



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Drive or Drunk - Design Document

Design and Implementation of Mobile Applications

Authors:

Armando Fiorini (10709856)
Ahmet Eren Genis (11062471)
Christian Mariano (10770302)

Academic Year: 2024-25

Contents

Contents	i
-----------------	----------

1 Introduction	1
1.1 Purpose	1
1.2 Scope	1
1.3 Definitions, Acronyms, and Abbreviations	1
1.3.1 Definitions	1
1.3.2 Acronyms	2
1.4 Reference Documents	2
1.5 Packages Used – Application	3
1.6 Document Structure	3
1.7 Entity Relationship Diagram	4
2 Architectural Design	7
2.1 Architectural Style	7
2.2 Key Features	8
2.3 High-level Components and Their Interaction	9
2.4 Component View	11
2.4.1 Components Description	12
2.4.2 Model View Controller (MVC)	15
2.5 Deployment View	16
3 User Interface Design	19
3.1 Introduction	19
3.2 Mobile Application Interface	19
4 Requirements	35
5 Implementation, Integration and Testing	37

5.1 Unit Tests 37

5.2 Widget Tests 37

5.3 Integration Tests 38

5.4 Coverage Analysis 39

List of Figures 41

1 | Introduction

1.1. Purpose

This document outlines the technical design and implementation strategies adopted in the development of the *Drive or Drunk* mobile application. It covers the structure of both the front-end user experience and the supporting back-end services.

The document also illustrates the application's main features through dynamic interaction flows and interface organization, providing insight into how users engage with the system during typical usage scenarios.

1.2. Scope

Drive or Drunk is a mobile platform developed to offer a smart and socially responsible solution to the challenges associated with alcohol consumption during nightlife activities. Whether at clubs, parties, or other social gatherings, individuals often struggle with the decision between abstaining from alcohol to drive or risking impaired driving—with potentially dangerous outcomes.

The app connects two types of users: those who remain sober (designated drivers) and those who have consumed alcohol and seek a safe way to return home. The system is built around a community-driven model that encourages networking, mutual trust, and social exchange.

1.3. Definitions, Acronyms, and Abbreviations

1.3.1. Definitions

- **Designated Driver:** A sober individual who volunteers or agrees to drive others who have consumed alcohol, ensuring their safe return home.
- **Firestore:** A cloud-based NoSQL database provided by Firebase, used to store user

data and event-related information in real time.

- **Flutter:** An open-source UI software development kit created by Google, used to develop cross-platform mobile applications.
- **Firebase Authentication:** A service that allows secure user authentication using email, social logins, or anonymous access.
- **Base64:** A method for encoding binary data (like images) into text format, useful for storing and transferring media via databases or APIs.
- **Widget:** A reusable UI component in Flutter that defines part of the app interface and behavior.

1.3.2. Acronyms

- **API:** Application Programming Interface
- **HTTPS:** Hyper Text Transfer Protocol over SSL.
- **TLS:** Transport Layer Security.
- **SSL:** Secure Socket Layer.
- **DD:** Design Document
- **ER:** Entity-Relationship.
- **UI:** User Interface
- **DBMS:** DataBase Management System.
- **FCM:** Firebase Cloud Messaging
- **JSON:** JavaScript Object Notation
- **IdP:** Identity Provider.
- **OAuth:** Open Authorization.
- **OCR:** Optical Character Recognition

1.4. Reference Documents

- Flutter SDK Documentation: <https://docs.flutter.dev/>
- Firebase Documentation: <https://firebase.google.com/docs>

- Google Maps Platform Documentation: <https://developers.google.com/maps/documentation>
- Azure Face API Documentation: <https://learn.microsoft.com/en-us/azure/cognitive-services/face/>
- pytesseract Documentation: <https://pypi.org/project/pytesseract/>
- Slides from the course *Design and Implementation of Mobile Applications*.

1.5. Packages Used – Application

The *Drive or Drunk* mobile application was developed using the Flutter SDK and integrates a variety of third-party packages to support its core functionalities. These packages enable authentication, data persistence and external service integration.

- **firebase_auth**: Provides authentication methods including email and password login. https://pub.dev/packages/firebase_auth
- **google_sign_in**: Allows users to authenticate using their Google accounts. https://pub.dev/packages/google_sign_in
- **cloud_firestore**: Used for storing and retrieving structured user and event data. https://pub.dev/packages/cloud_firestore
- **google_maps_flutter**: Embeds interactive Google Maps with user markers and navigation. https://pub.dev/packages/google_maps_flutter

1.6. Document Structure

- **Section 1: Introduction**

This section introduces the purpose and scope of the *Drive or Drunk* application, along with a summary of the required functionalities. It also defines key terms, acronyms, and abbreviations that appear throughout the document, and outlines the document's structure.

- **Section 2: Architectural Design**

Targeted primarily at developers, this section provides a comprehensive overview of the system architecture. It begins by describing the adopted architectural style and the logical division of the application into layers. It then details the behavior and data flow of the main functionalities offered by the app.

- **Section 3: User Interface Design**

This part focuses on the user-facing components of the app. It includes mockups and diagrams illustrating the application's key screens and navigation flow, outlining how users interact with different features in the mobile interface.

- **Section 4: Implementation, Integration, and Test Plan**

The final section covers the implementation strategy and integration steps, followed by testing procedures. It includes an in-depth description of the core modules and explains how they are developed, tested, and validated to ensure proper functionality.

1.7. Entity Relationship Diagram

Figure 1.1 presents the Entity Relationship (ER) diagram representing the core data model of the *Drive or Drunk* application, as implemented in Firebase. The diagram captures the main collections (entities), their attributes, and the relationships among them.

The application is structured around five principal collections:

- **User:** Each user has a unique identifier along with profile information such as email, username, name, age, and profile picture. Additional fields include verification status and references to their favorite users and events, as well as the events they are registered to attend.
- **Event:** This collection stores metadata about events, including their name, location (as a geo-point), image, description, and scheduled date.
- **Conversation** and **Message:** Users communicate via conversations, which are linked to two users (**user1** and **user2**) and contain a list of messages. Each message references its sender, has a timestamp, and a flag indicating whether it has been seen.
- **Review:** Users can receive reviews from others in the form of textual feedback and ratings. Each review is typed (e.g., **driver** or **drunkard**) and references its author.

The relationships among the collections include:

- A **User** can be registered to multiple **Events**, and an **Event** can have multiple registered users (many-to-many).
- A **User** can mark other users and events as favorites, modeled via arrays of references.

- Each **Conversation** connects two **Users**, and each conversation includes multiple **Messages**.
- **Messages** are authored by a **User**.
- **Reviews** are written by a **User** and are associated with the reviewed **User**.

This data model supports a scalable and flexible backend, taking advantage of Firebase's document-oriented structure while preserving relational consistency through references and denormalization where appropriate.

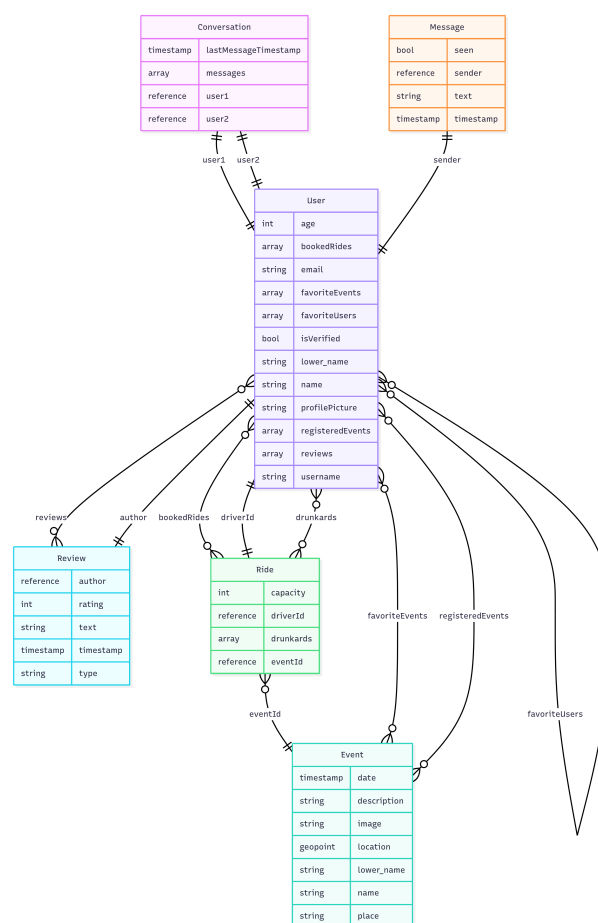


Figure 1.1: Entity Relationship Diagram of Firebase Collections

2 | Architectural Design

2.1. Architectural Style

The system is structured into three logical layers, with some cross-layer interactions due to the use of cloud-managed services.

