CSE431

Mobile Programming

PROJECT

HEDIEATY MOBILE APPLICATION

FINAL PROJECT DOCUMENTATION

Name	ID
Karim Bassel Samir Anby	20P6794

Contents

Introduction

Hedieaty is a mobile application designed to simplify the process of managing and sharing gift lists for various special occasions, such as birthdays, weddings, graduations, and holidays. The app provides users with an intuitive interface to create, modify, and organize their gift lists, offering functionalities such as adding gifts manually or using an integrated barcode scanner.

Hedieaty aims to enhance the joy of gift-giving by enabling users to share their lists with friends and family, who can pledge to purchase specific gifts in real-time. Key features include flexible gift management, cloud synchronization using Firebase Realtime Database, and notification systems for updates. By leveraging Flutter, the app ensures a seamless user experience across platforms, with deployment targeted for the Amazon App Store.

With robust database integration, advanced search and filtering capabilities, and features like real-time status updates and visual indicators for pledged items, Hedieaty addresses the challenges of coordinating gift-giving and promotes meaningful connections during special moments.

Survey of Similar Apps in the Market

1. Giftster

- **Platform**: Available on App Store, Google Play Store, and web.
- **Description**: Giftster allows users to create and share gift lists for any occasion. It supports group features where families or friends can create shared lists.
- **Key Features**:
    - List sharing with family and friends.
    - Marking items as purchased without notifying the creator.
    - Group management for collaborative gift lists.
    - Cross-platform compatibility (web and mobile).
- **Advantages**:
    - User-friendly interface.
    - Allows group collaboration, enhancing organization for families.
    - Cross-platform support ensures flexibility in accessing lists.
- **Disadvantages**:
    - Limited real-time synchronization features compared to Firebase-backed apps.
    - Does not integrate advanced scanning features, such as barcode scanning.
    - Free version contains ads and limited features.

---

2. MyRegistry

- **Platform**: Available on App Store and Google Play Store.

- **Description**: MyRegistry enables users to create universal gift registries that aggregate items from any store. It is commonly used for weddings, baby showers, and other major events.
- **Key Features**:
    - Universal gift registry.
    - Barcode scanning to add items from physical stores.
    - Real-time syncing across devices.
    - Allows cash gift contributions.
- **Advantages**:
    - Comprehensive registry capabilities across multiple stores.
    - Barcode scanning simplifies the process of adding physical items.
    - Integration with major retail platforms.
- **Disadvantages**:
    - Overwhelming for simple occasions like birthdays.
    - Requires user registration to access features.
    - Premium features are behind a paywall.

---

3. Elfster

- **Platform**: Available on App Store, Google Play Store, and web.
- **Description**: Elfster is a gift exchange platform that includes wishlist functionality, commonly used for Secret Santa or group gift exchanges.
- **Key Features**:
    - Gift exchanges for groups.
    - Wishlist creation and sharing.

    - Anonymous matching for gift exchanges.
    - Real-time notifications and reminders.
- **Advantages**:
    - Unique focus on group gift exchanges.
    - Built-in reminders and notifications improve organization.
    - Free to use with no major feature restrictions.
- **Disadvantages**:
    - Limited to group-based gift exchanges, making it less versatile for individual list management.
    - No barcode scanning or advanced search/filtering options.
    - Relies heavily on user engagement for full functionality.

---

4. WishUpon

- **Platform**: Available on App Store and Google Play Store.
- **Description**: WishUpon is a wishlist app that aggregates items from online stores, allowing users to track prices and share their lists with others.
- **Key Features**:
    - Price tracking for online items.
    - Wishlist sharing.
    - Notifications for price drops or availability changes.
- **Advantages**:
    - Focus on price tracking is beneficial for budget-conscious users.
    - Simplified list-sharing capabilities.
    - Clean and modern user interface.

- **Disadvantages**:
    - Limited focus on physical gifts or event-based lists.
    - No options for group collaboration or pledging.
    - Heavy emphasis on online shopping over in-person purchases.

---

5. Babylist

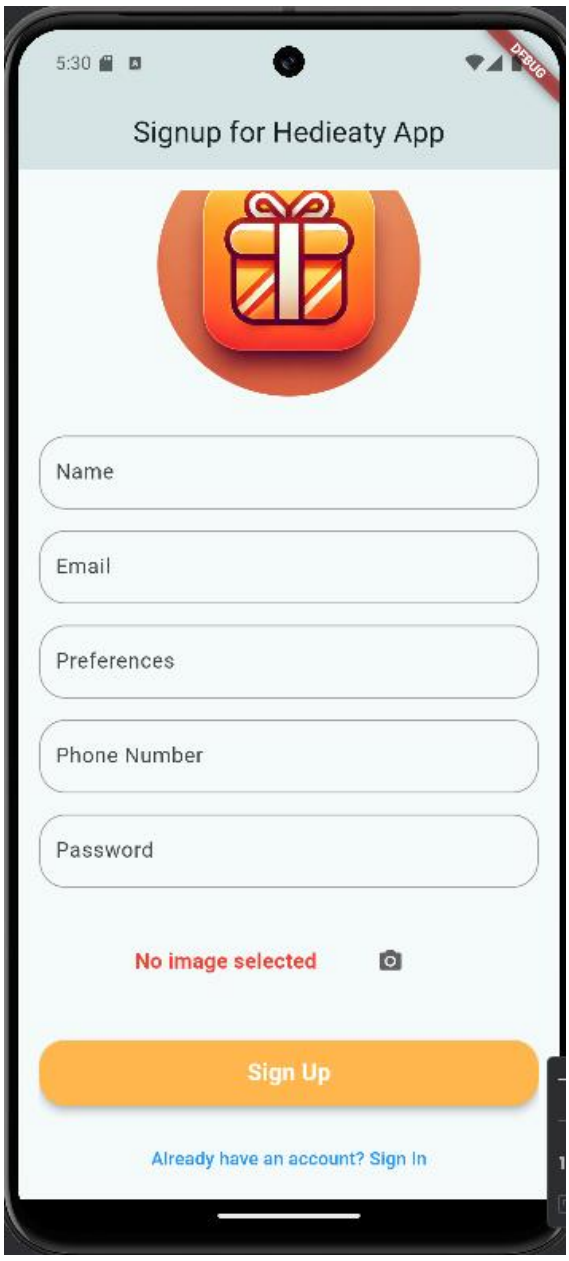
- **Platform**: Available on App Store and Google Play Store.
- **Description**: Babylist is a universal baby registry app that allows users to add items from any store and include personal services (e.g., babysitting or meal deliveries).
- **Key Features**:
    - Universal registry functionality.
    - Integration of non-product services (e.g., gift cards or contributions).
    - Barcode scanning for easy item addition.
- **Advantages**:
    - Well-suited for event-specific needs (e.g., baby showers).
    - Barcode scanning simplifies adding physical gifts.
    - Allows personalized service contributions.
- **Disadvantages**:
    - focused only on baby-related events.
    - Limited customization options for general-purpose gift lists.
    - No advanced notifications or collaboration features.

Comparative Analysis

Feature	Hedieaty	Giftster	MyRegistry	Elfster	WishUpon	Babylist
Purpose	General gift lists	Family lists	Universal registry	Gift exchanges	Online wishlists	Baby-specific registry
Barcode Scanning	Yes	No	Yes	No	No	Yes
Real-Time Sync	Yes	Limited	Yes	Yes	No	Yes
Platform Availability	Android	iOS, Android, Web	iOS, Android	iOS, Android, Web	iOS, Android	iOS, Android
Unique Feature	Firebase integration	Group sharing	Cross-store registry	Secret Santa	Price tracking	Non-product gifts

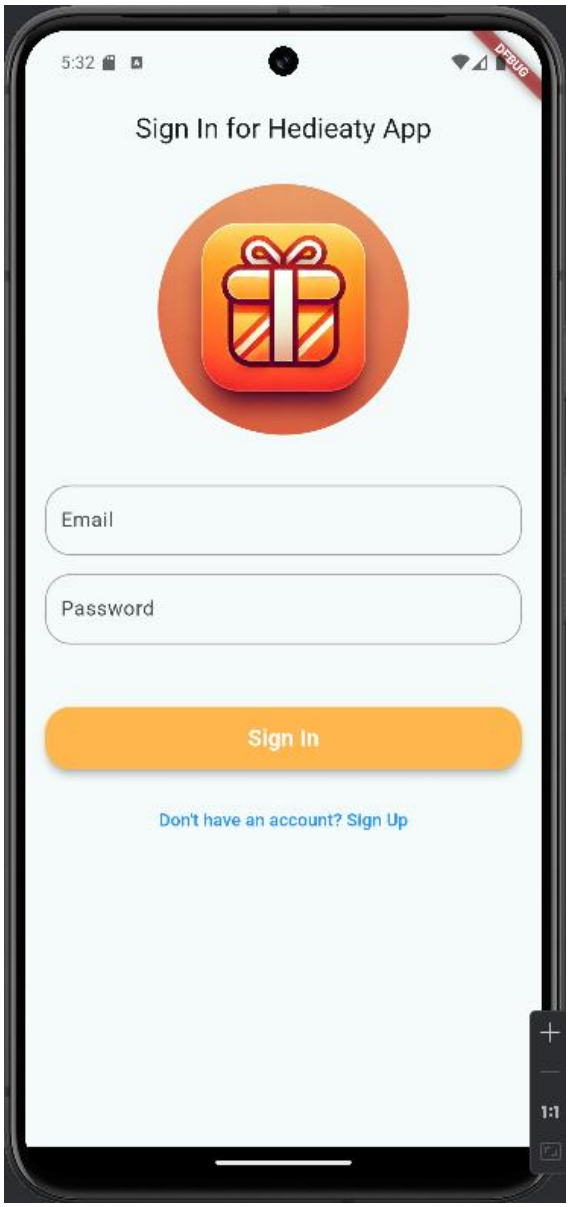
User Interface Design

Sign Up Page



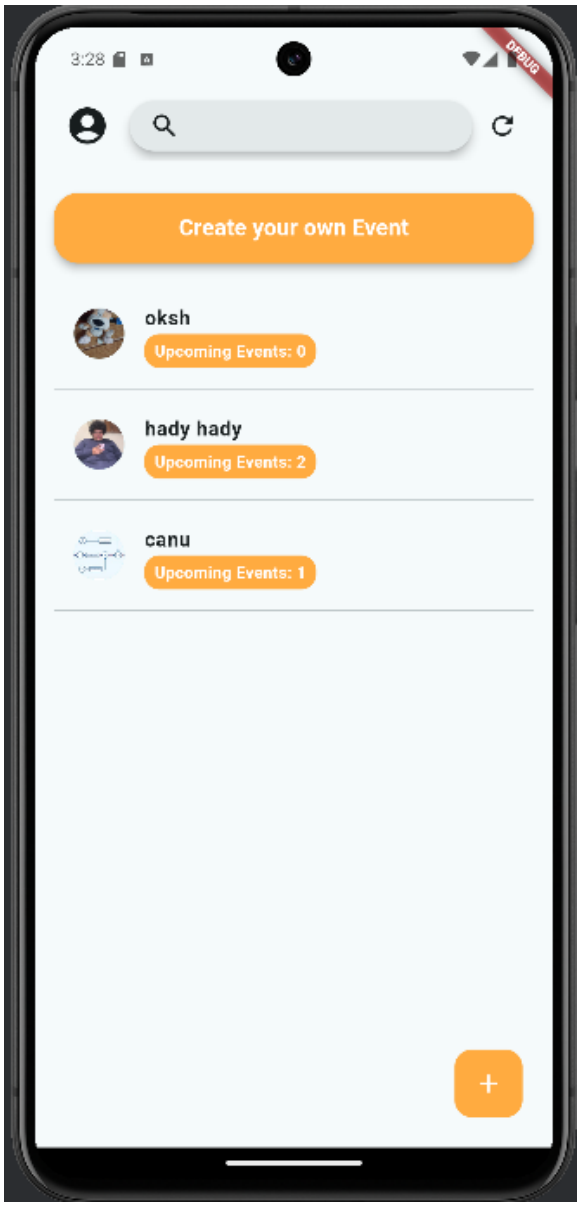
The **Signup** interface of the Hedieaty App offers a clean and user-friendly design that guides the user through the registration process. At the top, a prominent logo is displayed within a circular avatar. Below that, users are prompted to enter their details, including name, email, preferences, phone number, and password, with each field clearly labeled and featuring rounded input borders that change color when focused or when there is an error. There is also a section for users to select a profile image, accompanied by a camera icon for easy access to the image picker. The interface is designed with ample spacing between elements, ensuring a smooth and intuitive experience. At the bottom, the signup button is easily accessible, and users are given the option to navigate to the sign-in page if they already have an account. The design is responsive and ensures real-time feedback through form validation and interactive elements like snack bars.

Sign In Page



The **SignIn** interface of the Hedieaty App features a straightforward and user-friendly design for users to log into their accounts. The top of the screen displays a large circular avatar with the app's logo, providing a welcoming visual cue. Below the logo, users are prompted to enter their email and password, with each field offering rounded input borders that change color when focused or when an error occurs. The form is validated to ensure the email and password are properly filled in. A large **Sign In** button is prominently placed beneath the fields, changing to a loading indicator when the user presses it, giving feedback while the app processes the login. The button is styled with an orange background for visibility and easy interaction. At the bottom, there's a clear link to the Sign Up page for new users who don't have an account. The design is clean and efficient, ensuring that the user can easily navigate through the login process with clear instructions and real-time validation feedback.

Home Page

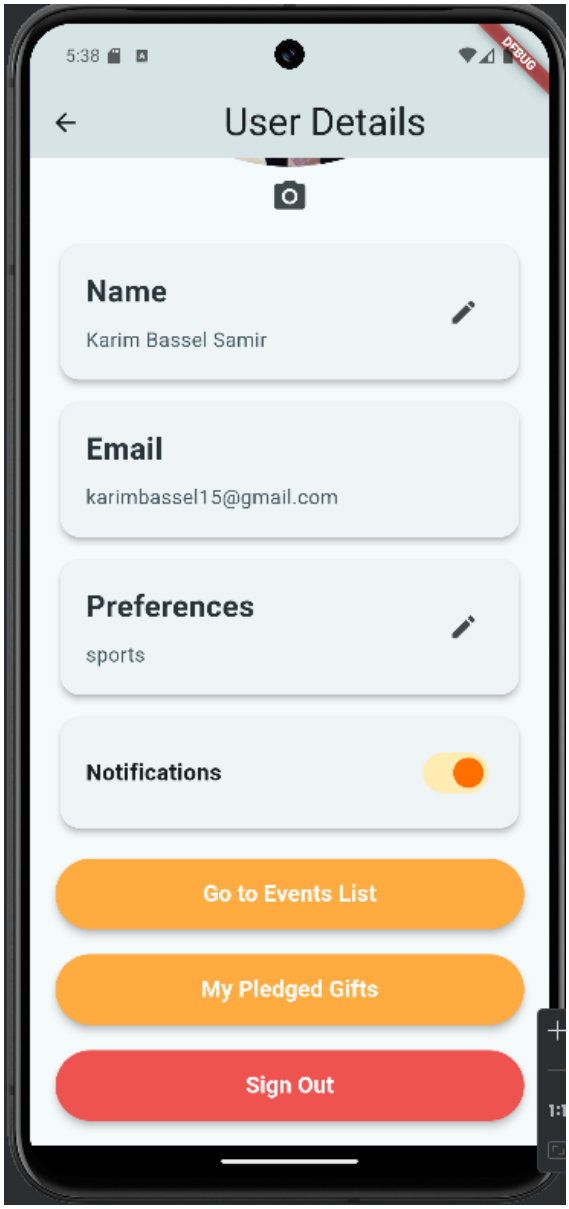


The **Home** screen of the Hedieaty App offers a dynamic and interactive interface for users to manage their friends and events. The top of the screen features a customizable AppBar, with a profile icon that allows users to access their personal profile when tapped. A search bar is also included, enabling users to filter their friend list by name in real-time. A refresh icon is used to get the most recent friends for the user. The main body of the screen displays a list of friends, each with a circular avatar and details such as the number of upcoming events. Tapping on a friend navigates to a detailed event list for that specific friend.

At the bottom, a floating action button (styled with a rounded rectangular background in orange) opens a menu to add a new friend, either manually by phone number or from contacts. The process of adding friends is facilitated with clear prompts, and the list updates dynamically after a friend is added. An "Add Friend Manually" option prompts a dialog for phone number entry, while the "Add Friend from Contacts" fetches contacts from the device. Additionally, the

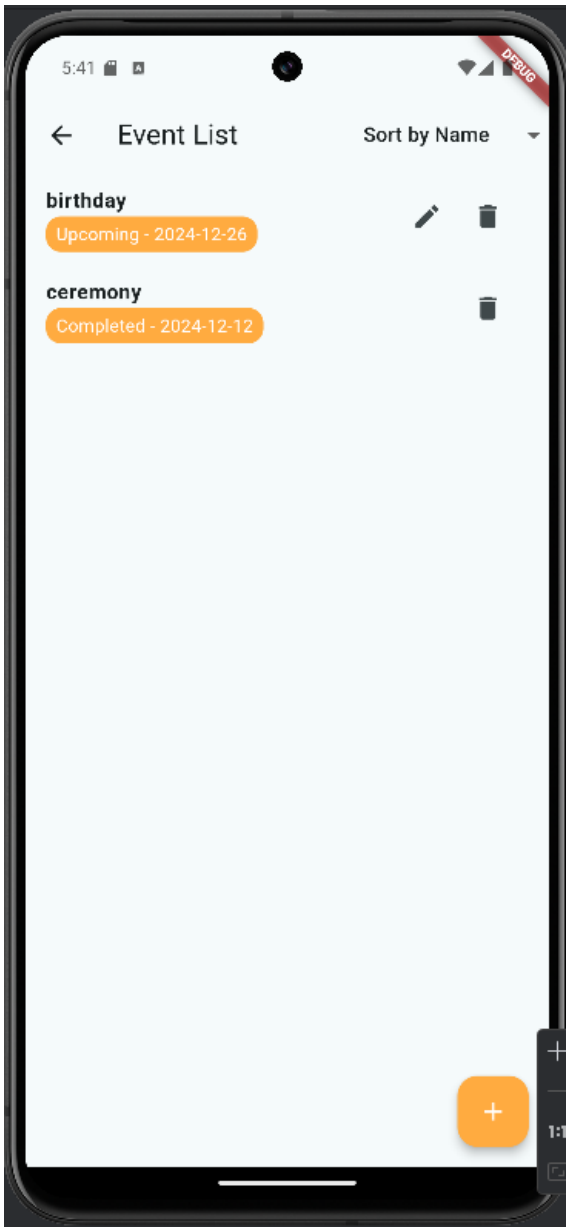
design is well-structured with clear calls to action, smooth navigation, and visually distinct buttons, ensuring an engaging and intuitive experience for the user.

Profile Page



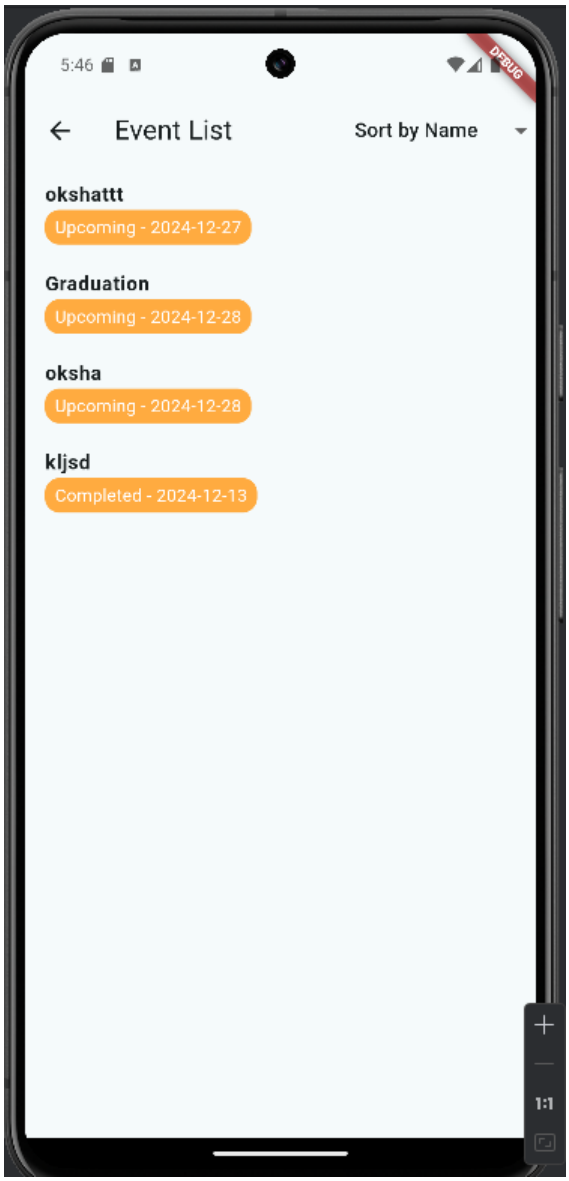
The Profile screen in this Flutter code displays and allows editing of a user's profile details, such as name and preferences. It also manages the user's profile image, which can be updated via an image picker. The screen includes a notification switch to toggle notifications on and off. The user can navigate to event lists and their pledged gifts, or sign out of the app. Editing functionality is provided through TextEditingController instances for each field, and changes are saved through the FriendController. The layout includes cards for profile information and buttons for navigation, providing a user-friendly interface for profile management.

