with this data and sets up task streams (_personalTasksStream and _familyTasksStream). It also handles errors by printing them to the console. This function is important for determining whether the user can create family tasks or manage other family-related tasks.

Setup Stream [_setupStreams()]

```
void _setupStreams() {
  final user = _auth.currentUser;
  if (user == null) return;

  // Personal tasks stream - always set up
  _personalTasksStream = _firestore
    .collection('todoList')
    .where('userId', isEqualTo: user.uid)
    .where('taskType', isEqualTo: 'personal')
    .orderBy('dueDate')
    .snapshots();

  // Family tasks stream - only if user has a family
  if (familyId != null) {
    _familyTasksStream = _firestore
      .collection('todoList')
      .where('familyId', isEqualTo: familyId)
      .where('taskType', isEqualTo: 'family')
      .orderBy('dueDate')
      .snapshots();
  }
}
```

Figure 33 Setup stream

This function sets up two streams of data: personal tasks and family tasks. The personal tasks stream is always set up for the logged-in user and listens for any changes in the user's personal tasks. If the user is part of a family, it also sets up a stream for family tasks, listening for tasks assigned to the family. These streams are used to update the UI whenever tasks are added, updated, or removed. It ensures the user sees both personal and family tasks in real time.

Convert Document to Task[_convertDocumentToTask()]

```
Task _convertDocumentToTask(DocumentSnapshot document) {
  final data = document.data() as Map<String, dynamic>;
  return Task(
    id: document.id,
    title: data['title'],
    dueDate: (data['dueDate'] as Timestamp).toDate(),
    isCompleted: data['isCompleted'] ?? false,
    userId: data['userId'],
    familyId: data['familyId'],
    taskType: data['taskType'],
    priority: data['priority'] ?? 'low',
  );
}
```

Figure 34 Convert document to task

This function converts a Firestore document snapshot into a Task object. It extracts data from the Firestore document, such as task title, due date, completion status, and other properties. It uses Timestamp to convert the Firestore date into a Dart DateTime. The function ensures that Firestore data is properly mapped to the Task model used in the app. It is essential for displaying tasks in the app's UI.

Add Task [_addTask()]

```
Future<void> _addTask(
  String title,
  DateTime dueDate,
  String taskType,
  String priority,
  String? assignedTo,
  String? description,
) async {
  try {
    final user = _auth.currentUser;
    if (user != null) {
      // Check if the task is a family task
      if (taskType == 'family') {
        // Only allow creators to create family tasks
        if (!isCreator) {
          ScaffoldMessenger.of(context).showSnackBar(
            const SnackBar(
              content:
                Text('Only family creators can create family tasks')),
            );
          return;
        }

        // If it's a family task, ensure the familyId is not null
        if (familyId == null) {
          ScaffoldMessenger.of(context).showSnackBar(
            const SnackBar(
              content: Text(
                'You must be part of a family to create family tasks')),
            );
          return;
        }
      }
    }

    // Add task to Firestore
    await FirebaseFirestore.instance.collection('tasks').add({
      'title': title,
      'dueDate': dueDate.toIso8601String(),
      'taskType': taskType,
      'priority': priority,
      'assignedTo': assignedTo,
      'description': description,
    });
  } catch (e) {
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(
        content: Text('Error adding task: $e'),
      ),
    );
  }
}
```

Figure 35 Add task

This function adds a new task to the Firestore database, either personal or family. It takes parameters such as task title, due date, priority, assigned family member, and description. If the task is a family task, it checks if the user is the creator and whether the familyId exists. It ensures family tasks are only created by creators and assigns tasks to specific family members if necessary. This function is used for task creation both for personal and family contexts.

Show Add Task Dialog [_showAddTaskDialog()]

```
void _showAddTaskDialog() {
  String newTaskTitle = '';
  DateTime selectedDate = DateTime.now();
  String taskType = 'personal'; // Default task type
  String priority = _selectedPriority;
  String? assignedTo; // Store the selected family member's ID
  String? description;

  List<String> taskTypes = ['personal', 'family'];

  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: const Text('Add New Task'),
        content: StatefulBuilder(
          // StatefulBuilder to dynamically rebuild UI
          builder: (context, setState) {
            return Column(
              mainAxisAlignment: MainAxisAlignment.min,
              children: [
                TextField(
                  onChanged: (value) {
                    newTaskTitle = value;
                  },
                  decoration: const InputDecoration(
                    labelText: 'Task Title',
                  ),
                ),
                TextField(
```

Figure 36 Show Add Task Dialog

This function displays a dialog where users can input details to create a new task. It provides fields for task title, description, priority, task type (personal or family), and assigned member (for family tasks). The dialog allows users to select a due date and choose a family member to assign a family task to if they are a creator. It dynamically rebuilds the UI when changing options using a StatefulBuilder. The function handles adding new tasks via a button click, including validation for required fields.

Update Task Status [_updateTaskStatus()]

```
Future<void> _updateTaskStatus(String taskId, bool isCompleted) async {  
  try {  
    await _firestore.collection('todoList').doc(taskId).update({  
      'isCompleted': isCompleted,  
    });  
  } catch (e) {  
    ScaffoldMessenger.of(context).showSnackBar(  
      SnackBar(content: Text('Error updating task: $e')),  
    );  
  }  
}
```

Figure 37 Update task status

This function updates the completion status of a task in Firestore. It takes the task ID and the new completion status as parameters. The function uses Firestore's update() method to mark the task as completed or not. It handles potential errors by displaying a snack bar with an error message if something goes wrong. It ensures that the task's completion status is synchronized with the database and UI.

Delete Task [_deleteTask()]

```
Future<void> _deleteTask(String taskId) async {  
  try {  
    await _firestore.collection('todoList').doc(taskId).delete();  
  } catch (e) {  
    ScaffoldMessenger.of(context).showSnackBar(  
      SnackBar(content: Text('Error deleting task: $e')),  
    );  
  }  
}
```

Figure 38 Delete task

This function deletes a task from Firestore using its task ID. It sends a delete request to Firestore to remove the specified task document from the todoList collection. The function handles errors by showing a snack bar with an error message. It ensures that the task is permanently removed from the database and UI once deleted. This function is used for task removal through the UI, such as when a task is swiped away.

Build Task List [_buildTaskList()]

```
Widget _buildTaskList(List<Task> tasks, bool isCompleted, String title) {
  final filteredTasks =
    tasks.where((task) => task.isCompleted == isCompleted).toList();

  return Column(
    crossAxisAlignment: CrossAxisAlignment.start,
    children: [
      Padding(
        padding: const EdgeInsets.all(16.0),
        child: Text(
          title,
          style: TextStyle(
            fontSize: 18,
            fontWeight: FontWeight.bold,
            color: isCompleted ? Colors.green : Colors.blue,
          ),
        ),
      ),
      if (filteredTasks.isEmpty)
        Padding(
          padding: const EdgeInsets.symmetric(horizontal: 16.0),
          child: Text(
            isCompleted ? 'No completed tasks' : 'No pending tasks',
            style: TextStyle(
              color: Colors.grey[600],
              fontStyle: FontStyle.italic,
            ),
          ),
        ),
    ],
  ),
}
```

Figure 39 Build task list

This function builds a list of tasks for display in the UI, filtering them by their completion status (completed or not). It accepts a list of Task objects, a boolean isCompleted, and a title to display. The function filters tasks based on their isCompleted property and creates a list of TaskCard widgets to show each task. It includes a message if no tasks are available for the selected status (pending/completed). The function handles task display, including user interactions like completion and deletion.

Show Copy Dialog [_showCopyDialog()]

```
void _showCopyDialog() {
  DateTime targetDate = _selectedDate;
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: const Text('Copy Meal Plan'),
        content: Column(
          mainAxisAlignment: MainAxisAlignment.min,
          children: [
            const Text('Select target date:'),
            const SizedBox(height: 16),
            ElevatedButton(
              onPressed: () async {
                final DateTime? picked = await showDatePicker(
                  context: context,
                  initialDate: targetDate,
                  firstDate: DateTime.now(),
                  lastDate: DateTime.now().add(const Duration(days: 365)),
                );
              },
            ),
          ],
        ),
      );
    },
  );
}
```

Figure 40 Show copy dialog

This function displays a dialog that allows users to select a target date for copying a meal plan. It uses `showDatePicker` to let the user pick a date, and once the date is selected, the meal plan is copied to that date. If the user presses the "Copy" button, the `_copyMealPlan` function is called with the selected date. The "Cancel" button dismisses the dialog without taking action. This dialog is useful for duplicating meal plans from one date to another.

Build Meal Plan Card [_buildMealPlanCard()]

```
Widget _buildMealPlanCard(DocumentSnapshot document) {
  final data = document.data() as Map<String, dynamic>;
  final date = (data['date'] as Timestamp).toDate();
  final isSelected = _selectedDate.year == date.year &&
    _selectedDate.month == date.month &&
    _selectedDate.day == date.day;

  return Container(
    margin: const EdgeInsets.all(8.0),
    padding: const EdgeInsets.all(16.0),
    decoration: BoxDecoration(
      color: Colors.white,
      borderRadius: BorderRadius.circular(12.0),
      border: isSelected
        ? Border.all(color: Theme.of(context).primaryColor, width: 2)
        : null,
    ),
  );
}
```

Figure 41 Build meal plan card

This function creates a card widget to display a meal plan for a specific date. It takes a `DocumentSnapshot` as input, which contains the meal plan data from Firestore. The meal plan's date is formatted and displayed, along with the meals for breakfast, lunch, dinner, and snacks. If the meal plan's date matches the currently selected date, the card is highlighted with a border. This card serves as a visual representation of a specific meal plan for a selected date.

Load User Family Data [_loadUserFamilyData()]

```

Future<void> _loadUserFamilyData() async {
  try {
    final user = _auth.currentUser;
    if (user != null) {
      final userData =
        await _firestore.collection('users').doc(user.uid).get();
      if (userData.exists) {
        setState(() {
          familyId = userData.data()?['familyId'];
          isCreator = userData.data()?['role'] == 'creator';
        });

        // Load family members if user has a family
        if (familyId != null) {
          final familySnapshot = await _firestore
            .collection('users')
            .where('familyId', isEqualTo: familyId)
            .where('role', isNotEqualTo: 'creator') // Exclude creators
            .get();

          setState(() {
            familyMembers = familySnapshot.docs
              .map((doc) => {
                'id': doc.id,
                'name': doc.data()['name'] ?? 'Unknown',
                'email': doc.data()['email'] ?? 'No email'
              })
              .toList();
          });
        }
      }
    }
  }
  _setupStreams();
}

```

Figure 42 Load user family data

This function is called in `initState` to load user-specific data from Firebase. It fetches the user's `familyId` and role (whether they are a creator or not), and checks if the user is part of a family. If the user is part of a family, it fetches the family members from Firestore. It then updates the state to reflect the `familyId`, `isCreator`, and `familyMembers` data. It also triggers the setup of Firestore streams for the personal and family tasks.

Build Family Management Button [_buildFamilyManagementButton()]

```
Widget _buildFamilyManagementButton() {  
  if (!isCreator || familyId == null) return const SizedBox.shrink();  
  
  return FloatingActionButton(  
    heroTag: 'familyManagement',  
    onPressed: () {  
      Navigator.push(  
        context,  
        MaterialPageRoute(  
          builder: (context) => FamilyManagement(  
            familyId: familyId!,  
            isCreator: isCreator,  
          ),  
        ),  
      );  
    },  
    child: const Icon(Icons.family_restroom),  
  );  
}
```

Figure 43 Build family management button

This function conditionally displays a floating action button for family management. It checks if the user is a family creator and if they have a familyId. If the conditions are met, it shows the button, which navigates to a family management screen when pressed. This button is used to manage family-related tasks or settings. If the user is not a family creator or doesn't have a family, this button is hidden. It helps manage family-related tasks efficiently.