- **Presentation Layer:** This layer corresponds to the Flutter mobile client, which handles the user interface and most user interactions. It communicates directly with Firebase services — such as Firestore for data access and Firebase Authentication for user login and session handling — reducing the need for an intermediate backend in standard flows.
- **Application Layer:** This includes both Firebase Authentication (which provides identity verification, token issuance, and session security) and a custom backend hosted on Render. The backend is responsible for managing tasks that cannot be safely or efficiently performed on the client, such as event fetching through the Ticketmaster API and facial recognition for ID verification.
- **Data Layer:** This layer consists of Firebase Firestore, which stores structured collections like users, events, ride history, and verification data. It supports real-time sync and is accessed directly by the client for most reads and writes, subject to Firebase security rules.

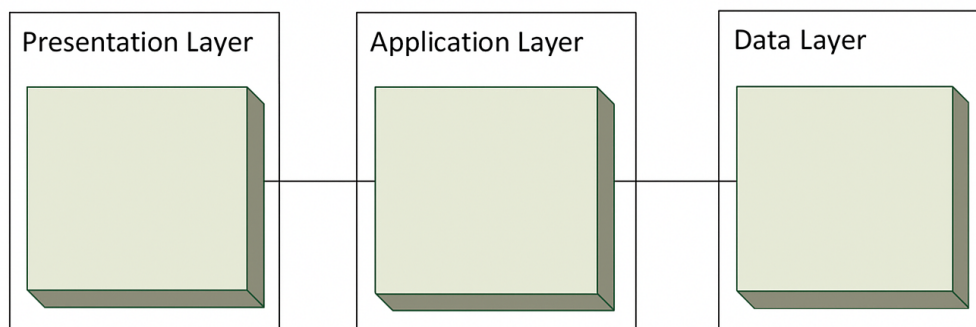


Figure 2.1: Black-box view of the *Drive or Drunk* system architecture

2.2. Key Features

The key features of the “Drive or Drunk” mobile application include:

1. **Find available drivers:** Users can find sober drivers or those who have not consumed alcohol nearby for a safe ride. - ****Booking mode**:** Users can book a ride in advance, allowing them to plan their evening with more peace of mind.
2. **Local event calendar:** Once a city is selected, users can access an interactive calendar with events scheduled daily (e.g., nightclub events, concerts, parties). By clicking on a specific event, all drivers available for that event are displayed.
 - All users can put events
 - Only verified users can fill up an event

3. **Event mapping:** Selected events can be viewed on a map, with the option to see available drivers nearby. This system helps users visually and intuitively plan their night by choosing the most convenient ride based on the event's location.
4. **Messaging area:** Users can communicate directly via the app to coordinate ride details, such as times and meeting points.
5. **Reviews and ratings:** Users can write reviews and rate drivers using a 0 to 5-star system. This helps ensure the quality of the service by highlighting the best drivers.
6. **User verification:** Users must register by entering their ID card information, which will be verified to ensure that only qualified users offer rides.
7. **Favorites:** Users can add events to their favorites to receive notifications if a driver becomes available for that event. This feature keeps users informed and ensures a safe ride for events of interest.

2.3. High-level Components and Their Interaction

The architecture of the *Drive or Drunk* application follows a layered client–cloud model designed for modularity, clarity, and extensibility. The client is a cross-platform mobile app developed in **Flutter**, where state is managed using the **provider** package. On the cloud side, the system relies primarily on **Firebase** for authentication and data persistence, while delegating specific high-complexity operations—such as facial verification, OCR, and third-party event integration—to a lightweight backend hosted on **Render**.

The Flutter application is organized using a **feature-first structure**, where each feature—such as authentication, events, rides, chat, or profile—encapsulates its screens, business logic, and state management. Providers are initialized in `main.dart` using **MultiProvider**, where core state objects like **AuthProvider**, **UserProvider**, and **Theme Provider** are injected before the app launches.

Routing is centralized in `lib/config/routes.dart`, and is dynamically generated through `AppRoutes.generateRoute`. This method uses the current authentication state to control access to protected screens such as the profile, chat, and ride history pages. The app uses a **declarative navigation model**, in which the rendered screen is determined directly by the user's current state, rather than being imperatively pushed or popped.

Upon successful login, the user is routed to the **NavigationMenu** widget `lib/navigation_menu.dart`, which manages the bottom navigation interface of the app. This widget

maintains the currently selected index and renders one of four main screens: `HomePage`, `EventsMapPage`, `ChatListPage`, and `ProfilePage`. Navigation is handled by a `NavigationBar`, and tapping a destination updates the index to show the corresponding screen within the scaffold's body. This structure provides seamless transitions between the app's core sections and acts as the main entry point after authentication.

The application follows a clear separation of concerns across the following macro components:

- **Models** (`lib/models/`): Contains plain Dart data classes representing domain entities such as `User`, `Event`, `Ride`, and `Conversation`, used throughout the app for serialization and business representation.
- **Services** (`lib/services/`): Encapsulates the app's business logic and external data access. `FirestoreService` manages CRUD operations on Firebase Firestore collections, while `GooglePlacesService` handles geolocation autocomplete via the Google Places API. Other services interact with REST endpoints on the Render backend for face verification and OCR tasks.
- **Features** (`lib/features/`): Provides user-facing functionality divided by domain (e.g., authentication, chat, events). Each module includes its own UI screens and connects to the appropriate services through providers and consumers. For example, the chat feature loads messages from Firestore and renders them with custom widgets.
- **Widgets** (`lib/widgets/`): Offers a wide set of reusable UI components including form fields, buttons, star ratings, Google Maps wrappers, and custom builders.
- **Utils** (`lib/utils/`): Contains general-purpose helper functions and utilities for formatting, validation, HTTP requests, and advanced features such as image-based identity checks.
- **Core** (`lib/core/`): Hosts application-wide constants, color themes, device info utilities, and global style definitions to maintain a consistent design across the app.

Authentication is handled by `AuthRepositoryImpl` and `FirebaseAuth`, while user identity and session state are maintained using the `UserProvider`. Theme switching between dark and light modes is controlled by `ThemeProvider`, and exposed through UI components like the `ThemeChangeButton`.

Overall, this architecture enforces a clear separation between presentation, logic, and data access, promoting reusability, testability, and scalability.

2.4. Component View

This section expands on the high-level architecture by analyzing the internal organization of all the macro components involved in the application: the arrows represent the navigation flow between the pages in the client view and the interactions with the model and all the external components. For readability reasons we modeled the the View-Model and Model-DB interactions since all the model components retrieve and load data in the respective collection in the DB and sends the data to the view where it is processed : the pages rendering logic is indeed embedded in the pages themselves and doesn't make use of a dedicated backend structure.

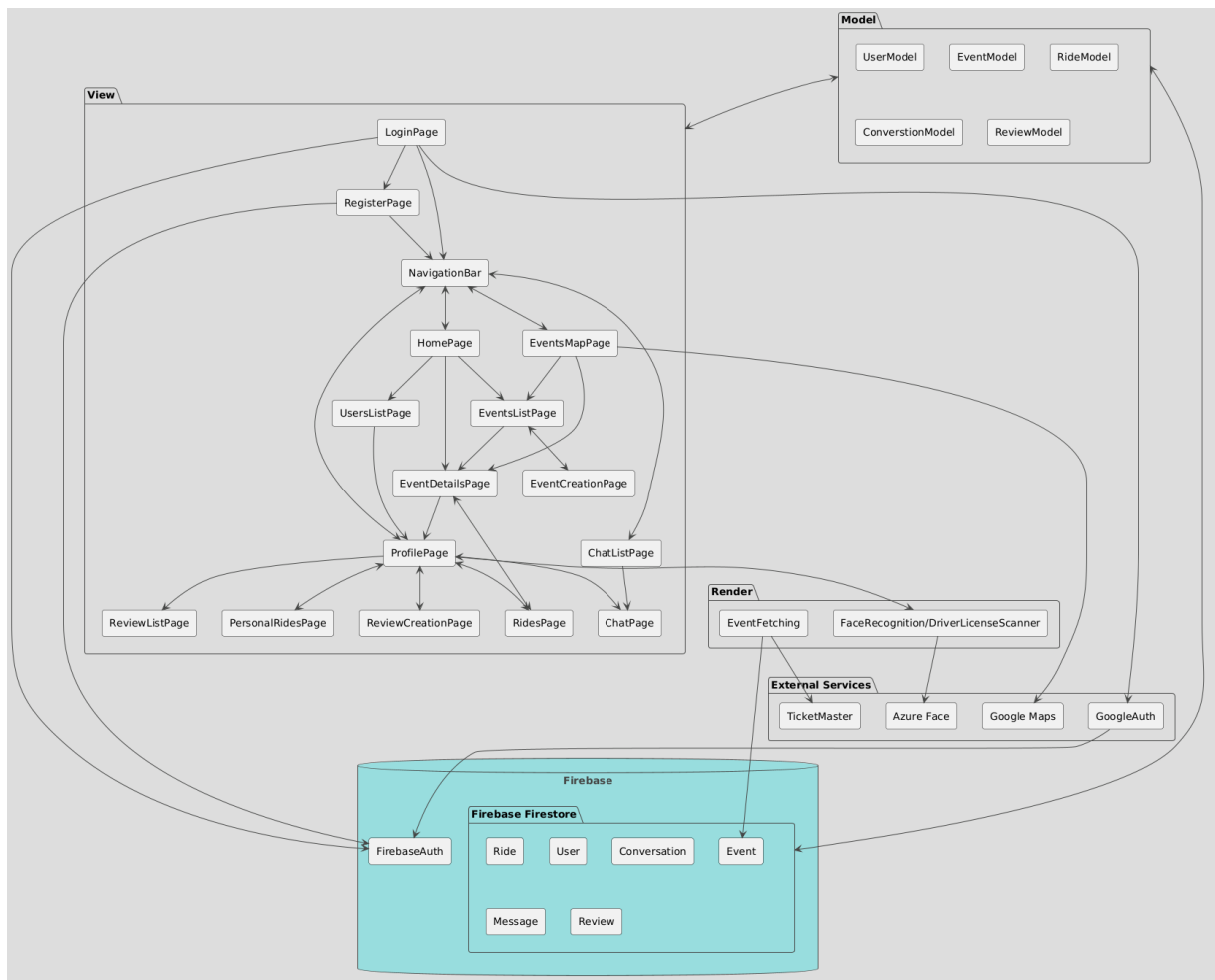


Figure 2.2: Architecture of the application

2.4.1. Components Description

The functionalities of the app are offered to the user through the interaction between local and remote components.