User Event List



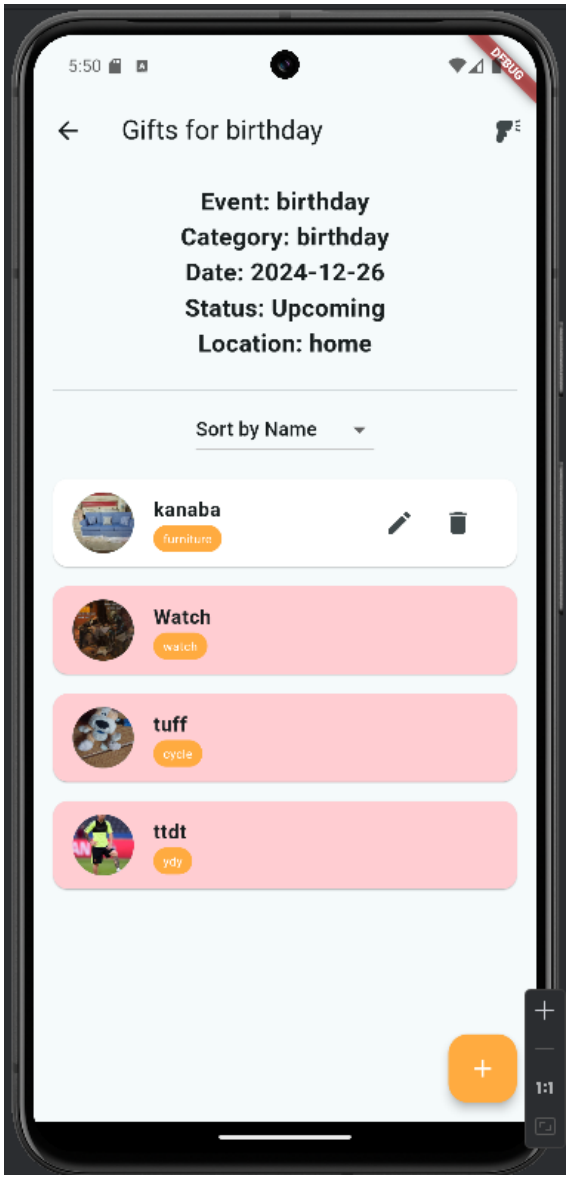
When the user accessing the event list is the owner, they have full control over the events, but they cannot edit past events. They can view, edit, and delete events for upcoming or ongoing events, with the ability to sort the list by name, category, or status. The owner can add new events using a floating action button, which opens a dialog to enter event details. Editing and deleting events is possible through icons next to each event, with changes synced to Firebase. The owner can also navigate to the associated gift list for each event. Overall, the owner has an interactive and dynamic experience to manage their events efficiently, with restrictions on past events to ensure they remain unchanged.

Friend Event List

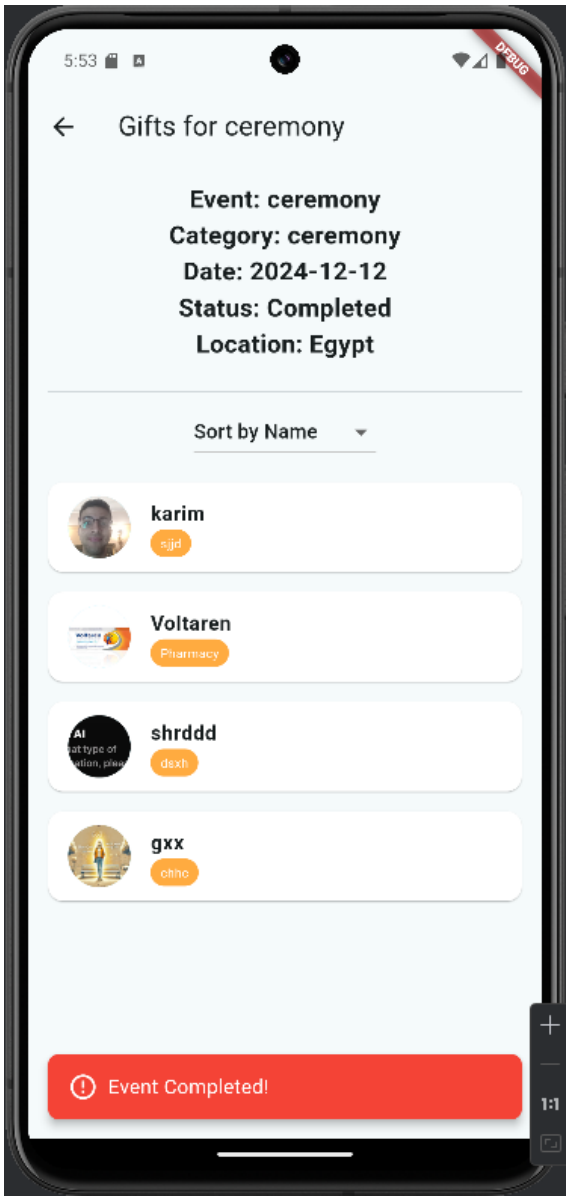


When the user accessing the event list is not the owner, they can only view the events associated with their friend. They have the ability to see details about each event, including its name, category, status, and date, but they cannot edit or delete any events. The events are sorted by the selected criteria, such as name, category, or status. The user can tap on an event to view its associated gifts, but the floating action button to add a new event is hidden, as they do not have the necessary permissions. This limited access ensures that the non-owner can only interact with the events in a view-only capacity, maintaining the integrity of the event management for the owner.

Owner Gift List for an Upcoming Event

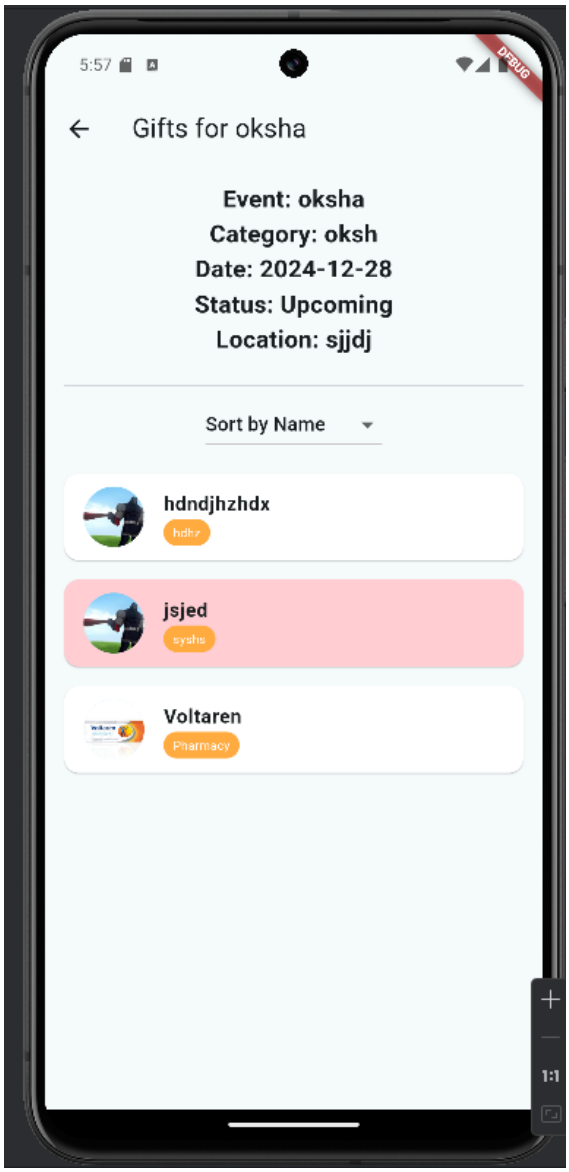


Owner Gift List for a completed event:



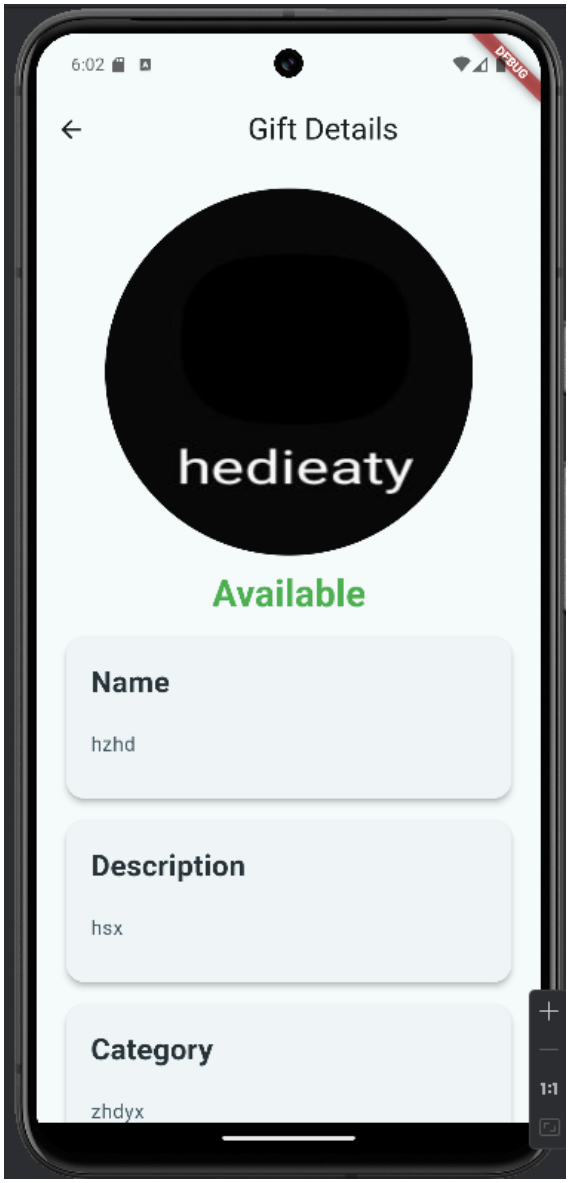
When the owner of the event accesses the gift list, they are presented with full control over the event's gifts. The page displays detailed information about the event, including its name, category, date, status, and location. The owner can sort the gifts by name, category, or status using a dropdown menu. The gift list shows each gift's details, including its name, category, and status (e.g., Available or Pledged). Pledged gifts are color-coded in red to make them visually distinct. The owner has the ability to edit or delete gifts that are marked as "Available" and are part of an "Upcoming" event. They can also add new gifts to the list by tapping the floating action button. If the owner taps on a gift, they can view its details and make any necessary updates. Additionally, the owner can use the barcode scanner to quickly scan and update gift information. This setup ensures that the owner has full administrative control over the event's gifts, making it easier for them to manage the event and its participants effectively.

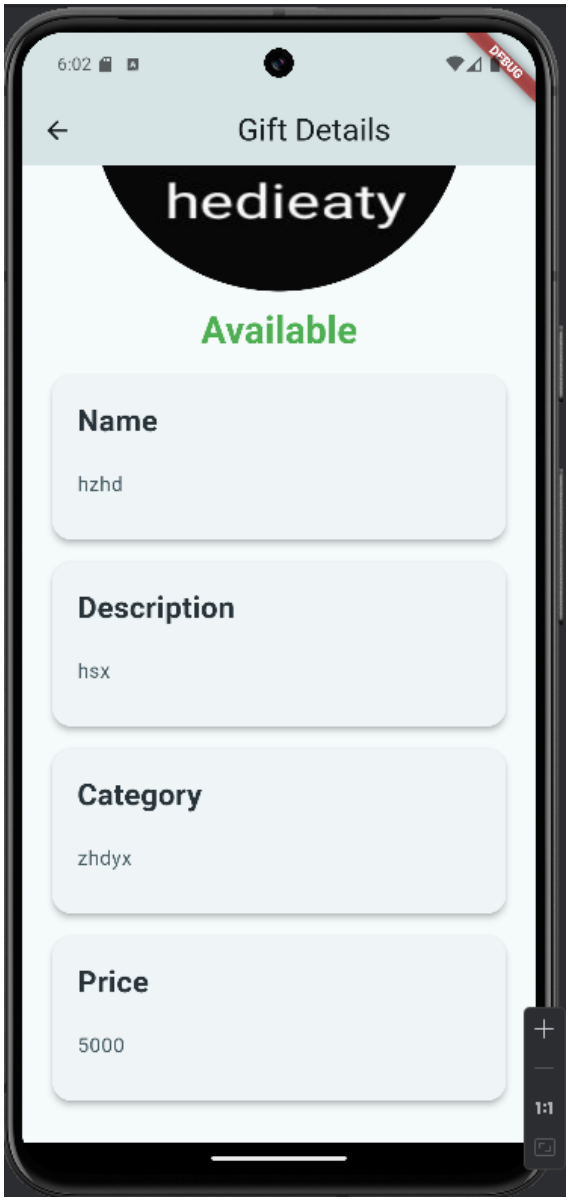
Non-Owner Gift List for an upcoming event:



When you access a friend's gift list for an event, you can view all the gifts they have been assigned or pledged to in the event. Each gift is listed with its name, description, and status, indicating whether it's available or already pledged by someone else. Pledged gifts are highlighted in red, making them easily distinguishable from available gifts. You can browse through the list to see what gifts your friend has chosen and check if they still need more. The gift list shows detailed information, including the category, gift description, and whether it's still available for others to pledge. This allows you to get a better understanding of what your friend is contributing to the event.

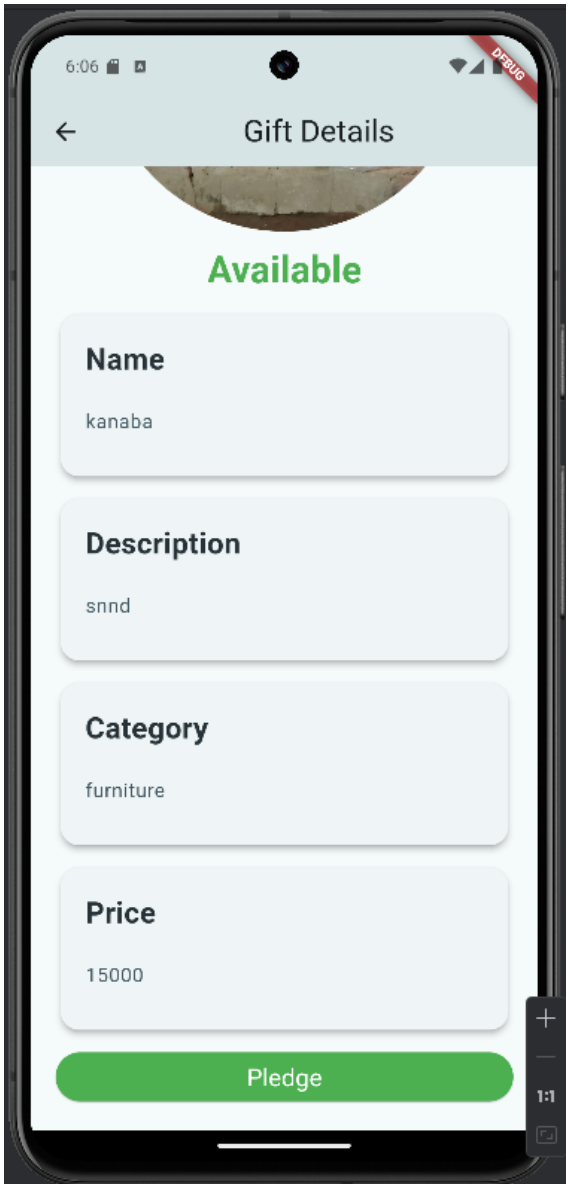
Owner Gift Details Page:

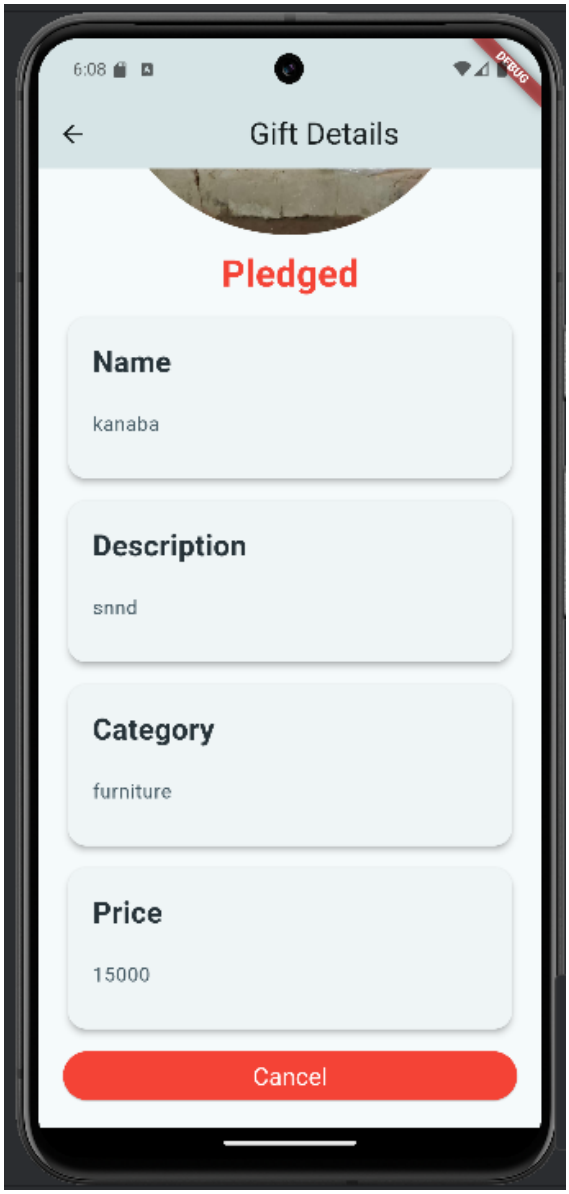




When the user, as the owner, accesses their own gift details, they can view and edit the information related to the gift they are offering for the event. The page displays the gift's name, description, category, price, and an image. As the owner, they have the ability to update the gift details, such as changing the description or price, as long as the gift's status is still "Available." If the gift has been pledged, they cannot edit certain fields. The "Pledge" and "Cancel" buttons are not visible to the owner. Additionally, if the event associated with the gift is marked as completed, the gift details cannot be accessed, ensuring that users can only interact with gifts from active events.

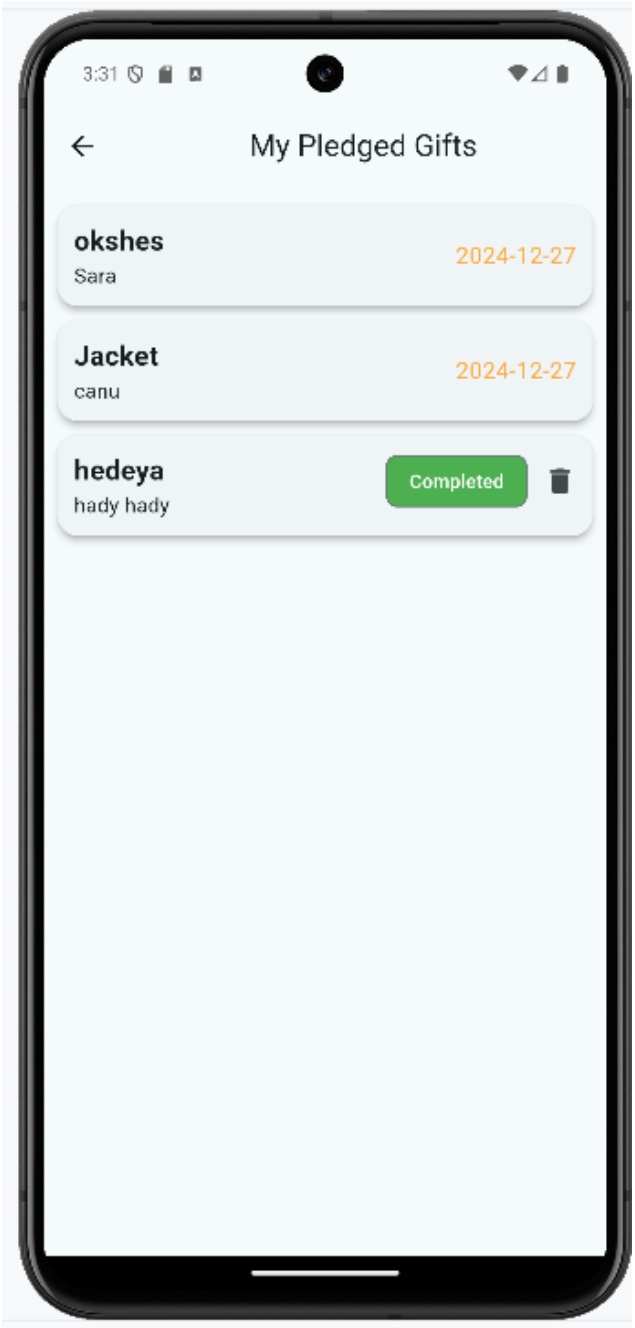
Non-Owner Gift Details Page:





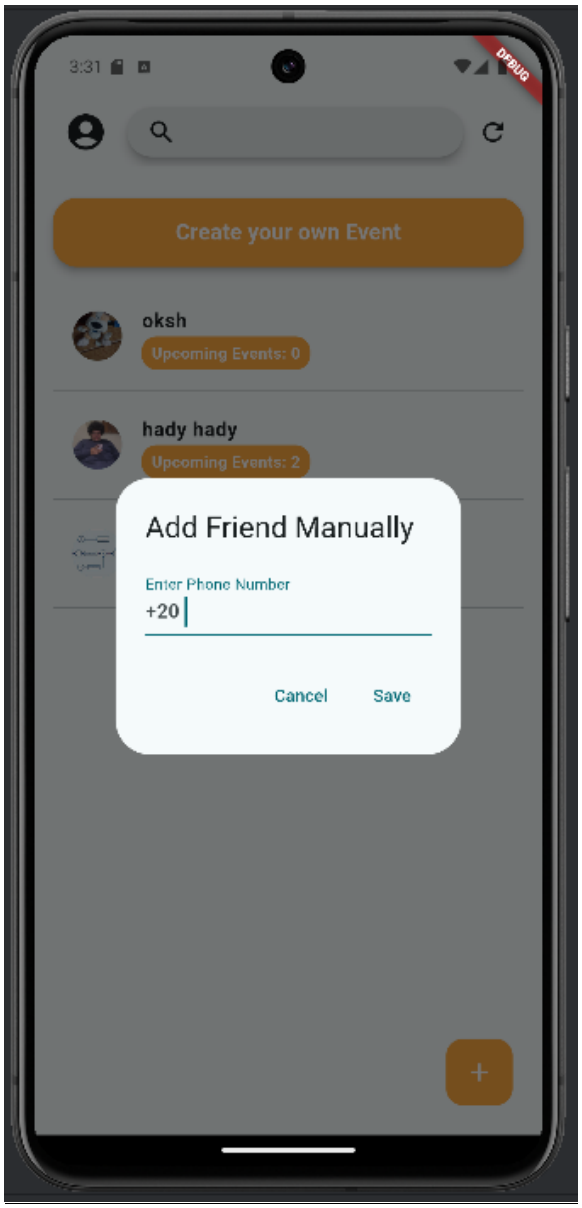
When accessing a gift that belongs to a friend, the user can view the gift's details, including its name, description, category, price, and an image. If the gift is available, the user has the option to pledge it, and the status will be updated to "Pledged." If the user has already pledged the gift, the status will reflect that, and they will have the option to cancel the pledge. However, if the user is the owner of the gift, they will not be able to interact with the pledge buttons. If the gift is part of a completed event, the user will not be able to access or modify the gift's details, ensuring that interactions are limited to gifts within active events.

