

UNIVERSITY OF BUEA

**FACULTY OF ENGINEERING
AND TECHNOLOGY**



REPUBLIC OF CAMEROON

PEACE-WORK-FATHERLAND

**COURSE TITLE: INTERNRT PROGRAMMING AND MOBILE
PROGRAMMING**

COURSE CODE: CEF440

COURSE INSTRUCTOR: Dr. VALERY NKEMENI

GROUP: 18

TASK 6: DATABASE DESIGN AND IMPLEMENTATION

GROUP 18 MEMBERS

S/N	NAME	MATRICULE
1	SINDZE DJAM ALAN WILFRIED	FE20A105
2	ENJEMA NDIVE PEACE	FE22A203
3	NGWINKEM KETTY NERITA	FE22A269
4	NJAMUTOH SHALOM BRENDA TASAHI	FE22A271
5	TABI ATEM ROGAN ESSEMBION	FE22A300

02/06/2025

Table of Contents

1. Data Elements	4
1.1 Overview	4
1.2 DATA ELEMENT CATEGORIES	4
1.3 DATA ELEMENTS BREAKDOWN	5
1.4. Security & Privacy Considerations	8
2. CONCEPTUAL DESIGN	9
2.1 System Overview	9
2.2 Major System Components	9
2.3 Entities and Attributes	10
2.4 Relationships	11
2.5. Functional Requirements (Logical Behaviors)	12
2.6 Use Case Examples	12
2.7 Security & Privacy	13
3.0 Entity-Relationship (ER)	14
3.1 Entities	14
3.2 Relationships	16
4.0 Database Implementation for Car Fault Finder App	19
4.1 Technology Stack	19
4.3 Integration with Firebase	19
4.4 Maintenance Book Functionality	20
4.5 Autre – Other Feature	20
4.6 Dashboard Scanning Feature	20
4.7 Engine Sound Recording & Storage	20
4.8 Security Rules	20
4.9 Optimization and Performance Tuning	21
5.0 Backend Implementation	22
5.1 Introduction to Backend Implementation	22
5.2 Firebase Services Used	22

5.3 Backend Data Model & Structure	23
5.4 Backend Logic – How It Works	24
5.5 Maintenance Book Feature	25
5.6 Security, Sync & Future Enhancements	25
6.0 Connecting Database to Backend	27
6.1 Introduction	27
6.2 Firebase Setup and Configuration	27
6.3 Connecting Firestore to Flutter	27
6.4 Maintenance Book Integration	28
6.5 Dashboard Scanning Connection	28
6.6 Real-Time Sync and Conclusion	29
6.7 Firebase Firestore Structure	29
7.0 Conclusion	41

1. Data Elements

1.1 Overview

This document outlines the data elements used in a mobile car diagnostic application. The app provides two modes of diagnosis—**automatic (via OBD-II)** and **manual (based on user input)**—as well as features like **camera-based dashboard light interpretation**, **audio-based fault detection**, and **car maintenance support**.

1.2 DATA ELEMENT CATEGORIES

Category	Description
Input Data	Data collected from the user or external devices (e.g., OBD-II, audio, camera)
Output Data	Data shown to the user (e.g., diagnosis result, video tutorial links)
Reference Data	Static or semi-static data used for interpretation (e.g., OBD codes, sound classification models)
User Data	Profile and interaction history

1.3 DATA ELEMENTS BREAKDOWN

A.. Automatic Diagnostic (OBD-II)

Element	Type	Source	Purpose
bluetooth_permission	Boolean	User device	To access OBD-II data
vehicle_vin	String	OBD-II or user input	Identifies vehicle type

✓ B. Repair Tutorials

Element	Type	Source	Purpose
tutorial_topic	String	App logic/manual choice	Issue or component (e.g., "engine overheating")
youtube_link	URL	Preconfigured reference	Link to repair video
view_count	Integer	YouTube API	Popularity measure
user_rating	Integer (1–5)	User input	Tutorial usefulness

✔ C. Maintainable Book

Element	Type	Source	Purpose
maintenance_task	String	Static	(e.g., "Change oil", "Check battery")
last_done_date	Date	User input	Track when task was last done
reminder_set	Boolean	User input	Whether reminders are enabled

✔ D. OBD-II Code Interpretation

Element	Type	Source	Purpose
input_code	String (e.g., "P0171")	User input	Code user enters
description	String	Reference lookup	Fault meaning
recommendation	String	Static/advisory	Suggested fix or check

✔ E. Camera-Based Diagnostic (Warning Lights)

Element	Type	Source	Purpose
captured_image	Image	User input	Photo of dashboard
light_icon_detected	String (e.g., "Check Engine", "ABS")	Image recognition model	Detected light
meaning	String	Reference	Meaning of the icon

Element	Type	Source	Purpose
urgency_level	Enum ("Low", "Medium", "High")	Static	How critical the light is

✓ F. Audio-Based Diagnostic

Element	Type	Source	Purpose
recorded_audio	Audio	User input	Engine sound
audio_classification	String	ML model	Identified fault pattern (e.g., "Knocking", "Misfire")
confidence_score	Float (0.0–1.0)	ML model	Accuracy of the prediction
recommended_action	String	Static	What to do next (e.g., “Check spark plugs”)

✓ G. Diagnostic Logs

Element	Type	Source	Purpose
timestamp	DateTime	System-generated	When diagnosis occurred
diagnosis_type	Enum ("Automatic", "Manual", "Camera", "Audio")	App logic	Mode of diagnosis
result_summary	Text	App logic	Final analysis shown to

Element	Type	Source	Purpose
			user
location	GPS Coordinates (optional)	Device	For regional service recommendations

1.4. Security & Privacy Considerations

- All user data should be stored securely (e.g., Firebase Firestore with proper rules).
- Media files (audio, images) should be protected with access control.
- No personal data should be shared with third parties without consent.