This paragraph expands the description of the single components as they are presented in the Component view of the app above.

View

The view includes all the pages the user can navigate through: in Drive or Drunk the view also embeds the controller logic since the rendering logics are implemented directly inside the pages and doesn't make use of dedicated backend components.

A detailed description of all the view components is provided below.

- **Login Page:** the Login Page is the first page shown to the user at the users: this page allows the user to log in using a DriveOrDrunk or a Google account (Google SSO login), to reset their password or create a new DriveOrDrunk account if they don't have one.
- **Register Page:** this page is shown to the user who wants to create a new DriveOrDrunk account: it implements the form where the user will insert all the necessary data for the creation of it (Name, e-mail, password)
- **NavigationBar:** when an user logs into the app, the Navigation bar is loaded: this component allows the user to move between the four main sections of the app: the Homepage the Events Map Page the ChatListPage and the Profile Page
- **Homepage:** the Homepage is the first page the user can see after accessing the app as it's the one loaded by the Navigation Bar by default. This page hosts the research functionalities of the app: through a tab the user can switch between Users and Events research. While the events can be searched for name place and date, only the name can be used to search for Users. In the trending section the user can have a view of the the most joined events in the database.
- **EventsMapPage:** thanks to this function, implemented through the integration with google Maps api, the user can search for any place in the map and see all the exact positions of the events available in that area, visualizable as pointers in the map or as a list in the EventsListPage
- **ChatListPage:** the chatlist page shows all the active chats between the current

user and other users in the app, clicking on a chat the **ChatPage** opens, which implements the actual messaging functions allowing the users to send messages to each other

- **ChatPage:** this page is shown when chatting with a user: it can be reached from the **ChatListPage** or from the profile page of an user through an apposite button. **ProfilePage** can also be reached from this page clicking on the name of the other user involved in the chat, visible in the page
- **ProfilePage:** This page is used to show the profile of a generic user, but, if accessed through the Navigation Bar, it shows the current user's personal profile. Through their personal page the user can modify his age and profile picture, see a preview or access the list (shown in **ReviewListPage**) of the reviews that other users left them, access the **PersonalRidesPage** to see the rides he has created as a driver and joined as a drunkard, and access the procedure to verify his profile through his driver license to be allowed to join events as a driver. This functionality is offered through a script in **Render**, that receives a picture of the user and of his driver license and relies on **Azure Face** to compare the two pictures.
- **ReviewCreationPage:** this page contains the form to create a review and can be accessed by the profile page of each other user in the system to leave him a review to rate the experience with that user as a driver or drunkard.
- **ReviewListPage:** this page is accessible from self and other users profile pages and offers a complete view of all the reviews received by that user. The two kinds of list (driver and drunkard reviews) are shown in two distincted reviewListPages accessed through different buttons in the profile page.
- **EventListPage:**)This page shows a list of events resulting from a research made from the **HomePage** or from the **EventMapPage**. When selecting an event from the results the user is redirected to their profile page, the list can be filtered by favorites
- **EventDetailsPage:** This page is accessed after the selection of one event from the homepage trending events section, an **EventListPage** or by Rides page when reached by an user's **profile page** (details below). It shows the name place and description of the event and allows the user to subscribe as a driver or drunkard to the event
- **EventsCreationPage:** In DriveOrDrunk users can also create their custom events: his page is accessible through a button on the **EventListPage** and it allows a user to create their personal event.

- **UsersListPage:** This page is shown as the result of an user research made using the homepage research form, when selecting an user from the results the user is redirected to their profile page, the list can be filtered by favorites
- **RidesPage:** this page is accessible by the `EventDetailsPage` and by the profile page of another user. In the first case it shows the list of the available rides for that event with info about the driver of each ride, whose profile page can be accessed tapping on the info box. In the second case instead, it shows all the rides that the user is offering as a driver, with info of each event: in this case the info box will redirect the user to the `EventDetailsPage` of that event.
- **PersonalRidesPage:** this page is shown when the user clicks on the available rides button on his personal profile page: it has two sections handled by a tab bar: the two sections show respectively the list of rides booked as a drunkard and created as a driver. In the first section for each ride informations about event and driver are available and linked to `EventDetails` and `Profile` page respectively of the driver and the event. In the second one only the one about the event are provided.

Model

The Model is composed by the User, Event, Ride, Conversations, and Review components. The model components is in charge of retrieving data from database and passing it to the view and viceversa.

Render

Render is a cloud service that

External Services

- **TicketMaster**
TicketMaster is used to retrieve data about events to be shown inside the application. Render periodically retrieves data from TicketMaster and pushes it into the database.
- **Google Maps**
Google Maps is used in the `EventsMapPage` to show the Map with the position of events in each selected area
- **Google Auth**
Google Auth is used to allow SSO login for users using their google account.

- **Azure Face**

Azure Face is used by render to perform face recognition during the identity verification procedure

Firestore

Firestore is a Baas (Backend-as-a-Service) for web and mobile applications development. For our purpose we used only two services among the ones offered.

- **FirestoreAuth**

This service offers automatization of the procedure of user authentication, all the created users are saved on the authentication service storage and synchronized with the User collection in the database.

- **Firestore cloud Firestore**

Through this service Firestore offers a NoSQL database for our application which contains all the data the application needs. The collection contained in the database are User, Event, Review, Conversations, Messages and Rides.

2.4.2. Model View Controller (MVC)

Model View Controller (usually known as MVC) is a software design pattern commonly used for developing user interfaces. It divides the related program logic into three interconnected elements to separate internal representations of information from the ways information is presented to and accepted from the user.

These three components are:

- **Model:** The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic, and rules of the application.
- **View:** Any representation of information that will be displayed to the end user.
- **Controller:** Accepts input and converts it to commands for the model or view, this component in Drive Or Drunk is actually integrated in the view, which encapsulates all the rendering logics.

2.5. Deployment View

Figure ?? shows the deployment topology of the *Drive or Drunk* system. The diagram illustrates how the system's components are distributed across various devices and cloud platforms. Each node represents a logical or physical entity, along with its underlying operating system or hosting service.