My Pledged Gifts Page:



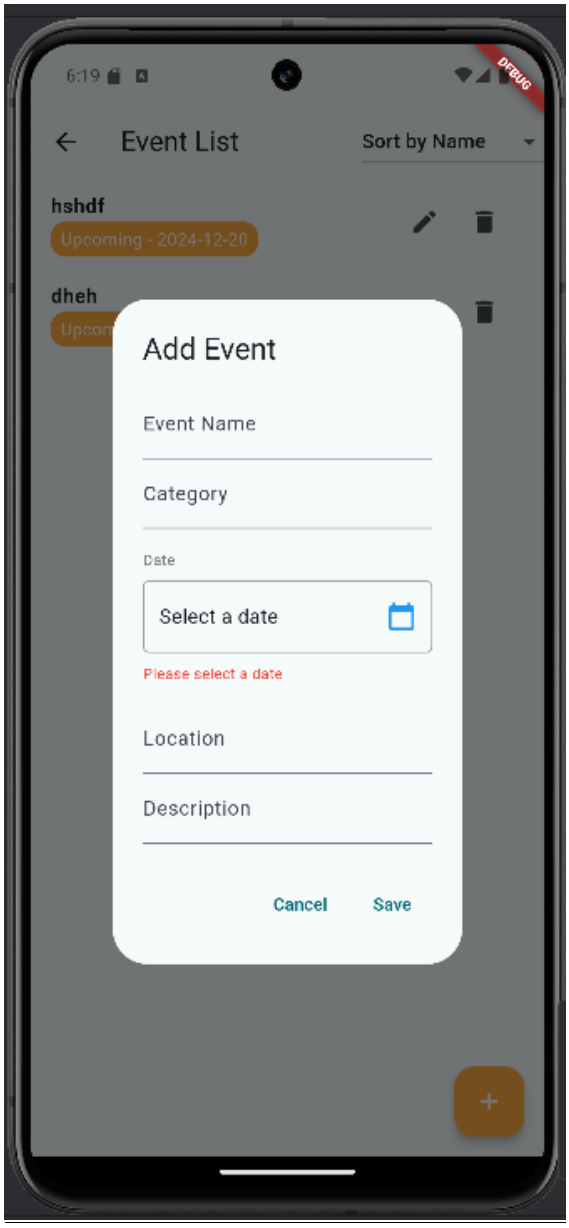
The MyPledgedGifts screen displays a list of gifts that the user has pledged in various events. Each gift is presented using a PledgedGiftCard, showing the gift's name, the name of the owner, and the due date for the gift (formatted in the YYYY-MM-DD format). The user can easily view all their pledged gifts, with each entry clearly distinguished and separated by space for better readability. This list helps the user keep track of their commitments in different events, ensuring they are aware of the due dates for the pledged gifts. If gift is related to a completed event, the user can delete it from his pledged gifts.

Add Friend Manually:

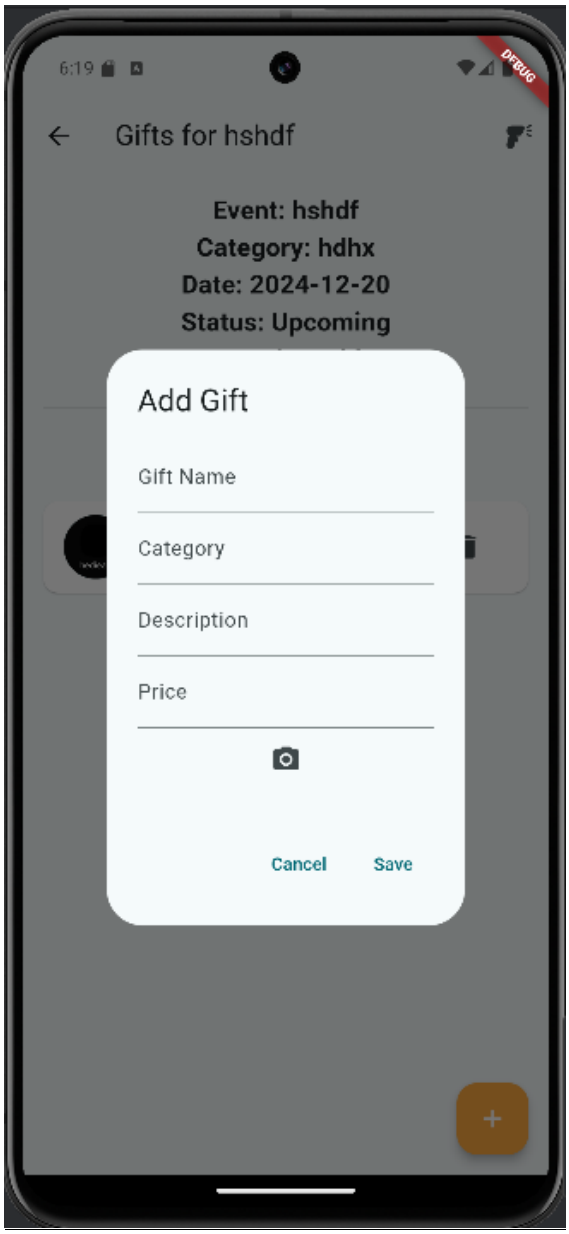


Add Friend from contacts:

Add New Event:



Add New Gift:


6:19
DEBUG
Gifts for hshdf
Event: hshdf
Category: hdhx
Date: 2024-12-20
Status: Upcoming
Add Gift
Gift Name
Category
Description
Price
Cancel
Save

Edit Profile Fields:

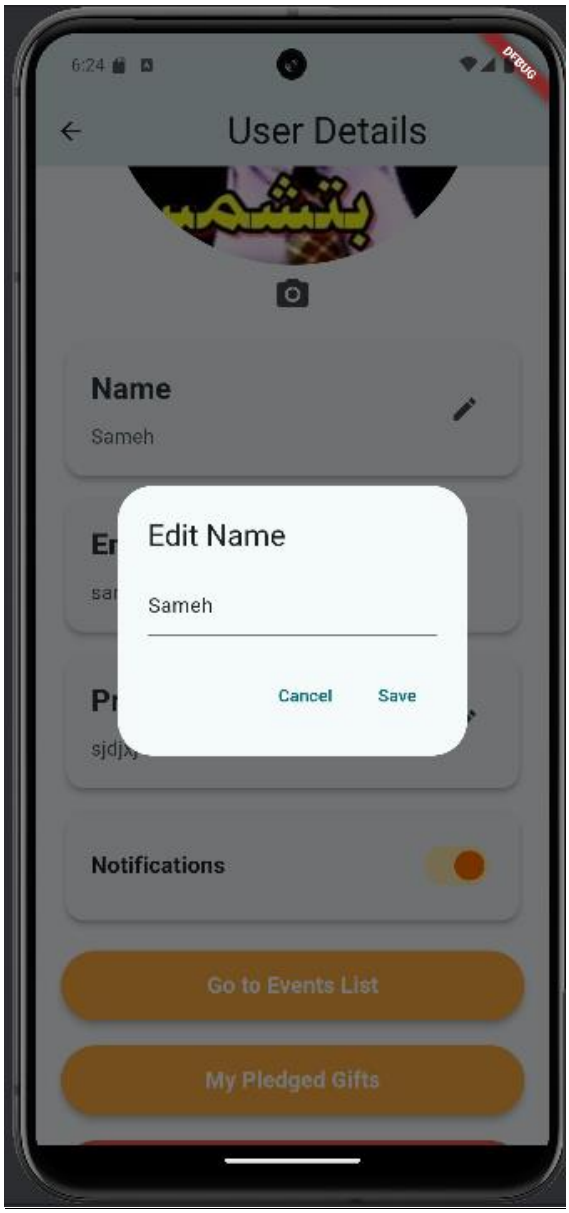
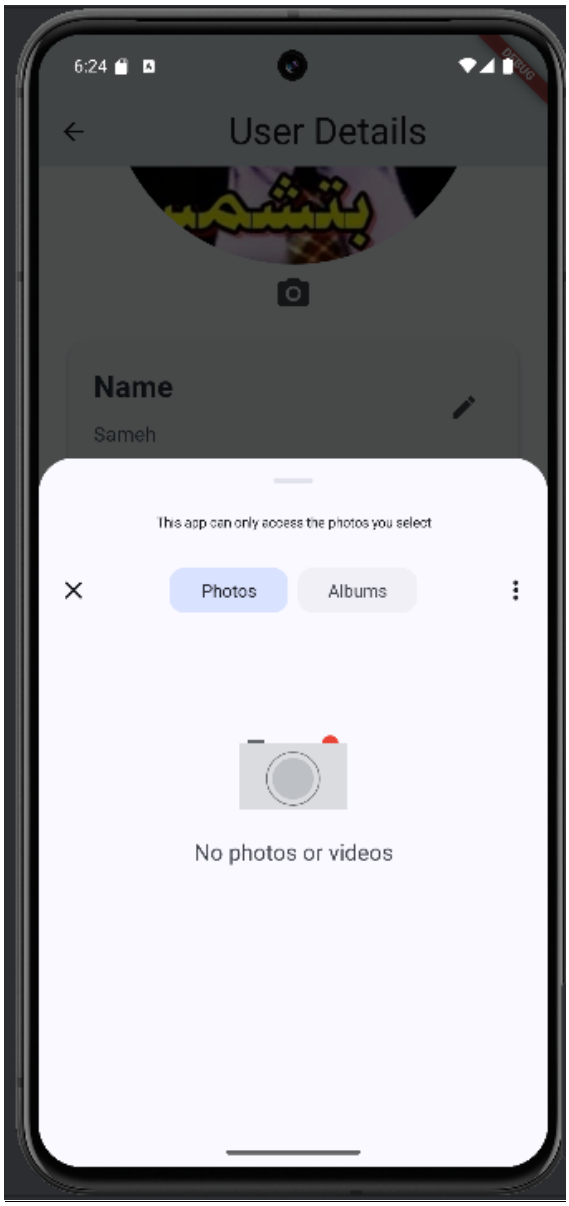
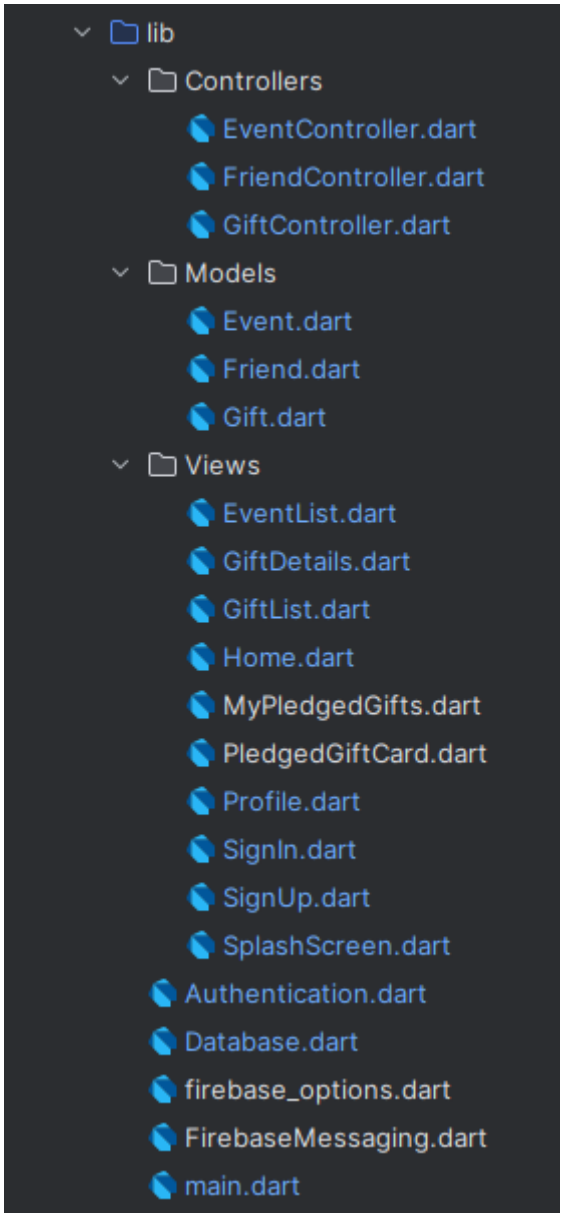


Image Picker:



Code Description



This Flutter project demonstrates a well-structured approach to app development. The Controllers folder likely handles application logic and data flow, while the Models folder defines data structures like Event, Friend, and Gift. The Views folder houses user interface components such as EventList, FriendCard, and GiftDetails. Firebase integration is evident through firebase_options.dart and FirebaseMessaging.dart. The main.dart file serves as the application's entry point. This clear separation of concerns promotes a modular and maintainable codebase.

Models

Friend.dart:

Description:

The Friend class is a central part of a social application that represents a user and their interactions within the app. It captures essential attributes such as a user's id, name, and optional details like email, preferences, PhoneNumber, and image. Additionally, it includes statistics like the number of notifications the user has and the count of upev (upcoming events). The class also organizes data into lists, including friendlist (a collection of Friend objects), eventlist (a list of Event objects), and PledgedGifts (a list of Gift objects representing the gifts the user has pledged).

The class offers a variety of methods for user management. The getUserById retrieves a user's details and populates their friends list. The getUserObject method provides a complete representation of a user, including their friends, events, and pledged gifts. Additionally, the getUserByPhoneNumber method checks whether a user exists in the local database by their phone number.

For managing friendships, the getFriends method retrieves all the friends of a specific user based on their user ID. The registerFriend method enables the creation of a new friendship by searching for a user in Firebase Realtime Database using their phone number and storing the relationship locally.

The class also facilitates event and gift management. The getEvents method fetches all events created by the user and populates each event with its associated gifts. Similarly, the getPledgedGiftsWithEventDetails method retrieves all gifts pledged by a user, including additional details such as the event date and the event owner's name.

Furthermore, the updateUser method allows updating specific fields in a user's database record, enabling seamless modification of user information.

This class integrates seamlessly with both Firebase Realtime Database and a local SQL database, offering robust support for CRUD operations and efficient data retrieval. Its comprehensive design makes it a foundational component of the application, supporting user interactions, relationships, and event management.

Event.dart:

Description:

The Event class is a comprehensive model that represents events within an application. It captures various details about an event, including its unique id, name, category, status, date, location, and description. The class also associates a list of Gift objects (giftlist) with each event and keeps track of the userId of the event's creator. These attributes make the class suitable for applications requiring event organization and management.

The Event class provides functionality to interact with a database, facilitating operations like creation, retrieval, updating, and deletion of event records. The fromMap factory constructor converts a database record (represented as a map) into an Event object, while the toMap method transforms an Event object into a map format suitable for database insertion or updating. The toMap method also determines the event's status as either "Upcoming" or "Completed" based on its date.

For adding new events, the insertEvent method constructs an SQL query to insert event details into the database. The updateEvent method allows modifying an existing event, ensuring that changes are reflected in the database. It checks if the update was successful and provides feedback accordingly.

The class also supports retrieving events from the database. The getEventById method fetches an event by its ID, converts the retrieved data into an Event object, and populates its giftlist with related Gift objects. This method ensures that events are retrieved with all their associated details.

For deleting events, the deleteEvent method constructs an SQL query to remove an event based on its ID. It checks if the deletion was successful and provides appropriate feedback.

The Event class relies on the Databaseclass for executing database queries, ensuring seamless integration between the class and the database. By combining robust attribute management with a set of database operations, this class serves as a foundational component for managing events in the application, enabling efficient CRUD operations and interaction with related Gift objects.

Gift.dart

Description:

The Gift class represents the gifts associated with events in the application. It is designed to store all the essential details about a gift, including attributes like id, eventId, name, category, status, description, price, image, PledgerID, Ownername, DueDate, and UserID. These attributes allow the application to maintain comprehensive information about each gift, such as its availability, its association with a specific event, and its ownership details.

The class provides a factory constructor fromMap to create a Gift object from a map (typically representing a database row) and a toMap method to convert the Gift object into a format suitable for database storage. This functionality ensures smooth integration between the application and the database, enabling easy data storage and retrieval.

To manage gifts, the class offers several static methods for performing Create, Read, Update, and Delete (CRUD) operations. For retrieval, the getGiftList method fetches all gifts linked to a specific event using its eventId, while the getGiftById method retrieves a single gift by its unique identifier. The addGift method allows adding new gifts to the database, ensuring attributes like status are correctly set as "Available" or "Pledged." Updating a gift's details is handled by the updateGift method, which modifies its attributes in the database. Additionally, the DeleteGift method removes a gift from the database using its id, with a clear indication of whether the deletion was successful.

A notable feature of the class is its support for barcode-based gift creation and validation. The CreateGiftByBarcode method retrieves gift details from the BarcodeGifts database table based on a provided barcode and creates a new Gift object associated with a specific event and user. Meanwhile, the CheckBarcode method verifies whether a given barcode exists in the database, enabling quick validation of barcode data.

The status attribute plays a dynamic role in the class. While it is represented as "Available" or "Pledged" in the application for readability, it is stored as 0 or 1 in the database for simplicity. This design ensures seamless communication between the database and the application.

Overall, the Gift class is a crucial component of the event management system, offering robust features for managing gift data efficiently. Its barcode integration and comprehensive CRUD operations ensure smooth data handling and enhance the functionality of the application.

Controllers

FriendController

Description:

This code defines a FriendController class in Flutter that interacts with Firebase, handles user authentication, manages user profiles, and facilitates friend-related actions. The controller includes various methods for managing navigation, user profiles, events, and friendships.

The methods in the controller handle different tasks. For example, ProfileIconTap navigates to the user's profile page, and CreateEventOnTap opens the event creation page for the user. These navigations use PageRouteBuilder with custom animations, such as slide transitions. The FriendCardOnTap method navigates to the event list page with a friend.

There are methods for adding friends either by selecting a contact from the device's contacts using the flutter_native_contact_picker package (AddFriendFromContacts), or manually entering a phone number (AddFriendManual). These methods check if the user is trying to add themselves or if a friendship already exists, showing relevant error messages through snack bars. If the friend is added successfully, the controller updates the user's friend list and syncs the data with Firebase.

The generateUniqueFriendId method generates a unique friend ID by checking existing IDs in the Firebase Realtime Database and incrementing until it finds an unused ID.

The controller also handles profile-related actions, such as picking a profile image using the image_picker package (pickImage), updating the user's notification preferences (NotificationSwitch), and handling user sign-out (SignOut). When a user signs out, the app removes the FCM token, deletes local data, and cancels real-time database listeners.

For authentication, the controller includes methods for signing in (SubmitSignInForm) and signing up (SubmitSignUpForm). When signing up, the controller validates inputs such as phone number and email, checks if they are already registered in Firebase, and if not, proceeds with account creation. After successful sign-up or sign-in, the user is redirected to the home page, and the necessary data is synced with Firebase.

The AlreadyAuthenticatedUser method ensures that real-time listeners are set up when the user is already authenticated, and refreshFriendsList refreshes the user's friend list by retrieving the updated data from Firebase.

There are also navigation methods like NavigatetoSignUp and NavigatetoSignIn to manage transitions between sign-up and sign-in screens. Additionally, methods like PopEditCard handle UI-related tasks, such as closing edit dialogs.

Overall, the FriendController class manages the flow of user interactions within the app, including profile management, friend management, authentication, and navigation, all while ensuring smooth integration with Firebase for real-time updates and data synchronization.

EventController

Description:

The EventController class is responsible for managing event-related operations in the application, including fetching, creating, updating, deleting, and synchronizing events to Firebase. It acts as the mediator between the database, Firebase, and the user interface, ensuring smooth event handling and data consistency.

The class includes methods like fetchEventsFromLocalDb, loadEvents, and getEvents for retrieving event data. These methods differentiate between events based on whether the current user is the event owner or a participant. If the user is the owner, events associated with their ID are fetched; otherwise, events tied to a friend's ID are retrieved. This separation ensures that the correct data is displayed based on user access levels.

The PickEventDate method allows users to select an event date using Flutter's showDatePicker widget. The method ensures that users can only pick dates within a specific range (from the year 2000 to 2100). The SaveEvent method performs input validation, such as ensuring the date is not empty. If the date is missing, a snack bar message informs the user. When saving, the event's status is determined based on whether the selected date is in the past ("Completed") or future ("Upcoming"). If the event already exists, it is updated; otherwise, a new event is created with a unique ID generated using generateUniqueGiftId. After saving, the syncEventsTableToFirebase method ensures the event data is synchronized with Firebase.