2. CONCEPTUAL DESIGN

2.1 System Overview

The system provides:

- **Automatic diagnosis** using OBD-II via Bluetooth
 - **Manual diagnosis** based on user-reported symptoms
 - **Camera-based interpretation** of dashboard lights
 - **Audio-based fault detection** using engine sound
 - **Reference tools** like repair tutorials and a maintenance guide
-

2.2 Major System Components

A. User Interface (Mobile App)

- Provides the front-end through which users interact with the app.
- Supports navigation between different diagnosis modes and tools.

B. Diagnostic Engine

- Logic layer responsible for processing input from users, sensors (OBD-II), and ML models (audio and image).
- Outputs diagnostic results and recommendations.

C. Media Analysis Module

- **Camera Module:** Detects and interprets dashboard warning lights.
- **Audio Module:** Classifies engine sounds and predicts faults.

D. Reference Services

- Maintains a database of OBD-II codes, repair tutorials, and maintenance tips.

E. Firebase Backend

- Manages user data, diagnostics history, media storage, and references.
 - Provides secure authentication and cloud data access.
-

2.3 Entities and Attributes

1. Vehicle

- vehicle_name (VIN or assigned)
- make, model, year
- obd_support_flag (Boolean)

2. Diagnostic Session

- session_id
- type (Automatic, Manual, Camera, Audio)
- timestamp
- result_summary

3. OBD Data

- obd_code
- description
- recommendation
- severity_level

4. Media Input

- media_id
- type (image/audio)
- file_path
- detected_feature (e.g., warning light, sound class)
- confidence_score

5. Tutorial Reference

- tutorial_id
- topic
- video_url
- relevance_score
- user_feedback

7. Maintenance Task

- task_id
- task_name
- recommended_interval
- last_done_date
- reminder_enabled

2.4 Relationships

- A **User** can own multiple **Vehicles**
- A **User** initiates multiple **Diagnostic Sessions**
- Each **Diagnostic Session** may generate one or more **Media Inputs**
- Each **Media Input** is linked to a **Diagnosis Result**
- A **User** can refer to multiple **Tutorials** and manage multiple **Maintenance Tasks**

2.5. Functional Requirements (Logical Behaviors)

Function	Description
F1: Start Automatic Diagnosis	Connects via Bluetooth to read and interpret OBD-II codes
F2: Upload Image	Processes dashboard light image and returns interpreted result
F3: Upload Audio	Classifies engine sounds to detect faults
F4: Retrieve OBD-II Code Info	Allows user to search and interpret codes manually
F5: View Repair Tutorials	Recommends videos based on symptoms or fault codes
F6: Log Maintenance Task	Records last maintenance actions and sets reminders
F7: Save Diagnostic History	Stores and retrieves past diagnostic sessions

2.6 Use Case Examples

Use Case 1: Automatic Diagnostic

➤ Actor: User

Precondition: OBD-II Bluetooth device is paired

➤ Steps:

- User taps "Automatic Diagnostic"
- App connects to OBD-II

1. App retrieves fault codes
2. App queries description for each code
3. Results are displayed with recommendations

Outcome: User understands detected issues

2.7 Security & Privacy

- All user media is stored with restricted access (Firebase rules)
- Personal diagnostic history is tied to authenticated users only
- All diagnosis results are local to the user (no data sold/shared)

3.0 Entity-Relationship (ER)

An ER diagram(Entity-Relationship diagram) is a visual tool used to design and represent the structure of a database. It shows Entities, Attributes, Relationships, and Cardinality. It helps planning how data is stored and connected.

3.1 Entities

1. User

- **Description:** Represents a user of the system.
- **Attributes:**
 - **car_model:** VARCHAR (Primary Key, likely composite with user_email) - The model of the user's car.
 - **user_name:** VARCHAR - The name of the user.
 - **user_email:** VARCHAR (Primary Key, likely composite with car_model) - The email address of the user.

2. RepairTutorial

- **Description:** Stores information about repair tutorials.
- **Attributes:**
 - **tutorial_id:** INTEGER (Primary Key) - Unique identifier for the tutorial.
 - **title:** TEXT - The title of the tutorial.
 - **description:** TEXT - A detailed description of the tutorial.

3. OBD2Code

- **Description:** Contains information about OBD2 diagnostic trouble codes.
- **Attributes:**
 - **code:** VARCHAR (Primary Key) - The actual OBD2 code (e.g., P0123).
 - **description:** TEXT - A detailed description of what the code signifies.