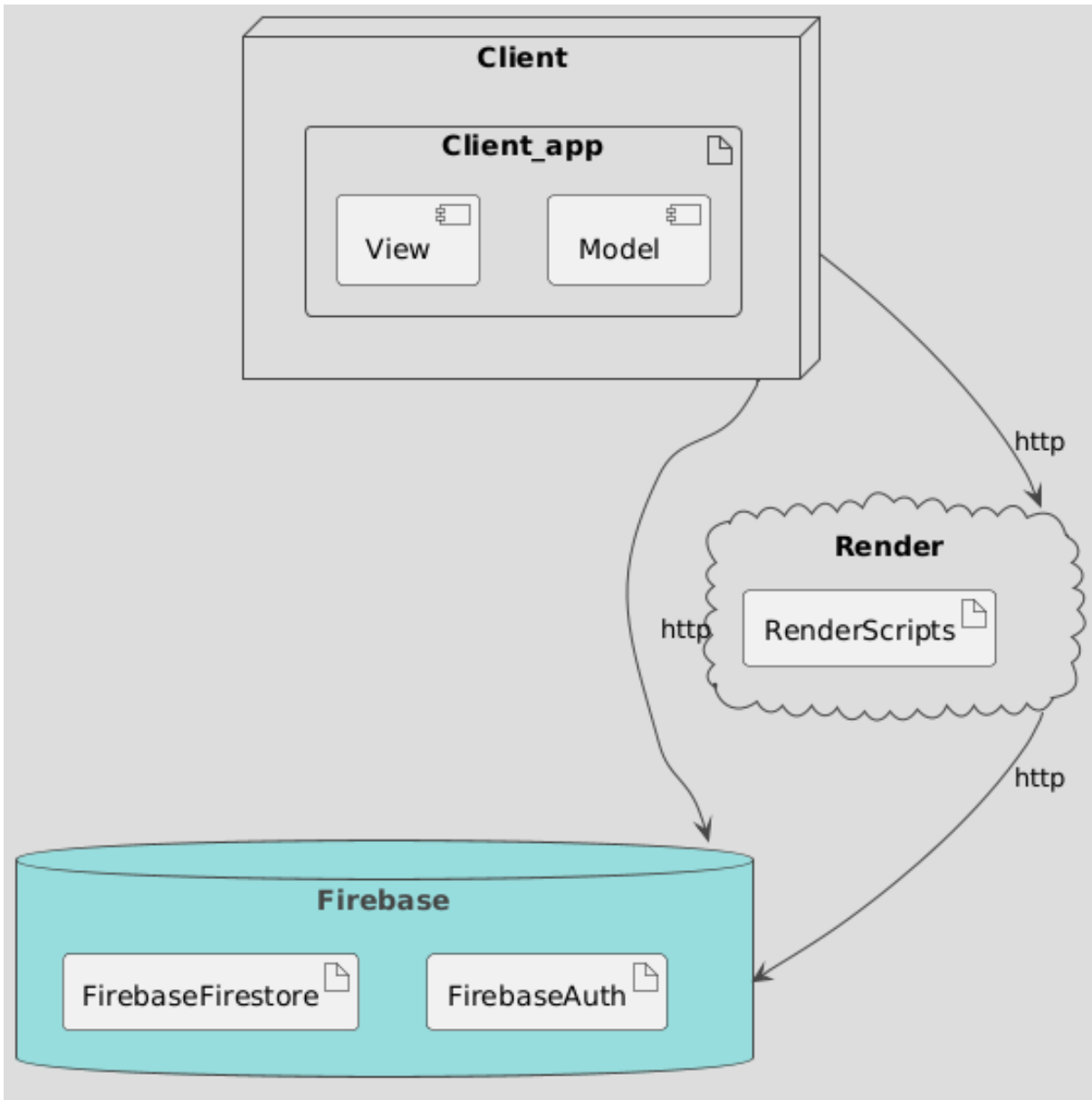


Figure 2.3: Deployment view of the application

- **Tier 1 – Client Device:** The client device is a smartphone running Android or iOS, where the Flutter application is installed. This app handles user interaction, UI rendering, and communicates directly with both Firebase services and the backend

server using secure HTTPS requests.

- **Tier 2 – Firebase Authentication:** This tier is hosted in the cloud and provides user authentication services. It is responsible for managing sign-in operations, including Google Sign-In, and issuing secure access tokens used by the mobile app to authenticate requests.
- **Tier 3 – Render Backend Server:** This tier hosts a custom backend deployed on Render. It handles application-specific logic that cannot or should not be executed on the client, such as facial verification, image processing, and fetching external events from the Ticketmaster API. It communicates with both the client and Firestore securely via RESTful interfaces.
- **Tier 4 – Firebase Firestore Database:** This cloud-based NoSQL database stores persistent information such as user profiles, events, ride requests, and verification metadata. It is accessed by both the mobile client and the backend through Firebase's secure SDK and API interfaces.

3 | User Interface Design

3.1. Introduction

The aim of this section is to show the design of the main screens of the user application, describing the flow of the main functionalities for which it is intended. The flow is created according to specific and illustrated input from the end user.

3.2. Mobile Application Interface

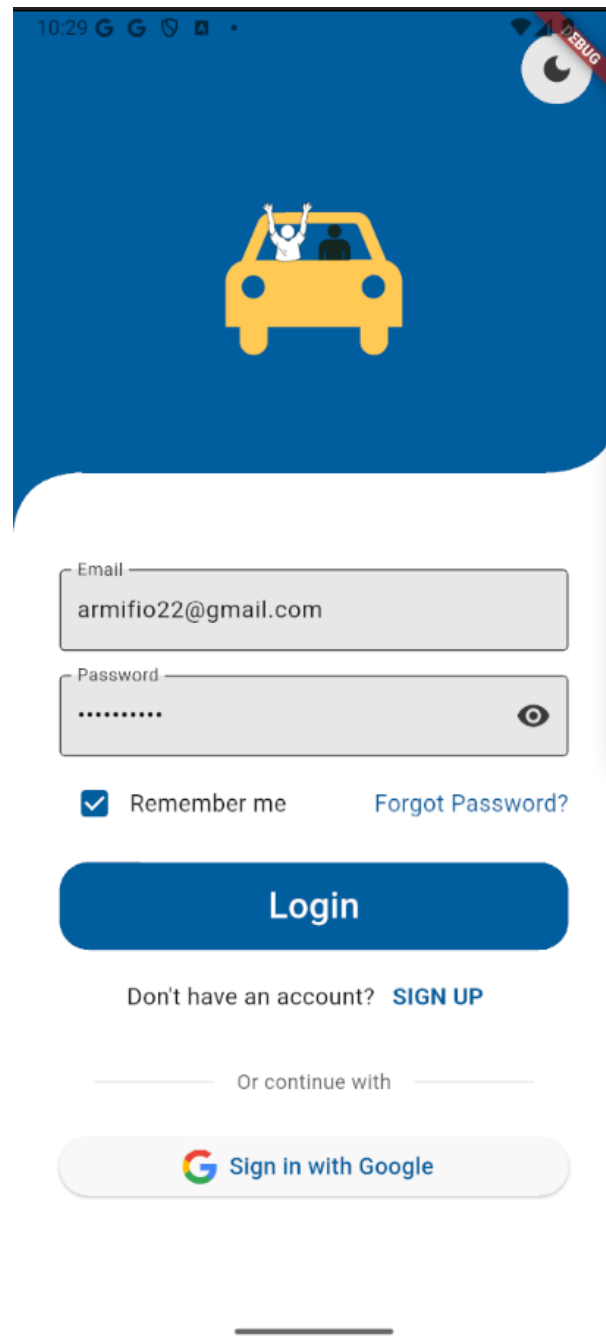
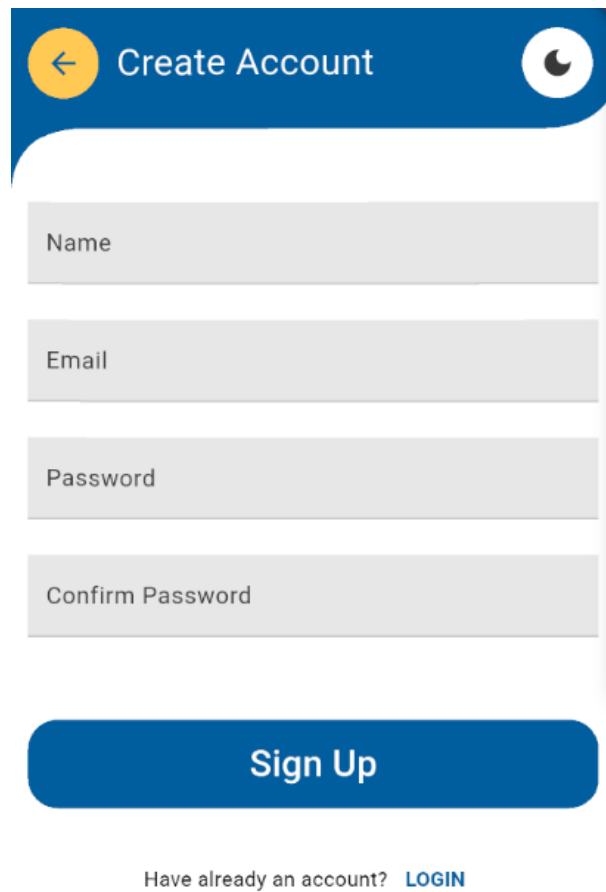


Figure 3.1: Login Page



The image shows a mobile app registration screen. At the top is a blue header bar with a yellow circular back button on the left, the text "Create Account" in the center, and a white circular profile icon on the right. Below the header are four light gray input fields stacked vertically, labeled "Name", "Email", "Password", and "Confirm Password". At the bottom of the form is a large blue button with rounded corners labeled "Sign Up". Below the button is the text "Have already an account?" followed by a blue link labeled "LOGIN".

