Report on

AutoHub - Car Enthusiast Community Mobile Application

Submitted to:

University Institute of Computing

Submitted by:

Roy Hillary Phiri (24MCA20512)

Chandigarh University, Mohali

CHAPTER 1 INTRODUCTION

1.1.Introduction

The car enthusiast community is a global, passionate, and highly social demographic. However, their digital experience is fragmented. Enthusiasts often juggle multiple platforms: Facebook for events, Instagram for showcasing cars, and forums for technical discussions. This diffusion of platforms creates a disjointed user experience, making it difficult to build a centralized community.

1.2.Problem Statement

The core problem is the lack of a dedicated, mobile-first, all-in-one platform for car enthusiasts. This results in:

- Difficulty in discovering and managing local car events.
- No centralized place to showcase a personal vehicle "build" and its modifications.
- Fragmented communication, with community interactions spread across disparate social media platforms.
- A high barrier to entry for new enthusiasts looking to find clubs and events.

1.3.The Solution: AutoHub

AutoHub is a Flutter-based mobile application designed to be the ultimate solution. It provides a "digital garage" for the modern car enthusiast, integrating all essential community features into a single, intuitive application. Users can find events, showcase their cars, vote on their favorites, and connect with other enthusiasts through a dedicated social feed and chat system.

1.4.Target Audience

- Car Enthusiasts & Modifiers: Individuals passionate about their vehicles who want to share their "builds," find parts, and get recognition.
- Event Organizers & Car Clubs: Groups looking for a streamlined tool to create, promote, and manage their car meets, cruises, and shows.
- Automotive Photographers & Content Creators: Individuals who want to share their media and connect with vehicle owners.

- Casual Observers: Fans who enjoy car culture and want to browse content and see local events.

1.5.Project Objectives

- Develop a high-quality, cross-platform mobile application for Android and iOS using Flutter.
- Implement a secure user authentication and profile management system using Firebase.
- Build a full-featured event management system with map integration.
- Create an engaging social feed for users to share posts, images, and updates.
- Design a "My Garage" feature for users to showcase their vehicles.
- Implement a real-time voting system and global leaderboard to foster friendly competition.
- Ensure the application is scalable, secure, and performs well under load.
- Integrate offline support to provide a continuous user experience.

1.6.Scope of the Project

In Scope:

- All 12 core features outlined in the project brief (Authentication, Garage, Events, Voting, Feed, Leaderboard, Notifications, Search, Offline, Profiles, Chat, Comparison).
- Full frontend development for Android and iOS.
- Complete backend development and integration using the Firebase suite.
- UI/UX design adhering to Material Design 3, including light and dark modes.

Out of Scope:

- E-commerce functionality (e.g., selling car parts or event tickets).
- In-depth vehicle diagnostics or integration with OBD-II devices.
- A dedicated marketplace for buying or selling vehicles.

CHAPTER 2 SYSYEM ARCHITECTURE & DESIGN

2.1.Architecture Overview: Clean Architecture

AutoHub adheres to the principles of Clean Architecture to ensure a robust, scalable, and testable application. This architectural pattern enforces a strict separation of concerns, dividing the app into distinct layers with specific responsibilities.

The core principle is the Dependency Rule: source code dependencies can only point inwards. This means nothing in an inner layer can know anything about an outer layer.

Presentation Layer: (Outermost Layer) Contains all UI-related components, such as screens, widgets, and state management logic (using Riverpod). It is responsible for displaying data and handling user input.

Data Layer (Services/Repositories): (Middle Layer) This layer contains the application's business logic. It includes services (e.g., AuthService, FirestoreService) that act as repositories, abstracting data sources from the rest of the app. It dictates what the app does (e.g., "fetch user profile").

Core/Models Layer: (Innermost Layer) Contains the core business entities (e.g., UserModel, EventModel) and utility/theme definitions. This layer has zero dependencies on other layers.

This separation ensures that a change in the database (e.g., migrating from Firestore) would only affect the Data layer, leaving the Presentation and Core layers untouched.