4. Diagnosis

- **Description:** Records each instance of a vehicle diagnosis.
- **Attributes:**
 - **diagnosis_id:** INTEGER (Primary Key) - Unique identifier for a diagnosis session.
 - **timestamp:** DATETIME - The date and time when the diagnosis was performed.

5. DashboardImage

- **Description:** Stores information about dashboard images used for diagnosis.
- **Attributes:**
 - **image_id:** INTEGER (Primary Key) - Unique identifier for the dashboard image.
 - **analysis_result:** TEXT - The result of analyzing the dashboard image.

6. EngineAudio

- **Description:** Stores information about engine audio recordings used for diagnosis.
- **Attributes:**
 - **audio_id:** INTEGER (Primary Key) - Unique identifier for the engine audio recording.
 - **analysis_result:** TEXT - The result of analyzing the engine audio.

7. MaintenanceLog

- **Description:** Maintains a log of vehicle maintenance activities.
- **Attributes:**
 - **log_id:** INTEGER (Primary Key) - Unique identifier for a maintenance log entry.
 - **car_model:** VARCHAR (Foreign Key, references User.car_model) - The model of the car for which maintenance was performed.
 - **maintenance_type:** VARCHAR - The type of maintenance performed (e.g., oil change, tire rotation).

■ **date:** DATE - The date when the maintenance was performed.

■ **notes:** TEXT - Any additional notes about the maintenance.

8. **DiagnosisResult**

○ **Description:** Stores the detailed results of a vehicle diagnosis.

○ **Attributes:**

■ **result_id:** INTEGER (Primary Key) - Unique identifier for a diagnosis result.

■ **diagnosis_id:** INTEGER (Foreign Key, references Diagnosis.diagnosis_id) - The ID of the diagnosis session this result belongs to.

■ **result_summary:** TEXT - A summary of the diagnosis findings.

■ **confidence_score:** FLOAT - A score indicating the confidence level of the diagnosis.

■ **suggested_action:** TEXT - The action recommended based on the diagnosis.

3.2 Relationships

Based on the connecting lines in the diagram, the following relationships can be inferred:

1. **User records Diagnosis:**

○ **Type:** One-to-Many

○ **Description:** A User can record multiple Diagnosis entries. Each Diagnosis is associated with one User.

○ **Foreign Key:** User's primary key (likely composite of car_model and user_email) would be a foreign key in Diagnosis (though not explicitly shown in the diagram, it's implied by the connection from User to Diagnosis labeled "records").

2. **User searches OBD2Code:**

○ **Type:** Many-to-Many (implied by the connection, often represented by a linking table not shown explicitly)

- **Description:** A User can search for multiple OBD2Codes, and an OBD2Code can be searched by multiple Users.

3. Diagnosis produces DiagnosisResult:

- **Type:** One-to-Many
- **Description:** A Diagnosis session can produce one or more DiagnosisResult entries. Each DiagnosisResult belongs to a single Diagnosis.
- **Foreign Key:** DiagnosisResult.diagnosis_id references Diagnosis.diagnosis_id.

4. Diagnosis uses DashboardImage:

- **Type:** One-to-Many (implied, as a diagnosis might involve multiple images, or an image is used for one diagnosis)
- **Description:** A Diagnosis can involve multiple DashboardImages. A DashboardImage is used in one Diagnosis.
- **Foreign Key:** Diagnosis's primary key (diagnosis_id) would likely be a foreign key in DashboardImage (not explicitly shown). Alternatively, DashboardImage.diagnosis_id could be a foreign key. The diagram shows the connection originating from DashboardImage to Diagnosis, which could also suggest a One-to-One or Many-to-One where an image is part of a diagnosis. Without arrowheads, it's ambiguous, but the most common scenario for "uses" here would be that a diagnosis "uses" images.

5. Diagnosis uses EngineAudio:

- **Type:** One-to-Many (similar to DashboardImage)
- **Description:** A Diagnosis can involve multiple EngineAudio recordings. An EngineAudio recording is used in one Diagnosis.
- **Foreign Key:** Similar to DashboardImage, Diagnosis's primary key (diagnosis_id) would likely be a foreign key in EngineAudio or vice-versa, depending on the specific cardinality.