To handle deletion, the deleteEvent method removes an event from the database and ensures all associated gifts are also deleted using DeleteEventGifts. Each gift deletion is then synchronized to Firebase to maintain data integrity. The SyncDeletiontoFirebase and SyncEventstoFirebase methods provide further mechanisms to ensure changes to events, such as deletions or updates, are consistently reflected in Firebase.

The GoToGiftList method navigates users to the gift list page associated with an event. It employs a custom sliding transition effect, creating a smooth navigation experience. For user feedback, the class includes a utility function showCustomSnackBar, which displays messages in a snack bar with customized styling, icons, and a floating behavior. This enhances user experience by providing real-time feedback, such as confirmation messages or error alerts.

Overall, the EventController class centralizes all event-related operations, providing robust and reusable methods to manage event data, validate user input, and ensure synchronization with Firebase. It is designed to handle edge cases gracefully, like missing dates or associated data, while maintaining a clean and responsive user experience.

GiftController

Description:

The GiftController class is responsible for managing the operations related to gifts within an event in the application. It facilitates adding, updating, deleting, and syncing gift data between the local database and Firebase. The class also manages the user interface interactions such as navigating to the gift details page and providing feedback through SnackBars.

One of the primary functions of the GiftController class is to handle the gift data. The fetchGiftsFromLocalDb method retrieves event details from the local database, including the list of gifts associated with the event. The OnGiftCardTap method is triggered when a user taps on a gift card; it navigates to the GiftDetails page and distinguishes between event owners and participants to show the appropriate UI. If the user is the owner, they get access to edit the gift, while non-owners can only view the gift and pledge it. This method also incorporates custom page transitions, providing a smooth sliding animation when navigating to the gift details screen.

The class is also responsible for syncing gift data to Firebase. The syncGiftsTableToFirebase method ensures that any changes made to the gift data locally are reflected in Firebase. Additionally, the OnSaveGiftPressed method handles saving or updating a gift, capturing data such as the gift's name, category, price, and image. If the gift is new, it generates a unique ID and adds it to both the event's gift list and Firebase. If it's an existing gift, the method updates the gift details accordingly. After saving or updating, a SnackBar is displayed to inform the user whether the operation was successful or not.

To generate unique gift IDs, the GiftController class includes the generateUniqueGiftId method, which checks existing IDs in Firebase to ensure no duplicates. This method iterates through the current gift IDs and increments the new ID until it finds one that hasn't been used. This guarantees that each gift has a unique identifier within the system.

The DeleteGift method enables the deletion of a gift, removing it from both the local database and Firebase. After a gift is deleted, the updated list of gifts is fetched to reflect the change. Similarly, the FetchGiftByID method retrieves a specific gift by its ID, allowing for detailed viewing or editing.

In addition to the core gift management features, the GiftController class supports barcode scanning for gift identification. The ScanBarcode method uses a barcode scanner to capture a gift's barcode and, if valid, creates a new gift based on the scanned data. This new gift is added to Firebase and the event's gift list is updated accordingly.

Finally, the showCustomSnackBar method is used throughout the class to display user-friendly notifications. Whether a gift is added, updated, or an error occurs, this method shows a SnackBar with a custom message and an appropriate background color (such as green for success and red for failure). This feedback mechanism enhances the user experience by providing immediate responses to user actions.

In conclusion, the GiftController class provides comprehensive functionality for managing gifts in the application. It interacts with both the local and cloud databases, ensuring that gift data is up-to-date and accessible across devices. By handling complex tasks like barcode scanning, unique ID generation, and custom notifications, it plays a crucial role in maintaining an efficient and user-friendly experience in the event gift management system

Views

EventListPage.dart

Description:

The EventListPage class is a stateful widget in Flutter that displays a list of events either created by the user (if they are the owner) or a friend (if the user is a participant). The page allows users to view, add, edit, and delete events, providing a comprehensive event management interface. The class takes two primary parameters: isOwner (a boolean indicating whether the current user is the event owner) and User (the current user or a friend of the user). If the friend parameter is provided, it represents the friend's information.

The state of EventListPage is managed using the _EventListPageState class. In this state class, a StreamSubscription listens to Firebase database changes for events related to the user or friend. The fetchEventsFromLocalDb function fetches the event data locally and updates the UI, while the getEvents method loads events either for the current user (if they are the owner) or for their friend. The event list can be sorted by various criteria like name, category, or status through a dropdown menu in the app bar.

The initState method sets up the Firebase listener and calls _loadEvents, which fetches and displays the list of events initially. If the event data changes in the Firebase database, the listener triggers a call to fetchEventsFromLocalDb to update the local list of events. The dispose method ensures that the Firebase listener is properly cancelled when the widget is no longer in use, preventing memory leaks.

The event list is rendered using a ListView.builder, where each event is represented by a ListTile that displays the event name, status, and date. For event owners, editing and deleting options are provided. If the event status is "Upcoming," the owner can edit the event using an IconButton. A delete button allows owners to remove events. Tapping on an event navigates the user to the GiftList page, where they can view or manage gifts associated with the event.

For adding or editing events, a dialog is displayed with a form containing fields for the event name, category, location, description, and date. The dialog is built using a Form widget, and each field is validated to ensure the user provides valid input. The Save button triggers the event saving process, either creating a new event or updating an existing one. The floatingActionButton is visible only if the user is the event owner, allowing them to add new events.

Event deletion is handled by the _deleteEvent method, which calls the EventController to perform the deletion operation. If the deletion is successful, a success message is shown via a SnackBar. The method also removes the deleted event from the displayed list, providing immediate feedback to the user.

To enhance the user experience, a custom SnackBar is shown for various actions, such as event deletion or errors. The showCustomSnackBar method is responsible for displaying these messages with appropriate background colors and icons to indicate success or failure.

Overall, the EventListPage provides a feature-rich and user-friendly interface for managing events, supporting both event owners and participants with robust event creation, editing, and deletion functionality. It integrates with Firebase to keep the event data synchronized across devices and provides visual feedback for a seamless user experience.

GiftDetails.dart

Description:

The GiftDetails class is a StatefulWidget that displays detailed information about a specific gift. It takes multiple parameters including isOwner, isPledged, gift, and User (a Friend object). These parameters define the context for the gift, whether the user is the owner, whether the gift has been pledged, and whether the current user is the one who pledged it. This widget manages the UI and business logic for updating and displaying gift details.

The state class _GiftDetailsState handles the logic of updating and displaying the gift details. It utilizes GiftController for interacting with the backend, such as fetching the gift from a local database and updating the gift's status. The widget also listens for real-time updates from Firebase via DatabaseReference, ensuring that any changes to the gift (such as a pledge status change) are reflected immediately in the UI. Additionally, the widget manages connectivity changes using the connectivity_plus package, updating the UI when the device's internet connection changes.

The main UI elements include a circular image representing the gift, editable fields for the gift's details (such as name, description, category, and price), and a button to pledge or cancel the pledge. The isOwner flag controls whether the fields are editable, while the isPledged flag determines whether the gift has been pledged or is available. When the user presses the pledge/cancel button, it triggers the _togglePledge() function, which updates the gift's status accordingly.

Finally, the GiftDetails page is designed to handle network connectivity issues. If the device goes offline, a message is displayed to inform the user. The app also listens for network reconnection events and provides feedback when the connection is restored. The logic ensures that users can interact with the app smoothly, even if there are temporary network issues.

GiftList.dart

Description:

The GiftListPage widget in this Flutter app is designed to manage and display the gifts associated with a particular event. It integrates Firebase Realtime Database to ensure that changes to the gift list are reflected in real-time across all users. The page is structured to display detailed information about the event, including the name, date, and a list of gifts. These gifts are fetched from the Firebase database, and the app displays them in a list format with their relevant details such as name, quantity, and whether they have been pledged by users.

The layout of the page is responsive, adapting to the screen orientation. When the screen is in portrait mode, the app shows a compact view, while in landscape mode, it adopts a split layout to utilize the extra screen space effectively. This dynamic design ensures an optimal viewing experience across different device sizes.

The gifts displayed on the page can be managed depending on the event's status. If the event is "Upcoming," the event owner can edit or add new gifts. The floating action button, which facilitates adding new gifts, is only visible to the event owner. The event owner can interact with the app through a custom dialog that allows for editing or creating new gift entries. When a gift is added or modified, the app updates the Firebase database to keep all users synced with the latest changes.

Gift items are also sortable by name or pledge status, providing an easy way for users to navigate through the list. Firebase real-time updates ensure that any changes, such as a gift being pledged by another user, are immediately reflected across all users' screens.

The app uses a snack bar to display error messages, offering feedback in case of issues, such as failed gift additions or database updates.

This integration with Firebase and the dynamic layout enhances user interaction by providing a seamless and interactive way to manage event gifts, ensuring that users, especially the event owner, can have full control over the gift list.

Home.dart

Description:

The Home class is a stateful widget that holds a list of friends and allows various user interactions such as searching for friends, refreshing the friends list, creating events, and adding friends manually or from contacts.
Upon initializing the Home widget, the list of friends for the current user (widget.User.friendlist) is assigned to the filteredfriends list. The app displays the user's profile icon, a search bar to filter friends, and a refresh button in the app bar. The refresh button triggers a function that fetches the latest friend data from the server and updates the UI with the new list.
In the body of the screen, a button allows the user to create their own event. Below that, a list of friends is displayed, with each friend shown as a ListTile containing a profile picture, name, and the number of upcoming events they have. Tapping on a friend's list item navigates to a detailed view of that friend.
The floating action button, represented as a popup menu, offers two options for adding a friend: "Add Friend Manually" and "Add Friend from Contacts." When the user selects one of these options, the appropriate action is taken. For manual addition, a dialog is displayed where the user can enter a phone number. The entered phone number is used to add a new friend to the list. The "Add Friend from Contacts" option allows the user to select a friend from their contacts using the FlutterNativeContactPicker package. After selecting a contact, the friends list is updated accordingly.
The filteredfriends list is dynamically updated based on user input in the search bar, filtering the displayed friends by their name. This allows the user to quickly find a friend within their list.
The setState method is called to ensure the UI is updated whenever the list of friends changes, whether from searching, adding a friend, or refreshing the list.

MyPledgedGifts.dart

Description:

The MyPledgedGifts widget is a stateful widget that displays a list of gifts that the user has pledged. The widget accepts a list of Gift objects, which it uses to populate the list view. Upon creation, the widget builds a screen with an app bar titled "My Pledged Gifts." The main body of the screen consists of a ListView that dynamically generates a list of PledgedGiftCard widgets, one for each Gift in the pledgedgifts list. Each PledgedGiftCard is customized with the name of the gift, the owner's name, and the due date, which is formatted to show only the date part (excluding the time). Between each PledgedGiftCard, a SizedBox is added for spacing. The PledgedGiftCard widget itself is used to display each individual gift's information, making it easy for the user to review their pledged gifts.

PledgedGiftCard.dart

Description:

The PledgedGiftCard widget is a stateless widget that presents information about a pledged gift in a card format. It takes three parameters: GName (the gift name), UName (the owner's name), and due (the due date). The card has rounded corners and a subtle elevation effect to make it visually distinct. Inside the card, there is a Row widget that arranges two columns of text. The first column displays the gift's name (GName) in a bold, larger font and the owner's name (UName) in a smaller font. The second column aligns the due date (due) to the right, styling it with a larger font size and a distinctive orange accent color. The layout ensures that the gift name and owner's name are left-aligned, while the due date is right-aligned, offering a balanced and visually appealing card design. This widget is designed to be reusable and can be used to display multiple pledged gifts within a list.

Profile.dart

Description:

The Profile widget is a stateful widget that displays the details of a user, represented by the Friend model. It allows for dynamic editing of the user's information and interacts with various controllers and methods to update or fetch data. The widget begins by initializing the user's data, including name, email, and preferences. A circular avatar is displayed, and users can update their profile picture by tapping a camera icon. Each user detail (name, email, preferences) is displayed within a card, and tapping an edit icon next to each field opens a dialog to edit the respective field. The widget also features a switch to enable or disable notifications, with changes reflected in the Friend model.

Users can navigate to other sections of the app, such as the event list or pledged gifts, through large, styled buttons. Additionally, there's a sign-out button that allows users to log out of the application. The layout is designed with cards and buttons, ensuring a visually appealing and user-friendly interface. Each card has rounded corners and a drop shadow for a modern, clean design. The switch for notifications uses custom colors, enhancing the visual experience. The overall structure focuses on clarity, functionality, and ease of use, ensuring users can efficiently manage their profile settings.

SignIn.dart

Description:

The SignIn widget in this code provides a sign-in interface for the "Hedieaty" app. It includes a form with fields for the user's email and password, both of which are validated before submission. The form uses a GlobalKey<FormState> to manage validation and trigger actions when the user interacts with it.

The user interface begins with a CircleAvatar at the top to display the app's logo. Below that, there are two fields: one for the email and one for the password. Both fields are styled with rounded borders and include validation logic. For the email field, the validator checks if the input is empty or if the email format is incorrect. For the password field, the validator ensures that the field is not empty.

Once the user fills out the form, the sign-in button becomes enabled. When clicked, the button triggers the onPressed function. If the form is valid, it checks whether the email and password match a user in the system by calling the IsUserFound method from the FriendController. If the user is found, the app proceeds to submit the sign-in request using the SubmitSignInForm method. If the credentials are incorrect, a custom snack bar message is displayed, indicating the issue.

The ElevatedButton shows a loading spinner while the sign-in process is ongoing, preventing multiple submissions. There is also a TextButton below the sign-in button that redirects users to the sign-up page if they don’t already have an account.

Overall, this sign-in screen ensures a smooth and user-friendly login process, with proper validation, error handling, and navigation to the sign-up page if needed. The app utilizes Firebase Authentication to verify the user's credentials before granting access.

Signup.dart

Description:

The Signup widget in this code defines the registration page for the "Hedieaty" app. It consists of a form where users input their name, email, preferences, phone number, and password. The user can also select an image from their gallery. The form uses a GlobalKey<FormState> to manage validation, ensuring all fields are filled correctly before submission.

At the top, the page features a CircleAvatar displaying the app's logo, followed by text fields for each user input. The name, email, preferences, phone number, and password fields are all styled with rounded borders, and each has validation to ensure that the input is non-empty and in the correct format. The email field also checks if the format is valid, and the phone number field validates that only digits are entered.

The image picker section allows the user to pick a profile picture. If no image is selected, a message saying "No image selected" appears in red; otherwise, it displays "Image selected" in green. The user can pick an image by tapping the camera icon, which opens the image picker.

Once the form is filled out, the user can tap the "Sign Up" button. Before submitting, the app checks if an image has been selected. If no image is selected, a snack bar with a warning message appears. If an image is selected, the SubmitSignUpForm method of the FriendController is called to submit the form. This method handles the user registration process, including saving the image and other data to a database.

Finally, a TextButton is provided that lets users navigate to the sign-in page if they already have an account. This setup ensures that the sign-up process is smooth, with proper validation, user feedback, and image handling.

SplashScreen.dart

Description:

The SplashScreen widget is designed to handle the initial app loading and user authentication flow. It checks whether a user is already authenticated using Firebase Authentication. If the user is authenticated, it fetches their corresponding Friend object and navigates to the Home screen. If no user is authenticated or the associated user data is not found, it navigates to the SignIn screen.

The widget uses two nested FutureBuilder instances. The first checks the Firebase Authentication state to determine if a user is logged in. If a user is authenticated, the second FutureBuilder fetches the user's data from the database as a Friend object. While data is being fetched, it displays a CircularProgressIndicator to indicate loading. The FriendController handles user-specific operations, such as resetting and setting up real-time listeners for authenticated users.

This implementation ensures a seamless and efficient transition between the splash screen and the appropriate application screen (either Home or SignIn) based on the user's authentication state and data availability.

Database.dart

Description:

Databaseclass is responsible for managing a local SQLite database and synchronizing it with Firebase Realtime Database. This class facilitates offline data storage while ensuring that the app receives real-time updates from Firebase.

The SQLite database comprises several tables, each serving a specific purpose. The Users table stores user-related data such as names, emails, phone numbers, and preferences. The Events table manages event details, including the name, date, location, and user ID to identify the event's creator. The Gifts table keeps track of gift-related information like name, category, price, image, and associated event or user IDs. Additionally, the BarcodeGifts table is dedicated to barcode-scanned gift data, while the Friends table establishes relationships between users by linking user IDs with their friend IDs. The database schema is defined using CREATE TABLE SQL commands, and the initialize method sets up the database, including pre-populating it with sample data for testing purposes.

The class offers helper methods to perform CRUD operations on the database. These include reading data through SELECT queries, inserting new records, updating existing ones, and deleting either specific records or the entire database. This functionality ensures that the local database remains up-to-date and manageable.

To enable real-time synchronization, the class integrates with Firebase Realtime Database. It begins by preloading data from Firebase, particularly focusing on the Friends node to retrieve and cache connections for the current user. Firebase listeners are then used to track changes across the Users, Events, Gifts, and Friends nodes. These listeners respond to events such as additions, updates, and deletions, applying corresponding operations to the local SQLite database to keep it in sync with Firebase. Synchronization is optimized by filtering data based on user and friend IDs, ensuring that only relevant updates are applied locally.

This class provides several features to enhance the user experience. It ensures that events, gifts, and other data associated with the user's friends are also synchronized, fostering a collaborative environment. Predefined events like "Birthday Party" and "Tech Conference" are included for testing purposes. The integration of SQLite enables offline access to data, allowing the app to function seamlessly even without an internet connection. A dedicated table for barcode-scanned gifts adds an extra layer of functionality. Multiple Firebase listeners are actively managed to maintain synchronization, ensuring that changes made either locally or on Firebase are consistently reflected in both environments.

Authentication.dart

Description:

AuthService class, which serves as the authentication and user management module for an application. This class integrates Firebase Authentication for user sign-in, sign-up, and session management while utilizing Firebase Realtime Database to store and retrieve user-specific information. The class simplifies the handling of authentication and user data synchronization.

The AuthService includes a static method, getUserIdFromRecord, that retrieves a user's ID from the Firebase Realtime Database based on their unique identifier (userId). It uses the once method to fetch data from the Users node and verifies if the data exists. If the user is found, their unique database ID is returned; otherwise, the method logs an error or a not-found message.

For user registration, the signUpWithEmailPassword method allows new users to sign up using their email and password. Upon successful account creation, a unique identifier (uid) is generated for the user by Firebase. The uid is converted into a hashed integer value to serve as a consistent user ID across the system. Additionally, a Friend object is instantiated to encapsulate user information, including name, email, preferences, phone number, password, and image. This object is then converted into a map and stored in the Firebase Realtime Database under the Users node, ensuring the user's data is readily accessible.

The signInWithEmailAndPassword method handles user login by verifying their credentials through Firebase Authentication. Upon successful sign-in, the method retrieves the user's hashed ID and searches the Firebase Realtime Database for their corresponding record. If the user exists in the database, their ID is returned; otherwise, an error or not-found message is logged. This ensures that authentication and user data retrieval are seamlessly integrated.

The class also includes additional utility methods for user session management. The getCurrentUser method retrieves the currently authenticated user, enabling the app to access the user's information and session state. The signOut method allows users to log out of their accounts, ensuring proper session termination.

Overall, this AuthService class provides a robust foundation for user authentication and data management, leveraging Firebase's capabilities to streamline both backend operations and user experience. It ensures data integrity and synchronization between Firebase Authentication and Firebase Realtime Database, enabling a secure and efficient user management system.

FirebaseMessaging.dart

Description:

FirebaseMessagingService class, which is responsible for managing Firebase Cloud Messaging (FCM) and local notifications within a Flutter application. This class integrates FCM to enable push notifications and handles both foreground and background notification scenarios. Additionally, it saves and removes FCM tokens in the Firebase Realtime Database to associate user-specific notifications.

The initNotifications method initializes the FCM service and sets up notification handling for a user identified by their userId. It uses the FlutterLocalNotificationsPlugin to configure the local notification settings, specifying the app's logo as the notification icon. The method retrieves the user's unique FCM token and stores it in the Firebase Realtime Database under the corresponding user's ID and it is called on sign in when the user is authenticated and redirected to Home page. This enables targeted notification delivery to specific users.

Foreground message handling is achieved through FirebaseMessaging.onMessage.listen, which listens for messages received while the app is active. When a message is received, the app checks the user's notification preferences stored in Firebase Realtime Database. If notifications are enabled for the user, the _showNotification method is invoked to display the notification locally. The notification includes a title, body, and additional payload data.

The class also handles notifications received when the app is in the background or terminated. The FirebaseMessaging.onMessageOpenedApp listener ensures that notifications open the app when tapped. Similarly, the FirebaseMessaging.onBackgroundMessage static method handles background notifications by displaying them as local notifications using the _backgroundMessageHandler. The notification's appearance is customized using AndroidNotificationDetails and NotificationDetails to ensure consistency across devices.

The _showNotification method leverages the FlutterLocalNotificationsPlugin to display local notifications. It uses customizable settings, such as importance and priority, to ensure that notifications are prominently shown. The method also allows the inclusion of additional data in the notification payload, enabling rich user interactions.

To manage user-specific FCM tokens, the removeFCMToken method removes the token associated with a user from the Firebase Realtime Database. This ensures that notifications are not sent to users who no longer require them or have logged out.

Overall, the FirebaseMessagingService class provides a comprehensive solution for integrating push notifications into a Flutter application. It combines the capabilities of Firebase Messaging and Flutter Local Notifications to deliver a seamless notification experience, whether the app is active, in the background, or terminated.