2.2.Project Structure

The project directory is organized to reflect the Clean Architecture pattern:

```
lib/
├── main.dart               # App entry point & Firebase init
├── firebase_options.dart   # Firebase platform configuration
│
├── core/                   # Core app logic, models, and utilities
│   ├── theme/
│   │   └── app_theme.dart  # Light/dark theme definitions
│   └── utils/
│       └── constants.dart  # App-wide constants (e.t., Firestore collection names)
│
├── data/                   # Data handling and business logic
│   ├── models/             # Data models (UserModel, EventModel, etc.)
│   └── services/           # Repositories/Services (AuthService, FirestoreService)
│
```

```
├── presentation/            # All UI components
│   ├── main_navigation.dart  # Bottom navigation bar controller
│   ├── screens/              # All application screens (e.g., feed, profile)
│   └── widgets/              # Reusable UI components (e.g., PostCard, CustomButton)
│
└── providers/               # Riverpod providers
    ├── app_providers.dart    # Global providers for services and data streams
    └── theme_provider.dart   # StateNotifier for theme switching
```

2.3.Design Patterns

Repository Pattern: Implemented via the services directory. For example, the FirestoreService acts as a repository that provides an API for the Presentation layer to interact with Firestore. The UI is completely unaware of how data is fetched or stored.

Provider Pattern: Utilized extensively via Riverpod. Services, controllers, and data streams are all "provided" to the widget tree, allowing for decoupled access and easy testing.

Observer Pattern: Implemented inherently by Riverpod's StreamProvider and StateNotifierProvider, allowing widgets to "observe" data streams or state and rebuild automatically when changes occur.

2.4.State Management: Flutter Riverpod

Flutter Riverpod was chosen as the primary state management solution due to its flexibility, compile-time safety, and reactive nature. Unlike Provider, Riverpod is decoupled from the widget tree, preventing common issues related to BuildContext.

Key Provider Types Used:

Provider: Used for providing service singletons (e.g., AuthService, FirestoreService) that are immutable.

Dart

```
// lib/providers/app_providers.dart
final authServiceProvider = Provider<AuthService>((ref) {
  return AuthService(FirebaseAuth.instance);
```

```
});
```

StreamProvider: Used for reactively listening to real-time data from Firestore. This is the backbone of the social feed, event lists, and chat.

Dart

```
// lib/providers/app_providers.dart
final eventsStreamProvider = StreamProvider<List<EventModel>>((ref) {
  final firestoreService = ref.watch(firestoreServiceProvider);
  return firestoreService.streamEvents();
});
```

StateNotifierProvider: Used for managing mutable UI state, such as form inputs or the currently selected theme.

Dart

```
// lib/providers/theme_provider.dart
final themeProvider = StateNotifierProvider<ThemeNotifier, ThemeMode>((ref) {
  return ThemeNotifier();
});
```

2.5.Data Flow Example (Joining an Event)

1. User (Presentation): Taps the "Join Event" button on event_detail_screen.dart.
2. Widget (Presentation): The onPressed callback reads the Event and User from their respective providers. It calls a method on the eventProvider, e.g., ref.read(eventProvider.notifier).joinEvent(eventId, userId).
3. Provider (State Management): The provider (likely a StateNotifierProvider or a simple Provider exposing a service) calls the FirestoreService.
4. Service (Data): firestoreService.joinEvent(eventId, userId) runs a Firestore transaction to atomically add the userId to the event's attendees array and the eventId to the user's joinedEvents array.
5. Firebase (Backend): The data is updated in the database.
6. StreamProvider (State Management): The StreamProvider listening to the event document automatically emits the new event data (with the updated attendee list).

7. UI (Presentation): Any widget watching this stream (e.g., the attendee list on event_detail_screen.dart) rebuilds instantly, showing the new user as an attendee.

CHAPTER 3 CORE FEATURES AND IMPLEMENTATION

3.1. User Authentication System

- Implementation: Leverages Firebase Authentication for secure and scalable user management. The primary method is Email/Password authentication.
- Core Flow:
    1. Registration: The AuthService calls FirebaseAuth.instance.createUserWithEmailAndPassword().
    2. Profile Creation: Upon a successful registration, a post-registration trigger (or a direct call from AuthService) creates a new document for the user in the users Firestore collection, using their uid as the document ID. This stores profile information like username, bio, etc.
    3. Login: Users sign in with signInWithEmailAndPassword().
    4. Session Management: Firebase Authentication automatically handles session persistence and token refreshes. The app uses FirebaseAuth.instance.authStateChanges() as a global stream (exposed via a Riverpod StreamProvider) to route users to the HomeScreen or LoginScreen automatically.
- Key Files: lib/data/services/auth_service.dart, lib/presentation/screens/auth/