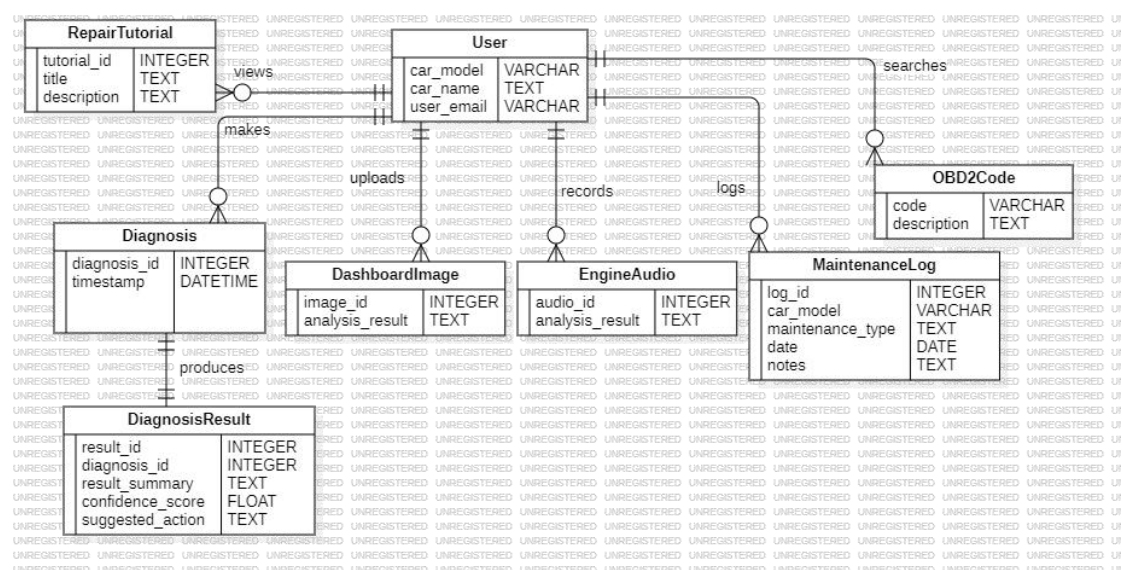
6. OBD2Code relates to RepairTutorial:

- **Type:** Many-to-Many (implied, as an OBD2 code might have multiple tutorials, and a tutorial might address multiple codes)
- **Description:** An OBD2Code can be associated with multiple RepairTutorials, and a RepairTutorial can be relevant to multiple OBD2Codes. This would typically require a linking table.

7. User has MaintenanceLog:

- **Type:** One-to-Many
- **Description:** A User (specifically their car_model) can have multiple MaintenanceLog entries. Each MaintenanceLog entry belongs to one User's car.
- **Foreign Key:** MaintenanceLog.car_model references User.car_model.

Below is the ER Diagram



4.0 Database Implementation for Car Fault Finder App

In order to build a scalable and efficient backend for the Car Fault Finder app, we decided to use Firebase as our primary cloud platform. This decision was based on Firebase's real-time data capabilities, ease of integration with Flutter, and robust infrastructure support for mobile applications. Firebase provided a unified backend service that enabled us to implement features such as user authentication, data storage, file storage, and synchronization across multiple devices.

4.1 Technology Stack

To implement the data layerbase backend for our application, we used several components of Firebase (cloud) including Cloud Firestore, Firebase (cloud) Realtime Database, Firebase (cloud) Storage, and Firebase (cloud) Authentication. The choice of Cloud Firestore as the main data layer storage was guided by its support for structured, scalable NoSQL storage. Firebase (cloud) Realtime Database was also integrated for components that required real-time synchronization, such as updating fault data layer or monitoring engine diagnostics.

4.2 Database Schema Design

We designed a modular and extensible schema to represent user profiles, vehicle data, maintenance records, scanned engine issues, and recorded sound files. Each user document in Firestore contained collections such as 'vehicles', 'maintenance_logs', and 'recordings'. For example, the 'vehicles' collection stored the car brand, model, and user-inputted details from the Autre – Other form. This structure enabled us to easily query and manage the user's data independently.

4.3 Integration with Firebase

To connect the Flutter frontend to Firebase, we used the official `firebase_core` and `cloud_firestore` packages. The initialization of Firebase was performed in the main function, ensuring all database functions were available across the app. Each screen that required data interaction, such as Maintenance Book and Dashboard Scanning, used Firebase CRUD operations through controller functions. These controllers handled create, read, update, and delete tasks based on user actions on the UI.

4.4 Maintenance Book Functionality

The Maintenance Book screen was backed by Firestore collections. When a user added a new maintenance task such as battery replacement or oil filter change, a document was created in the 'maintenance_logs' collection with fields including date, mileage, price, and repair center. Each entry was stored under the user's unique UID obtained from Firebase Authentication. We also implemented edit and delete functionality by using Firestore's update and delete operations respectively.

4.5 Autre – Other Feature

The Autre – Other form was used to collect additional car details like brand, model, and email. These values were saved to Firestore under the respective user document. When users submitted the form, the data was persisted and used to update the home screen UI. This ensured a consistent and personalized user experience, with data saved securely.

4.6 Dashboard Scanning Feature

To support dashboard fault scanning, we used a structure that mapped detected codes to explanations. Fault code entries were retrieved from Firestore when scanned, and matched against existing documents. This setup enabled users to get instant feedback. Data caching was added for efficiency, and error-handling mechanisms were included to gracefully manage missing or malformed fault codes.

4.7 Engine Sound Recording & Storage

We implemented local recording using flutter_sound and stored recordings in Firebase Storage. Upon stopping a recording, the audio file was uploaded to a Firebase Storage bucket, and its URL saved to the Firestore user document. This enabled future access and playback across devices. For standard comparison, a preloaded reference file was stored in Storage.

4.8 Security Rules

We implemented Firebase Authentication to manage user sign-in and secure their data. Only authenticated users could create, view, or modify their records. Firebase

Security Rules were customized to restrict access based on user IDs and protect sensitive fields. These measures ensured a high level of data integrity and user privacy.

