AY 2022/2023

HangOut
Design Document

Matteo Beltrante Marco Bendinelli

Project supported at Politecnico di Milano

Contents

1 Abstract

HangOut is a dynamic and engaging Flutter application that redefines social interactions in the post-pandemic era. With its modern and smooth design, it provides a platform for users to discover, join, and create a wide range of events and groups tailored to their interests. Whether it's a lively concert, a delightful dinner, an exciting game session, or a captivating cultural experience, HangOut ensures that users can easily connect and hang out with like-minded individuals who share their passions.

2 Introduction

2.1 Overview

HangOut is a modern mobile application that redefines socializing by providing a dynamic platform for users to connect, organize, and participate in events tailored to their interests.

The app offers a seamless user experience with its organized structure, consisting of four main pages: *My events*, *Explore*, *My groups*, and *Profile*. Through the *Explore* page, users can discover a diverse range of events and groups published by fellow users, filtering them based on specific interests such as food, sports, culture, music, nature, games, and more. The *My events* page keeps users informed about the events they have joined, while the *My groups* page provides a centralized hub for managing and engaging with the groups they are a part of. The *Profile* page allows users to personalize their account, providing a glimpse into their interests to facilitate connections with other users.

In addition to the main pages, HangOut incorporates interactive popups to enhance user interactions. The *Single Group* or *Event* popup allows users to view detailed information about a specific group or event, providing key details such as name, description, participants and so on. The *User Information* popup provides users with the ability to access and view the personal information of other users. The *Add event* and *Add group* popups offer users a way to create their own customized events and groups. Lastly, the *Notifications* popup allows you to keep track of all the messages and invitations received.

2.2 Objectives

Foster connections

HangOut aims to create a vibrant community where users can effortlessly connect with like-minded individuals who share their passions. By providing a platform that emphasizes shared interests, the app promotes meaningful connections and facilitates the formation of new friendships.

Simplify event discovery

It makes the discovery and the joint of events easy. By presenting a diverse range of activities and allowing filtering based on specific interests, the app ensures that users can easily find events that align with their preferences, providing opportunities to engage in activities they enjoy.

Encourage social engagement

It seeks to encourage users to step out of their comfort zones and engage in social interactions. The app provides a safe and inclusive space where individuals who may have distanced themselves socially can regain the courage to meet new people, fostering a sense of community and supporting personal growth.

Empower event organization

It empowers users to create and promote their own events, giving them the ability to share their passions and gather individuals who share the same interests. By enabling seamless event organization and communication within private groups, the app facilitates the planning and coordination of gatherings, making it easier for users to bring their event ideas to life.

Overcome social barriers

It aims to overcome the social barriers that individuals often face when trying to find companions for specific events. The app ensures that even those who struggle to find friends willing to attend a particular event have the opportunity to connect with others who share their enthusiasm, creating an inclusive environment where no one misses out on the chance to participate in desired experiences.

3 Functional Requirements

3.1 Features

3.1.1 Unified Sign-in Options

Users have the flexibility to sign in or sign up for the app using various authentication methods. They can choose to authenticate either through their email address or leverage the convenience of Single Sign-On (SSO) functionality, supported by compatible identity providers.

3.1.2 Exploration of groups and events

The *My Explore* page offers users a comprehensive platform to explore and discover a diverse range of groups and events. The app automatically curates personalized recommendations by pre-selecting the user's interests, ensuring relevant and accurate results. However, users have the flexibility to fine-tune their exploration by modifying the filters and selecting additional or fewer interests. Furthermore, the search bar enables users to easily locate specific events or groups by simply entering their names.

3.1.3 Group creation and customization

Users have the ability to create personalized groups tailored to their interests and preferences:

- **name and summary**: they can provide a distinct and meaningful name for their group and a descriptive summary to concisely portray the purpose and objectives of the group;
- **interests**: they can specify interests associated with the group to attract like-minded individuals;
- **image**: they are also encouraged to upload a custom image from their cell phone gallery, providing visual representation for the group.

To control access and maintain privacy, the users can specify the following options:

- **private groups**: users can select the private setting, which restricts access to the creator and enables manual addition of selected individuals as members. Note that private groups are excluded from the *Explore* page, ensuring exclusivity and limiting participation to invited members.
- **public groups**: the groups are visible in the *Explore* page which encourage open participation and engagement from any interested individual.

3.1.4 Event creation and customization

Users can plan and manage their events by specifying key details and settings. Here's an overview of the event creation and customization capabilities:

- **date, time, name, and description:** users have full control over setting the date and time of their events, ensuring accurate scheduling and coordination. Furthermore, they can specify event names and provide detailed descriptions that convey the event's purpose and essence;
- **location and privacy settings:** users have the option to set the location of an event through a dedicated **maps page**, allowing them to precisely specify the desired location. Additionally, users can configure privacy settings for events, choosing between public or private. Public events are discoverable in the *Explore* page, while private events maintain exclusivity by restricting access to invited participants;
- **event categories:** to streamline event discovery, the app allows users to assign events to a specific category. Whether it's food, sports, culture, music, nature, games, or study, users can align their events with relevant categories to attract like-minded individuals;
- **group invitations:** leveraging the power of community, HangOut enables users to send event invitations directly to their existing groups. This seamless integration ensures that events reach the right audience and fosters active participation among community members.

3.1.5 Efficient Event Organization

The integration of a comprehensive calendar feature in the *My Events* page ensures seamless event management and easy access for users. With the ability to view and track the events they have joined, users can effortlessly stay informed and in control of their event commitments.

3.1.6 Flexible Group and Event modification

The app offers users the flexibility to modify their created groups and events according to their evolving needs. The creator has the ability to make various adjustments to the group's or event's details and settings even after its creation.

3.1.7 Chat functionality

Within the application, both groups and events feature dedicated chat functionality that facilitates seamless communication among participants. Users can actively engage with other participants through the chat interface, which displays relevant information such as the timestamp of each message, the profile image, and the nickname of the senders.

3.1.8 Profile and app settings management

The *My Profile* page offers various customization options, allowing users to shape their online persona and manage their preferences effectively. Key features of profile and app settings management include:

- **personalization**: users can personalize their profile by customizing their nickname, bio, and specifying their interests. This enables users to showcase their unique identity and foster connections with individuals who share similar interests and values within the thriving app;

- **notifications**: users are empowered with the flexibility to customize their notification preferences according to their desired level of engagement. They can easily toggle push notifications and selectively enable or disable specific behaviors, such as receiving new messages from group or event chats, event invitations, group joins, and notifications about newly published public events or groups that match their interests. This thoughtful customization ensures that users stay informed with timely updates without being overwhelmed by an inundation of notifications;
- **logout and account deletion**: users can log out from their account when needed or choose to permanently delete their account if they no longer wish to use the application;
- **dark mode**: users can activate dark mode, either by adhering to their device's system preferences or manually setting it within the app. This feature enhances visual comfort and reduces eye strain, especially in low-light environments.

3.1.9 Unified Notifications Management

The application offers users a centralized hub for organizing and handling all notifications efficiently. Within the notifications popup, users can access a comprehensive list of notifications sorted chronologically and grouped by date. By tapping on a notification, users can directly navigate to the specific event, group, or chat associated with the notification. Additionally, users have the flexibility to delete notifications as desired.

3.2 Use Cases

In this section, a sequence diagram illustrating the process of adding a group by the user is shown. Please note that, for the sake of clarity, the functions depicted in the diagram may not correspond exactly to the ones in the actual code implementation.

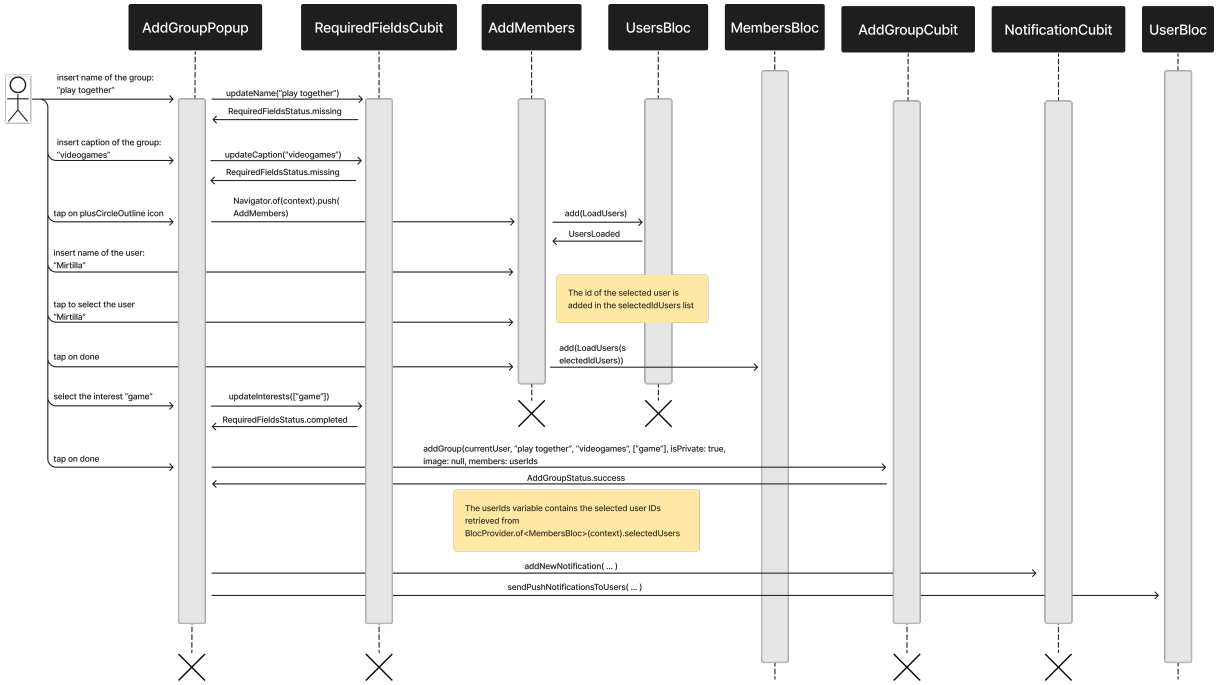


Figure 1: Add group sequence diagram

4 User Interface Design

The application is designed to be both **responsive** and **adaptive**, providing an optimal user experience across various devices. The app is responsive, meaning it adjusts and adapts to different screen sizes on both phones and tablets. Furthermore, the app is adaptive to cater specifically to tablet screens. The tablet landscape and portrait layouts have been carefully crafted to make the most of the larger screen real estate.