Dart

```
// lib/data/services/auth_service.dart (Simplified)
class AuthService {
  final FirebaseAuth _auth;
  final FirebaseFirestore _firestore;
```

```
  AuthService(this._auth, this._firestore);

  Future<void> signUp(String email, String password, String username) async {
    try {
      UserCredential userCredential = await _auth.createUserWithEmailAndPassword(
        email: email,
        password: password,
      );

      if (userCredential.user != null) {
        // Create user profile in Firestore
        UserModel newUser = UserModel(
          uid: userCredential.user!.uid,
          email: email,
          username: username,
          // ... other fields
        );
        await _firestore
            .collection('users')
            .doc(userCredential.user!.uid)
            .set(newUser.toJson());
      }
    } on FirebaseAuthException {
      // Handle errors
    }
  }
}
```

3.2.My Garage Feature

- Implementation: A core part of the user profile. It allows users to manage a digital collection of their vehicles.
- Data Model: The CarModel (with fields for make, model, year, mods, and imageUrls) is stored as an array within the UserModel document in Firestore.
- Image Handling:
    1. User selects images using image_picker.
    2. Images are uploaded to Firebase Cloud Storage in a user-specific folder (e.g., car_images/{userId}/{carId}/{imageName}).
    3. The public download URLs are retrieved from Storage and stored in the imageUrls list within the CarModel.
    4. Images are displayed efficiently using cached_network_image to reduce network usage and loading times.
- CRUD Operations: Users can add, edit, and delete car entries from their "My Garage" screen, which updates the cars array in their user document.
- Key Files: lib/presentation/screens/profile/my_garage_screen.dart, lib/presentation/screens/profile/add_car_screen.dart, lib/data/models/user_model.dart

3.3.Event Management System

- Implementation: Provides full CRUD functionality for car-related events.
- Features:
    1. Creation: Users fill a form with event details.
    2. Location: Maps_flutter is used to provide an interactive map where users can drop a pin to select the event location. This saves a GeoPoint (latitude/longitude) to the EventModel in Firestore.
    3. Geolocation: geolocator is used to get the user's current location, which can center the map or suggest nearby locations.
    4. Browsing: The EventsScreen uses a StreamProvider to listen for real-time changes to the events collection, with filters for "Upcoming" and "Past" events.

5. Attendance: Users can join/leave events. This action updates an attendees array in the EventModel, allowing for real-time attendee lists.

- Key Files: lib/presentation/screens/events/, lib/data/models/event_model.dart

3.4.Featured Car Showcase & Voting System

- Implementation: This system integrates events with the "My Garage" feature to create a competitive "car show" element.
- Core Flow:
  1. Submission: A user attending an event can "submit" a car from their "My Garage" to that specific event. This creates a new document in the eventSubmissions collection, linking the userId, carId, and eventId.
  2. Voting: Other event attendees (or all users, depending on rules) can vote on submissions.
  3. Data Integrity: To ensure one vote per user, the EventSubmissionModel contains a votes array (or subcollection) storing the uid of each user who has voted. A Firestore Transaction is used to add a vote, which first reads the document to see if the user has already voted, and only then writes the new vote. This prevents race conditions.
- Technical Challenge: Real-time vote counting and aggregation.
- Solution: The EventSubmissionModel stores a voteCount field, which is incremented during the transaction. This avoids needing to count the votes array on the client, which is inefficient.
- Key Files: lib/presentation/screens/events/submit_car_screen.dart, lib/data/models/event_submission_model.dart

3.5.Community Social Feed

- Implementation: A dynamic, real-time social feed inspired by platforms like Twitter and Threads.