4.9 Optimization and Performance Tuning

To ensure optimal performance, we indexed commonly queried fields in Firestore such as vehicle ID and user ID. We used batch operations to minimize network overhead and leveraged local caching for frequently accessed data. Lazy loading was introduced for large lists like maintenance history and fault logs, improving responsiveness.

5.0 Backend Implementation

5.1 Introduction to Backend Implementation

The backend of the Car Fault Finder app is responsible for managing persistent data, enabling cross-device access, comparing engine sounds, scanning dashboard lights, and storing car maintenance logs. Firebase, a robust Backend-as-a-Service (BaaS) platform, was chosen for its real-time data capabilities, cloud storage, scalability, and ease of integration with Flutter.

- Core backend roles in this app:
 - Storing car information (Autre – Other).
 - Saving and retrieving engine sound recordings.
 - Comparing recorded vs standard engine sounds.
 - Dashboard warning light scanning.
 - Recording and retrieving maintenance logs per car.
 - Optional user-specific data with Firebase Authentication.

5.2 Firebase Services Used

To implement a robust and scalable backend, the following Firebase services were used:

1. Firebase Firestore (NoSQL Database)
 - Stores structured car data and maintenance book entries.
 - Each user or car has its own document for clean separation.
2. Firebase Storage
 - Used to upload and retrieve engine sound recordings.
 - Helps store larger media files efficiently.
3. Firebase Authentication (Optional)
 - Identifies users and secures their data (if login is enabled).
 - Each user's data is scoped under their UID.

4. Firebase Cloud Functions (Future)

- Can be used later for automatic engine sound analysis or sending reminders for maintenance.

5.3 Backend Data Model & Structure

Car Info Storage (Autre – Other)

Collection: cars

Document ID: auto-generated or user ID

Fields:

- brand: "Toyota"
- model: "Corolla"
- email: "user@email.com"

Engine Sound Recordings

Collection: recordings

Document ID: userID/timestamp

Fields:

- userId: "abc123"
- localPath / cloudURL: "engine_sounds/user123/audio1.aac"
- comparedResult: "Match" / "Noise"

Maintenance Book Entries

Collection: maintenance_logs

Document ID: logID (auto-generated)

Fields:

- userId: "abc123"
- carBrand: "Toyota"
- carModel: "Corolla"
- description: "Oil changed"
- date: Timestamp

Dashboard Scan Results

Collection: dashboard_scans

Document ID: scanID (auto-generated)

Fields:

- userId: "abc123"
- carBrand: "Toyota"
- warningLight: "Check Engine"
- description: "Possible misfire or sensor issue"
- timestamp: DateTime

5.4 Backend Logic – How It Works

Autre – Other Flow

1. User taps the Autre – Other card on the HomeScreen.
2. App navigates to a form screen.
3. User selects brand, enters model, optional email.
4. On tapping Continue:
 - Data is stored locally (Shared Preferences) and synced to Firestore.
 - Home screen is updated to reflect the selected car info.

Engine Sound Comparison Flow

1. User records engine sound using flutter_sound.
2. Recording is saved locally and uploaded to Firebase Storage.
3. App compares it to a bundled standard sound file.
4. The result (“Noise” or “Match”) is stored under recordings.

Dashboard Scan Flow

1. User captures dashboard using camera.
2. Image is analyzed with image recognition logic.
3. Recognized warning light is mapped to known fault.
4. Result is stored in Firestore and shown on screen.

5.5 Maintenance Book Feature

Adding Maintenance Entries

1. User opens Maintenance Book screen.
2. Enters car brand, model, and description of maintenance.
3. Entry is saved to Firestore under maintenance_logs.

Retrieving Maintenance Logs

- On app startup or page load, logs are fetched from Firestore.
- Data is filtered by current car brand/model or user ID.
- Displays in list form: date, task, etc.

Structure Example:

```
{
  "userId": "abc123",
  "carBrand": "Toyota",
  "carModel": "Camry",
  "description": "Changed brake pads",
  "date": "2025-06-08T10:00:00Z"
}
```

5.6 Security, Sync & Future Enhancements

Firebase Security Rules

Access is restricted by userId:

```
match /cars/{carId} {
  allow read, write: if request.auth.uid == resource.data.userId;
}
```

Ensures that only the user who added data can access it.

● Data Sync and Persistence

- Local: SharedPreferences ensures the Autre info remains across sessions.
- Cloud: Firebase Firestore keeps car data, maintenance logs, and audio accessible across devices.