4.1 Wireframes

4.1.1 Phone

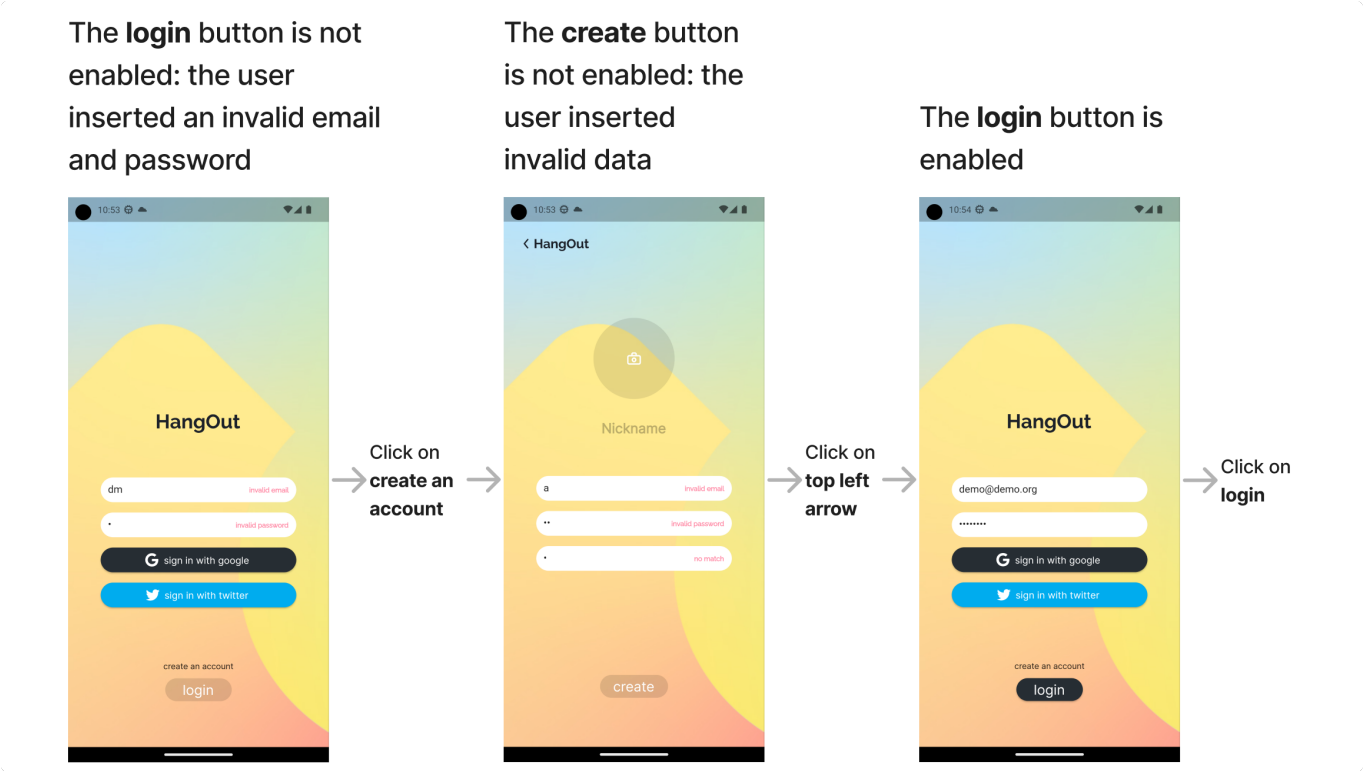


Figure 2: Login and sign up

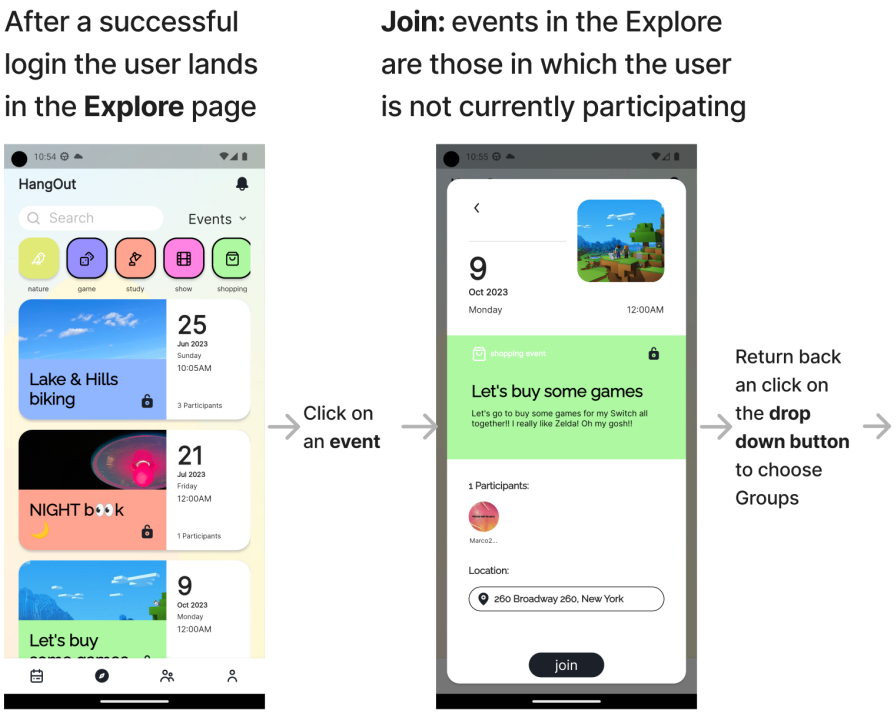


Figure 3: Explore - events

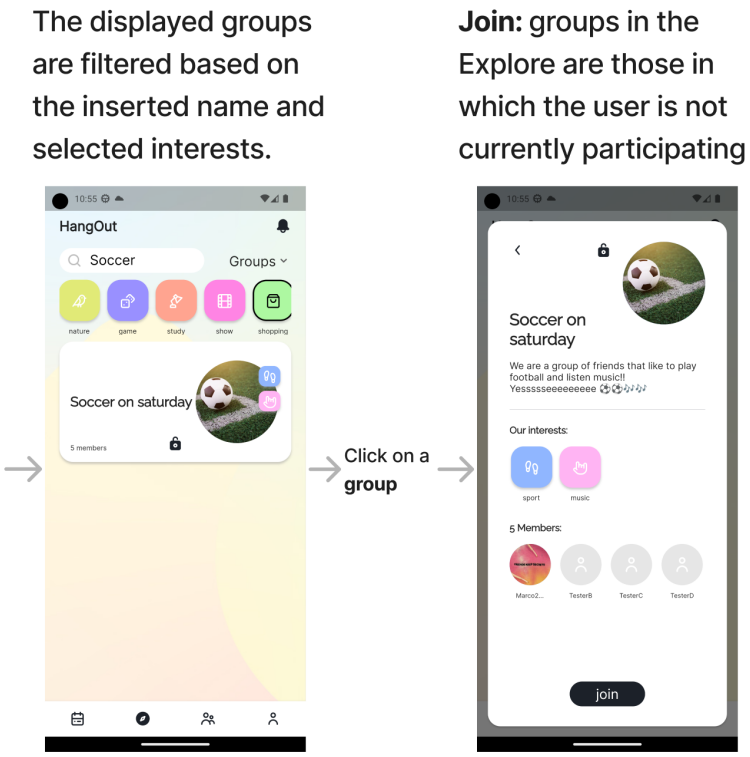


Figure 4: Explore - groups

My events page displays events in which the user is participating.

If the user has not filled in all the mandatory fields (marked with a star) the **done** button remains inactive and he/she cannot create the new event.

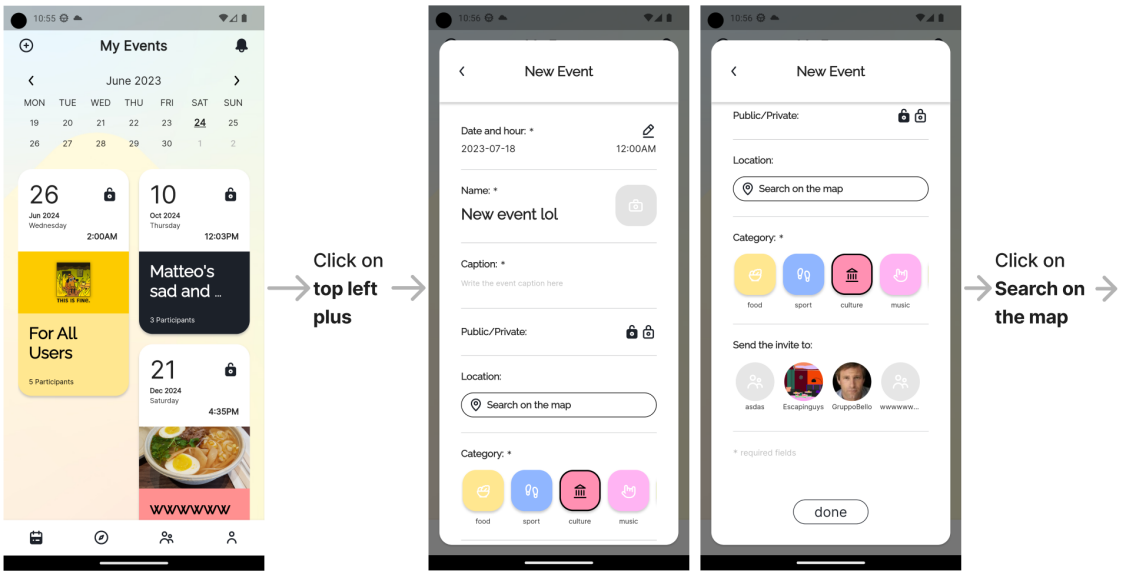


Figure 5: My events - new event

The user has the flexibility to navigate the map for precise location identification or simply utilize the search bar to quickly find the desired place.