Get Assigned User Email [_getAssignedUserEmail()]

```
// Function to fetch the email of the assigned user  
Future<String> _getAssignedUserEmail(String userId) async {  
  try {  
    final userSnapshot = await FirebaseFirestore.instance  
      .collection('users')  
      .doc(userId)  
      .get();  
    if (userSnapshot.exists) {  
      return userSnapshot.data()?['email'] ?? 'Email not found';  
    } else {  
      return 'User not found';  
    }  
  } catch (e) {  
    print("Error fetching user email: $e");  
    return 'Error fetching email';  
  }  
}
```

Figure 44 Get assigned user email

This function retrieves the email of the user assigned to a specific task. It fetches the user data from the Firestore database using the provided userId. If the user data is found, it returns the email; otherwise, it returns an error message indicating that the email or user was not found. It handles exceptions that may occur during the data fetch, providing a fallback error message. This function helps display the email address of the user assigned to the task on the task detail screen.

Get Invites Function [_getInvites()]

```
Stream<QuerySnapshot> _getInvites() {  
  final user = _auth.currentUser;  
  if (user == null) return const Stream.empty();  
  
  return _firestore  
    .collection('familyInvites')  
    .where('recipientEmail', isEqualTo: user.email)  
    .where('status', isEqualTo: 'pending')  
    .snapshots();  
}
```

Figure 45 get invites function

This function retrieves the family invitations for the current user by querying the familyInvites collection in Firestore. It filters the invites based on the recipient's email and the invite status (only "pending" invites). The function returns a stream of QuerySnapshot that listens for updates to the invitations, allowing real-time updates to the UI whenever new invites are received or statuses change.

Handling Invite Function [_handleInvite()]

```
Future<void> _handleInvite(String inviteId, bool accept) async {  
  final user = _auth.currentUser;  
  if (user == null) return;  
  
  try {  
    final invite =  
      await _firestore.collection('familyInvites').doc(inviteId).get();  
  
    if (accept) {  
      final familyId = invite.data()?['familyId'];  
  
      // Update user's family ID and role  
      await _firestore.collection('users').doc(user.uid).update({  
        'familyId': familyId,  
        'role': 'member',  
      });  
  
      // Add user to the family members array  
      await _firestore.collection('families').doc(familyId).update({  
        'members': FieldValue.arrayUnion([user.uid]),  
      });  
    }  
  
    // Delete the invitation  
    await _firestore.collection('familyInvites').doc(inviteId).delete();  
  } catch (e) {  
    print('Error handling invite: $e');  
  }  
}
```

Figure 46 handling invite function

The _handleInvite function processes a family invite when the user accepts or rejects it. It first fetches the invite document from Firestore using the provided inviteId. If the user accepts the invite (accept == true), it updates the user's profile with the new family ID and role, and also adds the user to the family's members array in Firestore. After processing the invite, whether accepted or rejected, the invitation document is deleted from the familyInvites collection. Any errors encountered during this process are caught and printed.

Load Family Member [_loadFamilyMembers()]

```
Future<void> _loadFamilyMembers() async {  
  final members = await _firestore  
    .collection('users')  
    .where('familyId', isEqualTo: widget.familyId)  
    .get();  
  
  setState(() {  
    _familyMembers = members.docs  
      .map((doc) => {  
        'id': doc.id,  
        'email': doc.data()['email'],  
        'role': doc.data()['role'],  
      })  
      .toList();  
  });  
}
```

Figure 47 Load family member

This function retrieves all users associated with the family using Firestore's query where the familyId matches the one passed into the widget. It fetches the members from the users collection and maps them into a list containing their id, email, and role. The function then updates the state by setting the _familyMembers list with the fetched data. It ensures that the family members are displayed correctly on the UI. This function is called in initState() to load family members when the screen is first displayed.

Load Pending Invites [_loadPendingInvites()]

```
Future<void> _loadPendingInvites() async {  
  final invites = await _firestore  
    .collection('familyInvites')  
    .where('familyId', isEqualTo: widget.familyId)  
    .where('status', isEqualTo: 'pending')  
    .get();  
  
  setState(() {  
    _pendingInvites = invites.docs  
      .map((doc) => {  
        'id': doc.id,  
        'email': doc.data()['recipientEmail'],  
        'timestamp': doc.data()['timestamp'],  
      })  
      .toList();  
  });  
}
```

Figure 48 Load pending invites

This function queries the familyInvites collection for invites that are pending, using the familyId passed to the widget and filtering by status: 'pending'. It fetches the pending invites and stores them in the _pendingInvites list with details like id, email, and timestamp. After fetching the data, it updates the state to reflect the list of pending invites on the UI. This function ensures that the list of invites is displayed and updated in real-time. It is also invoked in initState() when the screen is first loaded.

Check Date [_isDateValid()]

```
bool _isDateValid(DateTime date) {  
    final now = DateTime.now();  
    return date.isAfter(DateTime(now.year, now.month, now.day - 1));  
}
```

Figure 49 Check date

This function checks whether the given date is valid by ensuring that it is not in the past. It compares the selected date to the current date minus one day. If the date is in the future or today, it returns true; otherwise, it returns false. The purpose of this function is to prevent creating meal plans for dates that have already passed. It's used in the showCreateMealPlanDialog function to ensure users cannot create past meal plans.

Show Create Meal Plan Dialog [_showCreateMealPlanDialog()]

```
void _showCreateMealPlanDialog() {
  if (!_isDateValid(_selectedDate)) {
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(
        content: Text('Cannot create meal plan for past dates'),
        backgroundColor: Colors.red,
      ),
    );
    return;
  }

  // Reset controllers
  _breakfastController.clear();
  _lunchController.clear();
  _dinnerController.clear();
  _snacksController.clear();
}
```

Figure 50 Show create meal plan dialog

This function triggers a dialog that allows users to create a meal plan for a selected date. It first checks if the date is valid using the `_isDateValid` function. If the date is valid, it opens a form with fields for breakfast, lunch, dinner, and snacks, which are validated before submission. Upon form submission, the meal plan is either created or updated depending on whether a meal plan already exists for the selected date. If successful, a confirmation message is shown to us.

Create Custom Meal Plan [_createCustomMealPlan()]

```
Future<void> _createCustomMealPlan() async {
  try {
    final selectedDate = DateTime(
      _selectedDate.year,
      _selectedDate.month,
      _selectedDate.day,
    );

    // Check if a meal plan already exists for this date
    final existingPlan = await _firestore
      .collection('mealPlans')
      .where('userId', isEqualTo: _auth.currentUser?.uid)
      .where('date', isEqualTo: selectedDate)
      .get();

    if (existingPlan.docs.isNotEmpty) {
      // Update existing meal plan
      await existingPlan.docs.first.reference.update({
        'breakfast': _breakfastController.text,
        'lunch': _lunchController.text,
        'dinner': _dinnerController.text,
        'snacks': _snacksController.text,
        'updatedAt': FieldValue.serverTimestamp(),
      });
    }
  }
}
```

Figure 51 Create custom meal plan

This function creates or updates a custom meal plan for the selected date in the Firestore database. It first checks if a meal plan already exists for the selected date. If a plan exists, it updates the fields (breakfast, lunch, dinner, snacks) with the new data. If no meal plan exists, it creates a new document in Firestore. After successfully creating or updating the meal plan, a confirmation message is shown to the user.

Initialize Meal Plan Function [_initializeMealPlanStream()]

```
void _initializeMealPlanStream() {  
    final startOfWeek =  
        DateTime.now().subtract(Duration(days: DateTime.now().weekday - 1));  
    final endOfWeek = startOfWeek.add(const Duration(days: 7));  
  
    _mealPlanStream = _firestore  
        .collection('mealPlans')  
        .where('userId', isEqualTo: _auth.currentUser?.uid)  
        .where('date', isGreaterThanOrEqualTo: startOfWeek)  
        .where('date', isLessThan: endOfWeek)  
        .orderBy('date')  
        .snapshots();  
}
```

Figure 52 Customize meal plan function

This function initializes a stream of meal plans for the current week based on the selected date. It calculates the start and end dates of the current week and queries Firestore for meal plans that fall within that range. The query is filtered by the `userId` and sorted by the date. It then assigns the resulting stream to `_mealPlanStream` so the UI can be updated in real-time with the latest meal plans. This function ensures the user sees the meal plans for the current week.

Schedule Clean Up [_scheduleCleanup()]

```
void _scheduleCleanup() {  
    final thirtyDaysAgo = DateTime.now().subtract(const Duration(days: 30));  
    _firestore  
        .collection('mealPlans')  
        .where('userId', isEqualTo: _auth.currentUser?.uid)  
        .where('date', isLessThan: thirtyDaysAgo)  
        .get()  
        .then((snapshot) {  
            for (var doc in snapshot.docs) {  
                doc.reference.delete();  
            }  
        });  
}
```

Figure 53 Schedule clean up

This function schedules a cleanup task to delete old meal plans that are older than 30 days. It queries Firestore for meal plans older than 30 days and deletes them. The cleanup helps to manage the database and remove outdated meal plans that are no longer relevant. The function runs automatically when the page is initialized. This ensures the database is kept clean and only holds relevant data.

Check And Create Today Meal Plan[_checkAndCreateTodayMealPlan()]

```

Future<void> _checkAndCreateTodayMealPlan() async {
  final today = DateTime(
    DateTime.now().year,
    DateTime.now().month,
    DateTime.now().day,
  );

  final snapshot = await _firestore
    .collection('mealPlans')
    .where('userId', isEqualTo: _auth.currentUser?.uid)
    .where('date', isEqualTo: today)
    .get();

  if (snapshot.docs.isEmpty) {
    await _firestore.collection('mealPlans').add({
      'userId': _auth.currentUser?.uid,
      'date': today,
      'breakfast': '',
      'lunch': '',
      'dinner': '',
      'snacks': '',
      'createdAt': FieldValue.serverTimestamp(),
    });
  }
}

```

Figure 54 Check and create today meal plan

This function checks if a meal plan for the current day already exists. If no plan exists, it creates a new document with empty fields (for breakfast, lunch, dinner, snacks). It uses Firestore to query for a meal plan with today's date and the current user's ID. If no meal plan is found, it adds a new meal plan with default values. This function ensures that there is always a meal plan for today, even if the user hasn't entered one yet.

Get Meal Plan for Date [_getMealPlanForDate()]

```

Future<Map<String, dynamic>>> _getMealPlanForDate(DateTime date) async {
  final snapshot = await _firestore
    .collection('mealPlans')
    .where('userId', isEqualTo: _auth.currentUser?.uid)
    .where('date', isEqualTo: date)
    .get();

  if (snapshot.docs.isNotEmpty) {
    return snapshot.docs.first.data();
  }
  return null;
}

```

Figure 55 Get meal plan for date

This function retrieves the meal plan for a specific date from Firestore. It queries the mealPlans collection for a document matching the selected date and user ID. If a plan exists, it returns the data as a map; otherwise, it returns null. This function is used when copying meal plans or updating specific meal types for a date. It helps to fetch the correct meal plan data for operations that modify or display the user's meals.

Update Meal Plan [_updateMealPlan()]

```
Future<void> _updateMealPlan(String type, String meal) async {
  try {
    final selectedDate = DateTime(
      _selectedDate.year,
      _selectedDate.month,
      _selectedDate.day,
    );

    final snapshot = await _firestore
      .collection('mealPlans')
      .where('userId', isEqualTo: _auth.currentUser?.uid)
      .where('date', isEqualTo: selectedDate)
      .get();

    if (snapshot.docs.isNotEmpty) {
      await snapshot.docs.first.reference.update({
        type.toLowerCase(): meal,
        'updatedAt': FieldValue.serverTimestamp(),
      });
    } else {
      await _firestore.collection('mealPlans').add({
        'userId': _auth.currentUser?.uid,
        'date': selectedDate,
        type.toLowerCase(): meal,
        'createdAt': FieldValue.serverTimestamp(),
        'updatedAt': FieldValue.serverTimestamp(),
      });
    }
  }
}
```

Figure 56 Update meal plan

This function updates a specific meal (e.g., breakfast, lunch, dinner) for the selected date in the Firestore database. It first retrieves the existing meal plan for that date. If the plan exists, it updates the selected meal type with the new input. If the plan doesn't exist, it creates a new entry with the specified meal data. This function allows users to update individual meal types for a given date.