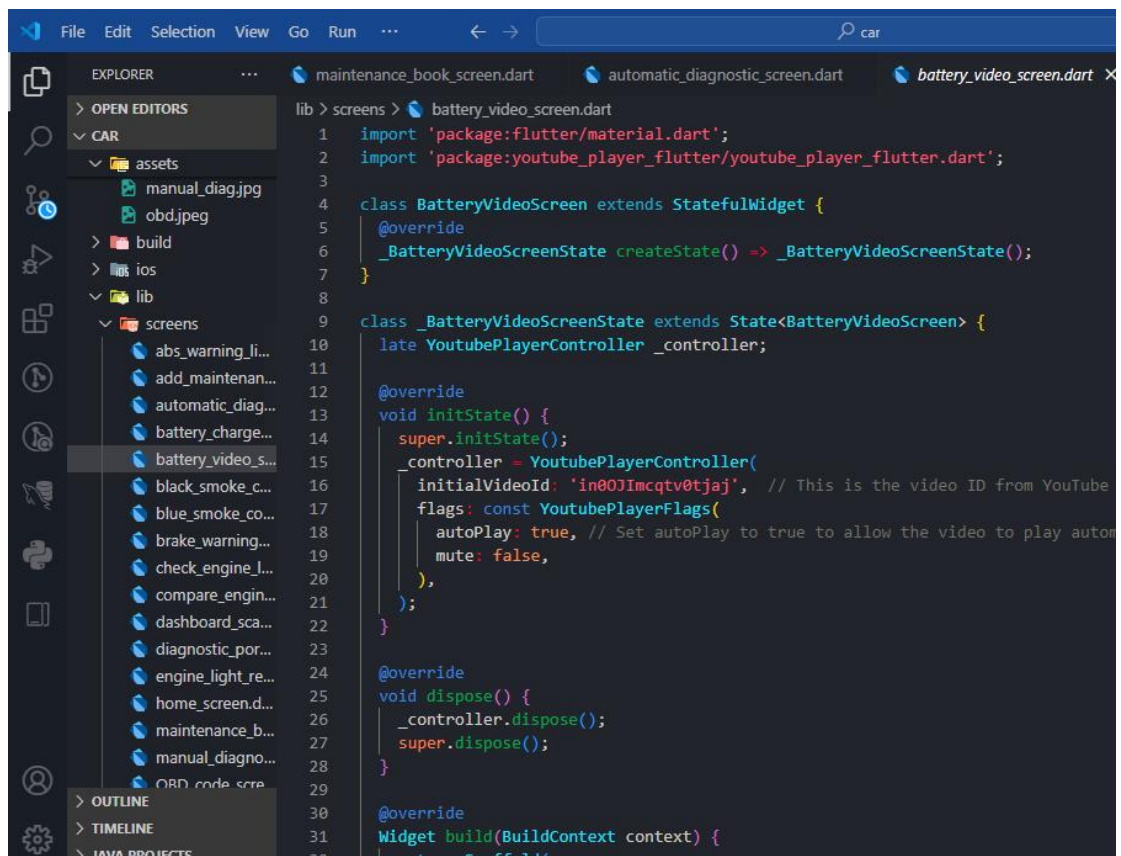
- Scalability and Future Enhancements

- Move to full offline/online sync with caching.
- Add AI fault detection using Firebase ML Kit.
- Schedule maintenance reminders using Cloud Functions.
- Add export/print of maintenance history in PDF format.
- Enhance dashboard light scanning with real-time suggestions.

Summary:

The Firebase backend handles car data, engine sounds, dashboard scans, and maintenance logs, ensuring data integrity, security, and real-time updates. This scalable backend lays a strong foundation for advanced diagnostic tools and personalized car care in future updates.

Below is the backend implementation for battery video.



```
File Edit Selection View Go Run ... car
EXPLORER
  OPEN EDITORS
  CAR
    assets
      manual_diag.jpg
      obd.jpeg
    build
    ios
    lib
      screens
        abs_warning_li...
        add_maintenan...
        automatic_diag...
        battery_charge...
        battery_video_s...
        black_smoke_c...
        blue_smoke_co...
        brake_warning...
        check_engine_l...
        compare_engin...
        dashboard_sca...
        diagnostic_por...
        engine_light_re...
        home_screen.d...
        maintenance_b...
        manual_diagno...
        ORB_code_scre...
    OUTLINE
    TIMELINE
    JAVA PROJECTS

lib > screens > battery_video_screen.dart
1  import 'package:flutter/material.dart';
2  import 'package:youtube_player_flutter/youtube_player_flutter.dart';
3
4  class BatteryVideoScreen extends StatefulWidget {
5    @override
6    _BatteryVideoScreenState createState() => _BatteryVideoScreenState();
7  }
8
9  class _BatteryVideoScreenState extends State<BatteryVideoScreen> {
10    late YoutubePlayerController _controller;
11
12    @override
13    void initState() {
14      super.initState();
15      _controller = YoutubePlayerController(
16        initialVideoId: 'in00JImcqtvt0tjaj', // This is the video ID from YouTube
17        flags: const YoutubePlayerFlags(
18          autoPlay: true, // Set autoPlay to true to allow the video to play autom
19          mute: false,
20        ),
21      );
22    }
23
24    @override
25    void dispose() {
26      _controller.dispose();
27      super.dispose();
28    }
29
30    @override
31    Widget build(BuildContext context) {
32      return Scaffold(
```

6.0 Connecting Database to Backend

6.1 Introduction

The Car Fault Finder app uses Firebase as a cloud-based backend solution. To support key features such as fault detection, maintenance record-keeping, engine sound comparison, and dashboard scanning, a solid connection between the mobile front-end (Flutter) and the backend (Firebase) is essential. This document outlines the steps and implementation of connecting the Firebase database to the Flutter backend.