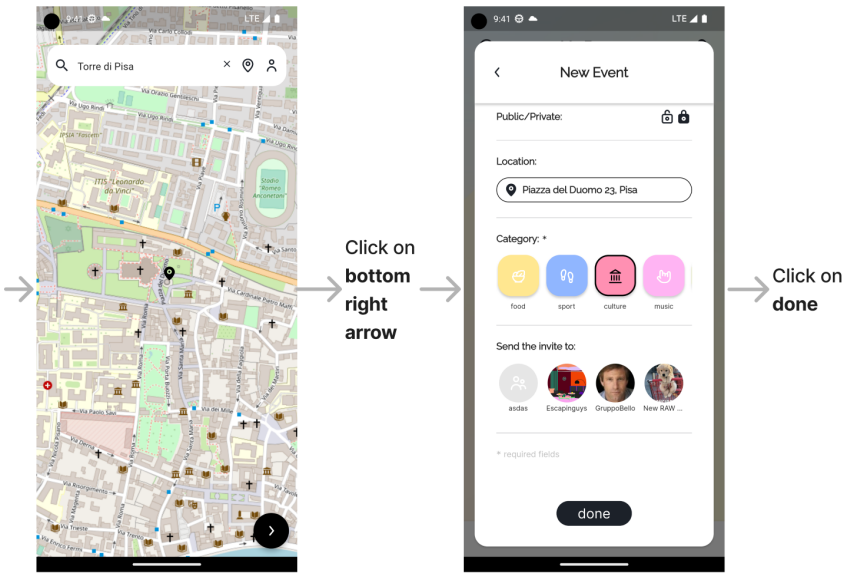


Figure 6: My events - new event

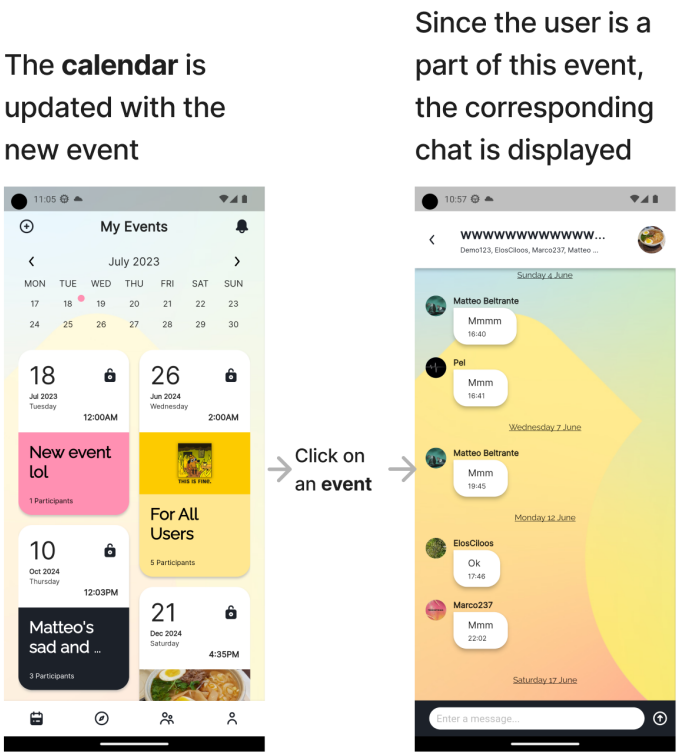


Figure 7: My events - chat

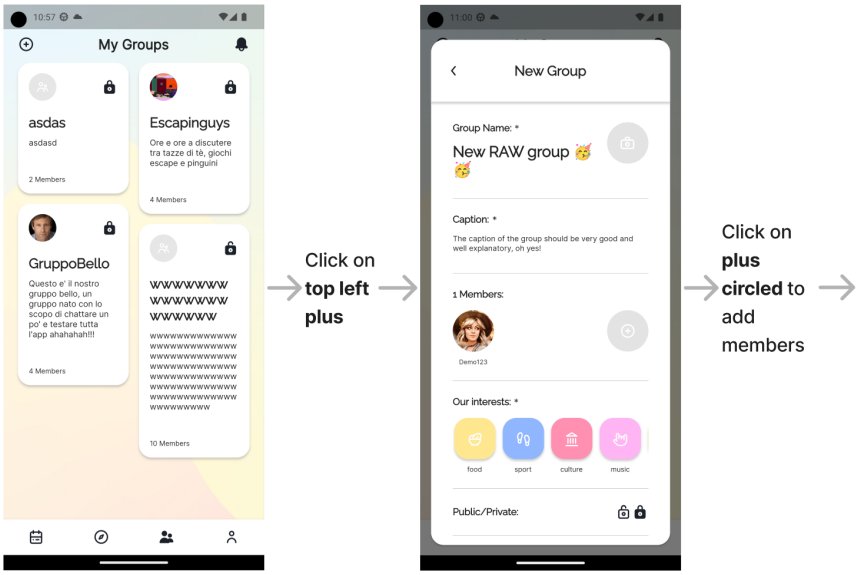


Figure 8: My groups - new group

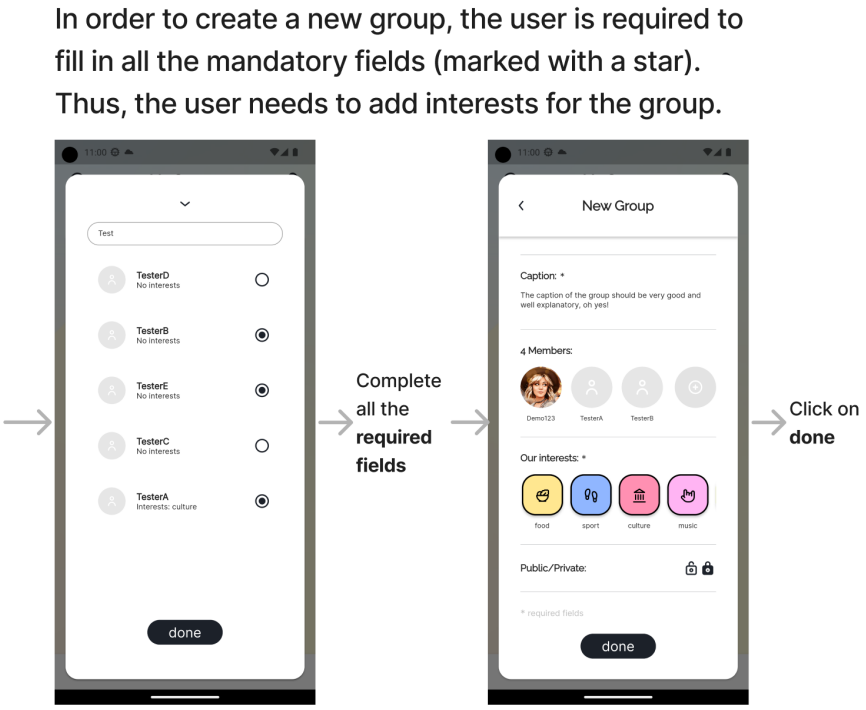


Figure 9: My groups - new group

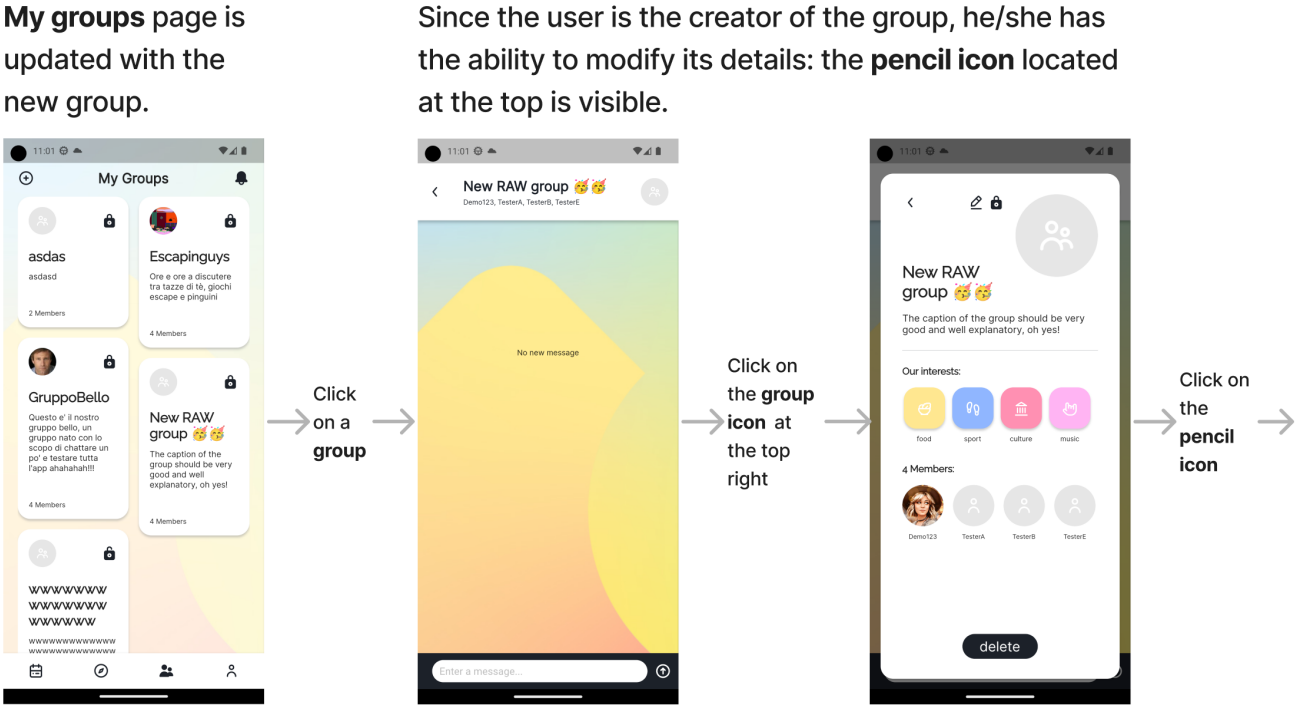


Figure 10: My groups - modify group

The user has the ability to modify the group, but it is important to note that the required fields cannot be removed under any circumstances.