Copy Meal Plan [_copyMealPlan()]

```
Future<void> _copyMealPlan(DateTime fromDate, DateTime toDate) async {
  final mealPlan = await _getMealPlanForDate(fromDate);
  if (mealPlan != null) {
    await _firestore.collection('mealPlans').add({
      'userId': _auth.currentUser?.uid,
      'date': toDate,
      'breakfast': mealPlan['breakfast'],
      'lunch': mealPlan['lunch'],
      'dinner': mealPlan['dinner'],
      'snacks': mealPlan['snacks'],
      'createdAt': FieldValue.serverTimestamp(),
    });

    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text('Meal plan copied successfully!')),
    );
  }
}
```

Figure 57 Copy meal plan

This function copies the meal plan from one date to another. It retrieves the meal plan for the fromDate and uses the data to create a new plan for the toDate. It checks if the fromDate meal plan exists and then creates a duplicate

for the `toDate`. If the operation is successful, a confirmation message is displayed to the user. This is useful for replicating meal plans from one day to another.

Generate Weekly Stats[_generateWeeklyStats()]

```
Future<Map<String, dynamic>> _generateWeeklyStats() async {
  final startOfWeek =
    _selectedDate.subtract(Duration(days: _selectedDate.weekday - 1));
  final endOfWeek = startOfWeek.add(const Duration(days: 7));

  final snapshot = await _firestore
    .collection('mealPlans')
    .where('userId', isEqualTo: _auth.currentUser?.uid)
    .where('date', isGreaterThanOrEqualTo: startOfWeek)
    .where('date', isLessThan: endOfWeek)
    .get();

  int totalMeals = 0;
  Map<String, int> mealTypeCount = {
    'breakfast': 0,
    'lunch': 0,
    'dinner': 0,
    'snacks': 0,
  };
}
```

Figure 58 Generate weekly stats

This function calculates and returns statistics for the current week's meal plans. It queries Firestore for meal plans within the current week and counts the number of completed meals for each type (breakfast, lunch, dinner, snacks). It also calculates the total number of meals planned and the completion rate (percentage of meals planned). The function returns these statistics in a map format for use in the UI. This is helpful for tracking the user's meal plan progress over the week.

Generate Meal Suggestion [_generateMealSuggestions()]

```
Future<List<String>> _generateMealSuggestions(String mealType) async {
  final snapshot = await _firestore
    .collection('mealPlans')
    .where('userId', isEqualTo: _auth.currentUser?.uid)
    .orderBy('date', descending: true)
    .limit(10)
    .get();

  final meals = snapshot.docs
    .map((doc) => (doc.data())[mealType.toLowerCase()])
    .where((meal) => meal != null && meal.toString().isNotEmpty)
    .toSet()
    .toList();

  if (meals.length < 3) {
    meals.addAll(_getDefaultSuggestions(mealType));
  }

  return meals.take(5).cast<String>().toList();
}
```

Figure 59 Generate meal suggestion

This function generates meal suggestions based on recent meals for a specific meal type (e.g., breakfast, lunch). It queries Firestore for recent meal plans and collects a list of meals for the selected meal type. If there are not enough recent suggestions, it falls back on default suggestions for that meal type. It returns a list of suggested

meals, ensuring the user has variety in their meal planning. This is useful for giving users ideas when they are unsure what to plan for a specific meal.

Get Default Suggestion [_getDefaultSuggestion()]

```
List<String> _getDefaultSuggestions(String mealType) {  
    switch (mealType.toLowerCase()) {  
        case 'breakfast':  
            return [  
                'Oatmeal with fruits and nuts',  
                'Yogurt parfait',  
                'Whole grain toast with avocado',  
                'Smoothie bowl',  
                'Eggs and vegetables'  
            ];  
        case 'lunch':  
            return [  
                'Grilled chicken salad',  
                'Quinoa bowl',  
                'Turkey wrap',  
                'Mediterranean pasta',  
                'Vegetable soup with bread'  
            ];  
    }
```

Figure 60 Get default suggestion

This function provides default meal suggestions for a given meal type (e.g., breakfast, lunch, dinner, snacks). It returns a predefined list of meal ideas based on the meal type requested. If there aren't enough recent meals to suggest, this function ensures the user has a set of healthy and varied options. The suggestions include popular, healthy meals like oatmeal, grilled chicken, and salads. This function helps users when they need meal inspiration for each type of meal.

Show Weekly Stats [_showWeeklyStats()]

```
void _showWeeklyStats() async {
  final stats = await _generateWeeklyStats();

  if (!mounted) return;

  showDialog(
    context: context,
    builder: (context) => AlertDialog(
      title: const Text('Weekly Statistics'),
      content: Column(
        mainAxisAlignment: MainAxisAlignment.start,
        children: [
          Text('Total Meals Planned: ${stats['totalMeals']}'),
          Text('Completion Rate: ${stats['completionRate']}%'),
          const SizedBox(height: 8),
          const Text('Meals by Type:',
            style: TextStyle(fontWeight: FontWeight.bold)),
          ...(stats['mealTypeCount'] as Map<String, int>).entries.map(
            (e) => Text('${e.key}: ${e.value}'),
          ),
        ],
      ),
    ),
  );
}
```

Figure 61 Show weekly stats

This function displays the weekly statistics in a dialog box. It calls the `_generateWeeklyStats` function to fetch the data for the week. Once the statistics are available, it creates a dialog showing the total number of meals planned, the completion rate, and the count for each meal type (breakfast, lunch, dinner, snacks). The function is useful for showing users how well they are sticking to their meal plans. It gives a quick summary of the week's meal planning performance.

Get Invites [_getInvites()]

```
Stream<QuerySnapshot> _getInvites() {
  final user = _auth.currentUser;
  if (user == null) return const Stream.empty();

  return _firestore
    .collection('familyInvites')
    .where('recipientEmail', isEqualTo: user.email)
    .where('status', isEqualTo: 'pending')
    .orderBy('timestamp', descending: true)
    .snapshots();
}
```

Figure 62 Get invites

This function returns a stream of pending family invitations for the current user from the Firestore `familyInvites` collection. It filters the invites based on the recipient's email and their invite status (pending). The data is ordered

by timestamp in descending order to show the most recent invites first. The stream listens for real-time changes and automatically updates the UI when new invites are added. If the user is not authenticated, it returns an empty stream.

Get Notification [_getNotifications()]

```
Stream<QuerySnapshot> _getNotifications() {  
  final user = _auth.currentUser;  
  if (user == null) return const Stream.empty();  
  
  return _firestore  
    .collection('notifications')  
    .where('uid', isEqualTo: user.uid)  
    .orderBy('timestamp', descending: true)  
    .limit(50)  
    .snapshots();  
}
```

Figure 63 Get notification

This function retrieves notifications for the current user from the Firestore notifications collection. It filters notifications by the user's UID, ensuring that the notifications belong to the current user. The notifications are ordered by timestamp, with the most recent appearing first, and are limited to 50 results. This stream also listens for real-time changes, automatically updating the UI when new notifications are added. If no user is authenticated, it returns an empty stream.

Handle Invite [_handleInvite()]

```
// Method to handle invite acceptance or decline  
Future<void> _handleInvite(String inviteId, bool accept) async {  
  setState(() => _isLoading = true);  
  
  try {  
    final user = _auth.currentUser;  
    if (user == null) return;  
  
    final inviteDoc =  
      await _firestore.collection('familyInvites').doc(inviteId).get();  
  
    if (inviteDoc.exists) {  
      final inviteData = inviteDoc.data() as Map<String, dynamic>;  
      final familyId = inviteData['familyId'];  
      final senderId = inviteData['senderId'];  
  
      // Handle the invite logic here  
    }  
  }  
}
```

Figure 64 Handle invite

This function processes the acceptance or rejection of a family invitation. It updates the user's familyId and role in the Firestore users collection if the invite is accepted. It also adds a notification to the family creator, letting them know that the invite was accepted. After handling the invite, it deletes the invite from the familyInvites collection. A Snackbar is shown to the user indicating whether they joined the family group or declined the invitation.

Get User Detail [_getUserDetails()]

```
// Method to fetch user details based on senderId (or use current user)
Future<Map<String, dynamic>>> _getUserDetails(String? senderId) async {
  try {
    final user = _auth.currentUser;
    final idToUse = senderId ?? user?.uid;

    if (idToUse == null) return null;

    DocumentSnapshot snapshot =
      await _firestore.collection('users').doc(idToUse).get();
    if (snapshot.exists) {
      return snapshot.data() as Map<String, dynamic>;
    }
  } catch (e) {
    print('Error fetching user details: $e');
  }
  return null;
}
```

Figure 65 Get user detail

This function fetches the details of a user based on their senderId or the current authenticated user's UID if no senderId is provided. It queries the Firestore users collection to retrieve the user's data and returns it as a Map. If no user details are found or there is an error during the fetch, it returns null. This function is used to display the sender's details when viewing an invitation. If the user details are successfully fetched, they are used to personalize the invite display.

Build Invite List [_buildInvitesList()]

```
Widget _buildInvitesList() {  
  return StreamBuilder<QuerySnapshot>(  
    stream: _getInvites(),  
    builder: (context, snapshot) {  
      if (snapshot.hasError) {  
        return Center(child: Text('Error: ${snapshot.error}'));  
      }  
  
      if (snapshot.connectionState == ConnectionState.waiting) {  
        return const Center(child: CircularProgressIndicator());  
      }  
  
      final invites = snapshot.data?.docs ?? [];  
  
      if (invites.isEmpty) {  
        return const Center(  
          child: Padding(  
            padding: EdgeInsets.all(16.0),  
            child: Text(  
              'No pending invites',  
              style: TextStyle(  
                fontSize: 16,  
                color: Colors.grey,  
              ),  
            ),  
          ),  
        );  
      }  
    },  
  );  
}
```

Figure 66 Build invite list

This function builds a list of pending family invitations using a StreamBuilder that listens to the `_getInvites` stream. If there is an error or loading, it shows appropriate messages or a loading indicator. When invites are fetched, they are displayed as a list of Card widgets, with each invite showing the sender's name and role. Each invite has two buttons for accepting or declining the invitation. The function also handles loading states and error handling within the UI.

Build Notification List [_buildNotificationsList()]

```
// StreamBuilder to display notifications list
Widget _buildNotificationsList() {
  return StreamBuilder<QuerySnapshot>(
    stream: _getNotifications(),
    builder: (context, snapshot) {
      if (snapshot.hasError) {
        return Center(child: Text('Error: ${snapshot.error}'));
      }

      if (snapshot.connectionState == ConnectionState.waiting) {
        return const Center(child: CircularProgressIndicator());
      }

      final notifications = snapshot.data?.docs ?? [];

      if (notifications.isEmpty) {
        return const Center(
          child: Padding(
            padding: EdgeInsets.all(16.0),
            child: Text(
              'No notifications',
              style: TextStyle(
                fontSize: 16,
                color: Colors.grey,
              ),
            ),
          ),
        );
      }
    },
  );
}
```

Figure 67 Build notification list

This function constructs a list of notifications using a StreamBuilder that listens to the `_getNotifications` stream. It handles error cases and shows a loading indicator when data is being fetched. The notifications are displayed as a list of Card widgets, where each notification has an icon based on its type and a message. If a notification is tapped and hasn't been read yet, it marks it as read. The function also handles cases where there are no notifications by displaying a message.