6.2 Firebase Setup and Configuration

To begin integrating Firebase:

1. Create a Firebase project via the Firebase Console.
2. Register your Android and iOS app.
3. Download and add the `google-services.json` (Android) and `GoogleService-Info.plist` (iOS) to respective directories.
4. Add required dependencies in `pubspec.yaml`:
 - `firebase_core`
 - `cloud_firestore`
 - `firebase_auth` (if user authentication is needed)
5. Initialize Firebase in `main.dart` using `WidgetsFlutterBinding.ensureInitialized()` followed by `Firebase.initializeApp()`.

6.3 Connecting Firestore to Flutter

Firebase Firestore is used to store structured data such as:

- Car brands, models, and user-specific details.
- Engine sound metadata.
- Maintenance book entries.
- Scanned dashboard images and results.

In our Dart code, the `FirebaseFirestore.instance` API is used to read and write data, We used the code below

```
await FirebaseFirestore.instance.collection('cars').doc(userId).set({  
  'brand': selectedBrand,
```

```
'model': carModel,  
'email': optionalEmail  
});
```

Example for reading:

```
var snapshot = await FirebaseFirestore.instance.collection('cars').doc(userId).get();
```

6.4 Maintenance Book Integration

The Maintenance Book feature lets users log tasks like oil change, tire checks, etc.

Each record is stored in Firestore:

- Collection: `maintenance_logs`
- Fields: `carId`, `taskType`, `date`, `description`, `status`

The code to add a new entry:

```
await FirebaseFirestore.instance.collection('maintenance_logs').add({  
  'carId': carId,  
  'taskType': 'Oil Change',  
  'date': DateTime.now(),  
  'description': 'Changed oil',  
  'status': 'completed'  
});
```

This allows dynamic and cloud-synced maintenance records.

6.5 Dashboard Scanning Connection

Dashboard scanning uses image input or text OCR (Optical Character Recognition) to detect warning signals.

Firebase Storage is used to store images:

```
final storageRef =  
FirebaseStorage.instance.ref().child('dashboard_images/$userId.jpg');  
await storageRef.putFile(File(imagePath));
```

Image analysis or warning icon results are stored in Firestore under `dashboard_logs` with fields like:

- `userId`, `imageUrl`, `warningsDetected`, `timestamp`

This ensures the backend tracks all scan results in sync with user profiles.

6.6 Real-Time Sync and Conclusion

Firebase provides real-time syncing. If a user adds or updates data (e.g., new maintenance task), the app UI updates automatically using snapshot listeners:

```
FirebaseFirestore.instance.collection('maintenance_logs')
  .where('carId', isEqualTo: carId)
  .snapshots()
  .listen((snapshot) {
    // Update UI with snapshot.docs
  });
```

This seamless integration allows for a responsive and dynamic experience. By connecting Firebase to the Flutter backend across features like engine sound comparison, maintenance records, and dashboard scanning, we provide reliable cloud support that scales with user needs.

6.7 Firebase Firestore Structure

In the Car Fault Finder app, Firebase Firestore is used as a NoSQL cloud database to store structured data. The collections used include:

- 'maintenance_records': includes records with fields such as name, date, mileage, cost, and service center
- 'engine_sounds': contains metadata about recorded audio files, possibly URLs if uploaded
- 'car_info': stores Autre – Other selections like brand, model, and email
- 'dashboard_scans': saves results from dashboard scanning

Each document is created or updated when a user performs a relevant action in the app. Unique document IDs are auto-generated or derived from UIDs.

6.7.1 Adding Maintenance Records to Firestore

When a user adds a new maintenance record in the app, it is saved to Firestore. The basic flow is seen below:

1. User fills out the form (e.g. battery change, date, mileage, cost).
2. A data object is constructed in Dart:

dart

```
Map<String, dynamic> data = {  
  'type': 'battery',  
  'date': '2025-06-08',  
  'mileage': 50000,  
  'cost': 100,  
  'center': 'SuperFix Garage'  
};
```

3. This is added to the `maintenance_records` collection:

dart

```
await FirebaseFirestore.instance.collection('maintenance_records').add(data);
```

This record can be later fetched, updated, or deleted as needed.

6.7.2 Engine Sound Storage & Metadata

The engine sound feature records audio locally, but if you want cloud comparison or backup, you can:

- Upload the `.aac` or `.wav` file to Firebase Storage
- Save a Firestore document with metadata:

``dart

```
await FirebaseStorage.instance.ref('sounds/user123.aac').putFile(audioFile);  
await FirebaseFirestore.instance.collection('engine_sounds').add({  
  'url': 'https://...firebaseapp.com/...',  
  'recorded_at': Timestamp.now()  
});
```

This allows comparing the sound with a standard sample (also stored in the app or cloud).

6.7.3 Saving Autre – Other Selections

When a user chooses their car info (brand, model, and email), the selections are stored locally first. To save to Firebase:

- Structure the data:

```
```dart
Map<String, dynamic> carData = {
 'brand': 'BMW',
 'model': 'X5',
 'email': 'user@example.com'
};
```
```

- Save it to Firestore:

```
```dart
await FirebaseFirestore.instance.collection('car_info').doc(userId).set(carData);
```
```

This allows personalization of the app interface based on stored car info.