- Data Model: A top-level posts collection in Firestore. Each PostModel document contains the authorId, text, imageUrls, timestamp, and likeCount. Comments are handled as a subcollection (comments) under each post for scalability.
- Real-Time Updates: The FeedScreen uses a StreamProvider to listen to the posts collection, ordered by timestamp. As new posts are added, they appear at the top of the feed instantly.
- Performance:
    1. Image Loading: cached_network_image is used extensively for all post images and user avatars to provide smooth scrolling and offline caching.
    2. Perceived Performance: shimmer loading effects are displayed while the initial batch of posts is fetched, providing a professional UI skeleton.
    3. Pagination: (Future enhancement, but critical) As the feed grows, "infinite scrolling" would be implemented by fetching posts in batches using Firestore's limit() and startAfter() queries.
- KeyFiles:lib/presentation/screens/feed/feed_screen.dart,
  lib/presentation/screens/feed/create_post_screen.dart,
  lib/data/models/post_model.dart

3.6.Global Leaderboard

- Implementation: A competitive ranking system that displays top users based on their "Featured Car" wins.
- Data Strategy: To avoid complex and slow aggregation queries, the UserModel document contains a winCount field.
- Backend Logic: When an event concludes, a (manual or automated) process determines the winner of the "Featured Car" showcase (based on voteCount from EventSubmissionModel). A Cloud Function (or admin panel) is triggered, which increments the winCount on the winning user's UserModel document.
- Display: The LeaderboardScreen performs a simple, efficient Firestore query:
  Dart
  // Query for the leaderboard

```
_firestore.collection('users')
  .orderBy('winCount', descending: true)
  .limit(50)
  .get();
```

This provides a highly scalable leaderboard that reads from pre-aggregated data.

- Key Files: lib/presentation/screens/leaderboard/leaderboard_screen.dart

CHAPTER 4 TECHNOLOGY STACK & INTERGRATION

This project's success relies on the tight integration of Flutter with a suite of powerful backend services and third-party libraries.

4.1.Frotend Framework

- Flutter SDK (3.8.1+): The foundation of the application. Chosen for its high-performance rendering, expressive UI, and the "write once, deploy anywhere" capability. The entire UI is built with Flutter's widget-based system.
- Dart: The modern, client-optimized language used to write all application code.

4.2.State Management

- Flutter Riverpod (2.6.1): The core state management solution. It provides a declarative, reactive, and testable way to manage all app state, from theme changes to complex real-time data streams.

4.3.Backend Services (Firebase)

The Firebase suite provides the entire serverless backend, handling data, authentication, storage, and more.

- Firebase Authentication: Manages user identity, providing secure login, registration, and session management.

- Cloud Firestore: The primary NoSQL database. Its real-time streaming capabilities power the social feed, event lists, and voting. The document-based structure is ideal for storing complex, hierarchical data like user profiles with nested car lists.
- Firebase Cloud Storage: Used for all user-generated content, including profile photos, car images, and images attached to social feed posts.
- Firebase Cloud Messaging (FCM): The backbone of the push notification system. Used to send targeted notifications for social interactions (likes, comments) and event reminders.
- Firebase Realtime Database: Utilized specifically for the Event Chat feature. Chosen over Firestore for this use case due to its extremely low-latency, JSON-based structure, which is ideal for ephemeral, high-velocity chat messages.
- Firebase Analytics: Integrated to track user behavior, feature adoption, and engagement, providing critical insights for future development.

4.4.Key Dependencies & Integrations

- Maps & Location:
    1. Maps_flutter (2.9.0): Provides the interactive map widgets used in event creation and event detail screens.
    2. geolocator (13.0.1): Used to request and retrieve the user's current GPS location.
- Media & UI:
    1. image_picker (1.1.2): Handles the native UI for selecting images from the device's camera or gallery.
    2. cached_network_image (3.4.1): A critical performance library. Caches network images to disk and memory, enabling smooth list scrolling and offline image access.
    3. shimmer (3.0.0): Provides elegant, skeleton-style loading animations.
    4. google_fonts (6.2.1): Used to load and apply the Inter font family for consistent typography.

- Notifications:
    1. flutter_local_notifications (18.0.1): Works with FCM. When a notification is received while the app is in the foreground, this library is used to display a local "heads-up" notification.
- Offline Storage:
    1. sqflite (2.4.0): A local SQLite database. Used by the OfflineService to cache critical data (like user profiles or upcoming events) for offline access.
    2. shared_preferences (2.3.2): Used for storing simple key-value pairs, such as the user's theme preference (light/dark) or the last sync time.
- Utilities:
    1. share_plus (10.0.2): Implements the native platform sharing dialog, allowing users to share links to events or profiles.