Get Notification Icon [_getNotificationIcon()]

```
// Helper method to get the icon based on notification type
IconData _getNotificationIcon(String? type) {
  switch (type) {
    case 'family_invite_accepted':
      return Icons.family_restroom;
    case 'task_completed':
      return Icons.task_alt;
    case 'task_assigned':
      return Icons.assignment;
    default:
      return Icons.notifications;
  }
}
```

Figure 68 get notification icon

This helper function determines the appropriate icon to display based on the notification's type. It takes in the notification's type as a parameter and returns a corresponding icon. For example, if the notification type is `family_invite_accepted`, it returns a `family_restroom` icon. This function helps in customizing the UI based on the type of notification. The function is called inside `_buildNotificationsList` to set the icon for each notification.

Notification Marker[_markNotificationAsRead()]

```
// Mark notification as read
Future<void> _markNotificationAsRead(String notificationId) async {
  try {
    await _firestore
      .collection('notifications')
      .doc(notificationId)
      .update({'read': true});
  } catch (e) {
    print('Error marking notification as read: $e');
  }
}
```

Figure 69 Notification marker

This function marks a notification as read by updating its `read` field in Firestore to `true`. It is called when a user taps on an unread notification in the list. The function catches any errors that occur during the update process and prints the error to the console. It helps manage the read status of notifications, ensuring that the UI reflects the user's actions. This function is used in the `_buildNotificationsList` to update the UI when a notification is tapped.

2.5 CRUD

Create:

Submit Feedback Function [submitFeedback()]

```
Future<void> _submitFeedback() async {  
  if (_titleController.text.isEmpty ||  
      _feedbackController.text.isEmpty ||  
      _emailController.text.isEmpty) {  
    ScaffoldMessenger.of(context).showSnackBar(  
      SnackBar(content: Text('Please fill out all fields')),  
    );  
    return;  
  }  
  
  setState(() {  
    _isSubmitting = true;  
  });
```

Figure 70 Submit feedback function

The `_submitFeedback` function first checks if any of the input fields (title, feedback, or email) are empty, showing an error message if so. It then fetches the current feedback counter from Firestore, increments it by 1, and adds the feedback data (including the new ID, title, message, email, and timestamp) to the feedback collection. After successfully adding the feedback, it updates the counter in Firestore with the new value. Finally, the function clears the input fields and resets the submission state, or shows an error message if the submission fails.

Adding Task Function [`_addTask()`]

```
void _addTask(String title) {  
  tasksCollection.add({  
    'title': title,  
    'completed': false,  
  });  
  _controller.clear();  
}
```

Figure 71 Adding task function

This function adds a new task to the Firestore "tasks" collection when the user submits a task title. It uses the `TextEditingController` to get the task title from the input field and then adds the task with a default "completed" status of false. After adding the task, the input field is cleared for the next task. This function ensures that new tasks are saved in the database and displayed in the UI.

Invite Member [_inviteMember()]

```
Future<void> _inviteMember(String email) async {
  setState(() => _isLoading = true);

  try {
    // Check if user exists
    final userQuery = await _firestore
      .collection('users')
      .where('email', isEqualTo: email)
      .get();

    if (userQuery.docs.isEmpty) {
      throw 'User not found';
    }

    final targetUser = userQuery.docs.first;

    // Check if user is already in a family
    if (targetUser.data()['familyId'] != null) {
      throw 'User is already in a family';
    }
  }
}
```

Figure 72: Invite Member

This function is responsible for sending a family invitation to a user by their email address. It first checks if the user exists by querying the users collection with the provided email, and verifies they are not already in a family. If the user is eligible and no invitation exists already, it creates a new invitation in the familyInvites collection with status: 'pending'. The function provides error handling and updates the UI with appropriate messages, such as "Invitation sent" or error details. It also reloads the pending invites list after successfully sending an invitation.

Read:

Get Feedback Details [_getFeedbackDetails()]

```
Future<DocumentSnapshot> _getFeedbackDetails() async {  
  return await FirebaseFirestore.instance  
    .collection('feedback')  
    .doc(id)  
    .get();  
}
```

Figure 73 Get Feedback Details

The _getFeedbackDetails function retrieves a specific feedback document from Firestore using the provided id. It returns a Future<DocumentSnapshot>, which represents the result of an asynchronous operation to fetch the document. The get() method is called on the DocumentReference to fetch the document from the feedback collection. This allows the app to access the detailed information of a specific feedback entry based on the given id.

Feedback List Page Function [_FeedbackListPageState()]

```
class _FeedbackListPageState extends State<FeedbackListPage> {  
  bool isAscending = true;  
  String searchQuery = '';  
  
  void _toggleSortOrder() {  
    setState(() {  
      isAscending = !isAscending;  
    });  
  }  
}
```

Figure 74 Feedback list page function

In the _FeedbackListPageState class, a boolean variable isAscending is used to control the sorting order of feedback items. The searchQuery string holds the current search text for filtering feedback by ID. The _toggleSortOrder method toggles the value of isAscending, which will switch between ascending and descending order when called. The setState method is used to update the UI whenever the sorting order changes, ensuring the UI reflects the latest state.

Build Calendar View [_buildCalendarView()]

```
Widget _buildCalendarView() {  
  return TableCalendar(  
    firstDay: DateTime.now().subtract(const Duration(days: 365)),  
    lastDay: DateTime.now().add(const Duration(days: 365)),  
    focusedDay: _focusedDay,  
    selectedDayPredicate: (day) => isSameDay(_selectedDate, day),  
    calendarFormat: _calendarFormat,  
    onFormatChanged: (format) {  
      setState(() {  
        _calendarFormat = format;  
      });  
    },  
    onDaySelected: (selectedDay, focusedDay) {  
      setState(() {  
        _selectedDate = selectedDay;  
        _focusedDay = focusedDay;  
      });  
    },  
    calendarStyle: const CalendarStyle(  
      todayDecoration: BoxDecoration(  
        color: Colors.blue,  
        shape: BoxShape.circle,  
      ),  
      selectedDecoration: BoxDecoration(  
        color: Colors.green,  
        shape: BoxShape.circle,  
      ),  
    ),  
  );  
}
```

Figure 75: Build calendar view

This function builds a calendar widget using the TableCalendar package. It allows the user to select a date, which updates the _selectedDate and _focusedDay variables. The calendar shows a range from one year before the current date to one year after it. It highlights the selected date with a green circle and today's date with a blue circle. The calendar's format can also be changed through the _calendarFormat variable.

Update:

Tracking Feature Usage Function [_logFeatureUsage()]

```
// Function to log feature usage in Firestore
void _logFeatureUsage(String featureName) {
  FirebaseFirestore.instance
    .collection('stats')
    .doc('featureUtilization')
    .update({
      featureName: FieldValue.increment(1),
    }).catchError((error) {
      print("Failed to log feature usage: $error");
    });
}
```

Figure 76 Tracking feature usage function

This function logs whenever the Feedback screen is accessed to track app usage statistics. It connects to Firestore's "stats" collection and updates a document called "featureUtilization" with a field matching the feature name. The field's value is incremented by 1 to record usage. If there's an error while updating, it catches the error and prints a failure message.

Dynamic List Ordering [_toggleSortOrder()]

```
class _FeedbackListPageState extends State<FeedbackListPage> {
  bool isAscending = true;
  String searchQuery = '';

  void _toggleSortOrder() {
    setState(() {
      isAscending = !isAscending;
    });
  }
}
```

Figure 77 Dynamic list ordering'

Located within the FeedbackListPage, this function controls the sort order for the feedback list. When triggered (such as by tapping a sort of button), it updates the "isAscending" state variable by toggling between true and false. This change causes the feedback list to reorder based on the timestamp field, either in ascending or descending order. The UI automatically refreshes due to the state change.

Remove Member [_removeMember()]

```
Future<void> _removeMember(String userId) async {  
  try {  
    await _firestore.collection('users').doc(userId).update({  
      'familyId': null,  
      'role': 'user',  
    });  
    _loadFamilyMembers();  
  } catch (e) {  
    ScaffoldMessenger.of(context).showSnackBar(  
      SnackBar(content: Text('Error removing member: $e')),  
    );  
  }  
}
```

Figure 78: Remove Member

This function allows the family creator to remove a member from the family by updating the member's familyId to null and their role to 'user' in the users collection. It performs this operation based on the member's userId and then updates the state to reflect the changes in the UI. After removing the member, it calls _loadFamilyMembers() again to reload the updated list of family members. The function ensures that only the creator can remove members, excluding the creator themselves from being removed. Error handling is also included to show messages if the removal fails.

Delete:

Delete Task Function [_deleteTask()]

```
void _deleteTask(String id) {  
  tasksCollection.doc(id).delete();  
}
```

Figure 79 Delete task function

This function deletes a task from the Firestore "tasks" collection when the user presses the delete icon. It performs the delete operation by referencing the task's document ID. The task is removed from the database, and the UI is updated automatically. This function allows users to manage their tasks effectively by removing completed or unwanted tasks.

Build Event List [_buildEventList ()]

```
Widget _buildEventList() {  
  final todayEvents = _calendarEvents.where((event) {  
    return event.dateTime.year == _selectedDate.year &&  
      event.dateTime.month == _selectedDate.month &&  
      event.dateTime.day == _selectedDate.day;  
  }).toList();  
  
  return Card(  
    margin: const EdgeInsets.all(8.0),  
    elevation: 2,  
    child: Column(  
      crossAxisAlignment: CrossAxisAlignment.start,  
      children: [  
        Padding(  
          padding: const EdgeInsets.all(16.0),  
          child: Text(  
            'Events for ${DateFormat('MMMM d, yyyy').format(_selectedDate)}',  
            style: const TextStyle(  
              fontWeight: FontWeight.bold,  
              fontSize: 16,  
            ),  
          ),  
        ),  
        const Divider(height: 1),  
        Expanded(  
          child: todayEvents.isEmpty  
            ? Center(  
              child: Text(  

```

Figure 80 Build event list

This function generates a list of upcoming events below the calendar grid. It displays each event with an icon, the event title, and the formatted date and time. The user can also delete events using a delete button on the right side of each list item. The function updates the event list whenever events are added or deleted. This provides an alternative view of the calendar events.

Cancel Invite [_cancelInvite()]

```
Future<void> _cancelInvite(String inviteId) async {  
  try {  
    await _firestore.collection('familyInvites').doc(inviteId).delete();  
    _loadPendingInvites();  
  } catch (e) {  
    ScaffoldMessenger.of(context).showSnackBar(  
      SnackBar(content: Text('Error canceling invite: $e')),  
    );  
  }  
}
```

Figure 81: Cancel Invites

This function cancels a pending family invite by deleting the invite document from the familyInvites collection. It first fetches the inviteId and deletes the corresponding invite. After deleting the invite, it reloads the pending invites list to update the UI. This function helps the creator manage and remove unwanted or expired invites. It also provides error handling and displays appropriate messages if the deletion fails

2.6 Validation

Validation function for Form Fields

```
},  
  validator: (value) {  
    if (value == null || value.isEmpty) {  
      return 'Please enter your name';  
    }  
    return null;  
  },  
,  
const SizedBox(height: 20),  
TextFormField(  
  controller: _usernameController,  
  decoration: const InputDecoration(  
    labelText: 'Username',  
    border: OutlineInputBorder(),  
  ),  
  validator: (value) {  
    if (value == null || value.isEmpty) {  
      return 'Please enter your Username';  
    }  
    return null;  
  },  
,  
,
```

Figure 82 Validation

These functions are used for validating the input fields in the form, ensuring that the user has entered the necessary information. They check that fields like name, username, email, password, and confirm password are not empty. In addition, they ensure that the password meets the minimum length requirement and that the passwords match. If any field is invalid, an error message is shown below the corresponding field. These functions help guide the user to complete the form correctly.