← Create Account

Name

Email

Password

Confirm Password

Sign Up

Have already an account? [LOGIN](#)

Figure 3.2: Registration Page

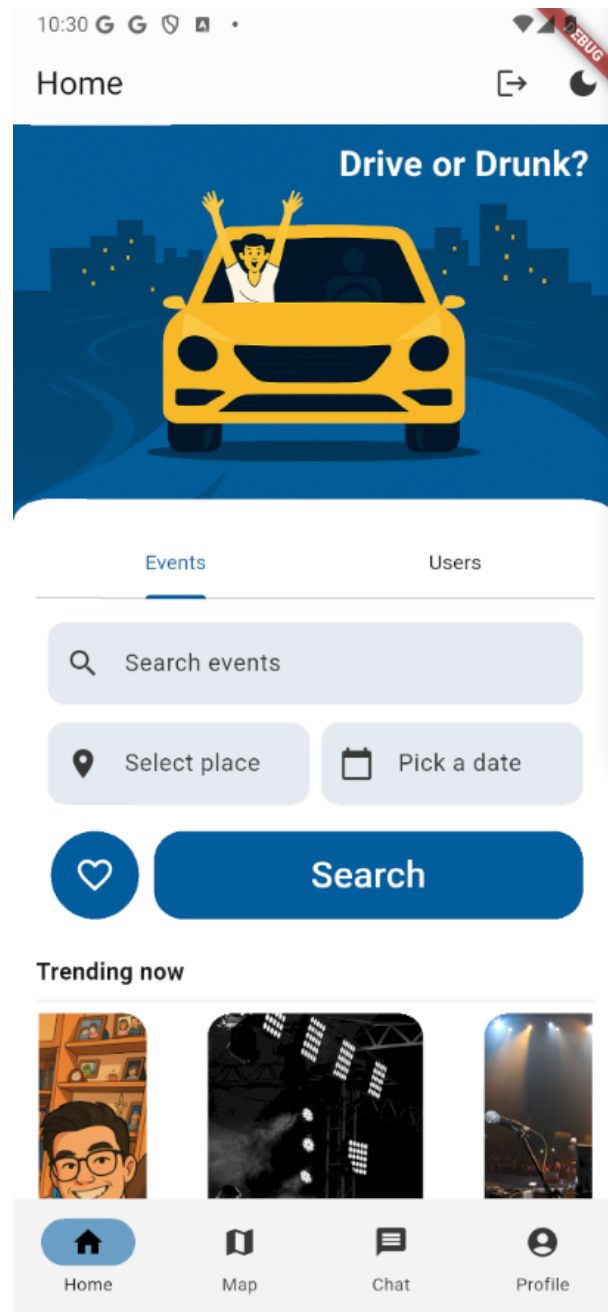


Figure 3.3: Homepage

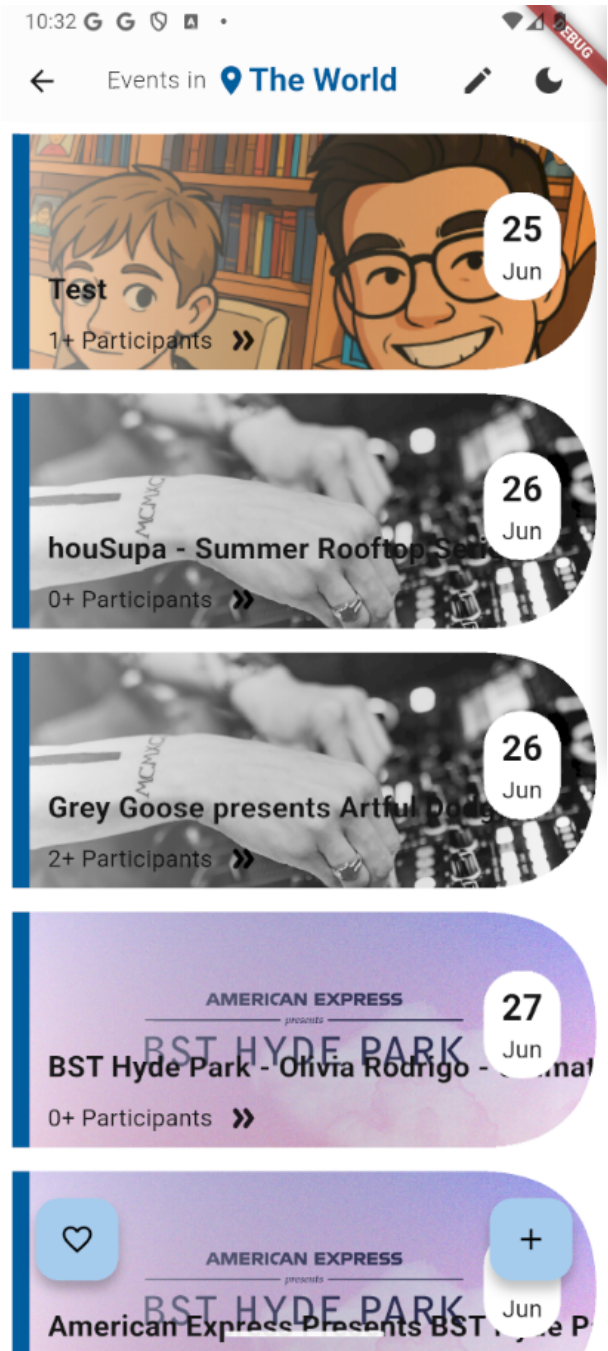


Figure 3.4: Event List Page

11:25 G G G G •

← Create New Event

Event Title*

Event Description

Event Date*

Event Place

Select Image

Save Event

Figure 3.5: Event Creation Page