CHAPTER 5 TESTING, DEPLOYMENT & FUTURE ENHANCEMENTS

5.1.Testing Approach & Strategies

A multi-layered testing strategy was employed to ensure application quality and stability.

- Unit Tests: Used to test individual functions, methods, and classes in isolation. This was heavily applied to the data/services layer. For example, testing AuthService logic by mocking FirebaseAuth.
- Widget Tests: Flutter's built-in widget testing framework was used to test individual UI components (PostCard, EventCard) and screens. This verifies that the UI reacts correctly to state changes and user interaction.
- Integration Tests: Used to test complete end-to-end user flows. For example, an integration test was written to simulate the entire "Login -> Go to Feed -> Create Post -> Verify Post Appears" flow, using flutter_driver and mock data.

5.2.Quality Assurance(QA) Measures

- Manual Testing: A dedicated QA process involved manual testing on a range of physical Android and iOS devices to catch platform-specific bugs and UI inconsistencies.

- Firebase App Distribution: Internal builds were distributed to stakeholders and beta testers for early feedback.
- Firebase Analytics: Post-launch, Firebase Analytics and Firebase Crashlyx are used to monitor app performance, stability, and user engagement in real-time.
- Security Rules: All Firebase services are protected by Firebase Security Rules. These rules are tested using the Firebase Emulator Suite to ensure users can only access their own data and that all interactions are authenticated and authorized.

5.3.Deployment Considerations

- CI/CD Pipeline: A Continuous Integration/Continuous Deployment pipeline (e.g., using GitHub Actions or Codemagic) is recommended for automating the build, test, and deployment process.
- Build Flavors: The app is configured with build "flavors" (e.g., dev, staging, prod) to connect to different Firebase projects, allowing for testing in a production-like environment without affecting live user data.
- Store Deployment: Standard release processes for the Apple App Store and Google Play Store are followed, including managing signing keys, versioning, and store listings.

5.4.Challenges Faced & Solutions

- Challenge: Managing complex, real-time state from multiple Firestore streams (e.g., posts, likes, comments) on a single screen.
    1. Solution: Riverpod was instrumental. By using StreamProvider and ref.watch, the UI could declaratively "listen" to data. Combining providers allowed us to merge data streams (e.g., a post stream and its like status) into a single, clean state for the UI to consume.
- Challenge: Implementing a robust offline-first experience.
    1. Solution: A dedicated OfflineService was created to cache data in SQFlite. A SyncService handles the "write-back" logic, queuing local changes (likes, posts) and executing them in a batch when a connection is restored.
- Challenge: Performance with long lists of high-resolution images.

1. Solution: cached_network_image was a complete solution, providing both memory and disk caching. This, combined with shimmer loaders, eliminated UI jank during scrolling.