Main.dart

Code:

```
import 'dart:ui';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:firebase_database/firebase_database.dart';
import 'package:firebase_messaging/firebase_messaging.dart';
import 'package:flutter/material.dart';
import 'package:hedieatymobileapplication/FirebaseMessaging.dart';
import 'package:hedieatymobileapplication/Views/EventList.dart';
import 'package:hedieatymobileapplication/Views/MyPledgedGifts.dart';
import 'package:hedieatymobileapplication/Views/SignIn.dart';
import 'package:hedieatymobileapplication/Views/SplashScreen.dart';
import 'package:image_picker/image_picker.dart';
import 'dart:io';
import 'Models/Authentication.dart';
import 'Views/GiftDetails.dart';
import 'Views/Home.dart';
import 'Views/Profile.dart';
import 'Views/GiftList.dart';
import 'Models/Gift.dart';
import 'Models/Event.dart';
import 'package:firebase_core/firebase_core.dart';
import 'firebase_options.dart';
import 'Models/Database.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'Models/Friend.dart';
void main() async{
  //Firebase init
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp(
    options: DefaultFirebaseOptions.currentPlatform,
  );


  //on app restart sync the cached data only
  FirebaseDatabase.instance.setPersistenceEnabled(true);
  FirebaseDatabase.instance.setPersistenceCacheSizeBytes(10000000);


  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.white),
        useMaterial3: true,
      ),
      home: SplashScreen(),
    );
  }
}
```

Notification Server(Node JS Script)

Description:

This Node.js script integrates Firebase Admin SDK to enable notifications for a gifting application. It uses the Firebase Realtime Database to listen for changes in the Gifts node and sends notifications when a gift's status is updated to "pledged."

First, the Firebase Admin SDK is initialized using a service account JSON file and the application's database URL. This setup provides access to Firebase's messaging and database services with admin privileges.

The core functionality includes the sendNotification function, which sends a push notification to a specific user using their Firebase Cloud Messaging (FCM) token. The notification contains a title and body, which are dynamically constructed based on the context (e.g., when a gift is pledged).

The listenForPledgedGifts function monitors the Gifts node for any updates (via the child_changed event). When a gift's Status field changes to 1 (indicating it has been pledged), the script:

1. Retrieves the EventID and GiftID from the gift data.
2. Looks up the corresponding event in the Events node to find the UserID associated with it.
3. Fetches the user's FCM token from the Users node using the UserID.
4. Sends a notification to the user, informing them that their gift has been pledged, using the sendNotification function.

Real-Time Notifications Even When App is Shutdown

The script ensures that notifications are sent to the user even if the app is shutdown, as long as the user is authenticated. Firebase Cloud Messaging (FCM) maintains the connection for push notifications in the background, so the user will receive the notification on their screen when the gift status is updated, regardless of whether the app is open or closed. This guarantees that users stay informed in real-time without requiring the app to be actively running.

This implementation ensures real-time notifications for users, enhancing the user experience by immediately notifying them about updates to their gifts. Additionally, it is designed with robust error handling to log issues like missing FCM tokens or invalid data.

The script initializes the listenForPledgedGifts function, making the application continuously monitor database changes and send notifications as needed. This is ideal for keeping users informed in real-time while maintaining clean and modular code.

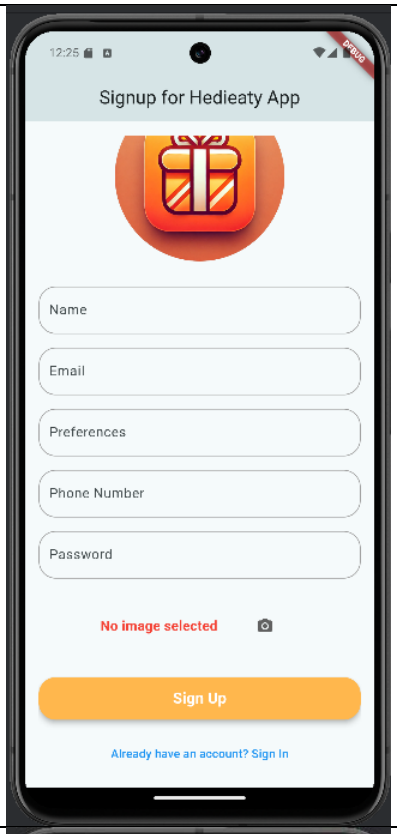
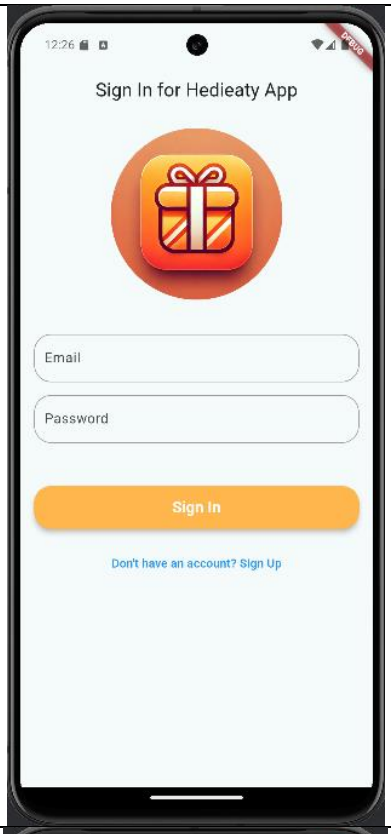
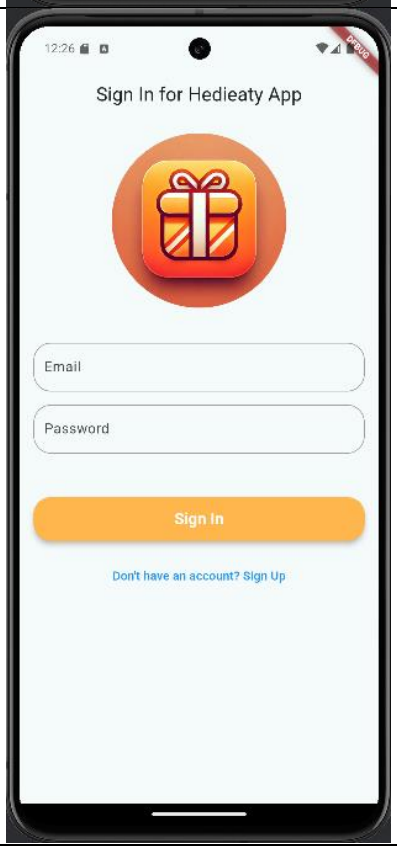
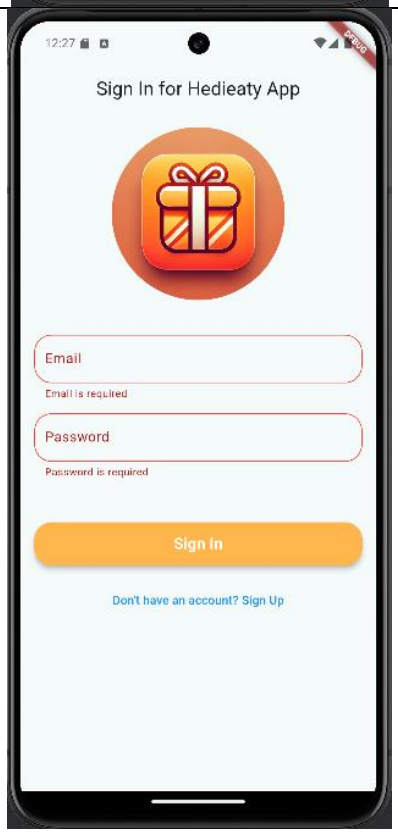
Code:

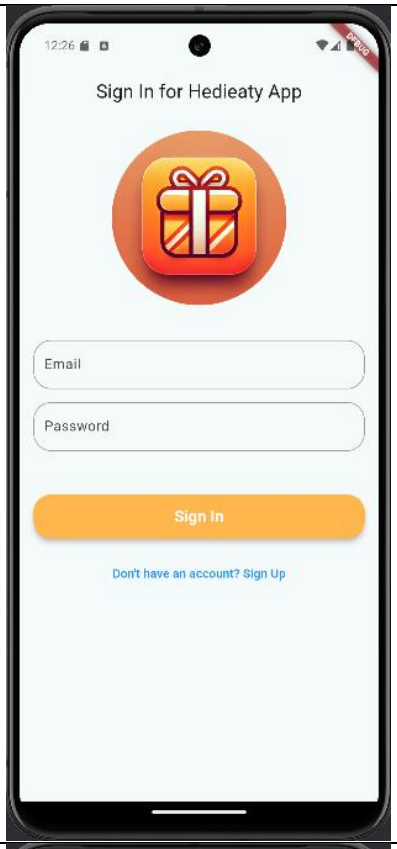
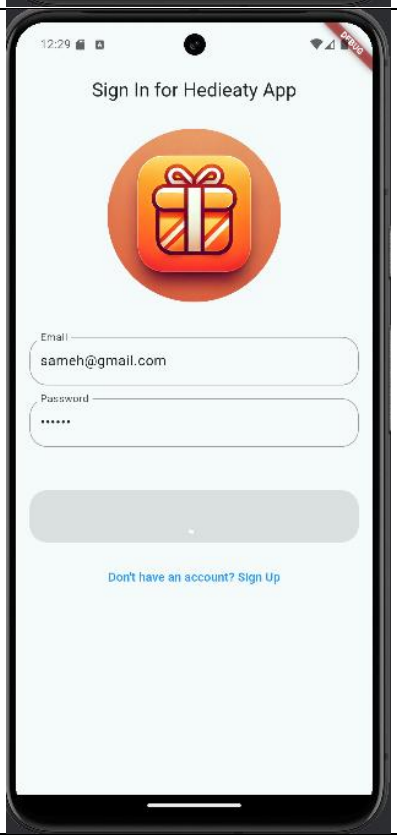
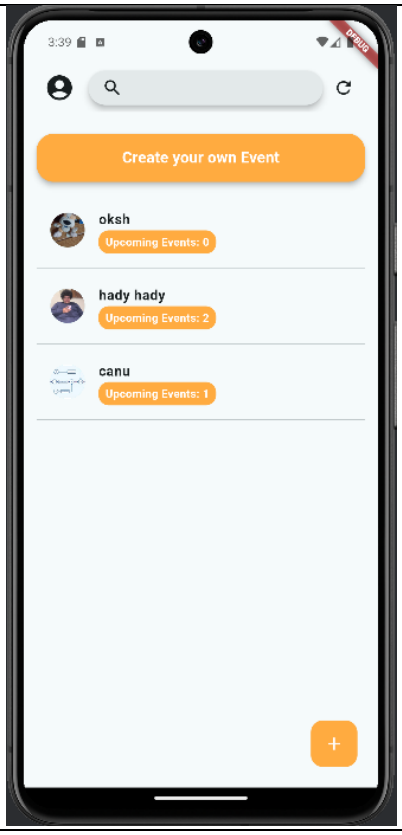
```
var admin = require("firebase-admin");

var serviceAccount = require("./hedieatyapp-firebase-adminsdk-bb0ur-
f241c08255.json");
```

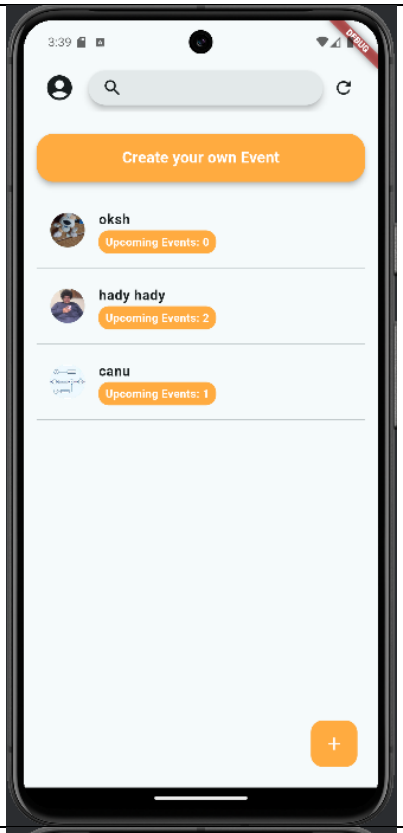
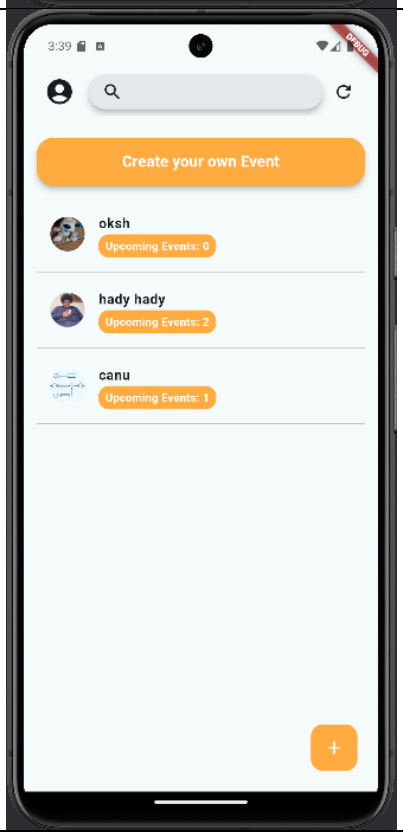
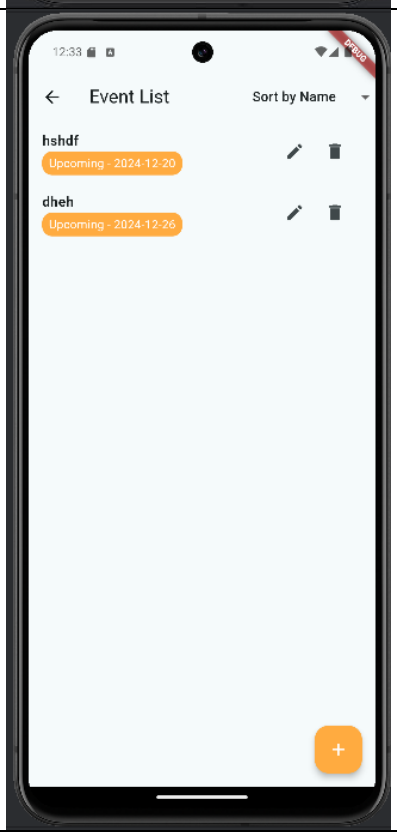
```
admin.initializeApp({
credential: admin.credential.cert(serviceAccount),
databaseURL: "https://hedieatyapp-default-rtdb.firebaseio.com"
});

// Function to send a notification to a user
const sendNotification = async (userToken, title, body) => {
const message = {
token: userToken,
notification: {
title: title,
body: body,
},
};

try {
// Send the message using Firebase Admin SDK
const response = await admin.messaging().send(message);
console.log("Successfully sent message:", response);
} catch (error) {
console.log("Error sending message:", error);
}
};

// Function to listen for changes in the 'Gifts' node and send notifications when a
gift is pledged
const listenForPledgedGifts = async () => {
const giftsRef = admin.database().ref("Gifts");

// Listen for updates to the 'Gifts' node (this checks when a gift is pledged)
giftsRef.on("child_changed", async (snapshot) => {
const giftData = snapshot.val();

// Check if the 'Status' is set to 1 (pledged)
if (giftData && giftData.Status === 1) {
const eventId = giftData.EventID; // Get the EventID from the gift data
const giftId = giftData.ID; // Get the Gift ID

// Fetch the Event data to get the associated UserID
const eventRef = admin.database().ref(`Events/${eventId}`);
const eventSnapshot = await eventRef.once("value");

if (eventSnapshot.exists()) {
const eventData = eventSnapshot.val();
const userId = eventData.UserID; // Get the UserID associated with the Event

// Fetch the User's FCM token from the 'Users' node
const userRef = admin.database().ref(`Users/${userId}/fcmToken`);
```

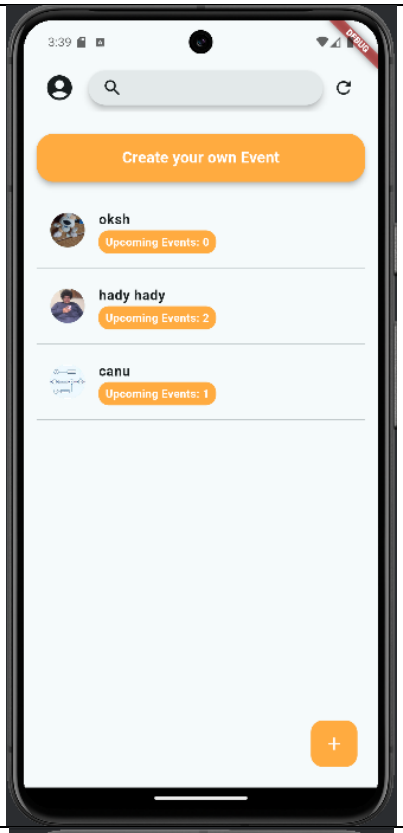
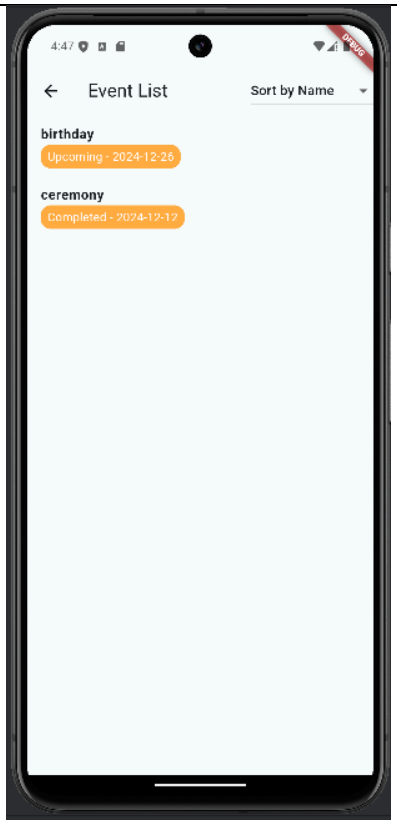
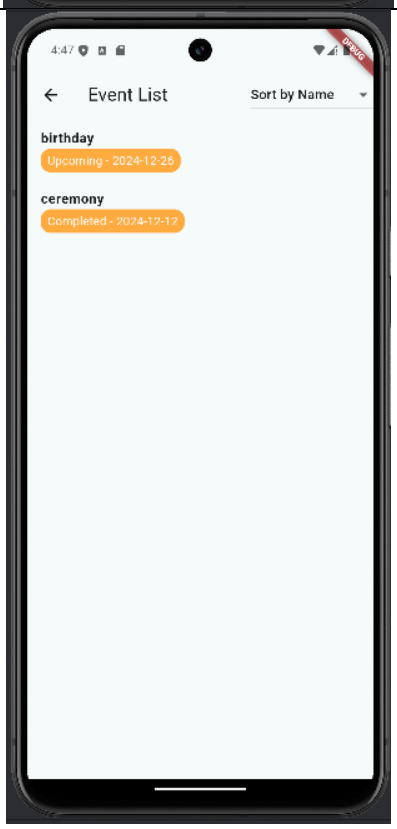
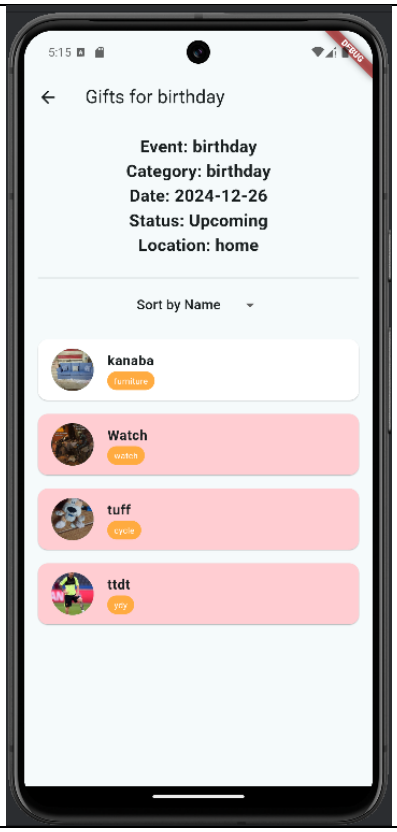
```
const userSnapshot = await userRef.once("value");

if (userSnapshot.exists()) {
const userToken = userSnapshot.val(); // The user's FCM token

// Send a notification to the user
const title = "Your Gift Was Pledged!";
const body = `Your gift ${giftData.Name} has been pledged successfully!`;

// Call the sendNotification function
await sendNotification(userToken, title, body);
} else {
console.log(`FCM token not found for user: ${userId}`);
}
} else {
console.log("Event not found.");
}
}
});

console.log("Listening for pledged gifts...");
};

// Initialize the function to start listening for pledged gifts
listenForPledgedGifts();
```