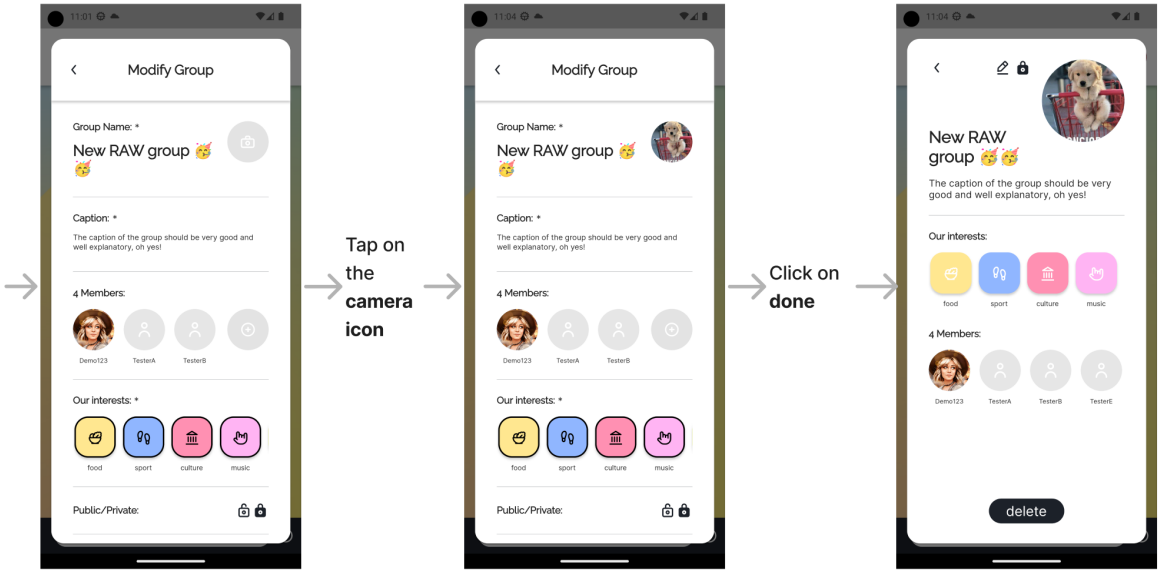


Figure 11: My groups - modify group

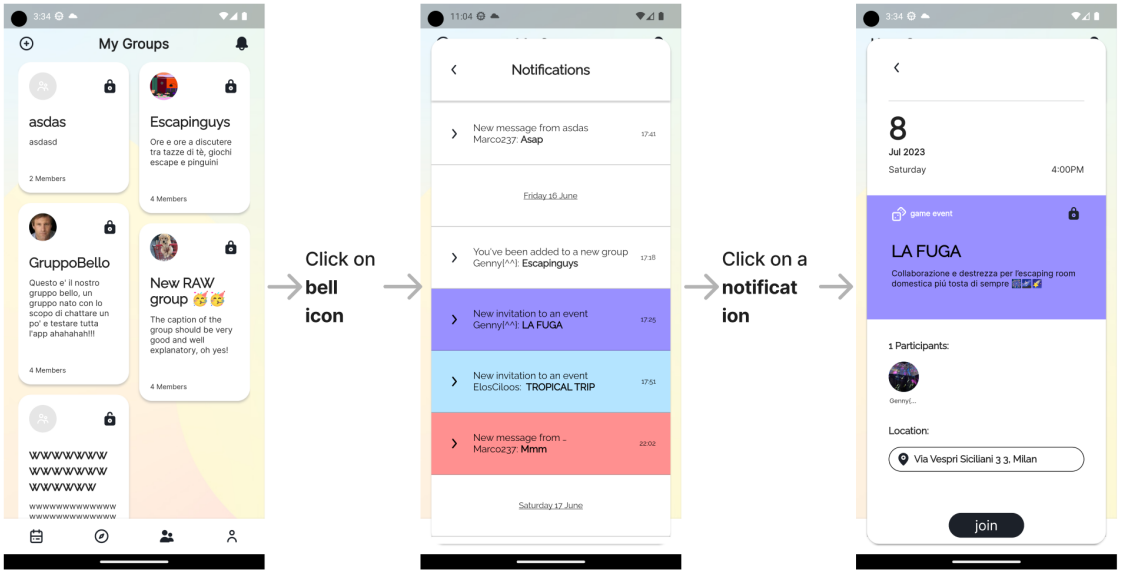


Figure 12: Notifications

In the **My profile** page, the user can update their data and access various app features. However, when changing their information, it is essential for the user to consider the required fields.

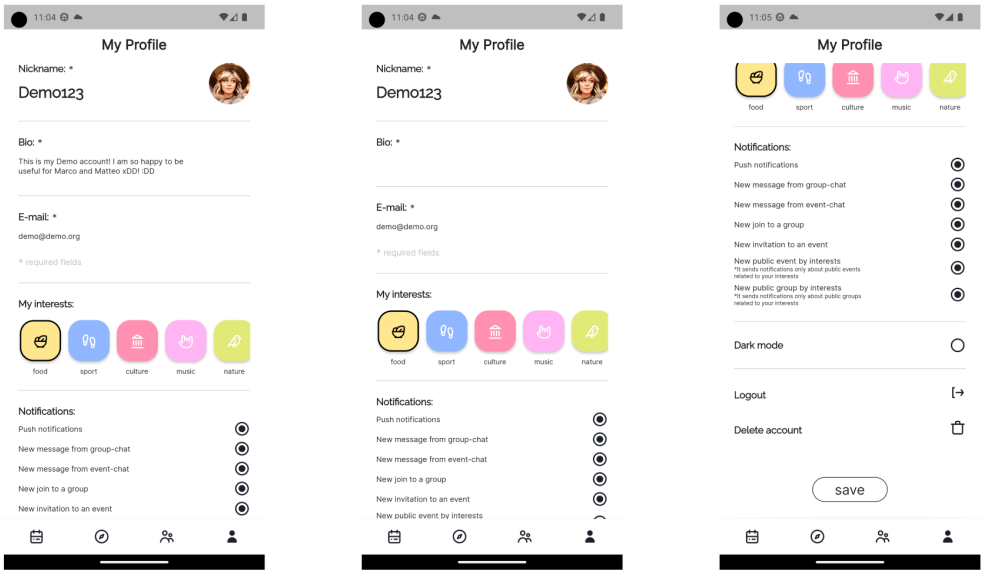


Figure 13: My profile

4.1.2 Tablet

This section highlights a subset of tablet wireframes to demonstrate the distinct design considerations for both landscape and portrait modes