5.5.Future Enhancements & Roadmap

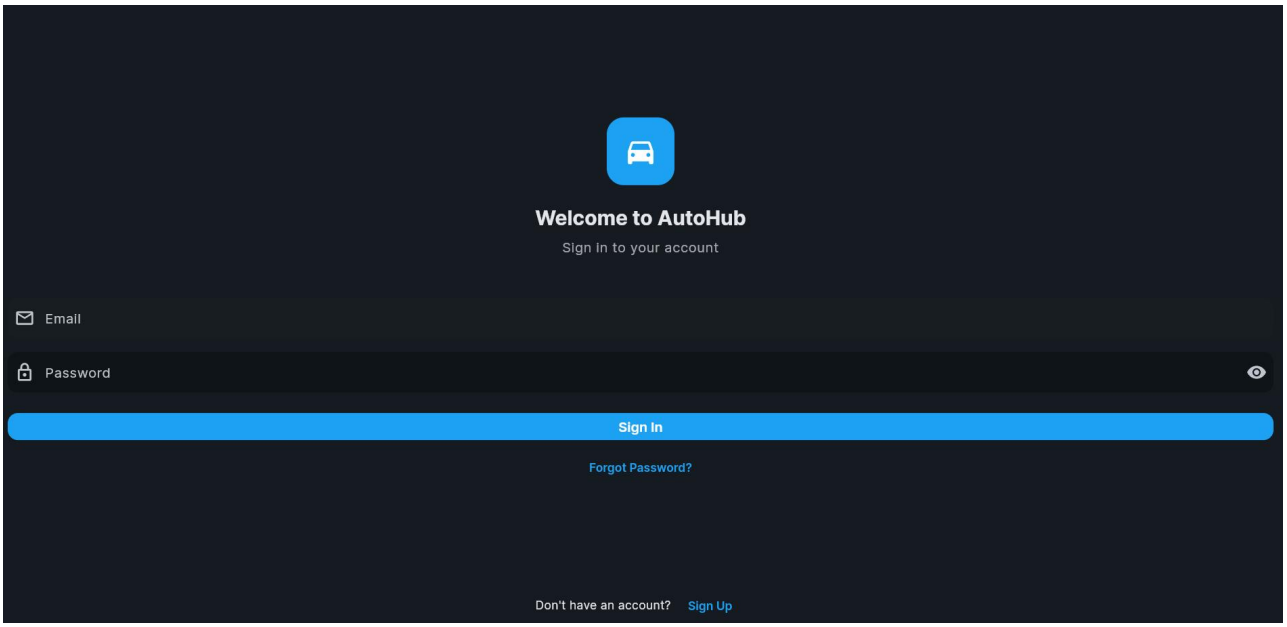
- Advanced Search: Integrate a dedicated search service like Algolia for faster, full-text search across all collections.
- Video Content: Allow users to upload videos to their garage and social feed posts.
- Car Clubs/Groups: Introduce a "Groups" feature, allowing users to create and manage private or public car clubs with their own discussion boards and event calendars.
- Marketplace: A "soft" marketplace for users to list parts for sale (though full e-commerce was out of scope, a classifieds section is a logical next step).
- Enhanced Car Comparison: Allow users to compare car specifications using a public car database API.