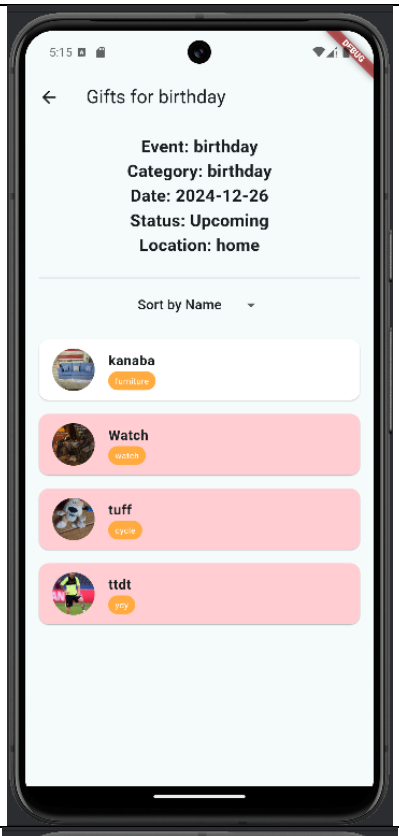
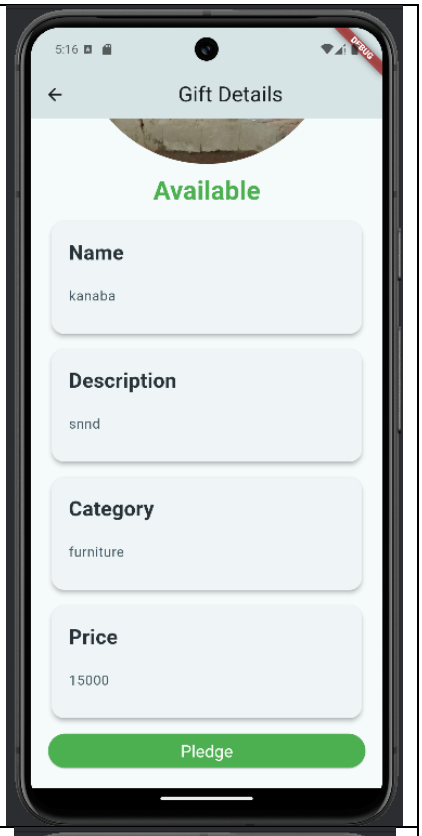
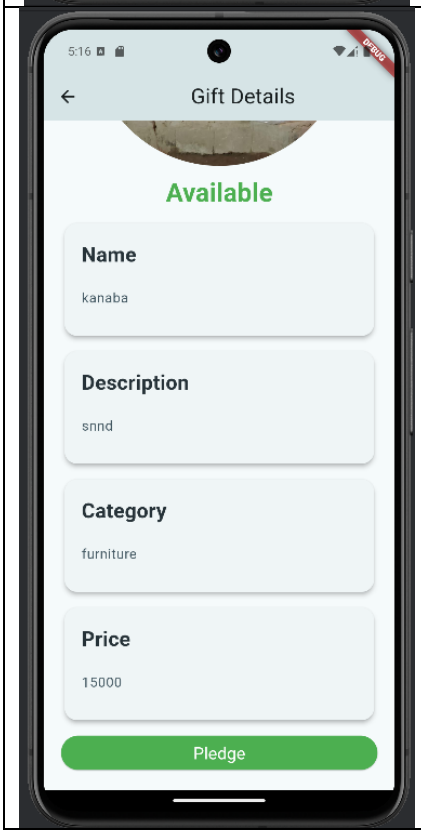
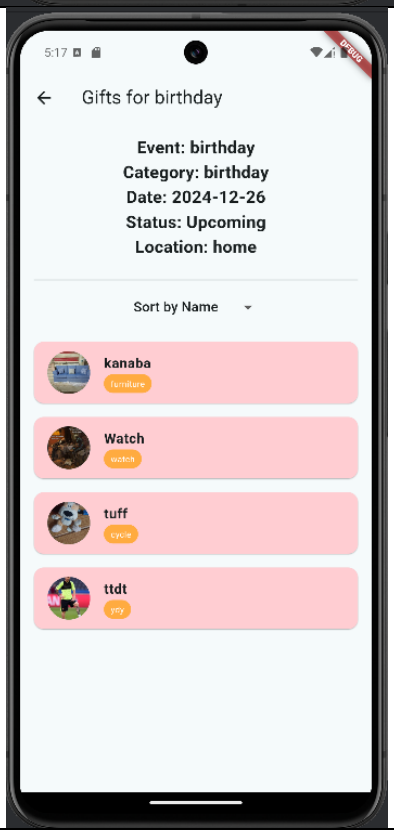
Navigation Scenarios

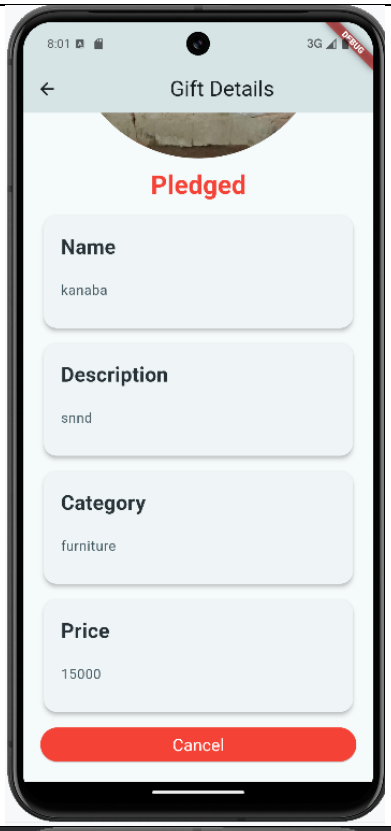
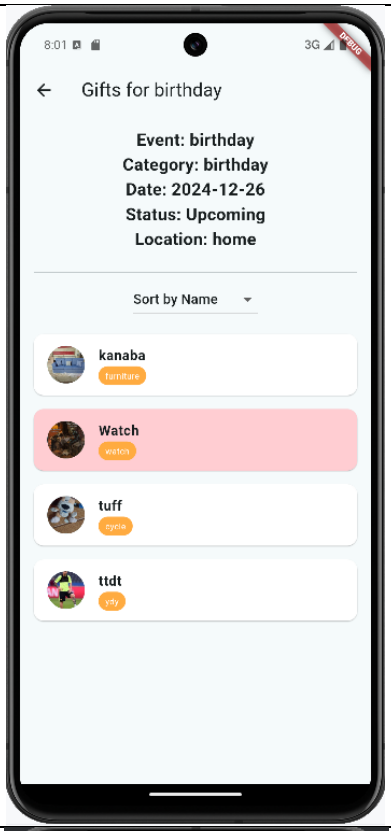
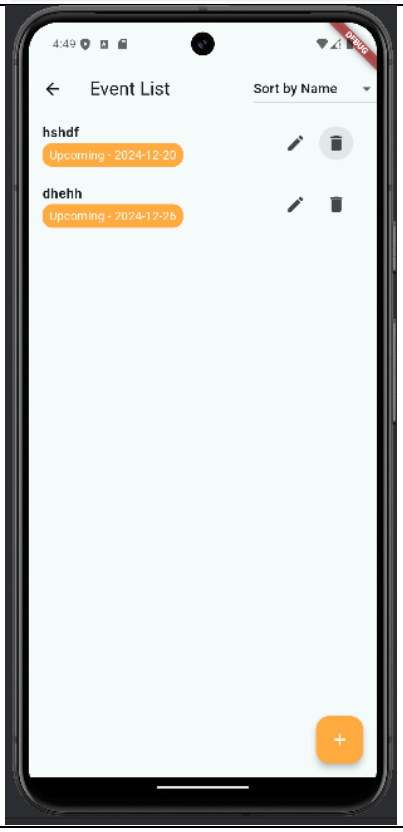
Screen	Action	Result
	Already have an Account? Sign In	
	Press Sign In(With empty fields)	

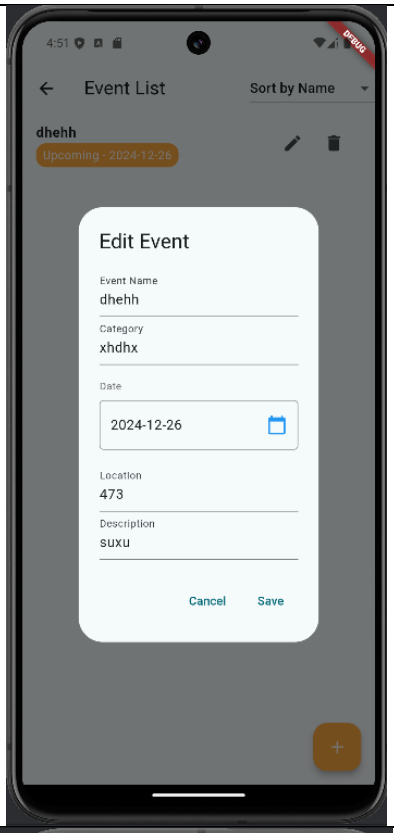
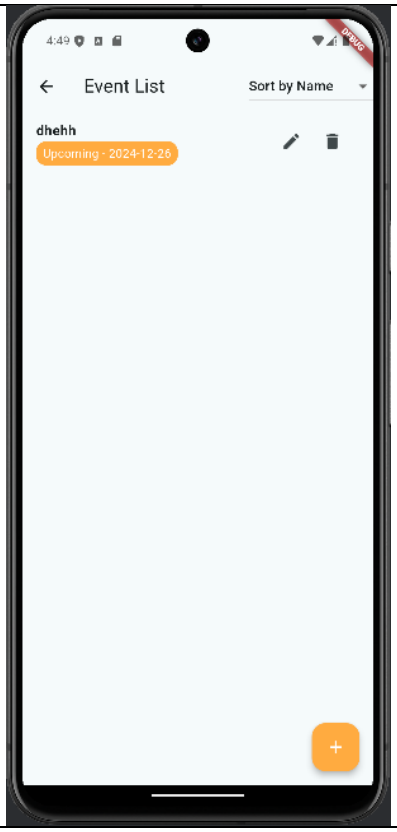
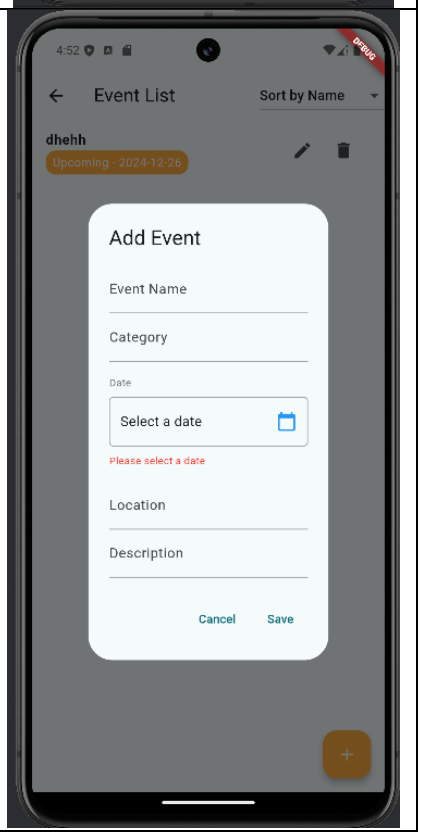
	Sign In(Incorrect Credentials)	
	Sign In(Correct Credentials)	

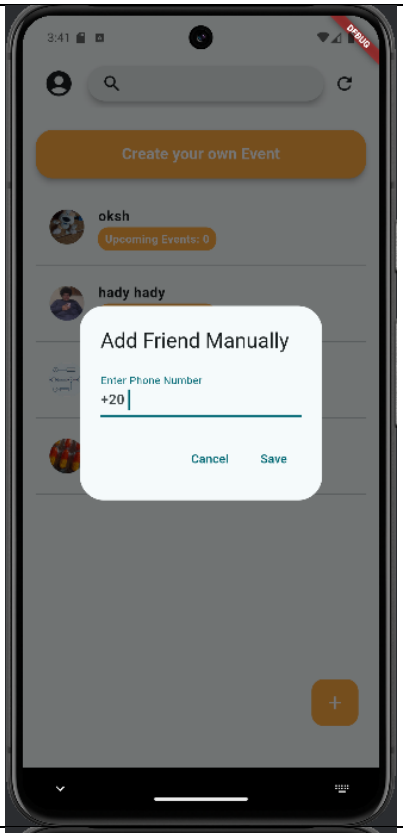
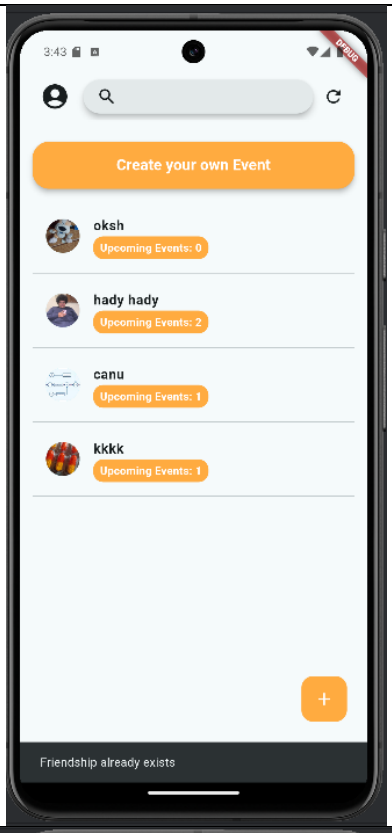
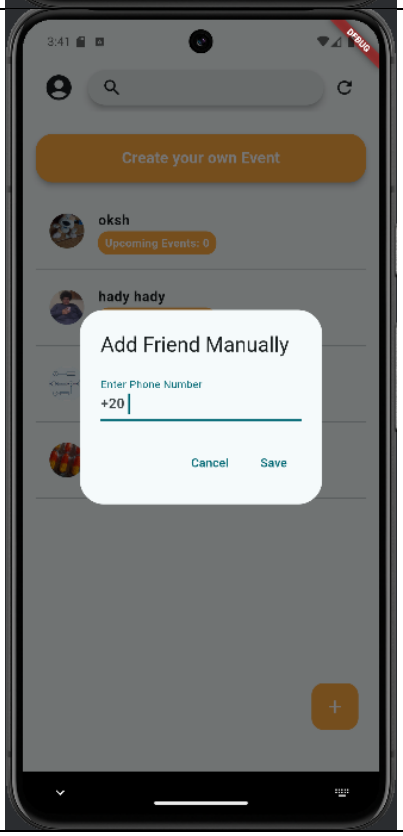
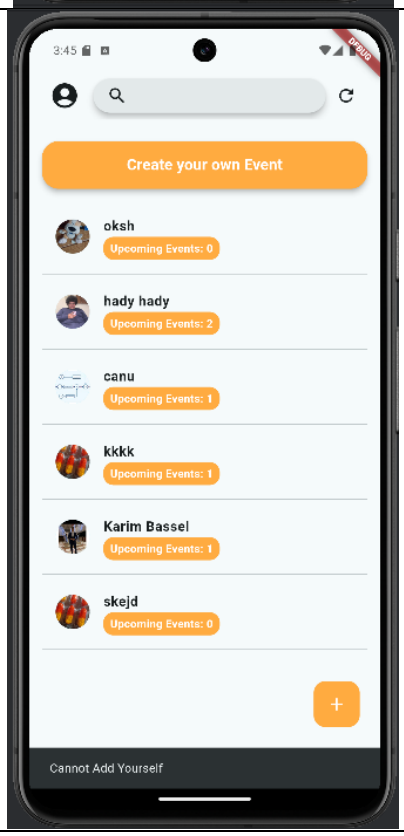
	Press Profile Icon	
	Create your own Event	

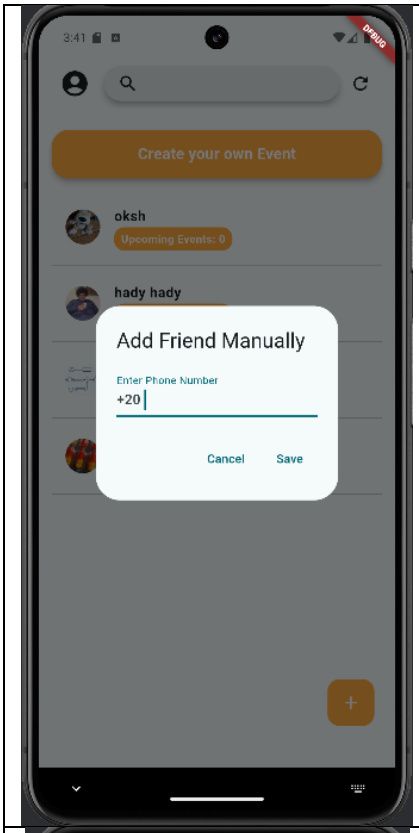
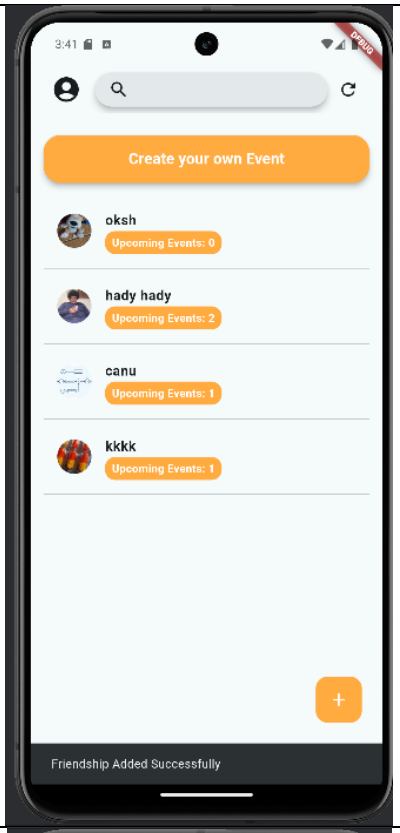
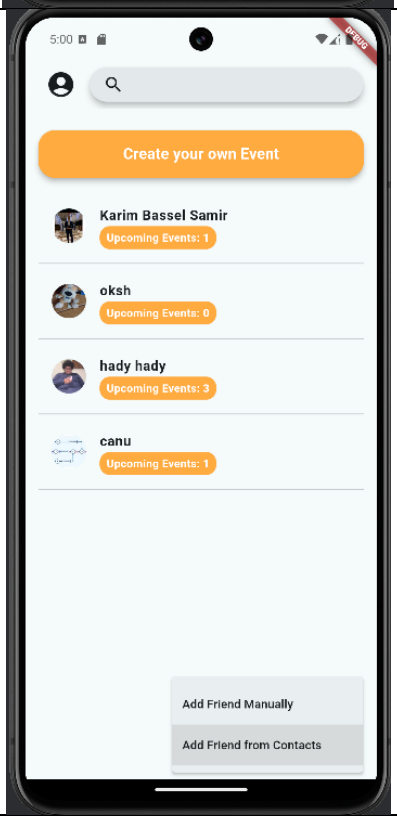
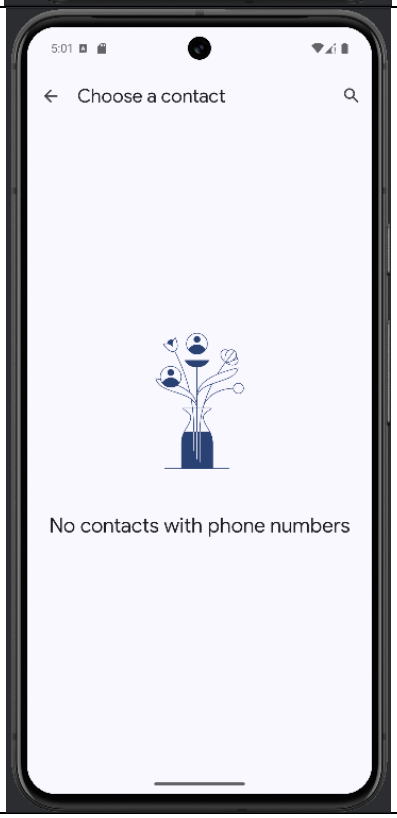
<table>
<tr>
<td> 
</td>
<td>Tap on Friend Card</td>
<td> 
</td>
</tr>
<tr>
<td> 
</td>
<td>Press on Event Card</td>
<td> 
</td>
</tr>
</table>

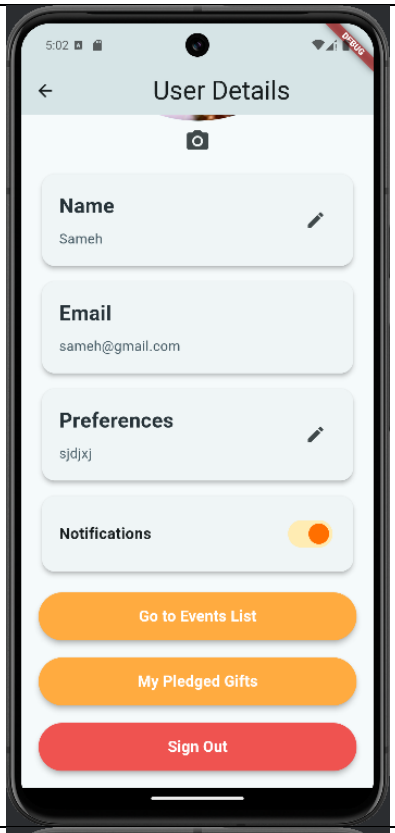
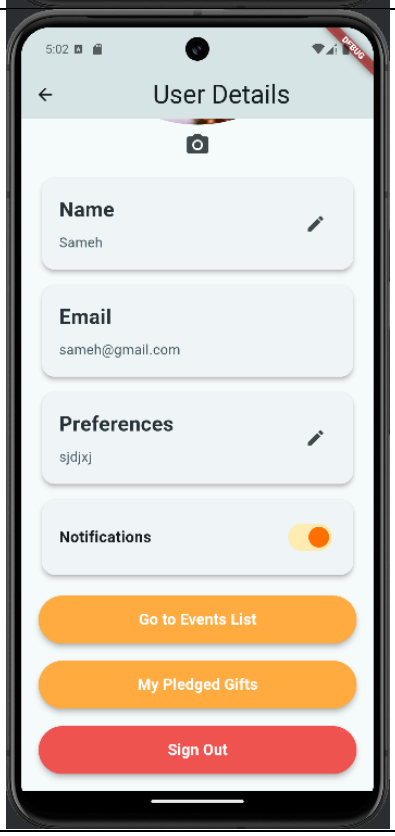
	Press on Gift Card	
	Pledge Gift(Gift Pledged Successfully)	