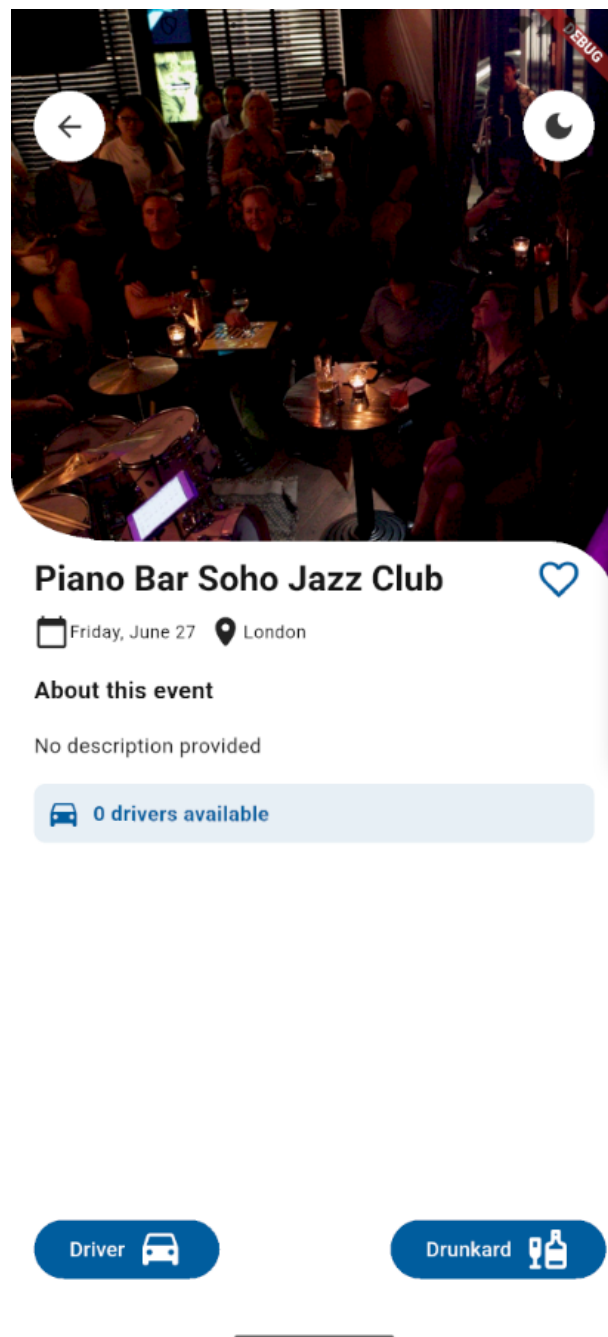


Figure 3.6: Event Detail Page

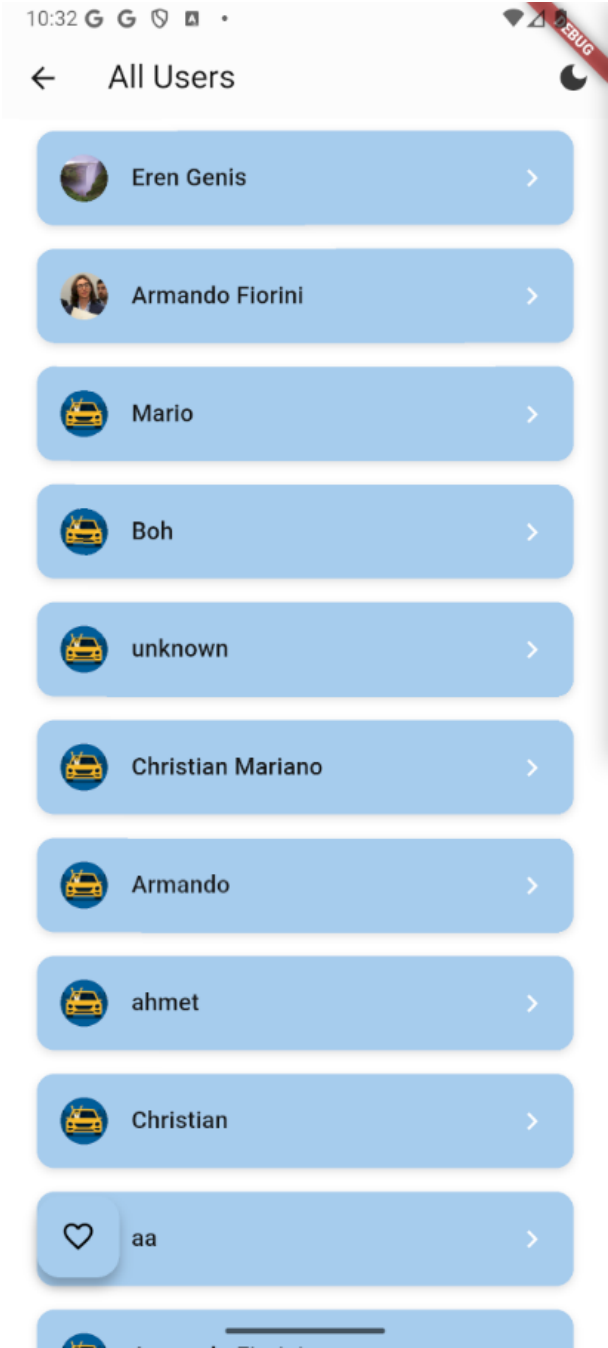


Figure 3.7: Users List Page

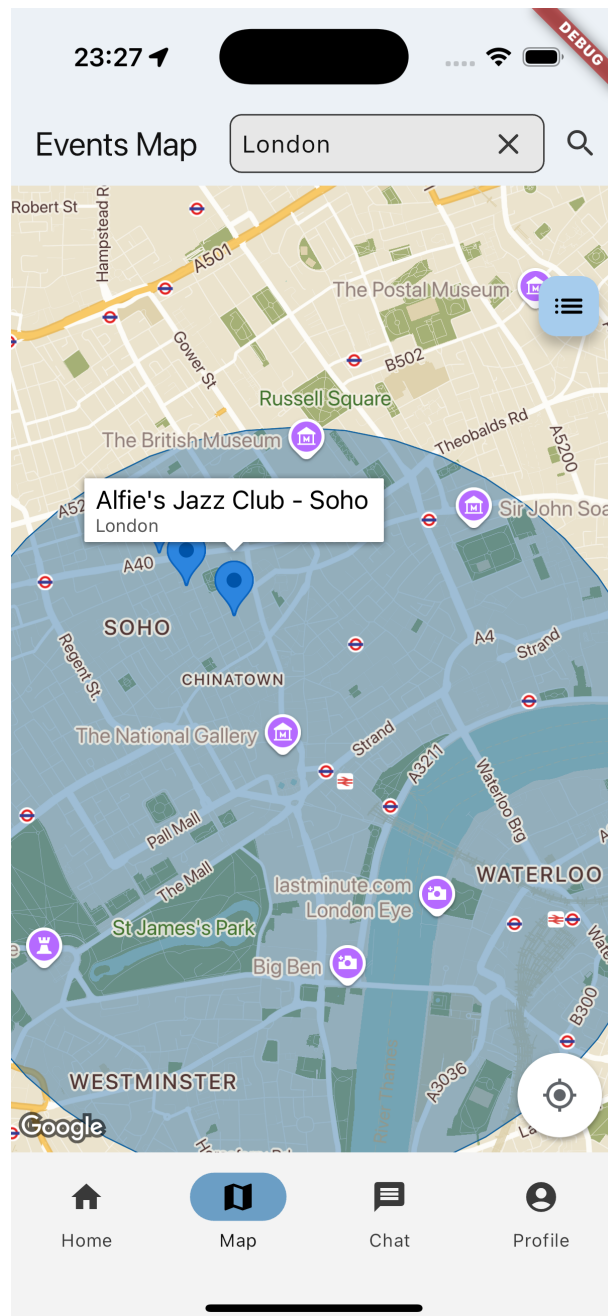


Figure 3.8: Events Map Page

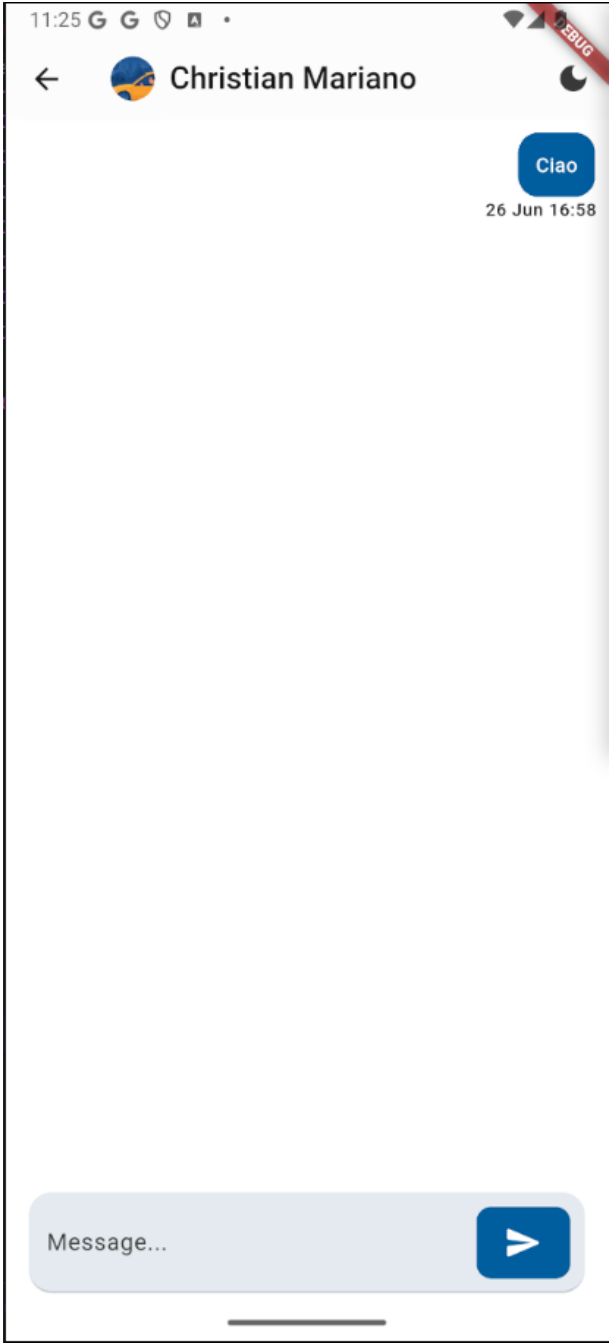


Figure 3.9: Chat Page

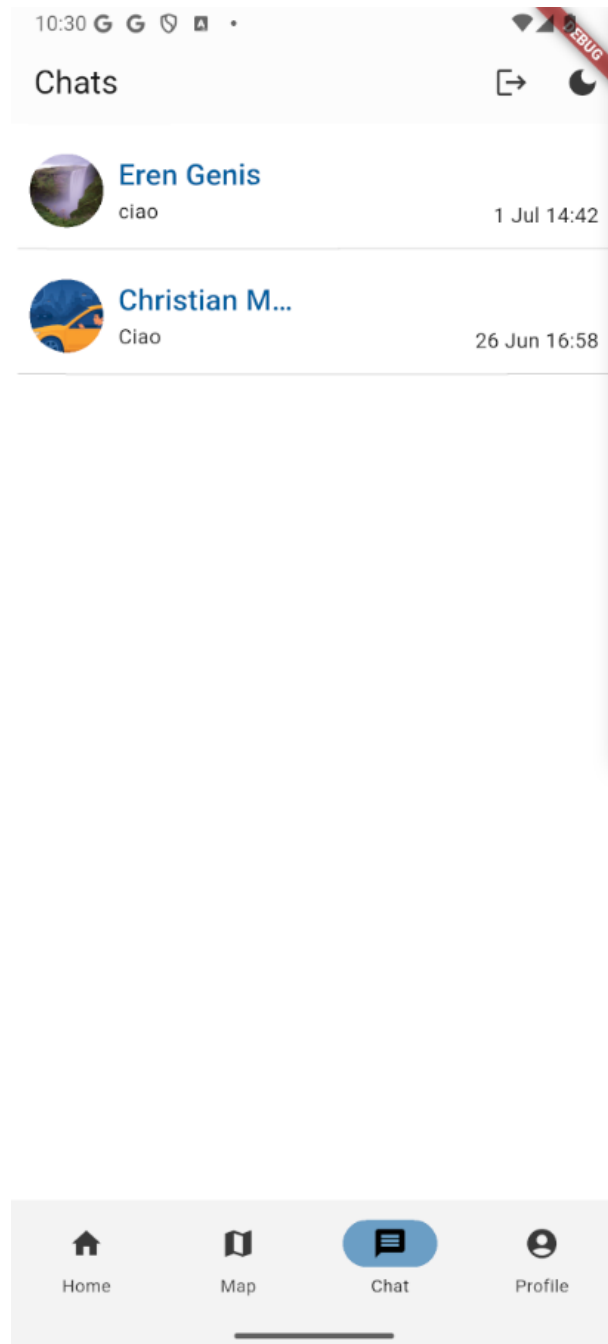


Figure 3.10: Chat List Page

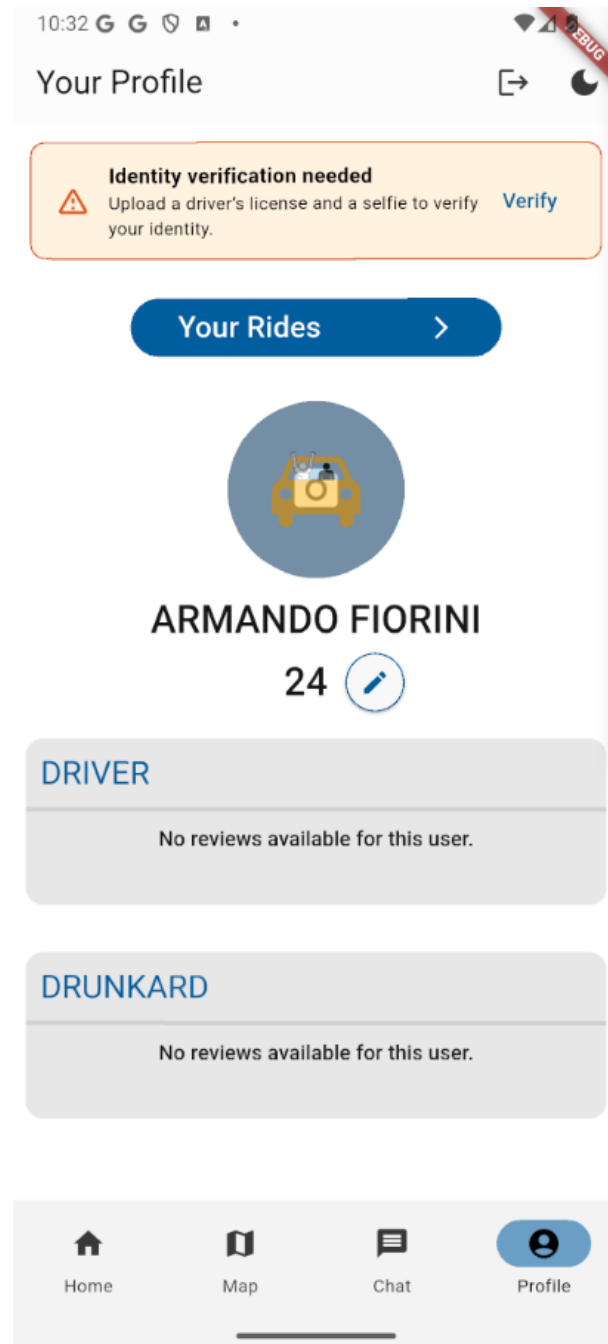


Figure 3.11: Profile Page

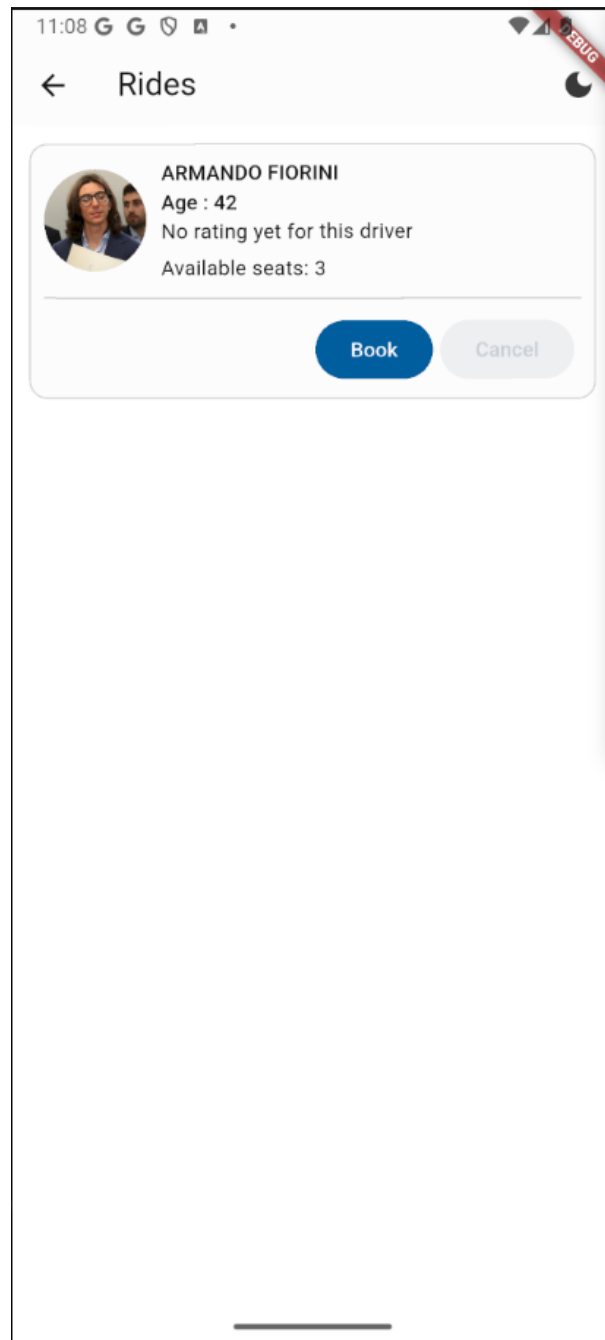