Portrait and dark mode

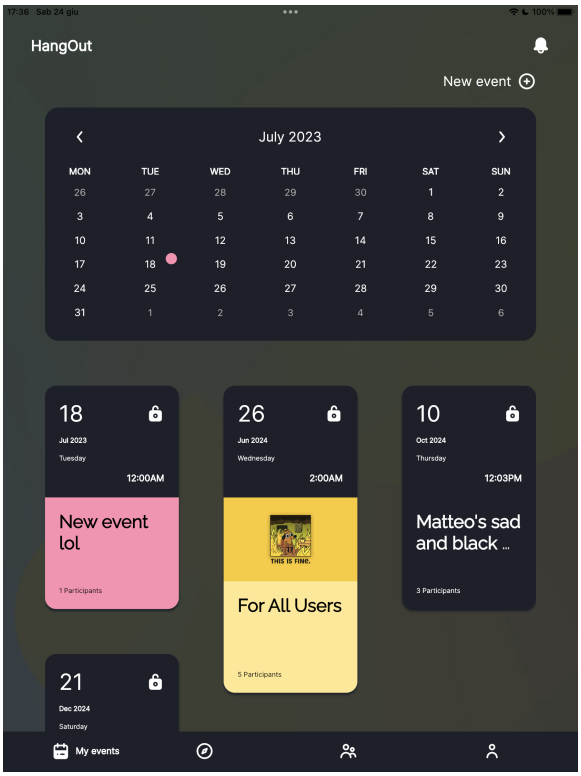


Figure 14: My events

Figure 15: Explore

Figure 16: My groups - chat

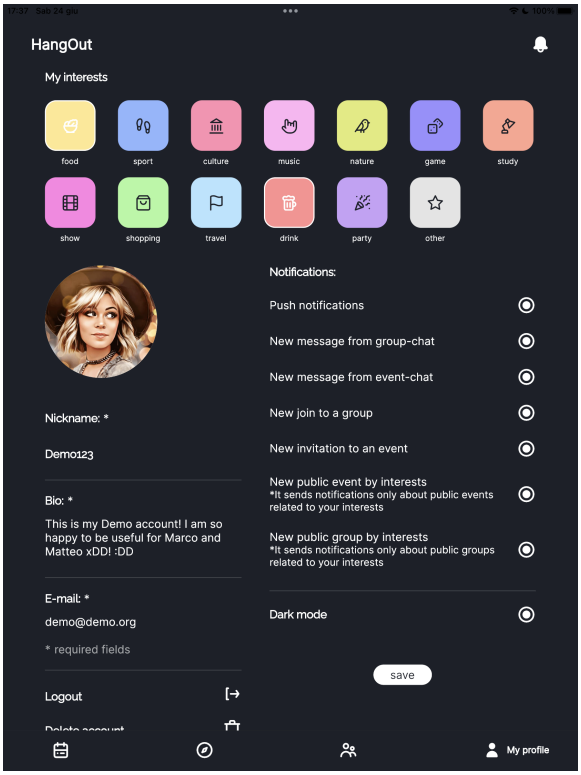


Figure 17: My profile

Landscape and dark mode

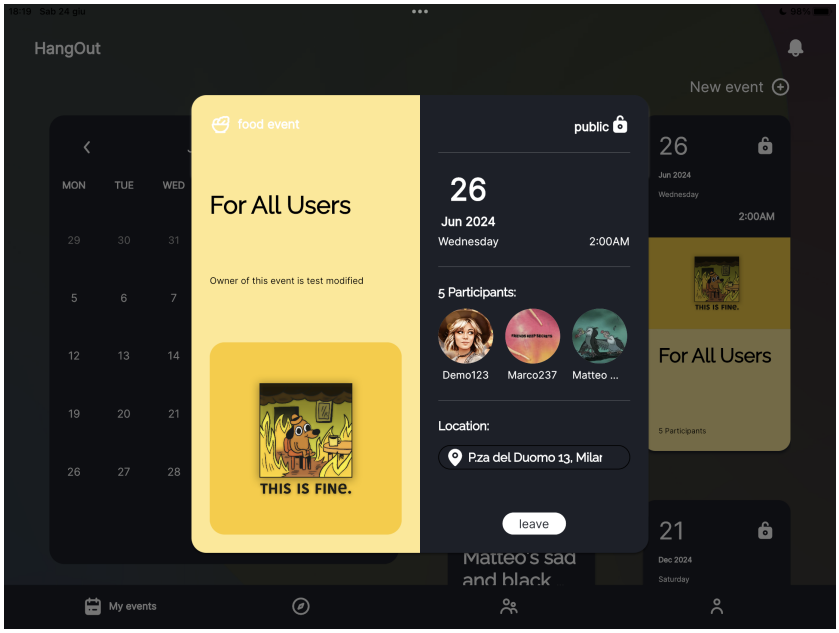


Figure 18: My events

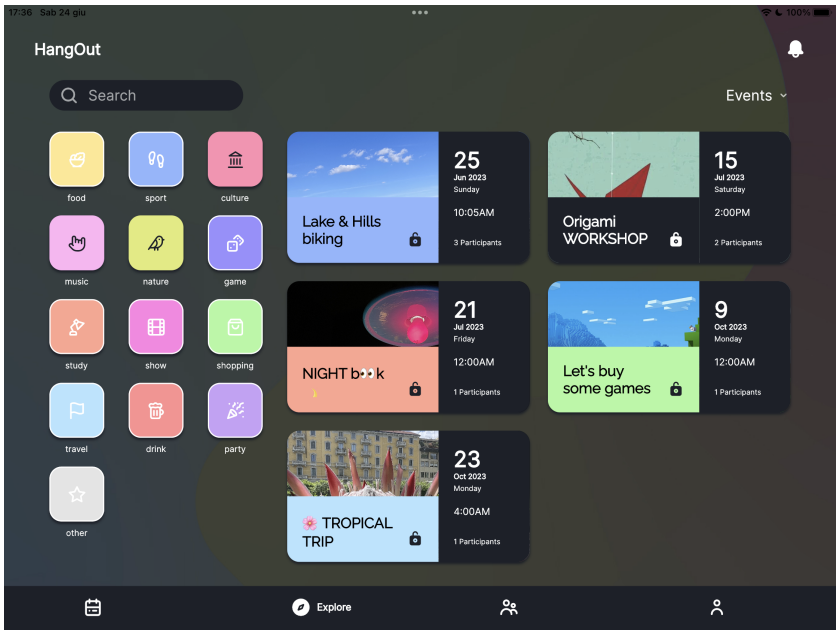


Figure 19: Explore

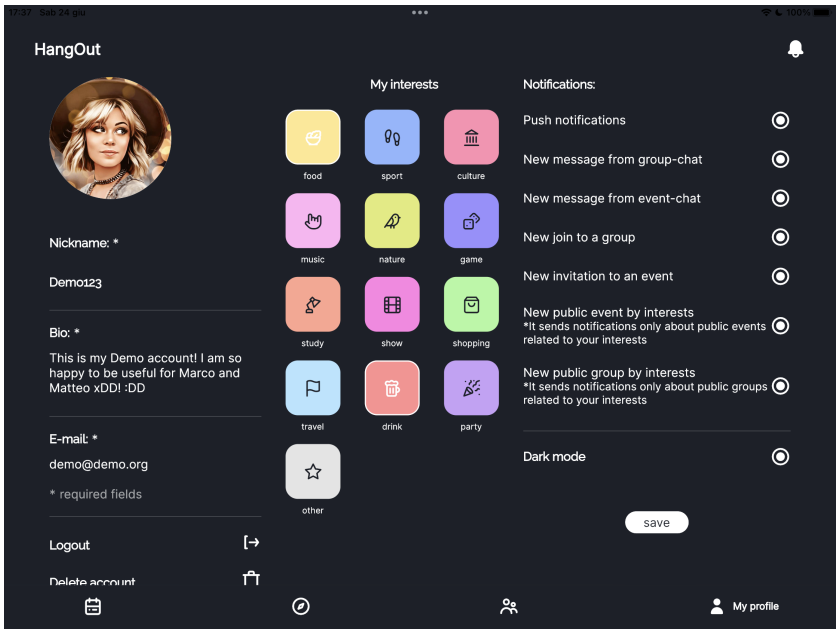


Figure 20: My profile

4.2 Visual Design

The application has been designed with a focus on maintaining a cohesive and visually appealing user interface. To achieve this, a set of consistent colors, fonts, and sizes were carefully selected and defined in a dedicated **Constants** file. This approach ensures that the entire application adheres to a unified visual style.

5 Information Architecture

5.1 Navigation Structure

This section provides a comprehensive overview of the user experience (UX) design for both phone and tablet devices. The diagrams include the most relevant buttons that distinguish the popups. These buttons are represented by red lines with a triangular shape at the top. The tablet UX diagram differs from the phone UX diagram in terms of the chat feature. While the phone has a dedicated chat page, the tablet utilizes a chat popup to optimizes the screen space.

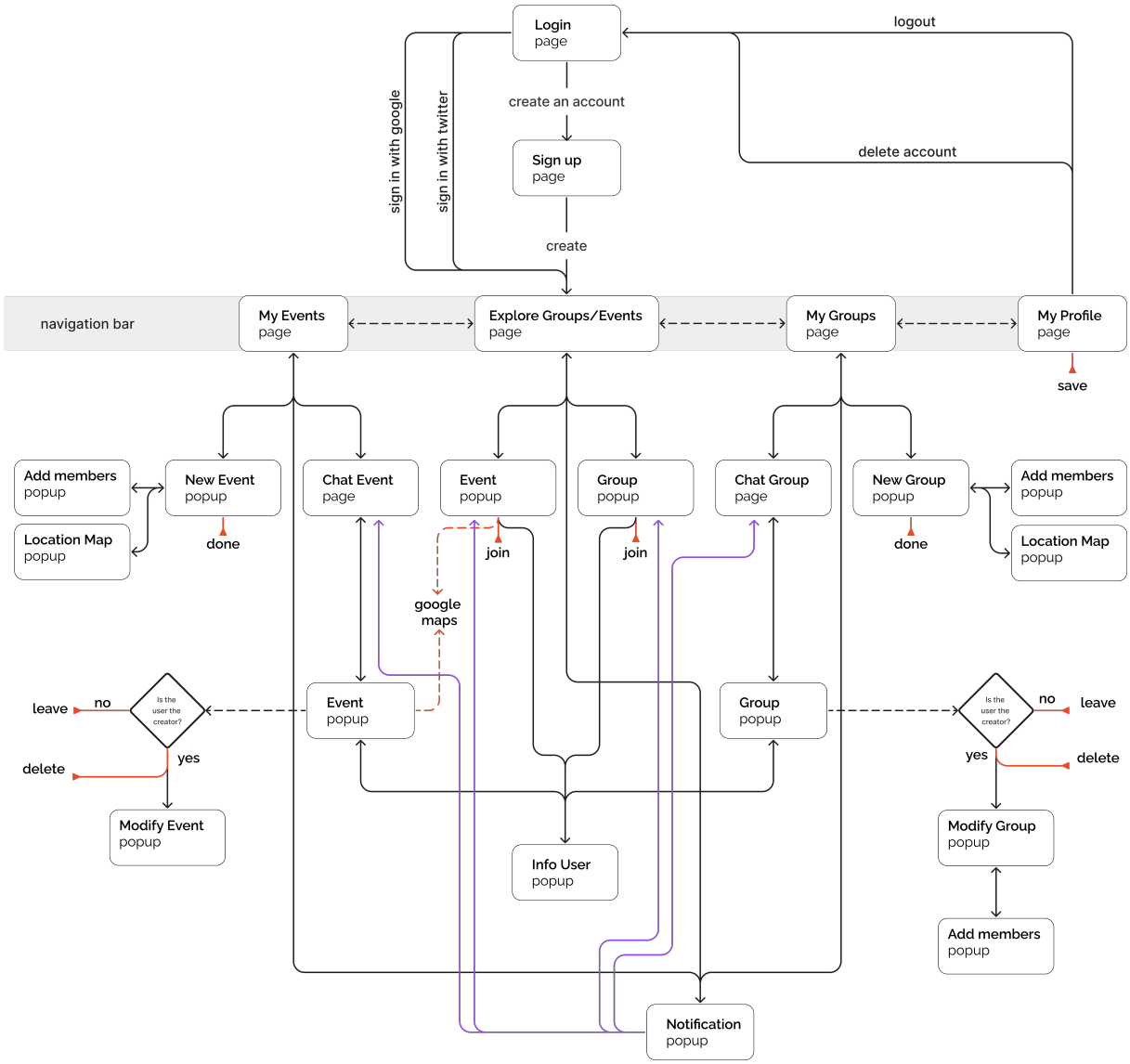


Figure 21: Phone UX diagram

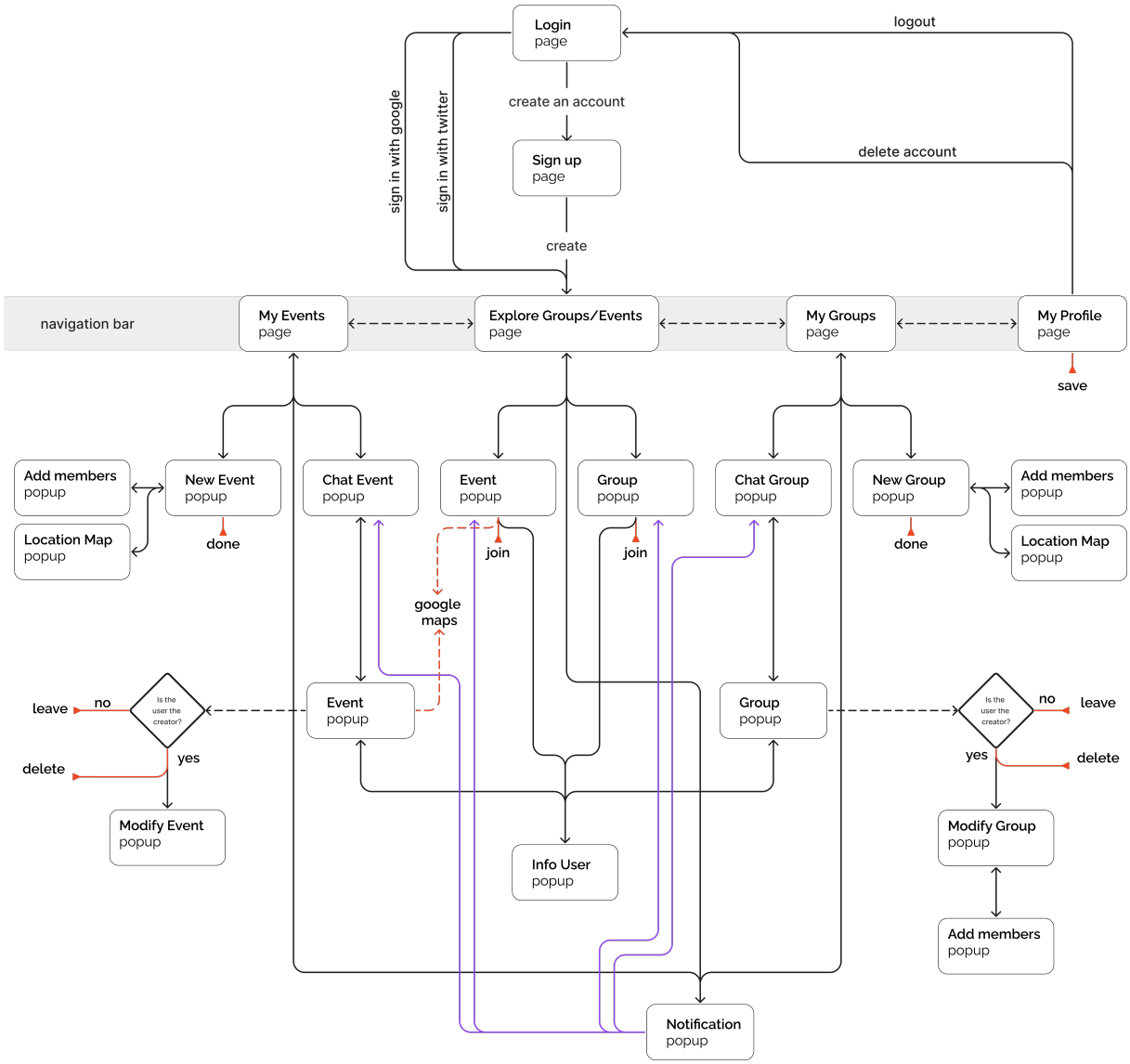


Figure 22: Tablet UX diagram

5.2 Content Organization

The Content Organization section of the app focuses on the systematic arrangement and categorization of content to ensure a coherent and intuitive user experience.

1. **Categorization of groups and events:** groups and events are organized into categories, enabling users to explore and discover content based on their specific interests. Categories may encompass diverse topics such as music, sports, or other relevant themes. This categorization facilitates efficient content discovery.

2. **My groups page:** to provide users with a centralized hub for managing their groups, the app features a dedicated *My Groups* page. Here, users can conveniently access all the groups they are part of and they can create new groups.

3. **My events page:** this page offers users an overview of the events they are attending. In addition, the page incorporates a calendar feature, enabling users to visualize their event schedule. Here, users can create new events.

4. **Explore page:** the *Explore* page serves as a platform for users to discover new groups and events they are not part of.