2.7 User Permission

Create Account Function [_createAccount ()]

```
// Handle account creation logic
Future<void> _createAccount() async {
  if (_formKey.currentState!.validate()) {
    try {
      // Step 1: Create a new user with Firebase Authentication
      UserCredential userCredential =
        await _auth.createUserWithEmailAndPassword(
          email: _emailController.text,
          password: _passwordController.text,
        );
    }
  }
}
```

Figure 83 Create account function

This function handles the account creation logic by first validating the input form. It uses Firebase Authentication to create a new user with the provided email and password. After successful authentication, the function stores the user data in Firestore, including the name, username, email, and a role. The function also handles the logic for creating or joining a family, depending on whether the user is creating a new family or joining an existing one. If the account creation is successful, the user receives a verification email and the screen transitions to the next page.

```
class _CreateAccountPageState extends State<CreateAccountPage> {
  final _formKey = GlobalKey<FormState>();
  final _nameController = TextEditingController();
  final _usernameController = TextEditingController();
  final _emailController = TextEditingController();
  final _passwordController = TextEditingController();
  final _confirmPasswordController = TextEditingController();

  final FirebaseAuth _auth = FirebaseAuth.instance;
  final FirebaseFirestore _firestore = FirebaseFirestore.instance;

  String? _familyId;
  String _accountType = 'user'; // Default to 'Regular User'
  //bool _isCreatingFamily = false;
}
```

Figure 84 Create account page state

This code defines the state for the CreateAccountPage widget in a Flutter app, managing user account creation. It uses TextEditingController instances to handle user input for name, username, email, password, and confirm password fields. A GlobalKey<FormState> is used for form validation and saving the form state. The code also initializes Firebase Authentication (_auth) for managing user authentication and Firebase Firestore (_firestore) for storing and retrieving user data. The accountType variable is set to 'user' by default, and there's a placeholder for a familyId, which could potentially be used for family-related features. There's also a commented-out variable (_isCreatingFamily) that suggests the app might have functionality for creating family accounts.

2.8 Wireframe

Admin

The wireframes serves as simplified depictions of the system interface. The wireframes represents the fundamental layout, structure, and components of the intended application.



Figure 85 User Feedback Details



Figure 86 Admin Dashboard

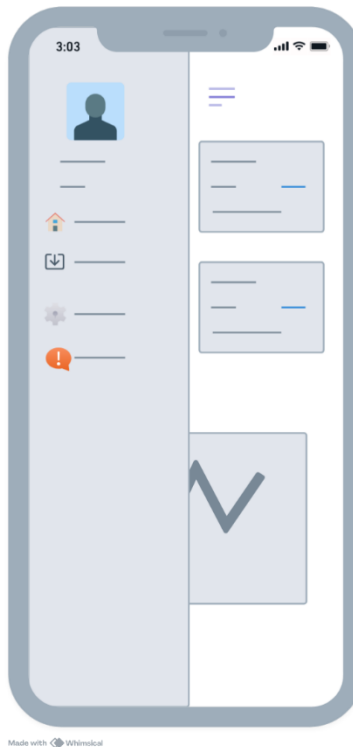


Figure 87 Admin Menu (Sidebar)

User and Family

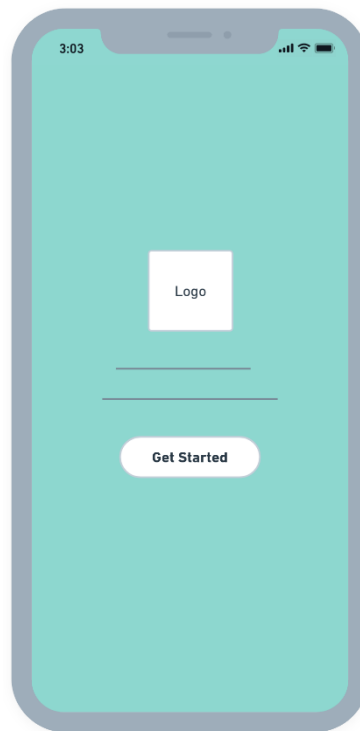


Figure 88 Welcoming Page



Made with  Whimsical

Figure 89 Create account page

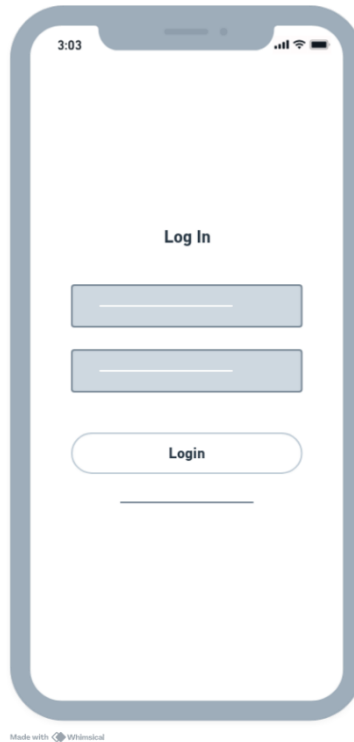


Figure 90 Login page

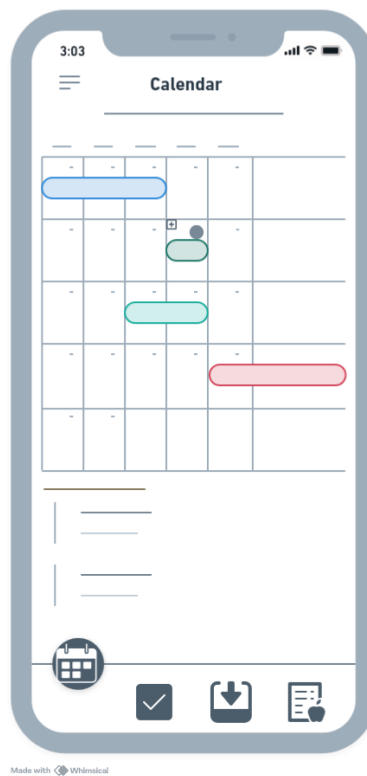


Figure 91 Calendar home page

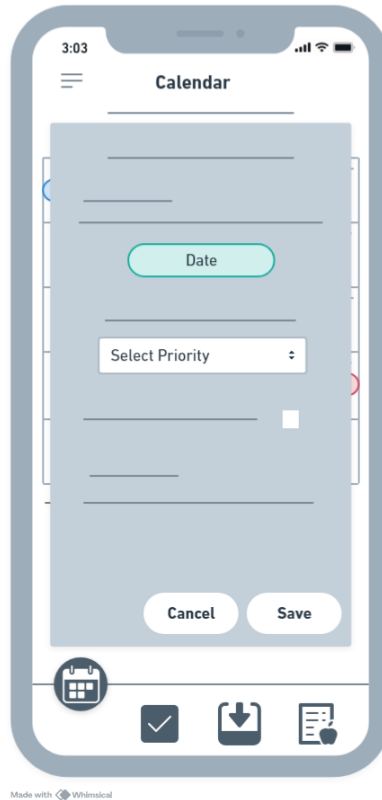


Figure 92 Add event

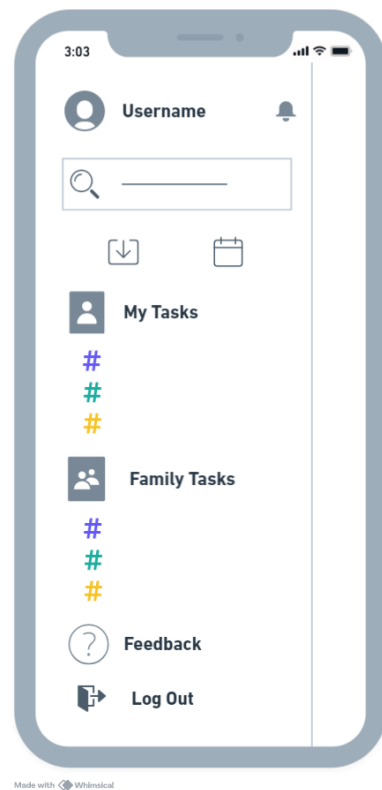
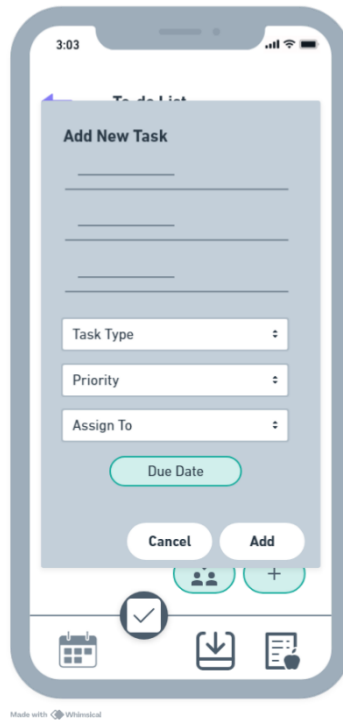
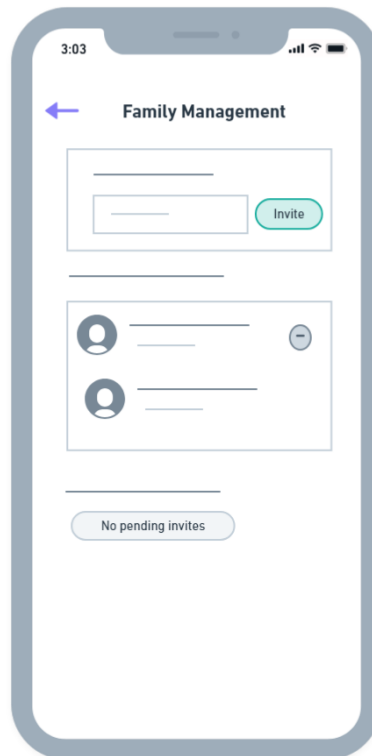