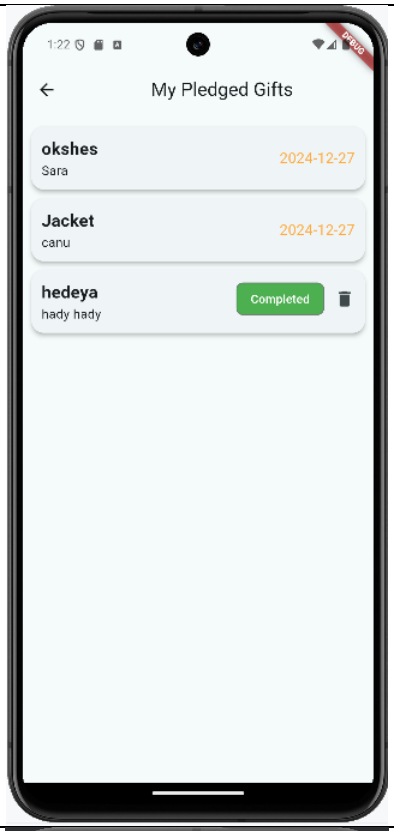
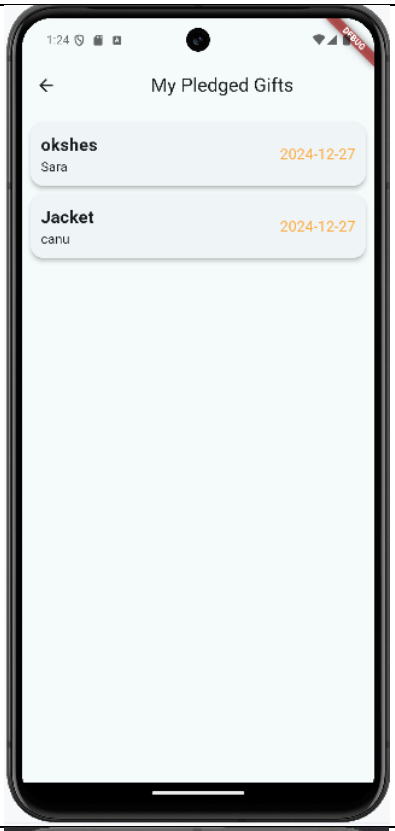
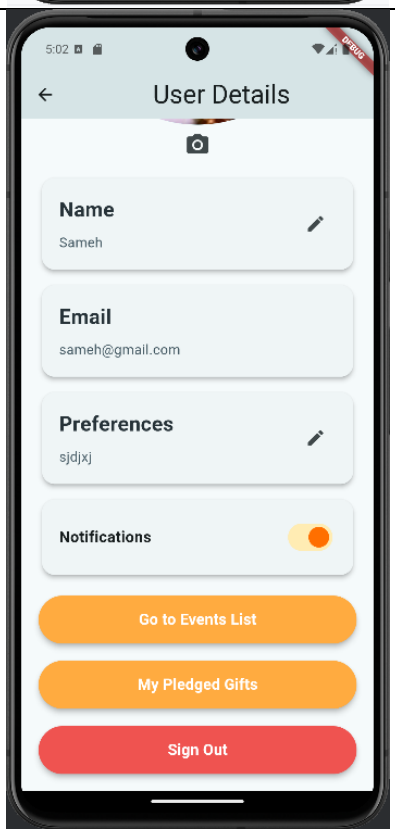
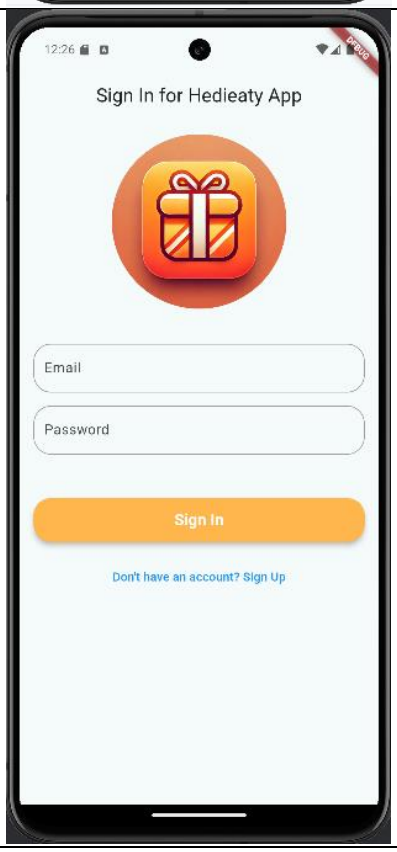
	Unpledge Gift	
	Delete Event	

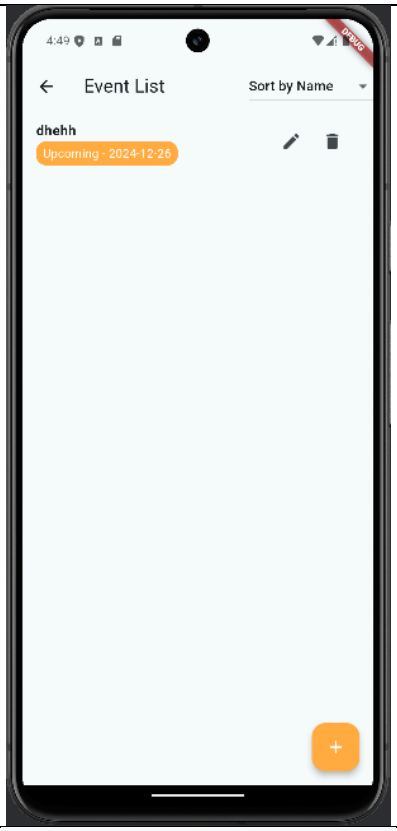
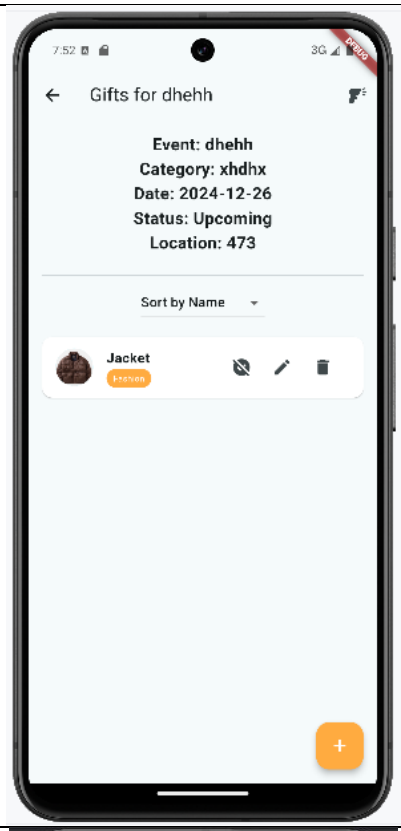
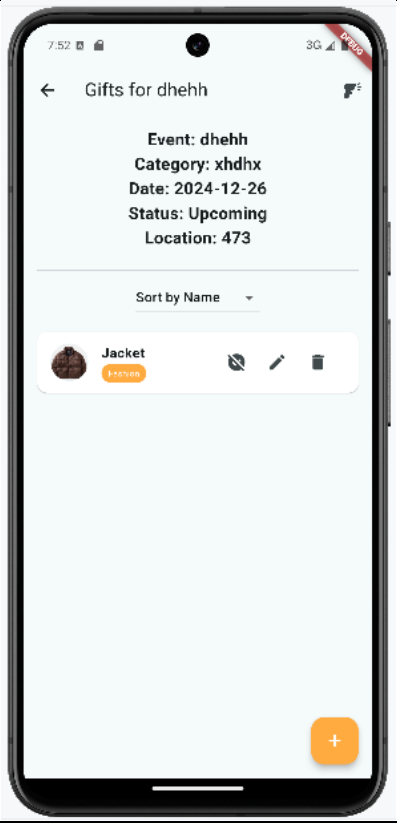
	Edit Event	
	Add Event	

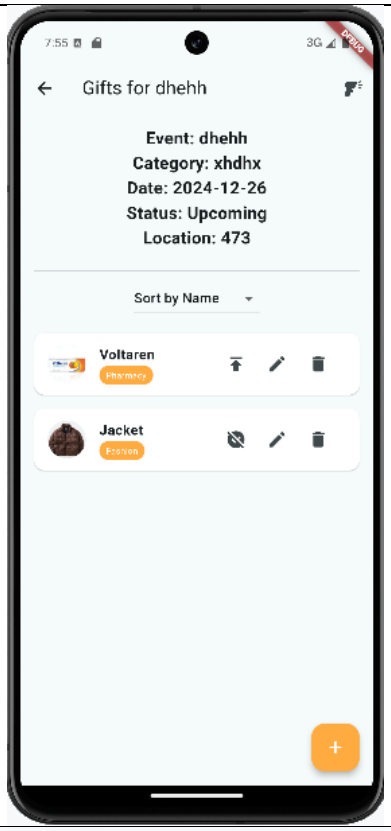
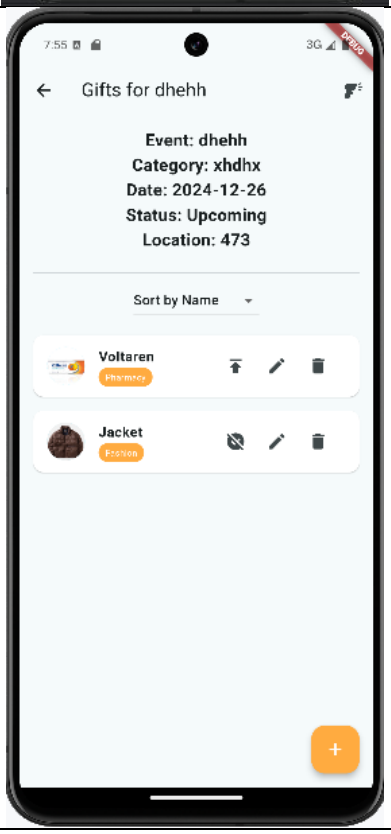
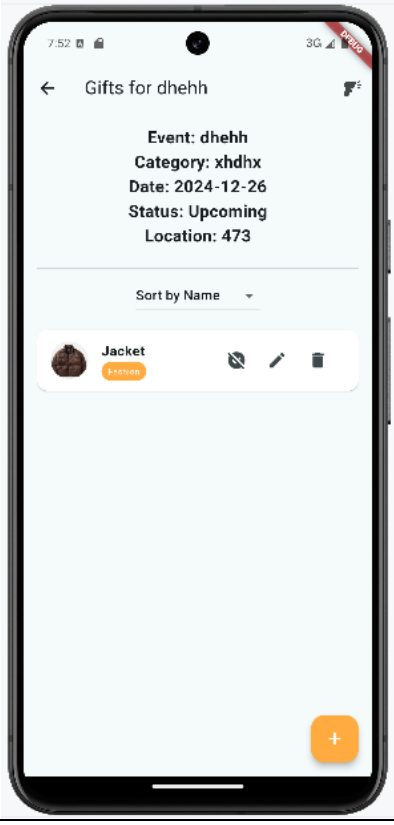
	Add Friend Manually(Friend already added)	
	Add Friend Manually(Adding myself)	

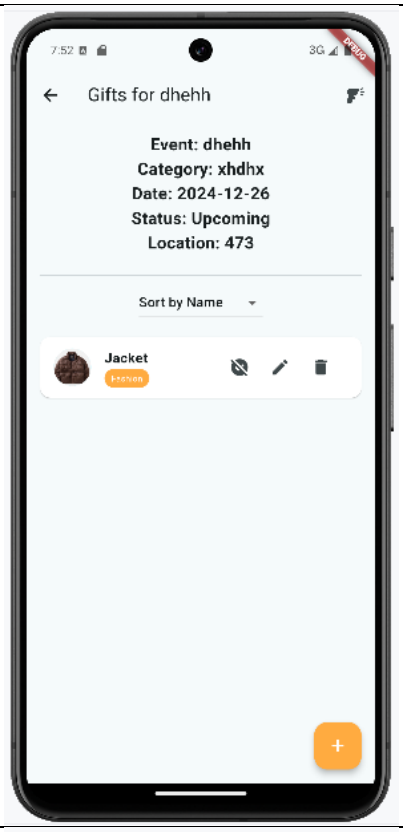
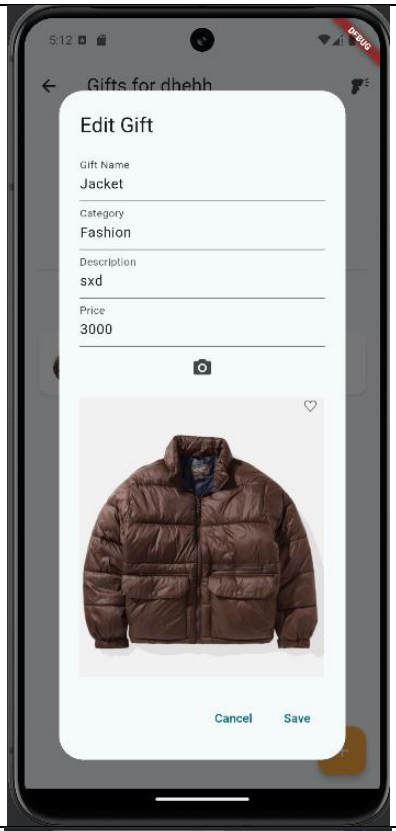
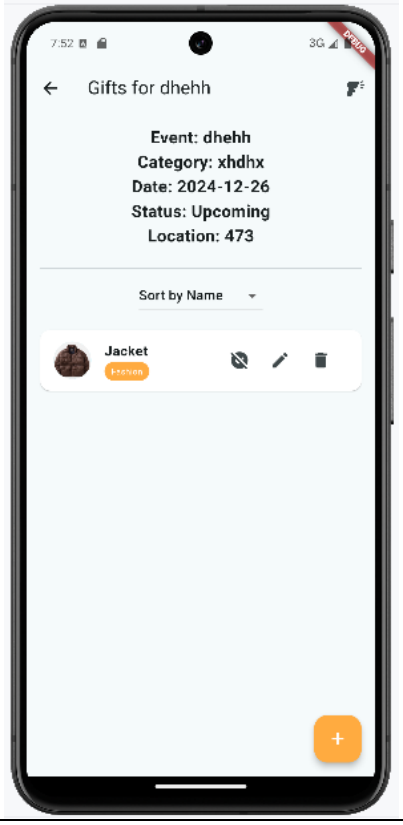
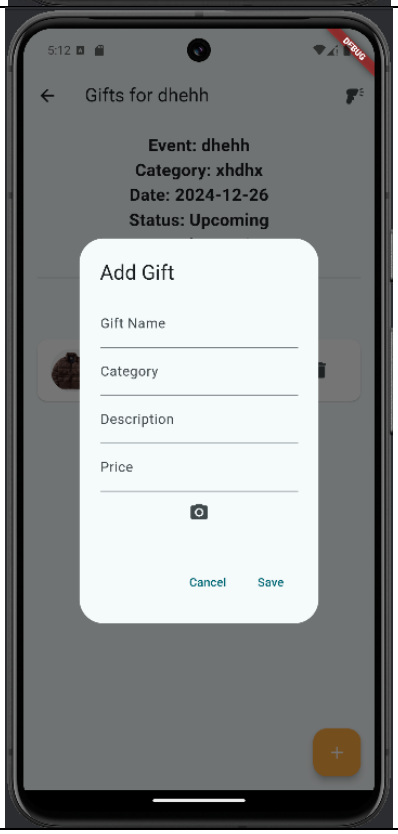
	Add Friend Manually(valid friend)	
	Add Friend from Contacts	

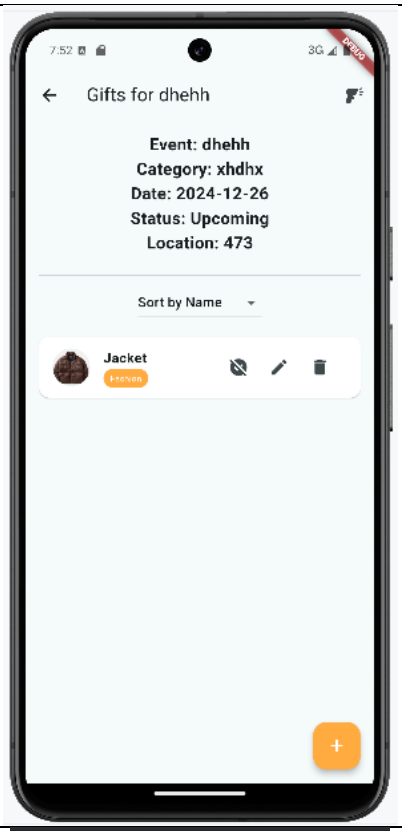
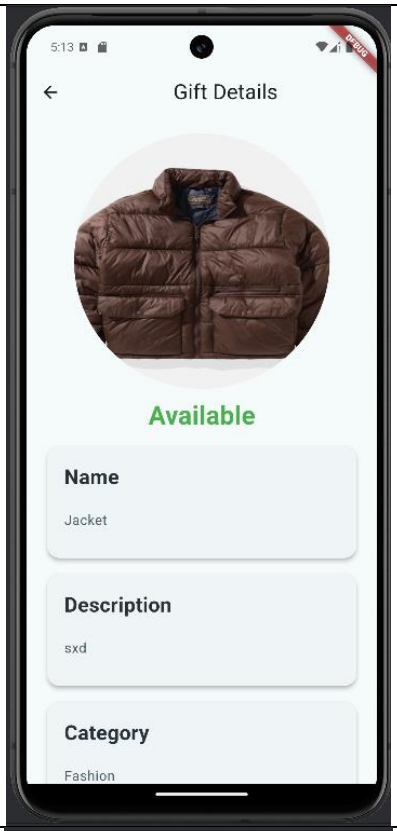
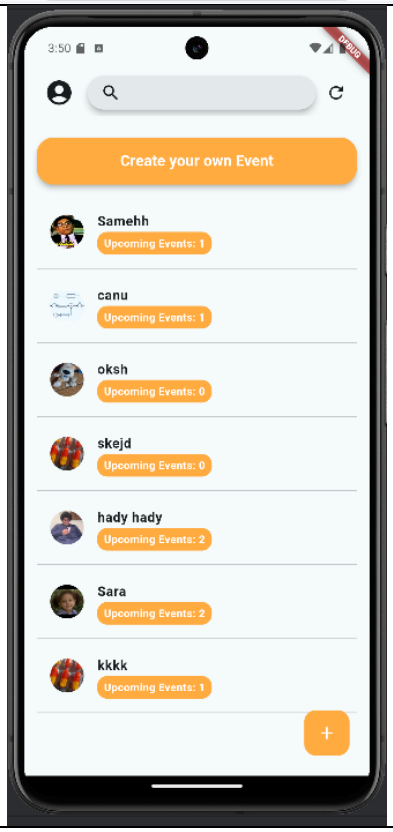
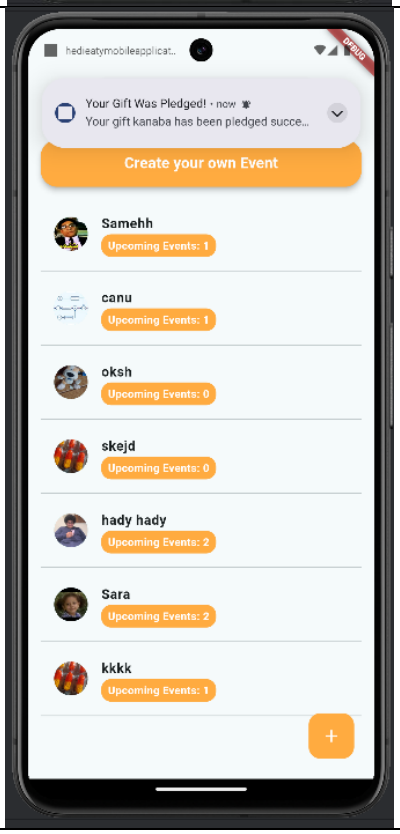
	Go to Events List	
	My Pledged Gifts	

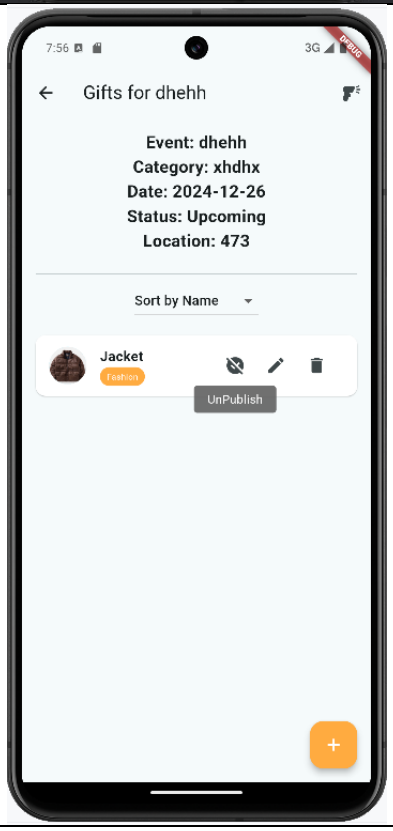
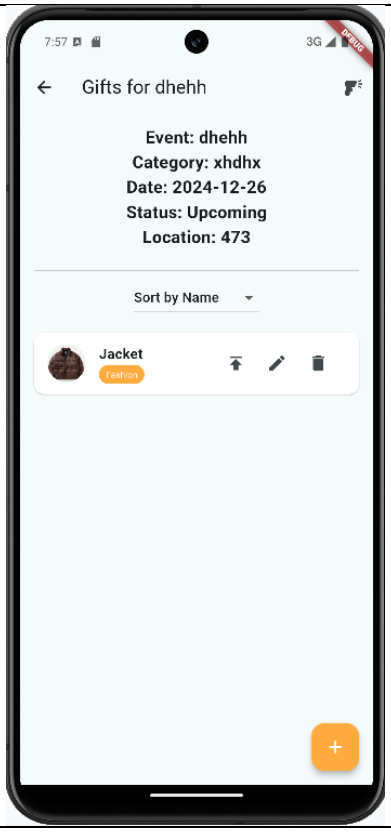
	Delete Icon(Delete Pledged gifts with completed status)	
	Sign Out	