5. **My Profile page and Settings:** all user-related settings, including profile customization and app preferences, are conveniently accessible within the *My Profile* page.

6. **Notifications:** the notifications are all organized inside a dedicated popup to easily access and manage the app notifications.

6 Technical Requirements

6.1 Platforms

The HangOut app is designed to cater to users on both the iOS and Android platforms.

- **iOS:** the app is compatible with iOS devices, including iPhone and iPad. However, it should be noted that some features, such as push notifications, may not function fully on iOS devices due to the absence of an Apple developer account during implementation. Additionally, the application is not available for download from the App Store due to the same reason.

- **Android:** the app was developed and tested on the Android platform, ensuring compatibility and stability across a wide range of Android devices. The capabilities of the Android emulator has been leveraged during the development process and the app underwent integration testing on Android devices to ensure optimal performance.

- **Screen Orientation:** HangOut is designed to provide a consistent user experience in portrait mode on both phones and tablets. The app remains locked to portrait mode on phones, while on tablets, it offers the flexibility to rotate and also support landscape mode for optimal viewing.

6.2 Technology Stack

The mobile application has been developed using **Flutter**, a cross-platform framework for building native applications. Flutter allowed us to create an application that is compatible with both iOS and Android platforms. For development, we utilized **Visual Studio Code** as our integrated development environment (IDE), which offers a rich set of features for coding and debugging. In addition, we leveraged the **Android emulator** provided by Android Studio to test and run the application during the development process. The following Dart packages were utilized in the app, organized based on their similar functionality:

- **UI and Design:** *animated_text_kit, assorted_layout_widgets, cupertino_icons, font_awesome_flutter, googl_fonts, eva_icons_flutter, lucide_icons, modal_progress_hud_nsn, flutter_staggered_grid_view, persistent_bottom_nav_bar*;

- **State Management and Architecture:** *bloc, flutter_bloc, equatable, flow_builder, hydrated_bloc*;

- **Database and Cloud Services:** *cloud_firestore, firebase_auth, firebase_core, firebase_core_platform_interface, firebase_storage, firebase_messaging*;

- **Date and Time:** *flutter_datetime_picker, table_calendar*;

- **Image and File Handling:** *image_picker, cached_network_image, flutter_cache_manager*;
- **HTTP and Networking:** *http*;
- **Localization and Internationalization:** *flutter_osm_plugin, intl, geocoding*;
- **User Authentication and Sign-In:** *google_sign_in, twitter_login*;
- **Form Validation:** *formz*;
- **Notifications:** *flutter_local_notifications*;
- **Testing and Mocking:** *mocktail, bloc_test, fake_cloud_firestore, firebase_storage_mocks, firebase_auth_mocks, google_sign_in_mocks*;
- **Other Utilities and Packages:** *flutter_screenutil, url_launcher, pointer_interceptor, collection, json_annotation, path_provider, flutter_device_type*;

6.3 Architecture and Design Patterns

6.3.1 High-level system architecture and component interaction

The **Bloc** (Business Logic Component) pattern is a state management pattern that helps to clearly structure the applications and manage the flow of data. The Bloc architecture promotes a clear separation of concerns, dividing the app into distinct layers. This separation enables better maintainability, testability, and scalability of the codebase. Each layer has its specific responsibilities and dependencies, ensuring a modular and organized code structure.

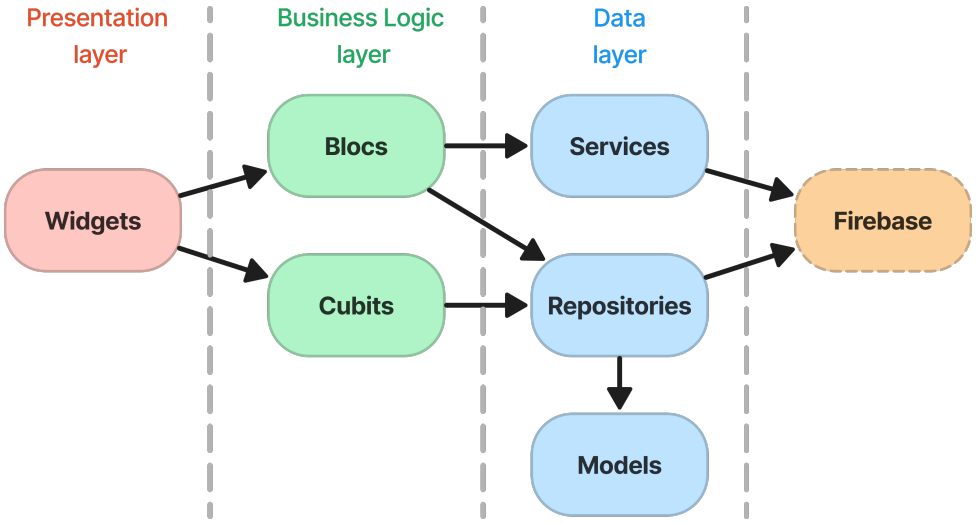


Figure 23: Components dependency

Data layer

The Data layer is responsible for handling data retrieval and manipulation from various sources, particularly from the **Firestore Database**. In our application, it consists of models and repositories. The models serve as blueprints for the data that the app interacts with, defining the structure and properties of entities such as groups, events, notifications, and more. Repositories are classes that fetch raw data and transform it into model instances. Here, data can be refined

through operations like filtering, sorting, and more. Finally, there is a single service, the Notification one, that is used to handle the push notifications and managing the communication between the app and **Firebase Messaging**, read more in the **Data layer - Services** paragraph inside the **In-depth components analysis** section.

Business Logic layer

The Business Logic layer encompasses all the Blocs and Cubits in the app. Its primary role is to bridge the gap between the user interface and the potentially volatile nature of the data layer. This layer is responsible for managing the application's logic, including handling errors that may arise from the data layer. By relying on repositories, the Blocs and Cubits retrieve data and provide it to the application.

Presentation layer

The Presentation layer focuses on everything related to the user interface. It encompasses widgets, user inputs, animations, and more. This layer is responsible for rendering the UI elements based on the current state of the Blocs. It receives data from the Business Logic layer and determines how to visually present it to the user.

6.3.2 In-depth components analysis

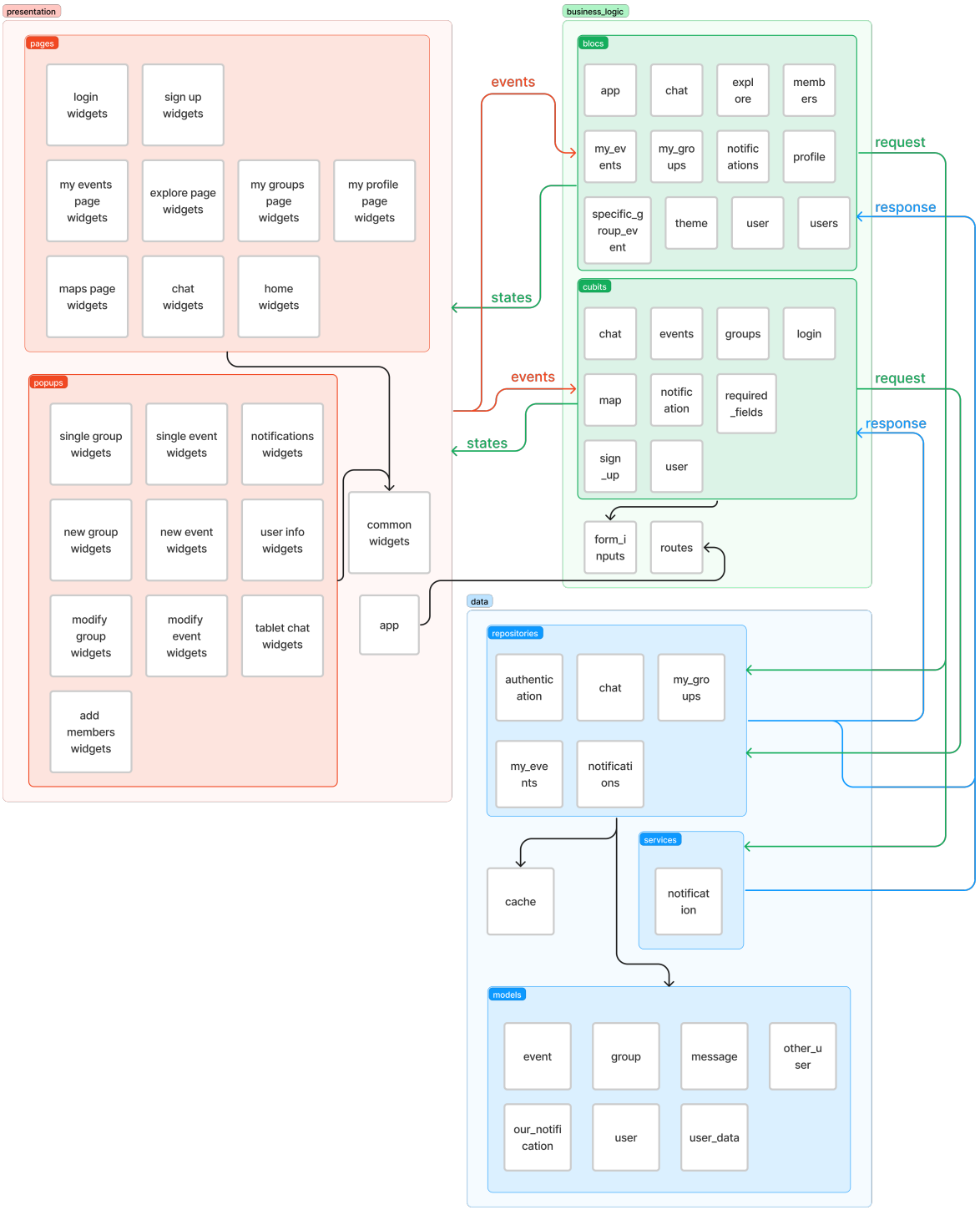


Figure 24: Code structure

Data layer - Models

Each model is equipped with two essential methods for facilitating queries in the Repositories. The first method takes a *DocumentSnapshot* as input and returns the corresponding object

instance, allowing for easy retrieval of the desired data. The second method returns a Map representation of the current class, providing a convenient way to access and manipulate the data within the model. These methods play a crucial role in streamlining data retrieval and manipulation processes within the Repositories.