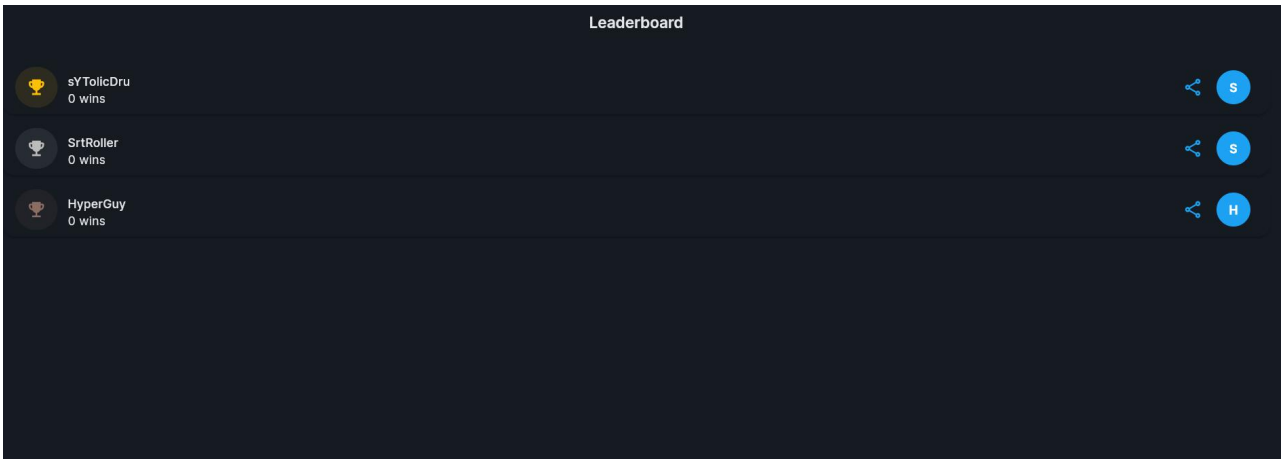
5.6.Conclusion

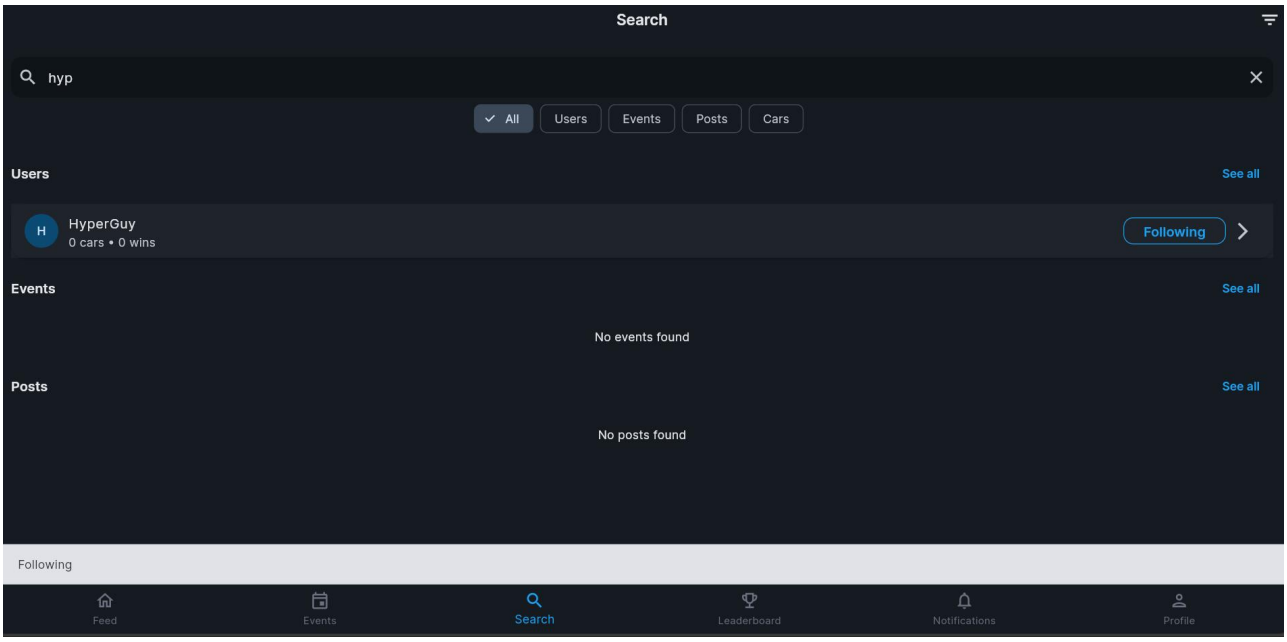
AutoHub successfully achieves its objective of creating a unified, feature-rich, and high-performance mobile application for the car enthusiast community. By leveraging the power of Flutter and Firebase, the project delivers a scalable, cross-platform solution with a robust, maintainable codebase based on Clean Architecture. The application provides a solid foundation for future growth and has the potential to become the definitive digital hub for car enthusiasts worldwide.

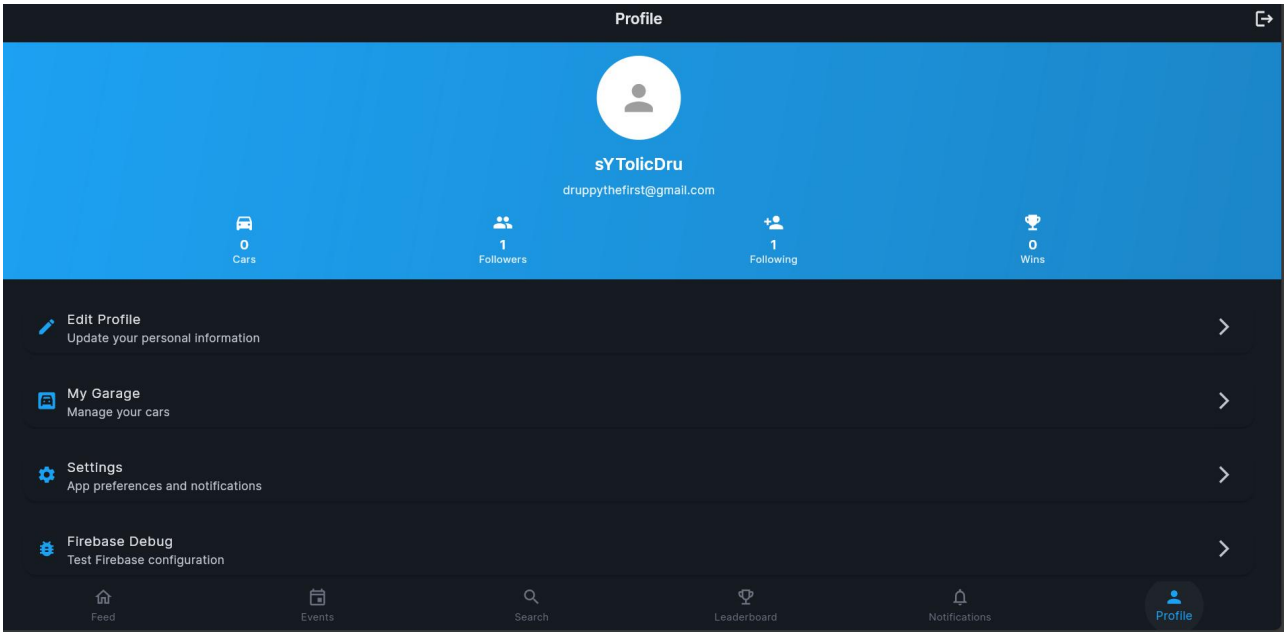
GITHUB LINK : https://github.com/Phiri-Roy/AutoHub