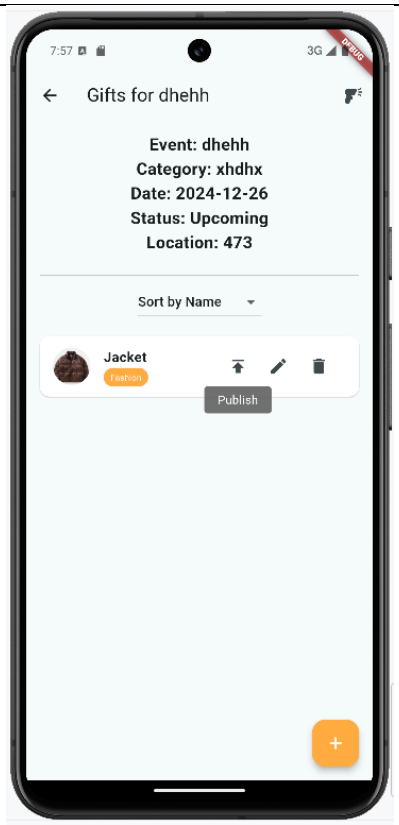
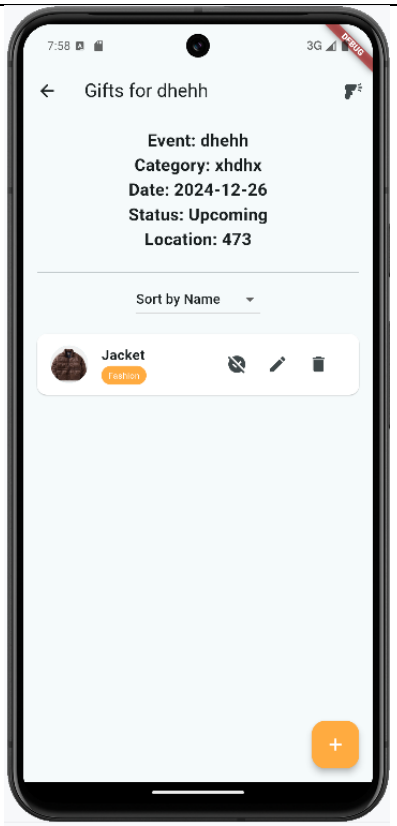
	Press on Event Card	
	Press Barcode Reader	

	Scan Barcode(only one barcode exists in the database which is a barcode for voltaren gel)	
	Delete Gift	

	Edit Gift	
	Add New Gift	

	Press on Gift Card	
	User Gift Pledged(Will receive foreground notification)	

	User Gift Pledged(Will receive background notification)	
	Unpublish gift(it will not be visible to other users)	

	Publish Gift(it will be visible to other users)	

Local Database Design(SQFLITE)

1. **Users Table:**

```
CREATE TABLE Users (
  ID INTEGER PRIMARY KEY AUTOINCREMENT,
  Name TEXT NOT NULL,
  Email TEXT UNIQUE,
  Preferences TEXT,
  PhoneNumber TEXT NOT NULL,
  Password TEXT,
  Image TEXT,
  Notifications BOOLEAN DEFAULT 0,
  UpcomingEvents INTEGER DEFAULT 0
);
```

2. **Events Table:**

```
CREATE TABLE Events (
  ID INTEGER PRIMARY KEY AUTOINCREMENT,
  Name TEXT NOT NULL,
  Date TEXT NOT NULL,
  Location TEXT,
  Description TEXT,
  Category TEXT,
  Status TEXT,
  UserID INTEGER NOT NULL,
  FOREIGN KEY (UserID) REFERENCES Users(ID)
);
```

3. **Gifts Table:**

```
CREATE TABLE Gifts (
  ID INTEGER PRIMARY KEY AUTOINCREMENT,
  Name TEXT NOT NULL,
  Description TEXT,
  Category TEXT,
  Price INTEGER,
  Image TEXT,
  Status INTEGER DEFAULT 0,
  EventID INTEGER NOT NULL,
  PledgerID INTEGER DEFAULT -1,
  UserID INTEGER NOT NULL,
  IsPublished INTEGER DEFAULT 0,
  FOREIGN KEY (EventID) REFERENCES Events(ID),
  FOREIGN KEY (PledgerID) REFERENCES Users(ID)
);
```

4. **Friends Table:**

```
CREATE TABLE Friends (
  ID INTEGER PRIMARY KEY AUTOINCREMENT,
  UserID INTEGER NOT NULL,
  FriendID INTEGER NOT NULL,
  FOREIGN KEY (UserID) REFERENCES Users(ID),
  FOREIGN KEY (FriendID) REFERENCES Users(ID)
);
```

5. BarcodeGifts Table:

```
CREATE TABLE BarcodeGifts (
  ID INTEGER PRIMARY KEY AUTOINCREMENT,
  Name TEXT NOT NULL,
  Description TEXT,
  Category TEXT,
  Price INTEGER,
  Image TEXT,
  Barcode TEXT
);
```

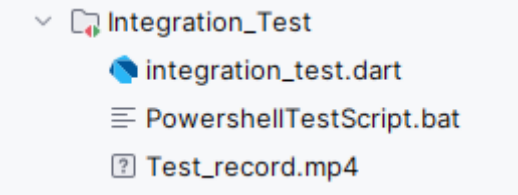
Firebase Real-time Database Structure(Sample)


```
{
"BarcodeGifts": [
id,
{
"Barcode": "6224007686263",
"Category": "Pharmacy",
"Description": "Voltaren gel",
"ID": 1,
"Image": "Base64Image",
"Name": "Voltaren",
"Price": 50
}
],
"Events": [
id,
{
"Category": "Birthday",
"Date": "2024-12-15",
"Description": "A fun celebration with friends and family",
"ID": 1,
"Location": "Johns House",
"Name": "Birthday Party",
"Status": "Completed",
"UserID": 1
}

],
"Friends": [
id,
{
"FriendID": 799103911,
"ID": 1,
"UserID": 327910316
}

],
"Gifts": [
id,
{
"Category": "sjjd",
"Description": "dhhed",
"EventID": 9,
"ID": 1,
"Image": "Base64Image",
"IsPublished": 1,
"Name": "karim",
```

"PledgerID": -1,
"Price": 400,
"Status": 0,
"UserID": 1003419596
}

],
"Users": {
"64858064": {
"Email": "person3@gmail.com",
"ID": 64858064,
"Image": "Base64Image",
"Name": "person333",
"Notifications": 1,
"PhoneNumber": "1287036438",
"Preferences": "sports"
}
}

Integration Testing



integration_test.dart

NOTE: USER HAS TO BE SIGNED OUT BEFORE EXECUTING THIS TEST TO AVOID THE AUTO LOGIN WHEN RUNNING THE TEST.

Description:

This Flutter integration test verifies the end-to-end functionality of a user journey in the app. It begins by launching the main application using app.main() and allowing it to initialize with a delay through pumpAndSettle. The test simulates user interactions, such as entering credentials into the email and password fields, which are located using keys like emailField and passwordField. Once the fields are populated, the test ensures the "Sign In" button is visible, taps it, and waits for the next screen to load.

After logging in, the test navigates through multiple app screens. It taps on a friend's event list, identified by the text "Samehh," and waits for the navigation to settle. Similarly, it proceeds to access a specific event gift list by tapping on an event labeled "dhehh." Once the gift list loads, the test further navigates to the gift details screen by tapping on the gift named "Jacket" and waits again to ensure the UI updates correctly.

On the gift details screen, the test searches for a "Pledge" or "Cancel" button. Using a ListView with a specific key GiftDetailsListView, the test scrolls down by simulating a drag action until the PledgeButton becomes visible. It taps the button and waits for the navigation to complete. To confirm the action was successful, the test checks that the app returns to the gift list screen, verifying the text "Gifts for dhehh" appears on the screen. This ensures that pledging or canceling a gift triggers the expected behavior. The test wraps up with this final validation step, confirming the flow works as intended.

Code:

```
import 'package:flutter_test/flutter_test.dart';
import 'package:hedieatymobileapplication/Views/SplashScreen.dart';
import 'package:integration_test/integration_test.dart';
import 'package:hedieatymobileapplication/main.dart' as app; //
Import your app entry point
import 'package:flutter/material.dart';
void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();
  group('App Integration Tests', () {
    testWidgets('Integration Test 1', (WidgetTester tester) async {

        app.main();

         await tester.pumpAndSettle(Duration(seconds: 10));
        final emailField = find.byKey(Key("emailField"));
```

```
        final passwordField = find.byKey(Key('passwordField'));
        final signInButton = find.byKey(Key('signInButton'));


        await tester.enterText(emailField,
'karimbassel15@gmail.com');
        await tester.enterText(passwordField, '123456');
        await tester.ensureVisible(signInButton);
        await tester.tap(signInButton);
        await tester.pumpAndSettle(Duration(seconds: 10));
        //Navigate to friend event list
        await tester.tap(find.text("Samehh"));
        await tester.pumpAndSettle(Duration(seconds: 5));
        //navigate to event gift list
        await tester.tap(find.text("dhehh"));
        await tester.pumpAndSettle(Duration(seconds: 5));
        //navigate to gift details
        await tester.tap(find.text("Jacket"));
        await tester.pumpAndSettle(Duration(seconds: 5));
        //search for pledge/cancel button and tap on it
        final listViewFinder =
find.byKey(Key('GiftDetailsListView'));
        final pledgeButtonFinder = find.byKey(Key('PledgeButton'));
// Target the button
        // Scroll until the button becomes visible
        await tester.drag(listViewFinder, const Offset(0, -800)); //
Scroll down
        await tester.pumpAndSettle(Duration(seconds: 3));
        //press the button
        await tester.tap(pledgeButtonFinder);
        await tester.pumpAndSettle(Duration(seconds: 5));

        //ensure gift pledged, when gift is pledged/cancelled the
user is navigated to the gift list again
        expect(find.text('Gifts for dhehh'), findsOneWidget);


      });

  });
}
```

PowershellTestScript.bat

Description:

The script automates the process of recording a screen on an Android device while running Flutter integration tests and managing the recorded file afterward. It begins by initiating screen recording on the Android device using the adb shell screenrecord command, saving the recording to the device's internal storage. This is done asynchronously in the background, allowing other processes to continue. Simultaneously, the Flutter integration tests are executed in a separate process by starting the flutter test command, which runs the test script provided in the file integration_test.dart.

The script then pauses for a specified duration, allowing time for the Flutter tests to complete while the recording continues. After waiting for 300 seconds, the script gracefully stops the screen recording process on the device using the adb shell pkill -l2 screenrecord command. Since stopping the recording process might take a moment, the script waits an additional 10 seconds to ensure the recording has fully terminated and the file is finalized on the device.

Once the recording stops, the script retrieves the recorded video file from the Android device to the local machine using the adb pull command. This pulls the file named Test_record.mp4 from the device's storage to the current working directory on the local machine. After successfully transferring the file, the script removes the recording file from the Android device using the adb shell rm command to clean up and free storage space.

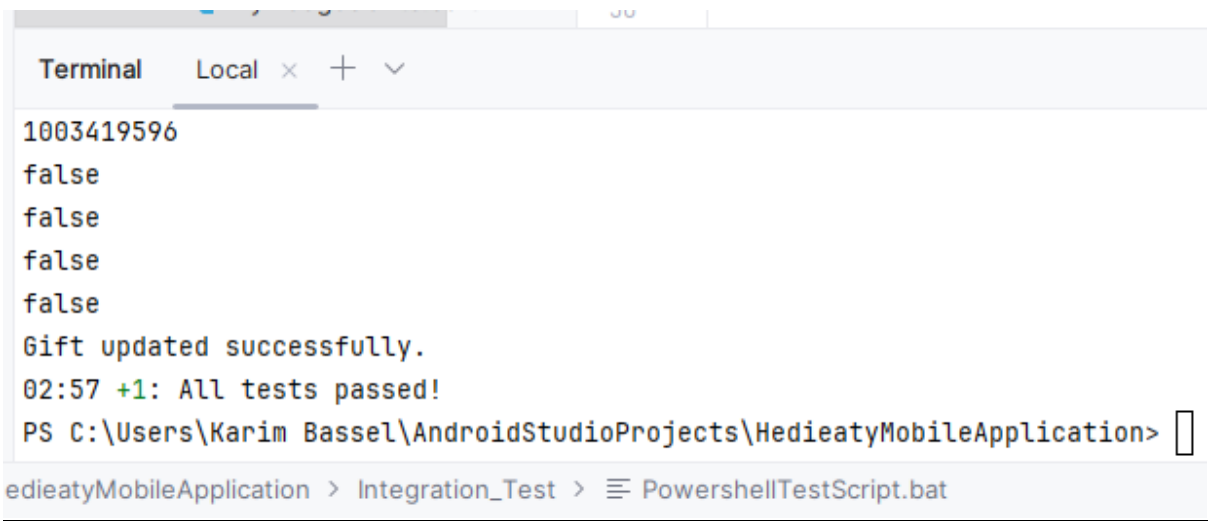
Throughout the execution, the script provides messages to the user via the echo command to indicate progress, such as starting the recording, running tests, stopping the recording, pulling the video, and cleaning up the file. Finally, a confirmation message is displayed to indicate the process has completed successfully, and the recording file is saved locally as Test_record.mp4. The script pauses at the end, allowing the user to review the output before the terminal closes.

Code:

```
@echo off

REM Start screen recording asynchronously on Android
echo Starting screen recording on Android...
start "" adb shell screenrecord  /sdcard/Test_record.mp4

REM Run Flutter integration tests in the background
echo Running Flutter integration tests...
start "" flutter test integration_test.dart

REM Wait for a brief moment to allow the tests to start running
timeout /t 300

REM Wait for the test to complete before stopping the recording
echo Test completed, stopping screen recording...
adb shell pkill -l2 screenrecord

REM wait till recording stops
timeout /t 10

REM Pull the video to the local machine
echo Pulling video to local machine...
```

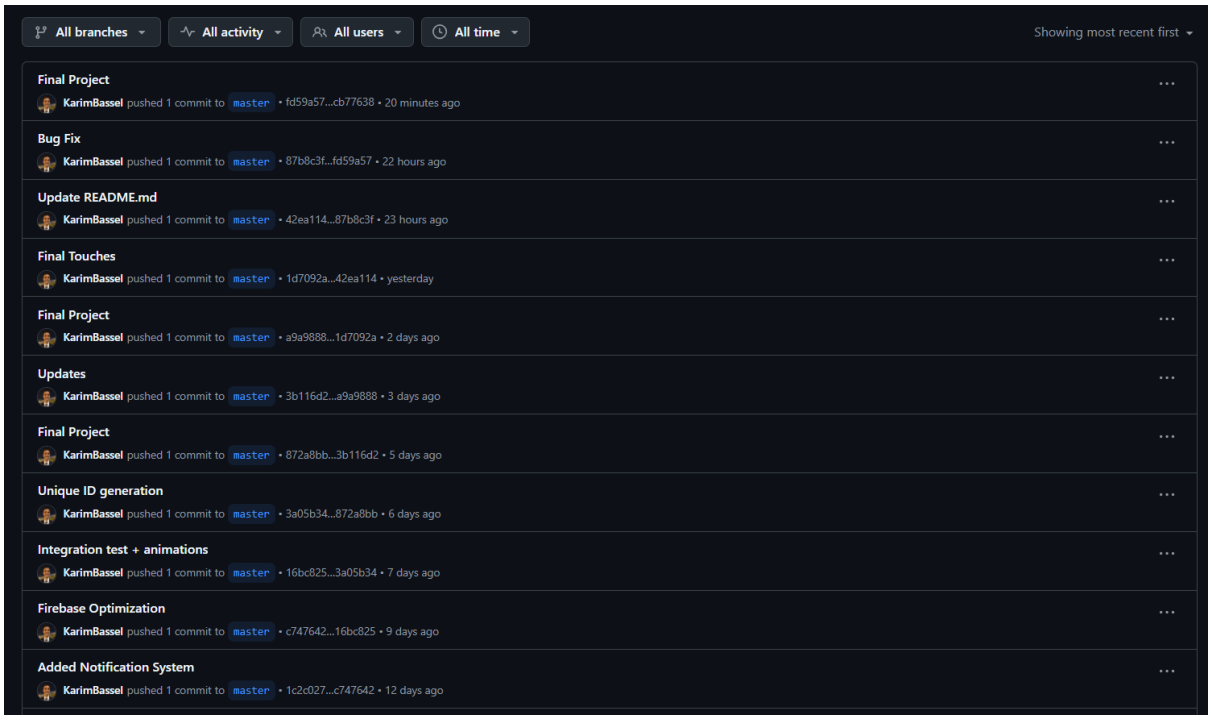
```
adb pull /sdcard/Test_record.mp4 .

REM Delete the recording file from the device
adb shell rm /sdcard/Test_record.mp4

echo Test completed. Recording saved as test_recording.mp4.
pause
```

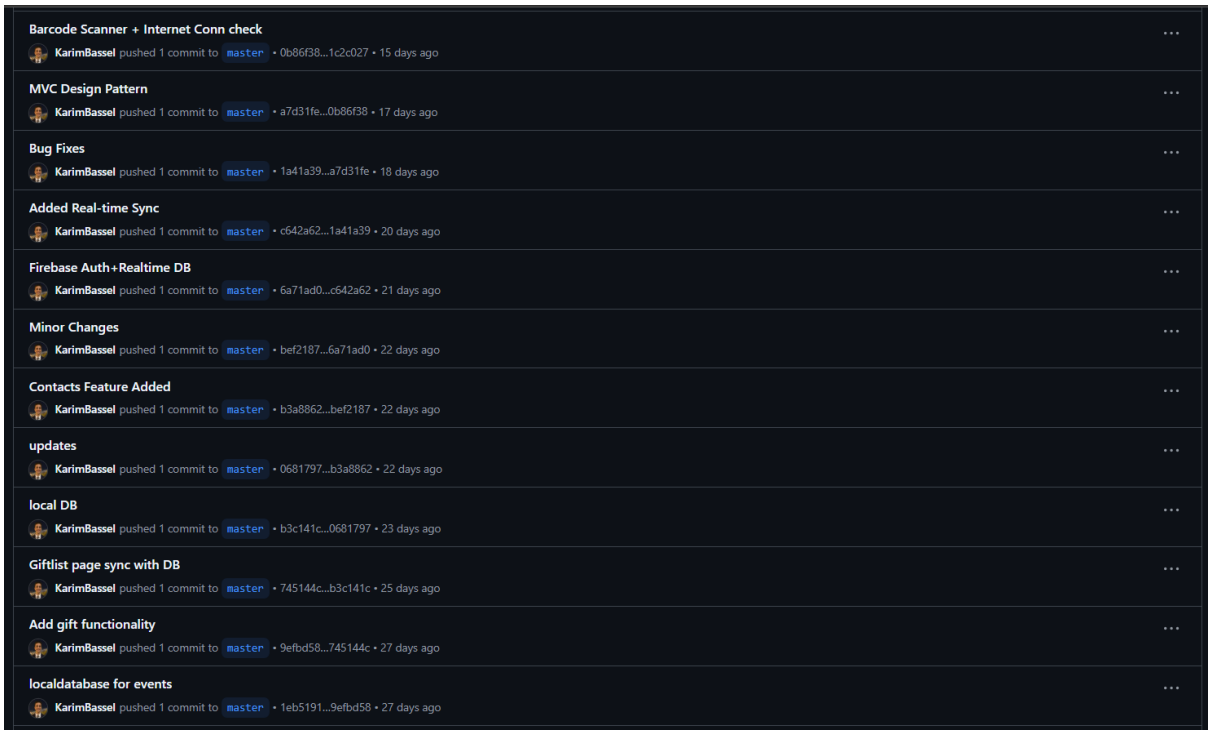
Test Execution:

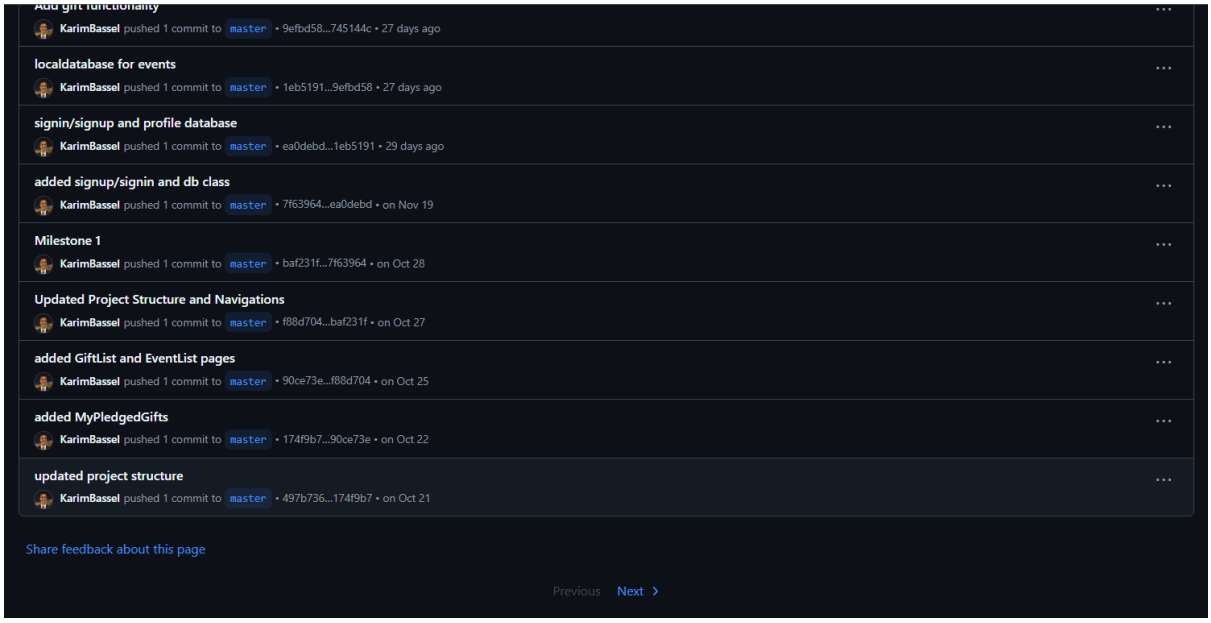


Known Bugs

Version Control Activity Report







Github Repository Link

Link to Repo

Demonstration Video Link

Link to Video

Application Landing Page

Link to App