Figure 93 User and family sidebar



Made with Whimiscal

Figure 94 Add new task



Made with Whimiscal

Figure 95 Family management

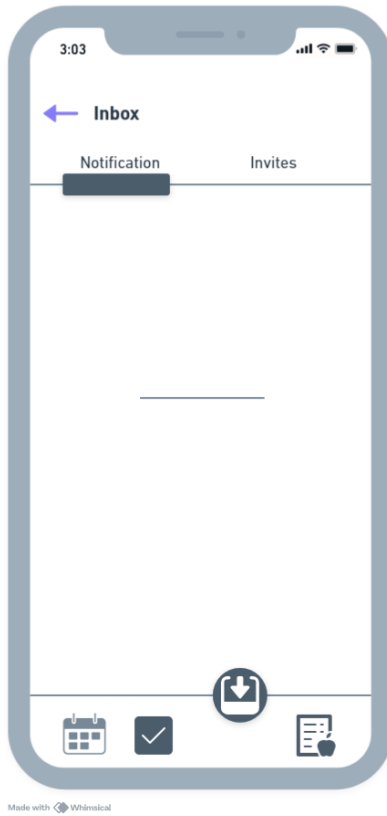


Figure 96 Users inbox

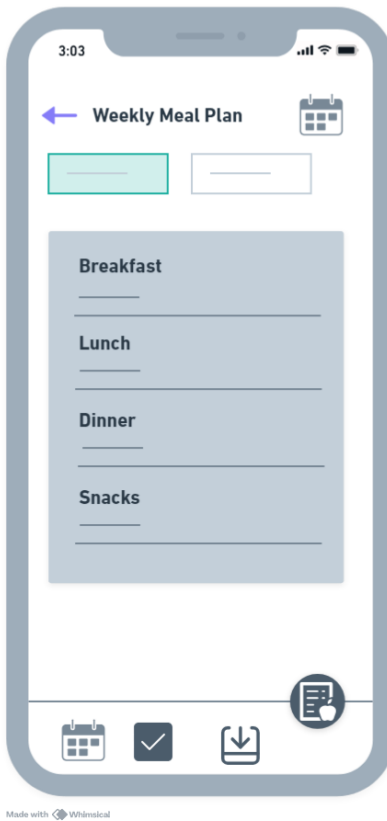


Figure 97 Weekly food plan interface

3.0 User Manual

Login and Sign-Up Screen

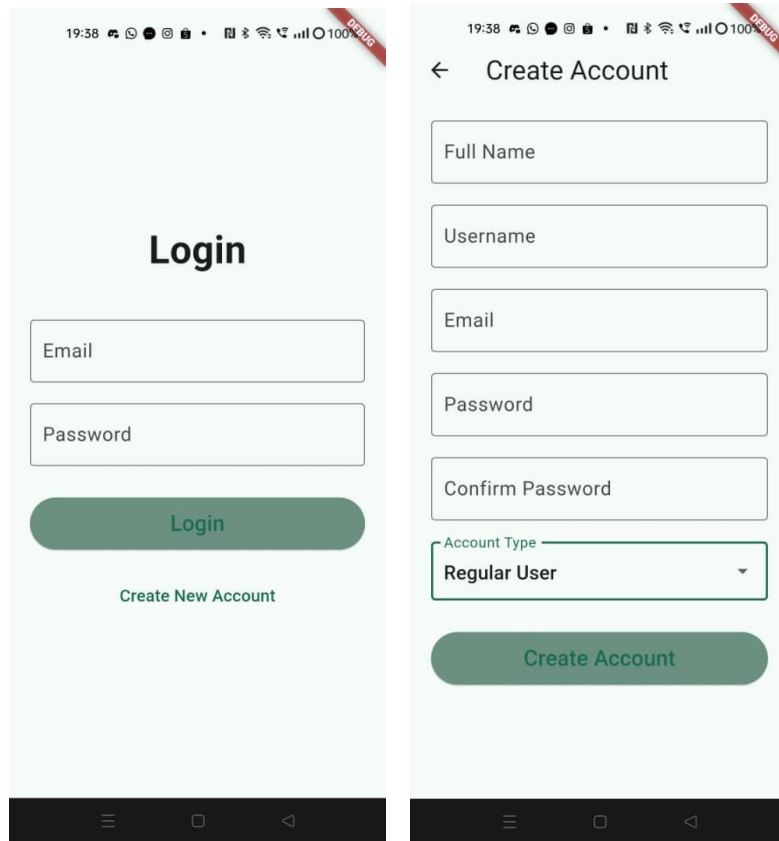


Figure 98 Login and Sign up screen

The images depict two screens of a mobile application. The first screen is for logging in, with fields for the Email and Password along with a Login button and a link to Create New Account for users who don't have an account. The second screen is for creating a new account, requesting the user's Full Name, Username, Email, Password, and a Confirm Password field to ensure accuracy. It also includes a dropdown to select the Account Type (defaulting to "Regular User") and a Create Account button to finalize the registration. These screens are designed to facilitate either logging in or account creation.

Main Page with a Calendar

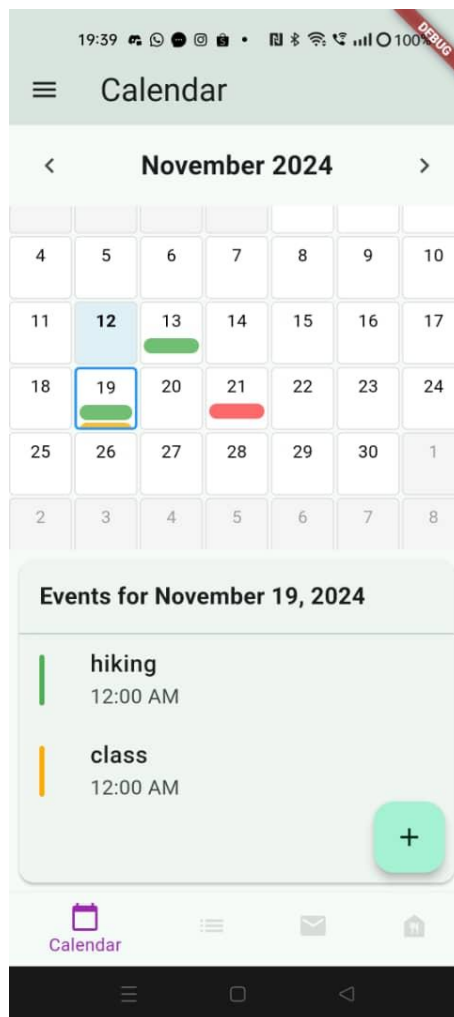


Figure 99 Main page with calendar

The page allows users to add tasks, appointments, special events, and other related items directly to the calendar on the main page. This feature enhances the transparency of the user's schedule by clearly displaying upcoming events. With this calendar view, users can easily access a summary of their upcoming schedule, which includes details about the event and its time, making it convenient for them to plan ahead.

Add Event on Calendar

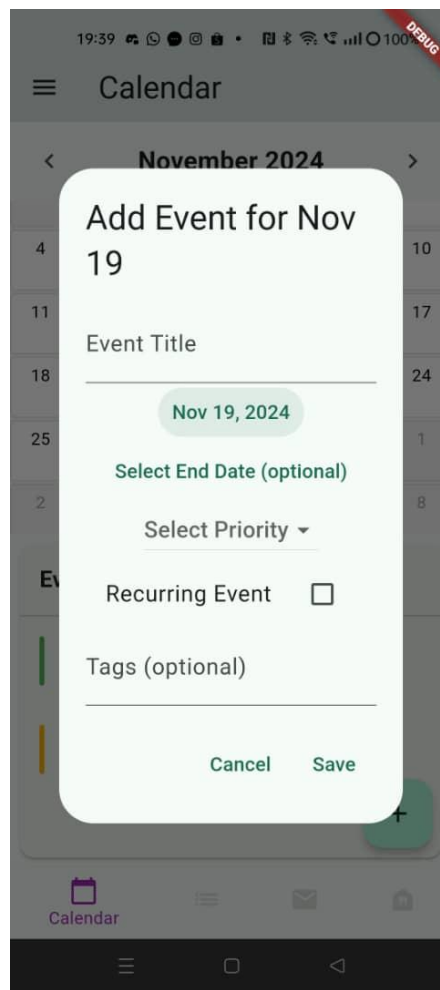


Figure 100 Add event on calendar

The image shows a screen from the calendar app where users can add a new event. The user is prompted to enter an Event Title, with an optional field to Select End Date. There's a dropdown to Select Priority for the event, and a checkbox for marking the event as a Recurring Event. An additional Tags field allows users to categorize the event, though it's optional. At the bottom, there are two buttons: Cancel to discard the changes and save to add the event to the calendar, providing a customizable and organized way for users to schedule their activities

Side Bar

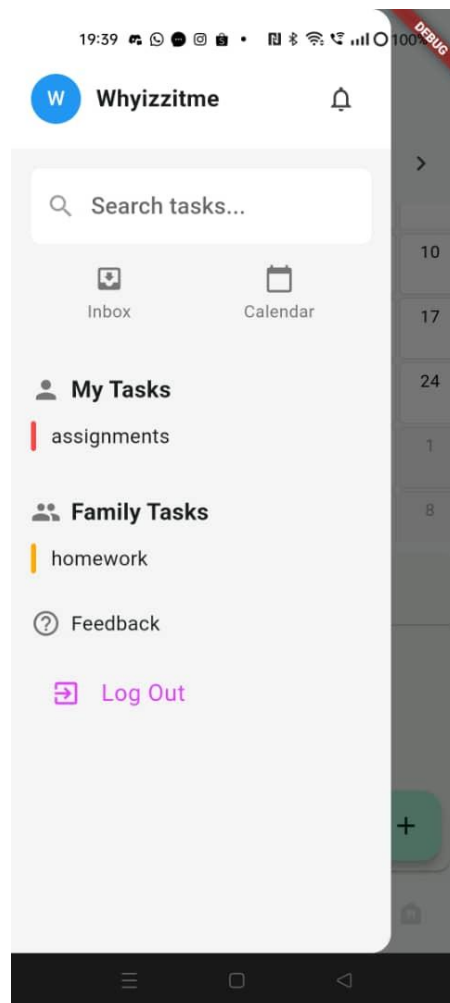


Figure 101 Side bar

The image shows a sidebar menu of a task management app. It includes a search bar at the top labeled Search tasks, allowing users to find tasks easily. Below the search bar, there are navigation options: Inbox for task-related messages, Calendar to view scheduled tasks, and sections for task categories. The My Tasks section lists tasks like assignments (marked with a red icon), while the Family Tasks section includes items like homework (indicated with an orange icon). Additionally, there is a Feedback option and a Log Out button at the bottom, providing the user with ways to log out of the app or provide feedback. The sidebar offers an organized layout for managing tasks.

To-Do List

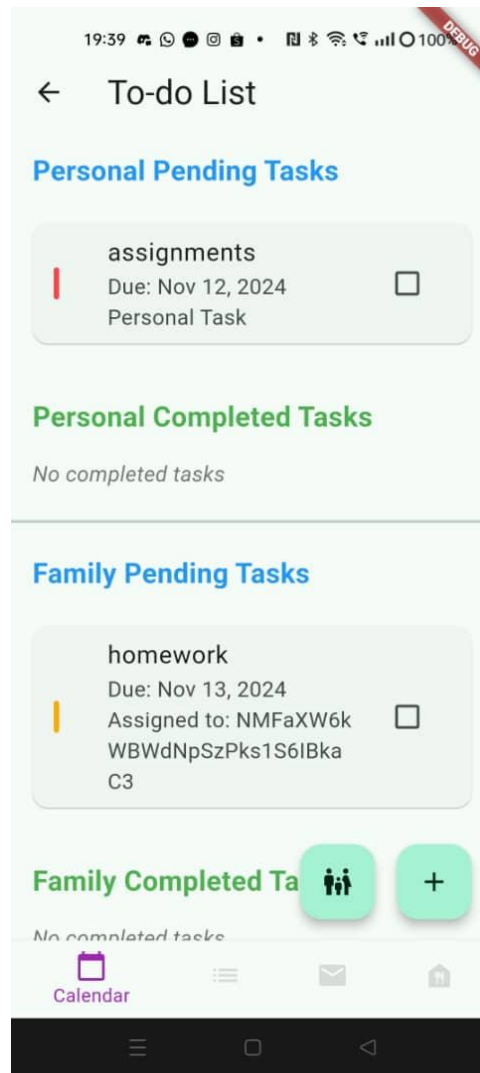


Figure 102 To-do List

The image shows a To-do List screen in a task management app, divided into sections for different types of tasks. Under Personal Pending Tasks, there is an assignments task listed, with the due date, with a red icon, indicating its importance. There is also a checkbox to mark the task as completed. The Personal Completed Tasks section is empty, as there are no completed tasks. Below that, the Family Pending Tasks section lists a homework task, due on November 13, 2024, with an orange icon. It also shows an Assigned to field, indicating who the task is assigned to. The Family Completed Tasks section is empty as well, with no completed tasks. At the bottom, there are buttons to access the Calendar and options to add new tasks or manage the task list.