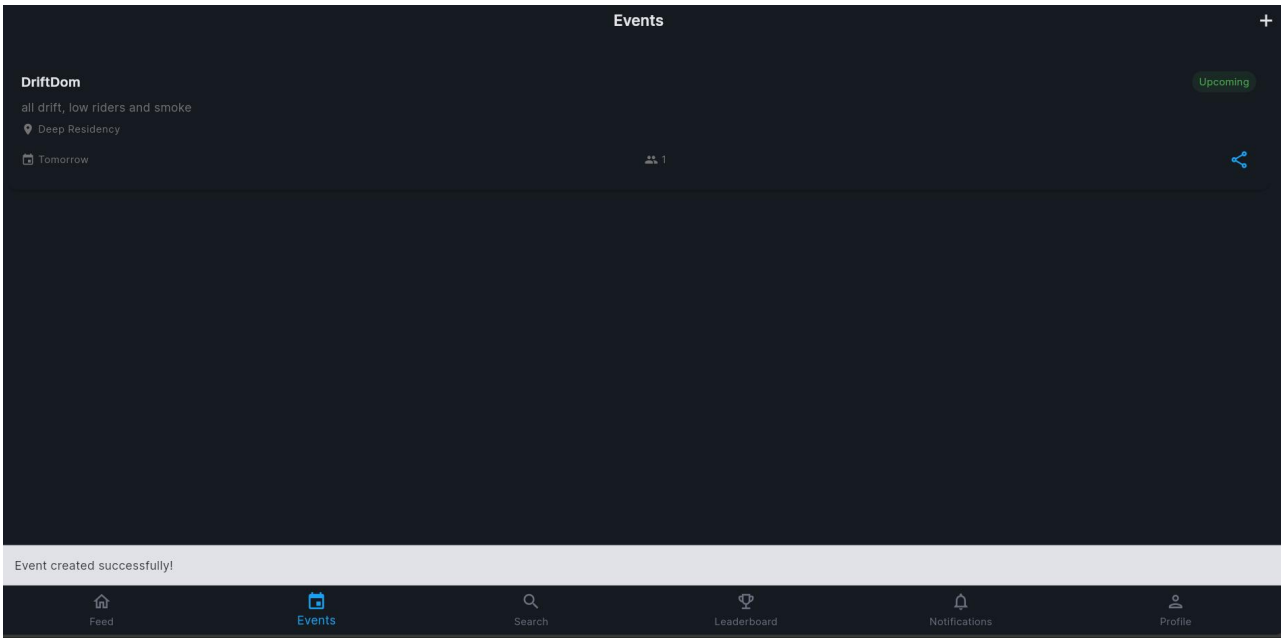
SCREENSHOTS








Profile
sYTolicDru
druppythefirst@gmail.com
0
Cars
1
Followers
1
Following
0
Wins
Edit Profile
Update your personal information
My Garage
Manage your cars
Settings
App preferences and notifications
Firebase Debug
Test Firebase configuration
Feed
Events
Search
Leaderboard
Notifications
Profile


Events
DriftDom
Upcoming
all drift, low riders and smoke
Deep Residency
Tomorrow
1
Event created successfully!
Feed
Events
Search
Leaderboard
Notifications
Profile