Figure 3.12: Rides from Profile Page

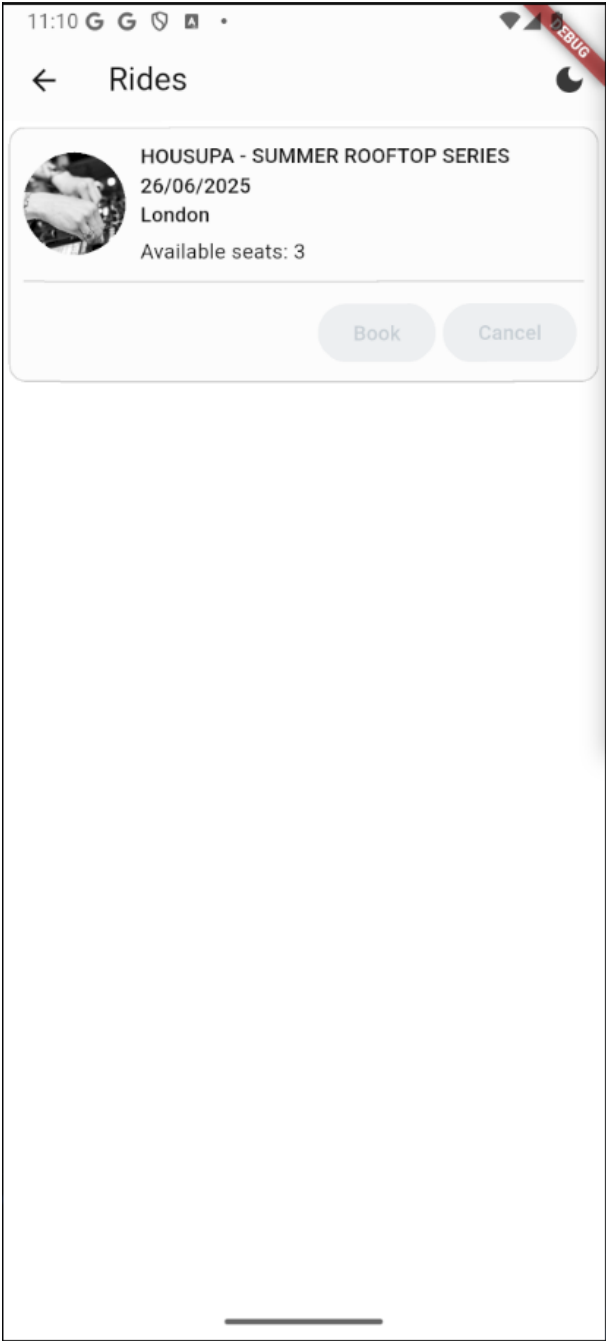


Figure 3.13: Rides from Event Page

11:09 G G G A •

← Create Review

Rating: ☆ ☆ ☆ ☆ ☆

Text:

Figure 3.14: Review Creation Page

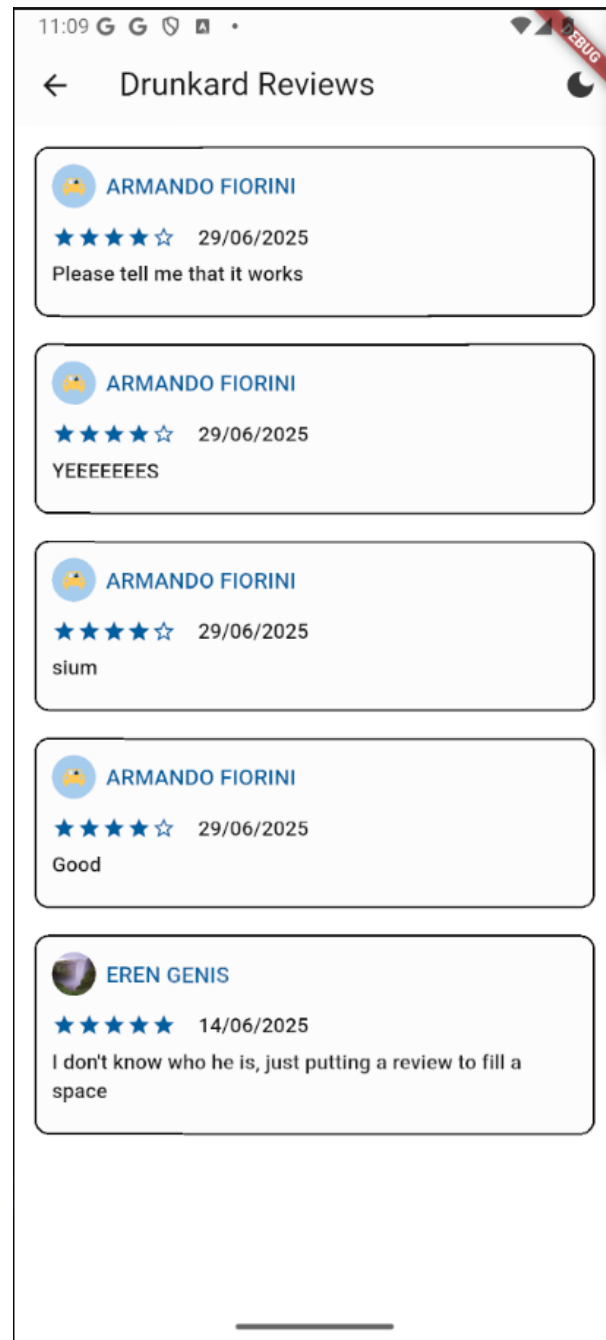


Figure 3.15: Reviews Page

4 | Requirements

The following list contains the requirements that the application should satisfy, together with a brief explanation of them.