- **Event** attributes: *id, name, category, numParticipants, photo, date, private, description, creatorId, location, locationName, members*;
- **Group** attributes: *id, name, creatorId, caption, numParticipants, isPrivate, interests, photo, members*;
- **Message** attributes: *timestamp, dateHour, senderId, senderNickname, senderPhoto, documentId*;
- **Other_user** attributes: *id, name, photo, description, interests.* This model serves the purpose of representing the participants or members associated with an event or group;
- **Our_notification** attributes: *userIds* (the ids of the members that have not yet seen the notification), *notificationId, thingToOpenId, thingToNotifyName, dateHour, timestamp, sourceName, eventCategory, chatMessage, public.* The *eventCategory* and *chatMessage* properties can be assigned an empty string value (""), indicating that the notification is not related to an event or does not pertain to a chat;
- **User_data** attributes: *id, name, email, photo, description, interests, notificationsPush, notificationsGroupChat, notificationsEventChat, notificationsJoinGroup, notificationsInviteEvent, notificationsPublicEvent, notificationsPublicGroup.* This model encapsulates all the data associated with the currently signed-in user;
- **User** attributes: *id, email, name, photo.* This model is utilized during the authentication phase to store and manage user authentication data.

Data layer - Repositories

In our application, the repository returns data as Streams instead of Futures. By utilizing Streams, the repository continuously provides a live stream of data, allowing the app to automatically update whenever there are changes to the underlying data source. This design choice ensures that the application remains up to date with the latest data, eliminating the need for manual app reloading by the user.

- **Authentication repository**: the Authentication repository is responsible for handling user authentication within the app. It manages the authentication process using email and password credentials and provides appropriate error messages when users input invalid or incorrect information. It also handles the integration with external providers for Single Sign-On (SSO) authentication. The repository takes care of saving user data in the database and managing important actions such as logout and account deletion. Additionally, it offers various methods to retrieve user information, events, and groups from the database.
- **Chat repository**: the Chat Repository is responsible for retrieving the chat messages of both events and groups, as well as adding new user messages to the database.
- **My events repository**: this repository manages event-related operations for a user. It handles saving, modifying, joining new events, retrieving participating and non-participating events, fetching specific event details, and facilitating event deletion and leaving.

- **My groups repository**: this repository handles various group-related operations for a user. It includes saving or deleting groups, modifying existing ones, retrieving participating and non-participating groups, joining and leaving groups, and fetching common or specific groups,
- **Notifications repository**: the Notifications repository is responsible for managing notifications in the application, specifically in the context of interacting with the database. Its main responsibilities include adding new notifications, removing users from the recipient list, deleting notifications when the recipient list is empty, and retrieving notifications that should be displayed to a specific user.
- **User repository**: the User repository is responsible for managing user information in the database. It handles various tasks related to user data, including retrieving data for specific users, modifying user information, and retrieving users who may be interested in a public notification based on shared interests.

Data layer - Notification Service

In our app, we utilize two distinct components for handling notifications: the notification repository and the notification service. The notification repository is responsible for managing notifications that are visible in the app's notification popup. It retrieves data from the database and converts it into our custom notification model. On the other hand, the notification service focuses on device-specific functionality and integration with **Firebase Messaging**. Its responsibilities include obtaining the device token from Firebase Messaging and storing it in the database; initializing the **Flutter Local Notifications Plugin**, which enables the app to display notifications on the user's device and configuring the Firebase Messaging object to listen for new notifications and triggers the appropriate actions when a notification is received. For example, when a chat notification is clicked, it opens the corresponding chat page, or when a public group notification is clicked, it displays the relevant group popup. Finally, it handles the sending of notifications by making a *POST* request to the **Firebase Messaging API**. It specifies the title, body, and token of the target device to ensure the notification is delivered to the intended recipient.

Business Logic layer - Blocs

- **App**: it manages the user-related actions in the app, such as user changes, logout requests, and account deletion;
- **Chat**: it manages the load of the group or event chat;
- **Explore**: its responsibility is to fetch and handle the retrieval of groups and events in *Explore* page;
- **Members**: it is responsible for handling various aspects related to the members of a group or event. It manages the retrieving and displaying of the list of existing members in a group or event, the retrieval of users to send invitations to join an event, as well as the loading of selected users for specific actions or interactions within the app;
- **My events**: it is responsible for managing the retrieval and loading of events specific to a user;
- **My groups**: it is responsible for managing the retrieval and loading of groups specific to a user;

- **Notifications**: it is responsible for managing the retrieval and loading of notifications that are relevant for a specific user;
- **Profile**: it manages the load of the common groups between two users;
- **Specific group event**: it is designed to handle the retrieval and management of all the relevant information for a specific group or event;
- **Theme**: a specialized **HydratedBloc** that manages theme selection and persistence, allowing users to customize the app's appearance and retain their chosen theme across sessions;
- **User**: it is responsible for loading user data and acting as an intermediary between the UI and the Notification Service;
- **Users**: it manages the load of all the users.

Business Logic layer - Cubits

- **Chat**: it manages the sending of new messages for groups or events;
- **Events**: cubits that manage the creation, the deletion and the modification of events;
- **Groups**: cubits that manage the creation, the deletion and the modification of groups;
- **Login**: it handles the login process, including authentication with credentials, login with Google and Twitter, and manages error messages associated with login attempts;
- **Map**: it manages the navigation inside the map page;
- **Notification**: it manages the creation and the deletion of the notifications;
- **Required fields**: this cubit manages the required fields during the creation or modification of a group or event. Moreover, it ensures that important information is not removed during profile modifications;
- **Sign up**: it handles the compilation of all fields during the sign-up phase and intermediates the creation of a new user account;
- **User**: it manages the modification of the user data.

Business Logic layer - Form inputs

We defined the login and sign up inputs using the Formz package. In this way we were able to assign a valid regex to each input for better control.

- **email:** accepts only inputs in standard email form;
- **name:** accepts only inputs with a Capital letter at the start and no special characters;
- **password:** requires a minimum security level for the user password;
- **confirmed password:** requires to be the same as the password field.

Business Logic layer - Routes

Here there is the primary routing logic of the app. It involves a function that generates a list of pages based on the state parameter provided. Within this function, a switch statement is utilized to evaluate the value of the **App bloc** state. If the state is determined to be *AppStatus.authenticated*, the function assembles a list consisting of a single page, namely the **HomePage**. Conversely, if the state is identified as *AppStatus.unauthenticated*, the function constructs a list comprising a single page, which corresponds to the **LoginPage**. This routing mechanism enables the app to determine the appropriate page to display based on the authentication status, ensuring that the user is presented with the relevant page at the appropriate time.

Presentation layer

The codebase is organized into several folders: *login*, *pages*, *sign_up*, *utils*, and *widgets*. The *login* and *sign_up* folders encompass all the widgets related to the login and sign-up pages, respectively. In the *pages* folder there are the pages that are accessible through the navigation bar. Each folder within the *pages* directory corresponds to a specific page and contains the essential widgets specific to that particular page. Additionally, any popups exclusive to that page can also be found within these folders. The *widgets* folder houses various custom widgets that have been developed for the application. This folder serves as an inventory for reusable components used across multiple pages. For example, we have implemented widgets for chat functionality, notifications, as well as popups related to groups and events.

7 Data Management

7.1 Database Services

We decided to implement our back-end using FireBase for it's multiple features easily integrated with flutter. In particular we utilized the following services:

- **FireStore:** is a NoSQL database which was a perfect fit for our app guaranteeing a scalable db to store the main data we needed. We created a series of collection which organization is described here.
- **FireStorage:** was used to store the images uploaded by each user. We created 3 main buckets named events groups and users, in each bucket related images are stored with an id equal to the event/group/user they are associated with for easy access and possible modifications later on.
- **FirebaseAuth:** provided an easy solution to handle user authentication and login via third party APIs.
- **Messaging:** was used to handle app notifications.

7.2 Database Structure

We divided our main FireStore database in 5 collections:

- **users:** contains one document for each user storing all his associated data. It has the following fields: *name,description,email,events,groups,interests,photo*,and a series of boolean fields for each notification type. The 2 fields *events* and *groups* in particular are arrays

containing the ids of each group and event the user is a part of, while the *interests* field is an array containing all the user interests like food or music;

- **events:** contains one document for each event storing all its associated data. It has the following fields: *name, id, numParticipants, photo, private, category, creatorId, date, description, location, locationName, members*; In particular the field members contains an array with all participating users ids.
- **groups:** contains one document for each group storing all its associated data. It has the following fields: *name, numParticipants, photo, private, caption, creatorId, id, interests, members*; In particular the field members contains an array with all participating users ids while the field interests contains the name of all the defined group interests;
- **notifications:** consists of individual documents, each representing a notification. Each document includes an *userIds* array field that contains the IDs of the users who should see the notification. The document also includes the following fields to cover various types of notifications: *chatMessage, dateHour, eventCategory, id, public, sourceName, thingToNotifyName, thingToOpenId, timestamp*;
- **tokens:** contains a document for each user with an array of tokens as its only field. Each token is associated with only one physical device and is used when a user needs to receive a new notification.

Both group and event documents may also have a sub collection named chat which has one document for each message sent in that specific chat. the document has the following fields: *senderId, senderNickname, senderPhoto, text, timeStamp, type, dateHour*. Some fields contain duplicated data which is useful to improve performance as described here.

8 Third-Party Integrations

8.1 APIs

Our app relies on several external APIs to implement different complex services.

- **Twitter API:** to allow users to login within the app using a twitter account;
- **Google API:** to allow users to login within the app using a google account;
- **Open Street Maps API:** is used to load and display locations when a user is creating or modifying an event position. It's a free alternative to google maps for this specific use case;
- **Google Maps API:** When a user clicks on an event location, our app launches the Google Maps app on their device, providing them with a direct path to reach the desired location.

9 Performance Optimization

Our app prioritizes delivering a visually pleasing user experience by incorporating high-quality images. However, we understand that repeatedly loading these images during navigation could negatively impact the user experience with unnecessary loading times. To mitigate this issue, we have implemented **several strategies**.

Firstly, we have leveraged the **device cache** to store every loaded image for a duration of 24 hours since the last load request. This approach allows us to retrieve the images quickly from the cache instead of initiating multiple loading requests, thereby enhancing the overall performance and reducing loading times.