Family Management System Page

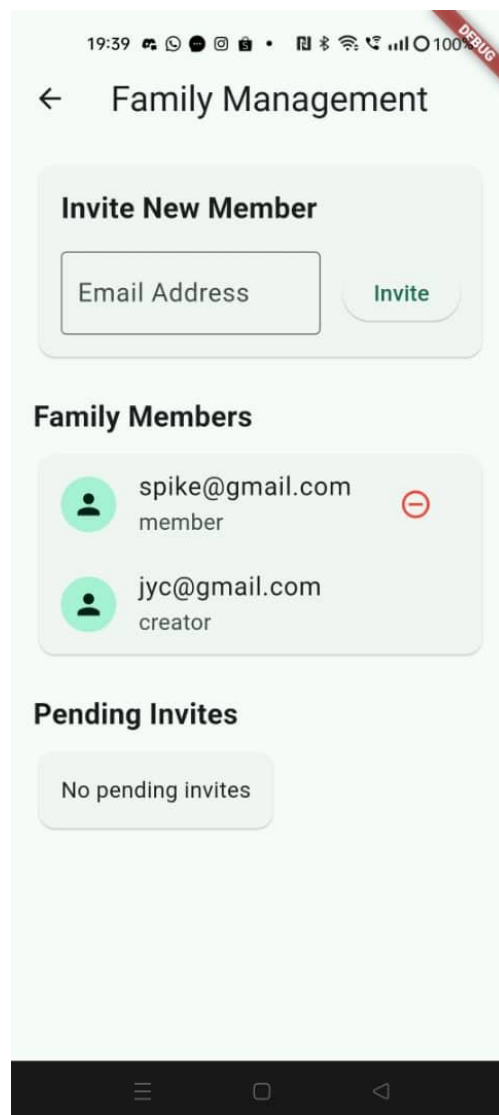


Figure 103 Family management system page

The image shows a Family Management screen in an app. At the top, there is an option to Invite New Member by entering an Email Address and clicking the Invite button to send an invitation. Below, the Family Members section lists two members: one labelled as a member and the other as a creator. Next to each name, there is an icon representing their role, and the creator member has an additional option to remove or edit. In the Pending Invites section, it shows that there are No pending invites, indicating no outstanding invitations have been sent. This screen allows users to manage family members and invitations within the app.

Inbox Page

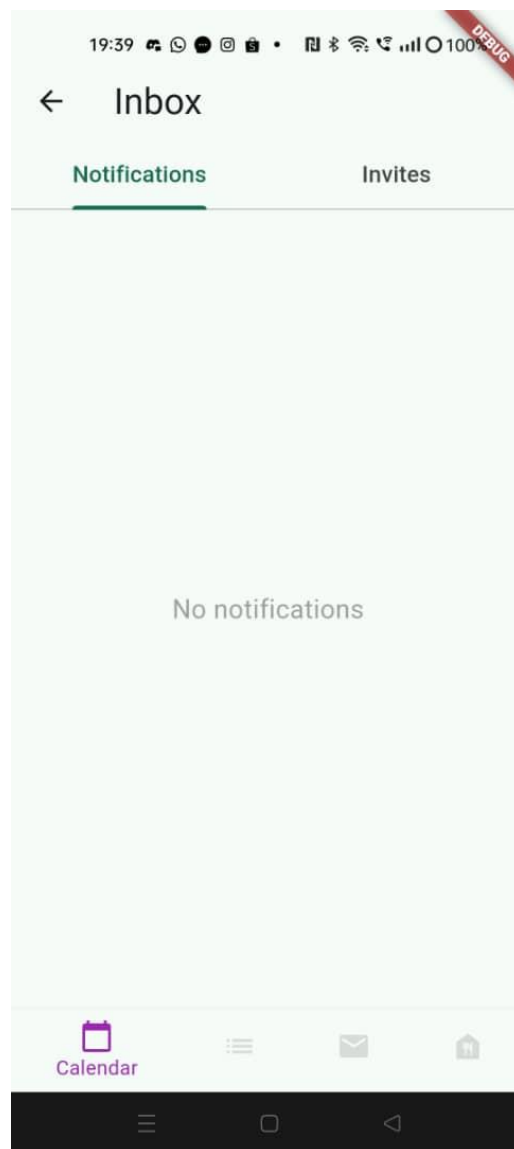


Figure 104 Inbox page

The image shows the Inbox screen of the app, with two tabs at the top: Notifications and Invites. Currently, the Notifications tab is active, but there are No notifications displayed, indicating that there are no recent updates or messages for the user. The bottom of the screen has the Calendar button, along with other icons for accessing different sections of the app. This screen provides users with a way to view notifications or invites related to their tasks or events.

Weekly Meal Plan

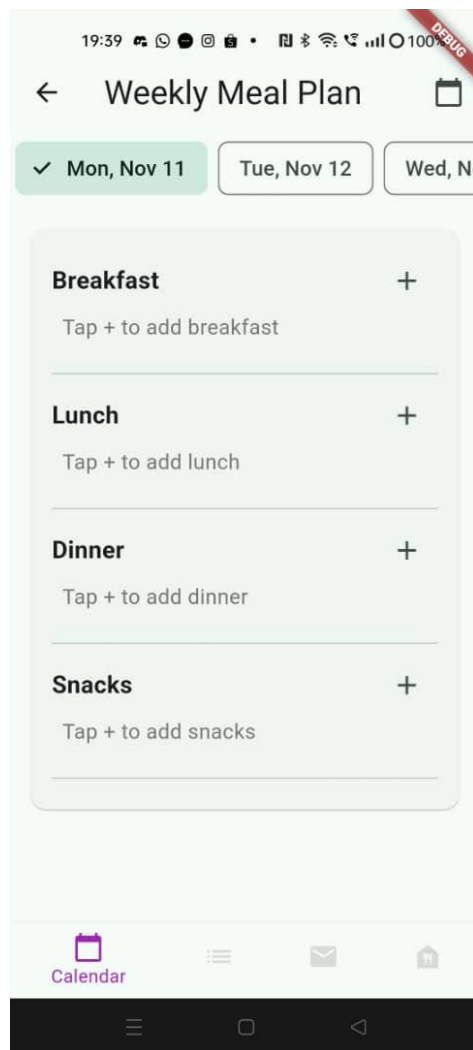
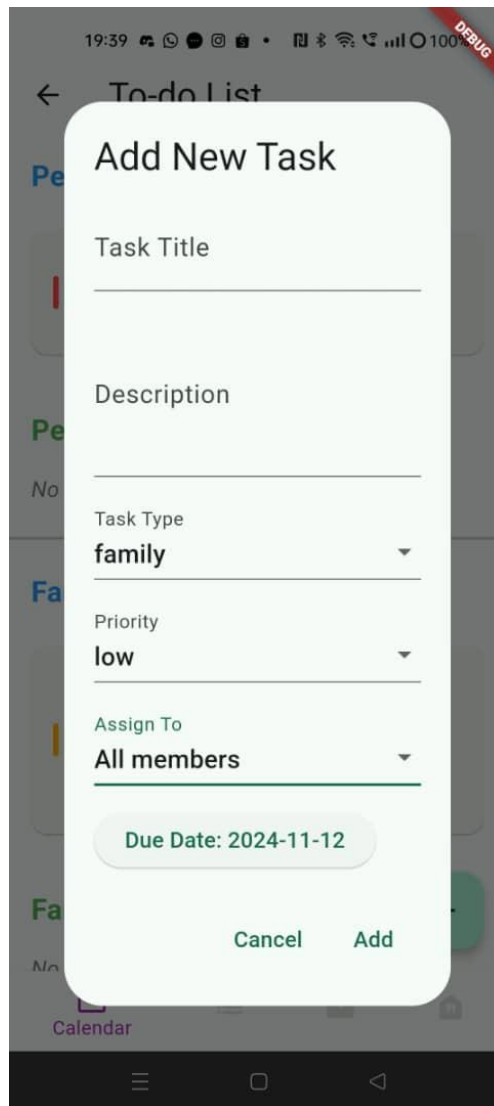


Figure 105 Weekly meal plan

The image shows a Weekly Meal Plan screen in an app, where the user can plan meals for each day of the week. The current day selected is displayed at the top, with options for other days available. For each day, there are sections to add meals: Breakfast, Lunch, Dinner, and Snacks. Each meal section has a + button that allows the user to tap and add details about the meals for that specific time of day. This screen helps users organize and track their meals, making meal planning more efficient.

Add New Task

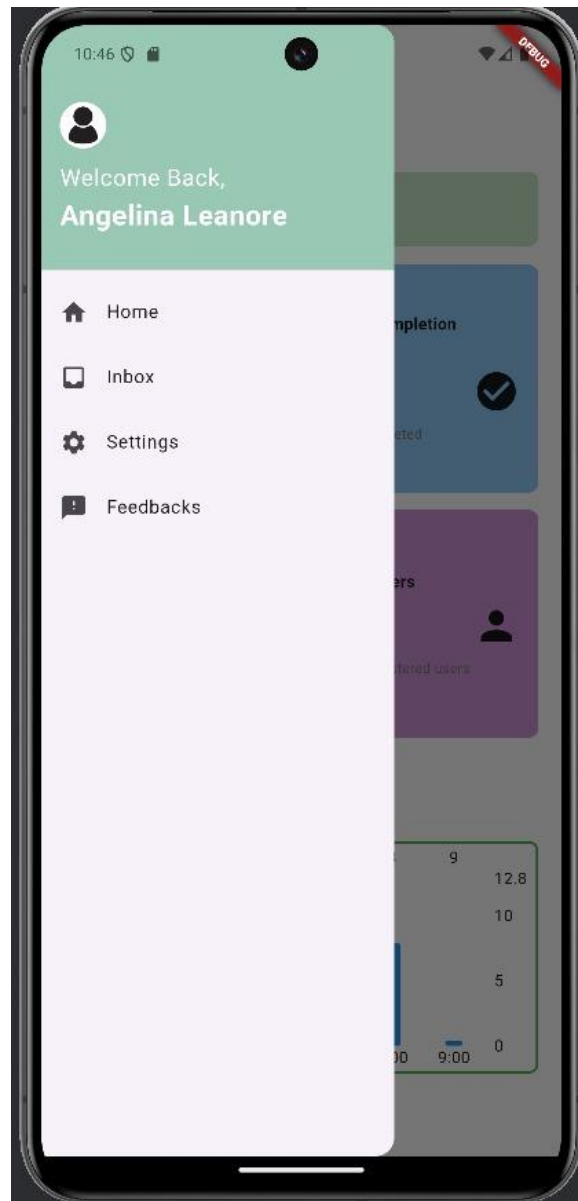


The screenshot displays a mobile application interface with a 'To-do List' screen in the background. A white modal dialog titled 'Add New Task' is centered on the screen. The dialog contains the following elements: a 'Task Title' text input field; a 'Description' text input field; a 'Task Type' dropdown menu currently showing 'family'; a 'Priority' dropdown menu currently showing 'low'; an 'Assign To' dropdown menu currently showing 'All members'; a green pill-shaped button indicating the 'Due Date: 2024-11-12'; and two green buttons at the bottom labeled 'Cancel' and 'Add'. The background app shows a list of tasks with categories like 'Pe', 'No', 'Fa', and 'No' visible. The top status bar shows the time as 19:39 and a 'DEBUG' banner in the top right corner.

Figure 106 Add new task

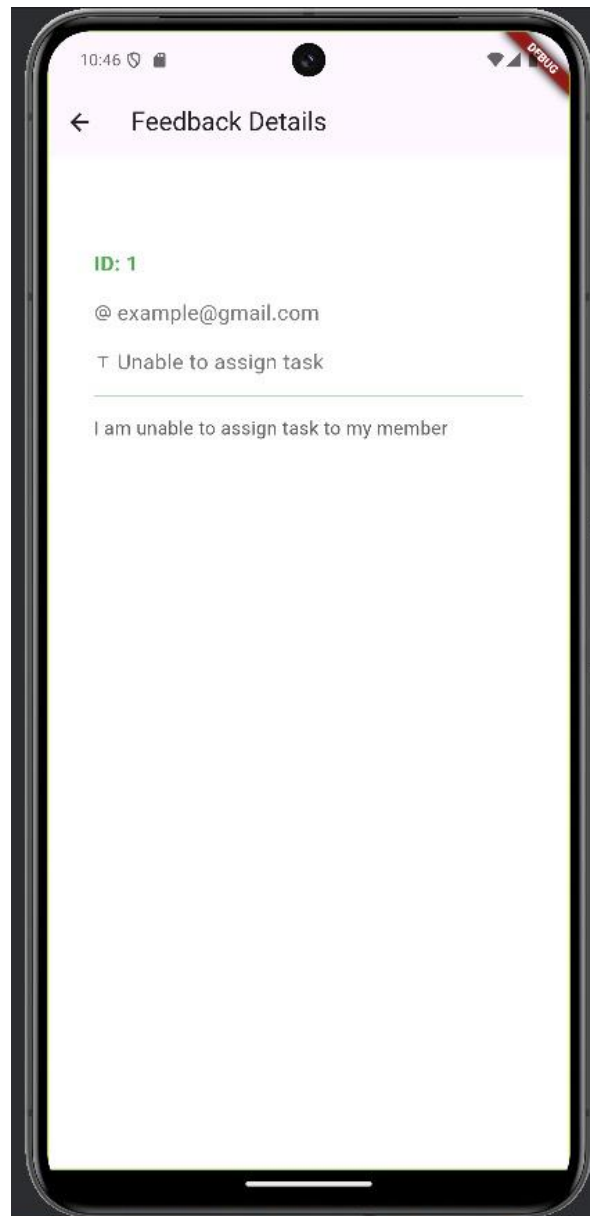
The image shows a screen for adding a new task in a to-do list app. Users can input a task title and description, select a "Task Type" (currently set to "family"), set the "Priority" level (set to "low"), and choose an assignee (set to "All members"). At the bottom, there are options to cancel or add the task. This setup is typical for a task management app, allowing users to organize and prioritize tasks by category, importance, and responsibility within a group or family context.