1. User Registration and Login

The system allows the registration and login of a user. Users can create an account by providing their personal details and verifying their identity through facial recognition and an ID document (e.g., using a free API like Microsoft Azure Face API).

2. Viewing Events

The system allows users to view an interactive calendar of local events based on their selected city. These events, such as nightclub parties or concerts, are created and managed by event organizers or venue owners, not by users.

3. Driver Registration

The system allows users to register as drivers by providing their personal information and verifying their identity using a government-issued ID. The app requires users to upload their driver's license for verification.

4. Driver and User Verification

The system verifies drivers by checking their driver's license and ID document against their selfie using an identity verification service. This ensures the authenticity of both drivers and riders on the platform.

5. Ride Booking

The system allows users to book a ride from a driver based on their location and the available drivers nearby. Users can book rides in advance or on the spot, depending on availability.

6. Driver Availability

The system allows drivers to mark their availability for specific events. This ensures that users can see when a driver is available to give them a ride to a particular event.

7. Event and Driver Matching

The system matches users with available drivers based on their location and the event they plan to attend. Users can also view a map showing the drivers available near the event location.

8. Messaging

The system allows users to communicate with drivers via an integrated messaging system to coordinate ride details such as pick-up points, times, and special instructions.

9. Reviews and Ratings

The system allows users to rate their drivers using a 0 to 5-star system and leave feedback. This helps maintain a high level of service quality and trustworthiness within the platform.

10. Notifications

The system sends push notifications to users to inform them about available drivers, ride confirmations, and reminders about upcoming events. Notifications are also sent to drivers when a new user books a ride with them.

11. Favorites and Notifications for Events

The system allows users to add events to their favorites, so they receive notifications when a driver becomes available for that event. This ensures that users are notified of ride availability for the events they're interested in.

5 | Implementation, Integration and Testing

In this section, we describe the testing process conducted for the `Drive or Drunk` application. We detail the environments used, the dummy data loaded into the database, the display formats tested (phone, tablet portrait, tablet landscape), and how external services were mocked.

The following subsections explain the different types of tests performed: unit tests (used only to a limited extent due to the app's heavy reliance on external services), widget tests (where we thoroughly tested graphics, function calls, and interactions with primary-level widgets in a controlled environment), and integration tests (run on physical devices with less control but more realism).

5.1. Unit Tests

Since the application depends heavily on external services (e.g., Firebase Auth, Firestore, network images, geolocation APIs), only a small portion of the codebase is suited for pure unit testing. We focused on testing the core models (`User`, `Review`, `Event`, etc.), using dummy input and mocking external responses.

The average coverage for files in the `lib/models/` directory reached **89.5%**, as reported by LCOV. This was achieved thanks to the clean separation of logic and the ability to inject simulated data.

5.2. Widget Tests

This was the most extensive part of the testing campaign. All major widgets in the project were tested in controlled environments, with comprehensive use of mocking. Mocked services included:

- `FirebaseAuth` (via `Mock/FakeUser`)

- `Firestore` (via the `FakeFirestore` package)
- Network image loading (using the `networkImageMock` package)

We were unable to fully mock Google Maps APIs; instead, we used dummy coordinates to simulate geolocation functionality.

Each test loaded the necessary dummy data into the mocked database to simulate specific user flows (e.g., “user1 created an event and invited user2”). For each widget, we tested UI rendering, function calls, and first-level interactions with related widgets. When applicable, we tested the widget across phone and tablet (portrait and landscape) screen sizes.

5.3. Integration Tests

We conducted end-to-end testing of the application on physical devices to verify real-world functionality. This included interactions with live Firebase services and network operations, testing how the app handled delays, errors, and external responses. We populated the Firebase database to simulate real situations (with users, events, reviews, etc.) and tested the main functionalities of our application. Integration testing enables the simulation of more complex behaviors, navigation between various screens, and, in our case, allowed us to test using real location data to filter nearby places.

Key integration tests included:

- User authentication
- Searching for events
- Booking rides
- Searching for users
- Sending and displaying chat messages

File Name	Description	Screen Path
login_flow_test.dart	Login	Login → Home
event_flow_test.dart	Search and register for an event	Login → Home → SearchEvent → EventsList → EventDetail → RidesPage
chat_flow_test.dart	Search for an user and send a message	Login → Home → SearchUser → UsersList → UserProfile → ChatPage

5.4. Coverage Analysis

Using LCOV, we measured an overall line coverage of **85.5%** across **4,197 lines**, as seen in Figure 5.1. Several directories reached full or near-full coverage, including constants, theme, and widget files.

Some lower-coverage areas (e.g., `lib/features/authentication`, `lib/features/events`) were due to hard-to-mock dependencies or edge-case logic.

Excluding code strictly dependent on external APIs (e.g., location/mapping services), we estimate that the actual testable logic coverage exceeds **90%**, providing strong verification for the app’s core functionality.

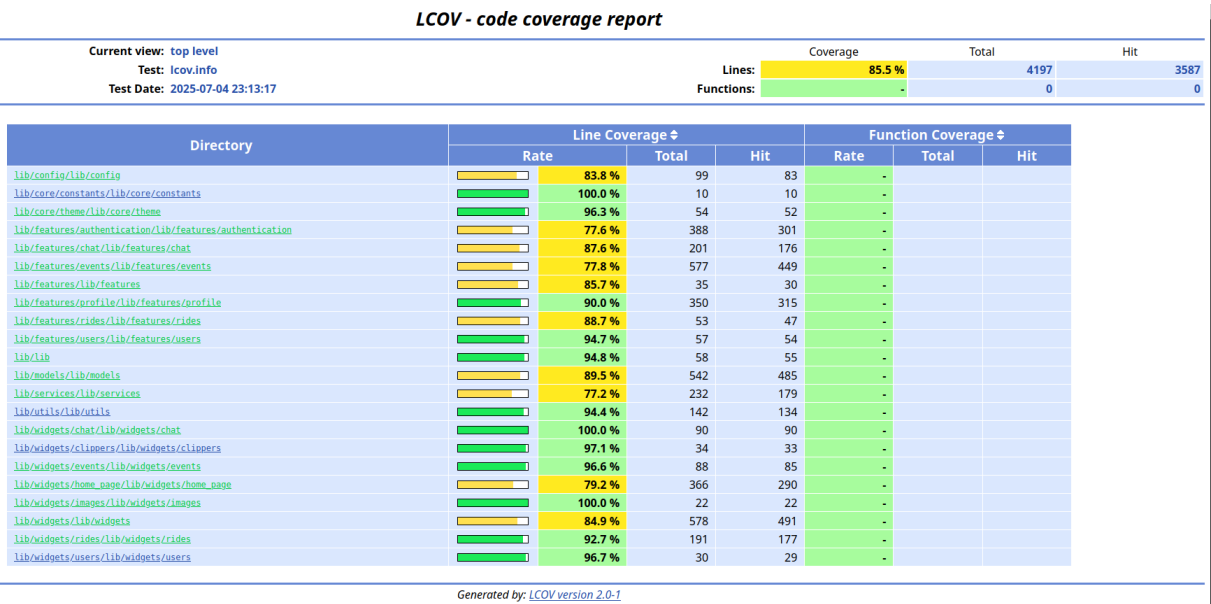


Figure 5.1: Code coverage summary generated by LCOV.

List of Figures

1.1	Entity Relationship Diagram of Firebase Collections	5
2.1	Black-box view of the <i>Drive or Drunk</i> system architecture	7
2.2	Architecture of the application	11
2.3	Deployment view of the application	16
3.1	Login Page	20
3.2	Registration Page	21
3.3	Homepage	22
3.4	Event List Page	23
3.5	Event Creation Page	24
3.6	Event Detail Page	25
3.7	Users List Page	26
3.8	Events Map Page	27
3.9	Chat Page	28
3.10	Chat List Page	29
3.11	Profile Page	30
3.12	Rides from Profile Page	31
3.13	Rides from Event Page	32
3.14	Review Creation Page	33
3.15	Reviews Page	34
5.1	Code coverage summary generated by LCOV.	39