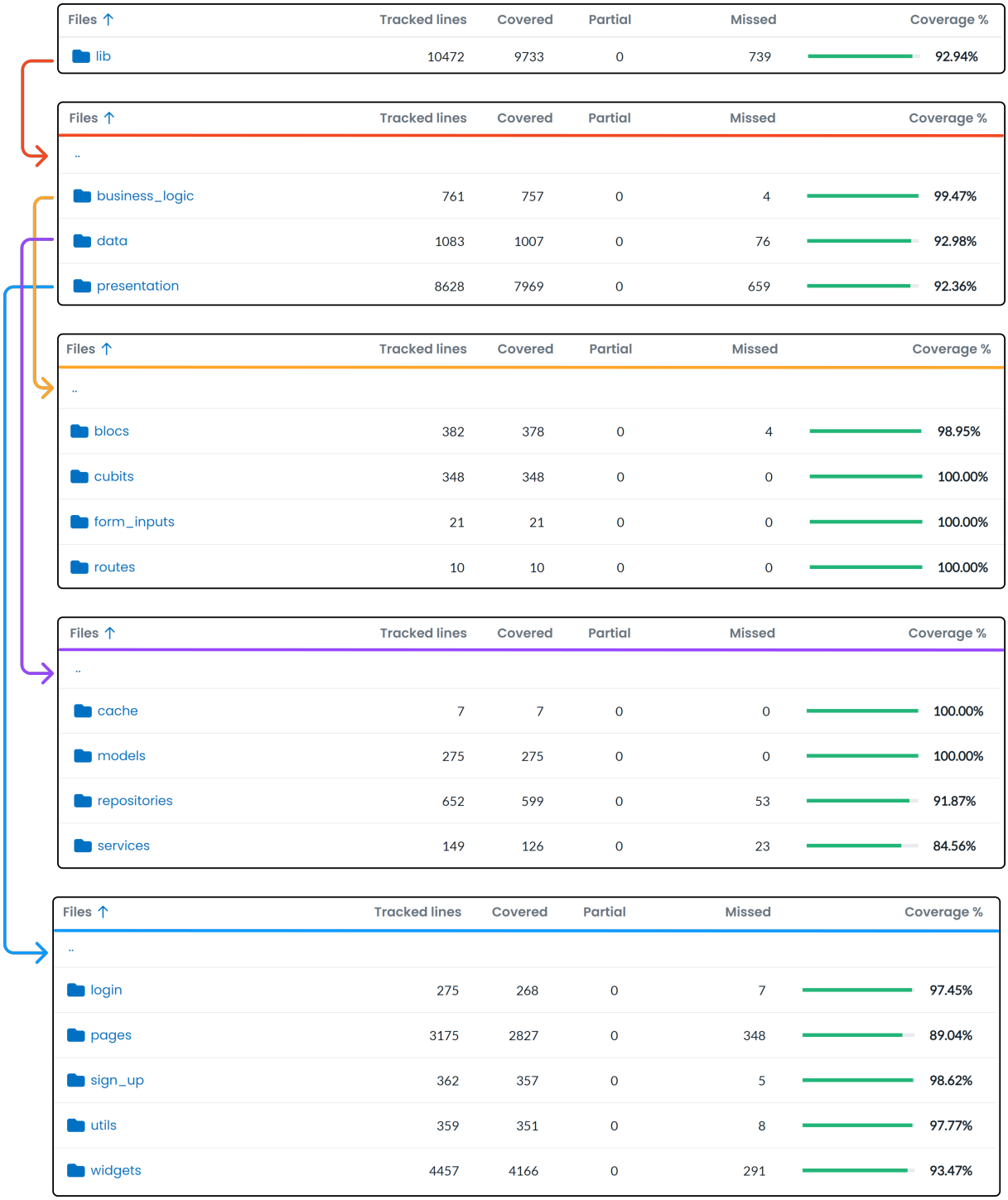
Additionally, we have implemented a **pre-caching** mechanism for background images during app startup. By preloading these images, we ensure that their loading is nearly instantaneous when they are required, further enhancing the smoothness of the user experience.

Considering the billing implications associated with Firebase, our chosen back-end, we have taken measures to minimize future costs. **Firebase** billing is calculated based on document reads by each user. To optimize this, we have designed our **queries** to be as lightweight as possible on document reads. For instance, we filter elements as early as possible in the query process and, if necessary, duplicate data strategically to minimize the need for frequent document reads.

By employing these strategies, we aim to enhance the performance and efficiency of our app, providing users with a visually appealing experience while keeping costs under control.

10 Testing and Quality Assurance

During the development process of our app, We implemented a comprehensive testing strategy to ensure its stability and reliability. To achieve a high level of confidence in the codebase, we conducted thorough testing using a combination of ***Unit***, ***Widget***, and ***Integration*** tests reaching 92.9% line coverage with 480 tests, guaranteeing robustness and minimizing the risk of potential bugs and errors.



Files ↑	Tracked lines	Covered	Partial	Missed	Coverage %
lib	10472	9733	0	739	92.94%

Files ↑	Tracked lines	Covered	Partial	Missed	Coverage %
..					
business_logic	761	757	0	4	99.47%
data	1083	1007	0	76	92.98%
presentation	8628	7969	0	659	92.36%

Files ↑	Tracked lines	Covered	Partial	Missed	Coverage %
..					
blocs	382	378	0	4	98.95%
cubits	348	348	0	0	100.00%
form_inputs	21	21	0	0	100.00%
routes	10	10	0	0	100.00%

Files ↑	Tracked lines	Covered	Partial	Missed	Coverage %
..					
cache	7	7	0	0	100.00%
models	275	275	0	0	100.00%
repositories	652	599	0	53	91.87%
services	149	126	0	23	84.56%

Files ↑	Tracked lines	Covered	Partial	Missed	Coverage %
..					
login	275	268	0	7	97.45%
pages	3175	2827	0	348	89.04%
sign_up	362	357	0	5	98.62%
utils	359	351	0	8	97.77%
widgets	4457	4166	0	291	93.47%

Figure 25: Codecov report coverage

10.1 Unit testing

By employing unit tests, we focused on verifying the functionality of individual components, ensuring that each one behaved as expected in isolation abstracting from the real UI implementation of our app. We conducted this tests on some type of components, in particular:

- **Blocs:** responsible for a big part of the state management logic in our app.

- **Data Repositories and Data Models:** responsible for the fetching and storing of the data received from our back end.

10.2 Widget Testing

Widget tests allowed us to simulate user interactions and validate the behavior of single UI elements, in our case Flutter Widgets. We also created a variety of bigger tests utilizing a greater number of widgets at once in order to simulate more complex interactions by the user.

10.3 Integration Testing

Finally, integration tests were employed to assess the integration and cooperation of various modules and services within the app as well as the connection with the database. We were able to test a great number of complex actions including:

- **login, logout and deletion** of multiple accounts;
- **create, join and leave** actions of multiple events/groups by different users;
- **editing** of events and groups data;
- **chatting** in groups and events.

10.4 Tools

To enhance the effectiveness of our testing efforts, we utilized a range of supplementary tools and technologies:

- **Mocktail** played a crucial role in our testing process by enabling us to simulate the behavior of components that were not directly under test. This allowed us to isolate the specific functionality we aimed to test without relying on the actual implementation of dependent components.
- **Firebase Local Emulator Suite** provided a realistic back-end while running our integration tests. This allowed us simulate the execution of our app in an environment as realistic as possible.
- **Codecov** was then used to generate an in depth report from the lcov.info file, providing us with a comprehensive overview of the code coverage achieved by our tests.

11 Project Timeline and Resources

11.1 Development Timeline

1. **Design Phase:**
    - Define app concept and requirements.
    - Create wireframes and prototypes.
    - Finalize visual design and user interface.
2. **Development Phase:**
    - Set up project structure and environment.

- Implement data layer and Firebase integration.
- Develop user authentication and authorization features.
- Build core functionality for events, groups, and notifications.
- Implement main pages.
- Implement chat functionality.
- Implement main popups.
- Develop user profile management features.
- Implement search functionality.
- Implement map functionality.
- Integrate push notifications.
- Implement tablet application.
- Conduct thorough testing and bug fixing.

3. **Refinement Phase:**
    - Gather feedback and conduct user testing.
    - Fine-tune user interface and user experience.
    - Address any performance or compatibility issues.
    - Perform additional testing and bug fixing.
    - Conduct integration testing.
4. **Deployment and Maintenance:**
    - Prepare app for deployment on Android and iOS platforms.
    - Handle app release and distribution.

11.2 Resource Allocation

During the development of the app, our team of two individuals maintained a high level of collaboration and coordination. We actively engaged in continuous communication, regularly scheduling calls and meetings to exchange ideas and share progress updates. We divided the workload by assigning specific features and tasks to each team member, ensuring a balanced distribution of responsibilities. However, it is important to note that our work was not isolated, as we frequently collaborated and provided assistance to one another whenever needed. This collaborative approach allowed us to leverage our combined skills and knowledge, resulting in well-informed decisions and a cohesive development process.

12 Deployment and Maintenance

During the development of our app, we decided to utilize Git and GitHub for version control. To further enhance our development workflow, we used the power of GitHub Actions to establish a reliable Continuous Integration/Continuous Deployment (CI/CD) flow.

12.1 Branch organization

To maintain an organized and efficient development process, we structured our repository with two primary branches:

- **main:** served as our production-ready branch, it contained the stable and tested codebase, which was ready for deployment to the live environment.
- **development:** was designated for ongoing development work. This branch allowed us to introduce new features, make improvements, and conduct tests in an isolated environment without affecting the stability of the main branch.

By adopting this branching strategy, we ensured a clear separation between our production-ready code and the ongoing development efforts.

12.2 CI/CD

We created 2 different GitHub Actions flows, the first one which execute all our tests every time a push is received on the development branch and the second and more complex flow is executed after a merge in the main branch is requested, it is divided in the following 4 jobs:

- **Test:** executes all tests on the codebase and updates the CodeCov token on our README with the new line coverage.
- **Version:** automatically extract the current app version from the repository commit history and updates the target build version.
- **Build:** builds the app and loads the updated app bundle as a release on GitHub.
- **Deploy:** uploads the final app loaded on the play store for the end users to download and enjoy.

This allowed us to create a secure environment were merges are not allowed if an error is detected by any of our tests and were deployment to production is fast and reliable.

12.3 Security

To successfully upload our app to the Google Play Store, it is crucial to sign the app during the build phase and have access to the team developer account. However, due to the sensitive nature of this data, it cannot be shared publicly, posing a challenge within our CD flow. To address this issue, we have implemented a robust solution utilizing GnuPG encryption. We securely encrypted a folder containing all the necessary files and incorporated it into our repository, ensuring the protection of sensitive information. During the flow execution a script is called which uses GitHub Actions Secrets to securely decrypt the files and moves them to the required locations within the project. This approach provides a robust security measure, ensuring that even if our repository became public, the encrypted files would remain inaccessible and unauthorized individuals would be unable to upload any app updates.

13 Conclusion

13.1 Next Steps

As the app evolves, there are several opportunities to enhance the user experience and expand its functionality. Some suggested next steps include:

- **Implement Advertising Integration:** Integrate advertising platforms or networks to display targeted ads within the app. This can provide an additional source of revenue and support for the app.
- **Sponsored Events:** Introduce a feature that allows companies or organizations to sponsor events by paying a fee. This could provide businesses with increased visibility and promotional opportunities while generating revenue for the app.
- **Enhanced Event Discovery:** Improve the event discovery capabilities by implementing advanced search filters, personalized recommendations based on user preferences, and location-based suggestions. This can help users find relevant events more easily.
- **Social Sharing:** Enable users to share events or group activities on their social media profiles, allowing for increased exposure and potential participation from their social network.
- **Event Ticketing and Payment Integration:** Integrate a ticketing system that enables users to purchase event tickets directly within the app. This can streamline the event registration process and provide a seamless payment experience.
- **User Ratings and Reviews:** Implement a rating and review system for events and groups, allowing users to share their experiences and provide feedback. This can help users make informed decisions and enhance the overall app community.