Admin Sider Bar



The image displays an admin sidebar menu in an app, welcoming the user. The sidebar provides navigation options, including "Home," which likely leads to the main dashboard; "Inbox," where messages or notifications may be stored; "Settings," for adjusting app configurations or personal preferences; and "Feedback," possibly for viewing or managing user feedback. The sidebar appears over a background with different coloured sections that may represent various admin panels or dashboard metrics, giving the admin quick access to essential management tools and insights.

Feedback Section



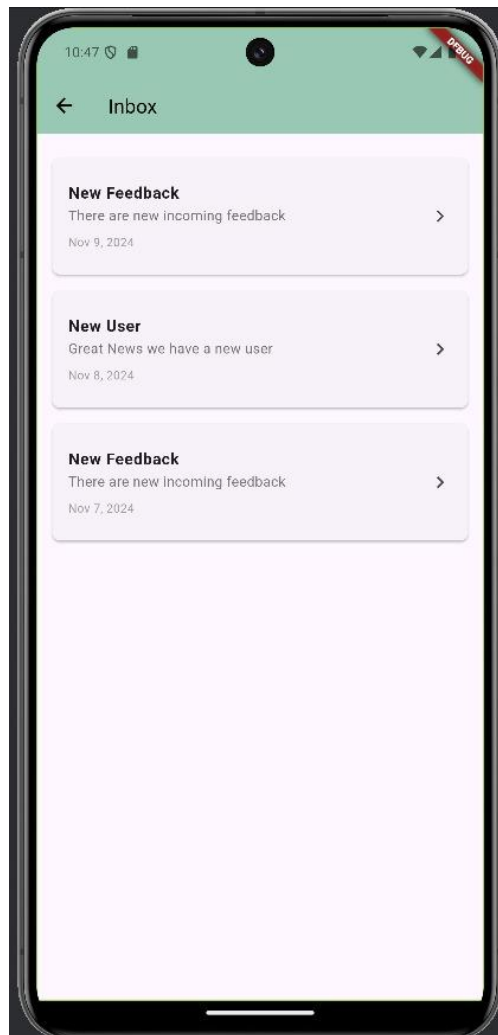
The depicted diagram shows a "Feedback Details" screen from a mobile app, where a user has submitted feedback regarding a task assignment issue. The feedback entry is identified by an ID of 1 and linked to the email "example@gmail.com." The title "Unable to assign task" summarizes the issue, with a further description stating the user is unable to assign tasks to their team member. This information helps the development team understand and address the problem.

Search Feedback ID Section



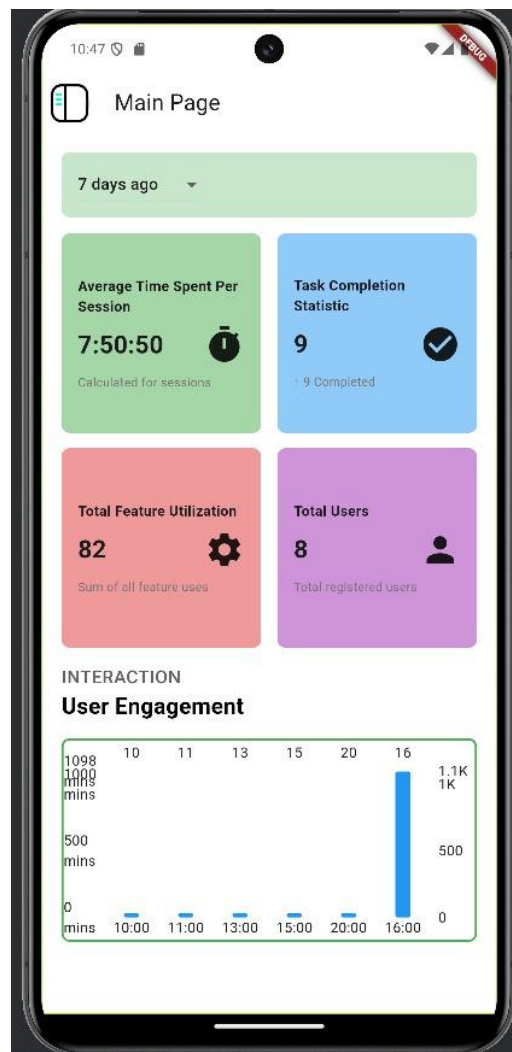
The depicted diagram shows a "Search by ID" screen in a mobile app, allowing users to locate specific feedback entries by ID. The screen displays two feedback entries with IDs "1" and "2", each listed alongside a timestamp indicating they were submitted "2 days ago". Each entry is accompanied by a checkbox, likely for selecting or managing multiple feedback items. The presence of a search bar at the top suggests users can quickly find feedback entries by typing in an ID.

Inbox Notifications List



The mobile app screen displays an “Inbox” section with a list of notifications, each shown as a card with title, brief message summary, and chevron icon, indicating that tapping will open detailed views. The notifications include messages like “New Feedback” and “New User”. The design is clean and user-friendly, providing a straightforward way to view incoming messages.

Main Page Admin



This mobile app screen titled "Main Page" provides an overview of user engagement and usage statistics through coloured cards and a bar graph. At the top, a green card indicates the last update as "7 days ago." Below, four cards display key metrics: "Average Time Spent Per Session," "Task Completion Statistic," "Total Feature Utilization," and "Total Users," with respective values for each. The "User Engagement" section at the bottom includes a bar graph showing usage patterns throughout the day. The clean, color-coded layout allows users to quickly interpret each metric.

4.0 Automated System Testing

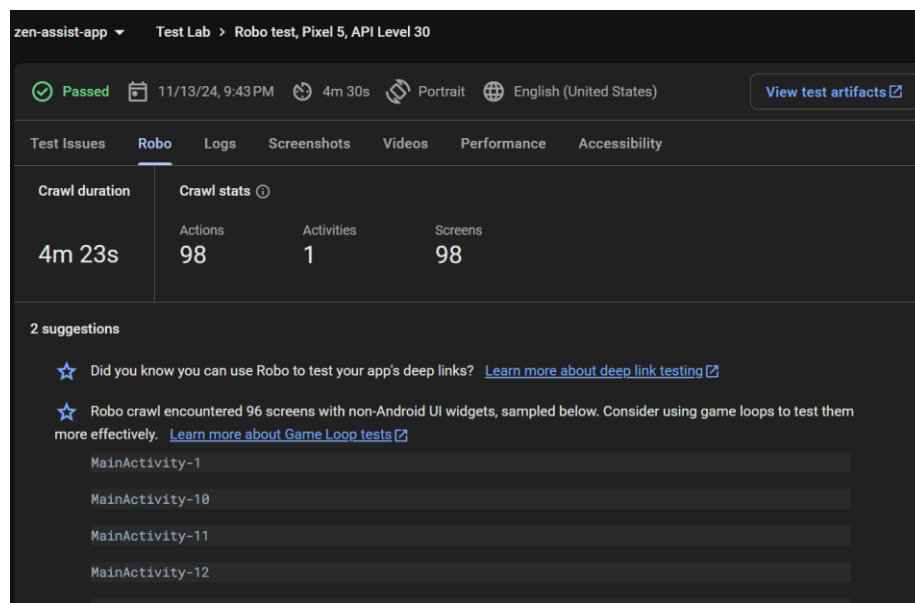


Figure 107: Firebase Test Results

The outcomes of a successful Robo test conducted on a Pixel 5 running Android API Level 30 in Google's Firebase Test Lab are shown in this screenshot. On November 13, 2024, at 9:43 PM, the exam was administered. It lasted 4 minutes and 30 seconds in portrait mode, with English (United States) selected as the language. The Robo crawler explored the app for 4 minutes and 23 seconds, completing 98 actions on 98 screens, however it only came across one unique activity during the test.

Several recommendations for enhancing the app's test coverage were given by Firebase. In order to make sure that links leading to certain content within the app work properly, it first suggests utilizing Robo for deep link testing. Second, the study mentions that 96 screens with non-Android UI widgets that might not react well to conventional UI interactions were discovered by the Robo crawl. To better cover these non-standard UI aspects, it recommends taking game loop testing into account. This is especially helpful for apps with intricate interfaces, like games. Examples of various states or instances of the main activity that were visited throughout the crawl are shown in the sample list of screens (e.g., MainActivity-1, MainActivity-10).

In conclusion, the Robo test was successful, but it also identifies areas that might be improved with more testing options, particularly for deep link navigation and non-standard UI elements, to guarantee full app functionality.

5.0 Conclusion

The ZenAssist project is very relevant in today's society because it helps users find that fine line between productivity, mental health and physical health without compromising any of the three aspects. Stress levels have reached unprecedented levels, and Zenassist as a solution to this problem has incorporated task management, health, and even diet in a single structure. Its components, which include custom to-do lists, weekly meals organization, and health prompts, are aimed at improving efficiency with regard to health. The application is based on Firebase, which provides the app with ease of updates and data transfer to the users without any lag.

ZenAssist features a colorful and appealing interface, multiple user roles including an admin who oversees the performance and usage of the system and a family who share the responsibility of task management. The application is equipped with efficient tracking functionalities that help in evaluation of the app and also how the users of the app engage in it, such as how long do users spend in a session, how much of the features are utilized etc.

To sum up, the ZenAssist app is all inclusive as a productivity enhancing app in alleviating modern problems wherever efficiency and health are pitched against each other for the user's attention. This unique offering files for a constructive order in the usage of time more so for the users who are productive and active in the present era.

6.0 References

Ray, J. (2024, October 18). *World Unhappier, more stressed out than ever*. Gallup.com.
<https://news.gallup.com/poll/394025/world-unhappier-stressed-ever.aspx>

Betune, S. (2023, March 9). *What a new study from the American Psychological Association (APA) tells us about stress since COVID-19*. American Psychological Association.
<https://drgiamarson.com/what-a-new-study-from-the-american-psychological-association-apa-tells-us-about-stress-since-covid-19/>

7.0 Workload Matrix

Name (TPNumber)	User role	Task
Angelina Leanore (TP072929)	Admin	<ul style="list-style-type: none"> • Create User Feedback • View User Feedback Details • Monitor the system performance • Examine the number of new users • Examine the number of tasks completed by users • Examine the number of features utilized by the app • Examine the user interaction statistics • View Admin inbox
Chua Jun Yi (TP065882)	User	<ul style="list-style-type: none"> • Browse the calendar on the main page • Create to-do lists • Weekly meal plans • Create inbox • Ability to create a new account • Ability to Create a login page
Chang Zheng Fang (TP066899)	Family	<ul style="list-style-type: none"> • Invite users to contribute on the family to-do list • User role and permission • Remove people from collaborating

		<ul style="list-style-type: none">• Transfer admin to user• Delegating task
--	--	----------------------------------------------------------------------------------------------------