6.7.4 Dashboard Scanning and Firebase

For dashboard scans, you can store scan results for analytics or reference. Each scan might contain details like scanned icon, result, and time:

```
```dart
Map<String, dynamic> scan = {
 'symbol': 'engine light',
 'result': 'Check engine',
 'timestamp': FieldValue.serverTimestamp()
};
await FirebaseFirestore.instance.collection('dashboard_scans').add(scan);
```
```

This data can later be visualized or shared with mechanics for remote diagnostics.

6.7.5 Backend Security and Access Rules

When integrating Firebase, it's essential to protect data with Firestore rules. The code is seen below

```
bash
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId} {
      allow read, write: if request.auth.uid == userId;
    }
    match /maintenance_records/{docId} {
      allow read, write: if request.auth != null;
    }
  }
}
```

● Advanced Data Modeling in Firebase

In the context of the Car Maintenance App, Firebase Firestore is used to model car information, maintenance records, and user data. The collections are structured as follows:

- `/users/{userId}/vehicles/{vehicleId}/maintenance/{maintenanceId}`
- `/users/{userId}/profile`

Each `vehicleId` can store sub-collections such as `maintenance`, where each entry has fields like `maintenanceType`, `date`, `mileage`, `cost`, and `serviceCenter`.

This structure provides scalability and clarity, allowing efficient querying and aggregation of maintenance records per vehicle.

● **Firebase Cloud Functions for Business Logic**

Firebase Cloud Functions can be integrated to offload complex tasks from the client app, such as:

- Validating maintenance data before it is stored.
- Sending push notifications after a record is added.
- Comparing recorded engine sounds against a standard template.

This serverless approach ensures security and performance while centralizing key logic.

● **Integrating Dashboard Fault Scanning with Firebase**

Dashboard scanning results, like warning lights or fault codes, can be stored under:

``/users/{userId}/vehicles/{vehicleId}/scans/{scanId}``

Each scan contains timestamped diagnostic results which are categorized by severity and component (e.g., engine, battery, exhaust). This lets users track the frequency of issues and correlate them with maintenance activity.

● **Securing Firebase Access**

Firebase Rules are configured to restrict access to each user's data:

```
```json
match /users/{useremail} {
 allow read, write: if request.auth != null && request.auth.uid == userId;
}
```

This ensures data privacy and integrity. Additionally, sensitive audio files (engine recordings) can be stored securely using Firebase Storage with appropriate access control.

## ● **App-to-Backend Authentication Flow**

The app uses Firebase Authentication to sign in users. Once authenticated, a secure token allows read/write operations on Firestore and Storage. The flow:

1. User signs in via email/password.
2. Token issued by Firebase Auth.

3. Token passed on every request to Firestore or Cloud Functions.

This protects the backend and associates all operations with a user identity.

### ● **Logging and Analytics for Maintenance Events**

Firebase Analytics can be used to track actions like “Add Maintenance,” “Delete Recording,” or “Run Diagnostic Scan.” These insights help developers optimize the user experience and identify which features are most used.

### ● **Real-Time Sync of Maintenance Records**

By leveraging Firestore’s real-time listeners, the app updates the Maintenance Book screen instantly when new entries are added. This removes the need for manual refresh and enhances interactivity.

### ● **Offline Support and Sync Strategy**

Firestore offers local caching. When a user is offline and submits a maintenance entry, it is cached locally and synced when the network returns. This ensures seamless usage in areas with poor connectivity.

### ● **Audio File Processing with Firebase Storage**

Recorded engine sounds are saved as `.aac` files locally and uploaded to Firebase Storage under:

```
`/users/{userId}/recordings/{fileName}`
```

Later, they can be fetched to compare with standard samples using a machine learning model or heuristic filters on the backend.

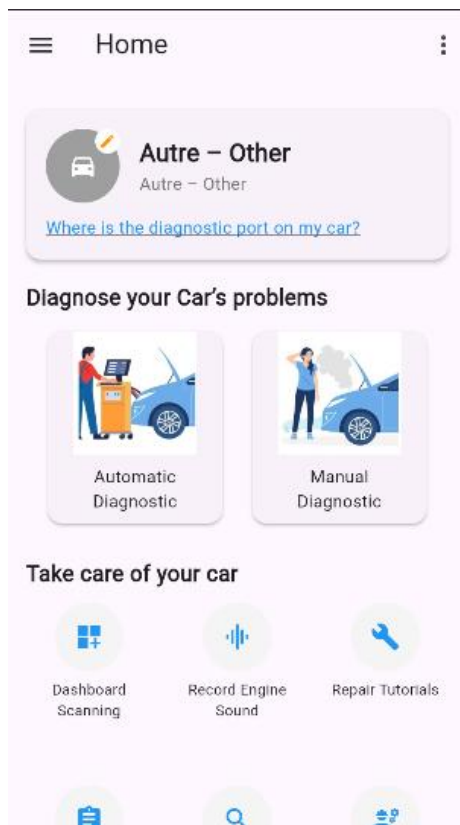
### ● **Admin Dashboard for Analytics**

An admin portal (built with Firebase Hosting + Firestore) could allow developers to:

- View user maintenance trends.
- Review fault scan frequency.
- Manage car brand datasets.

This closes the loop between mobile app users and backend analysis.

Below is the home page

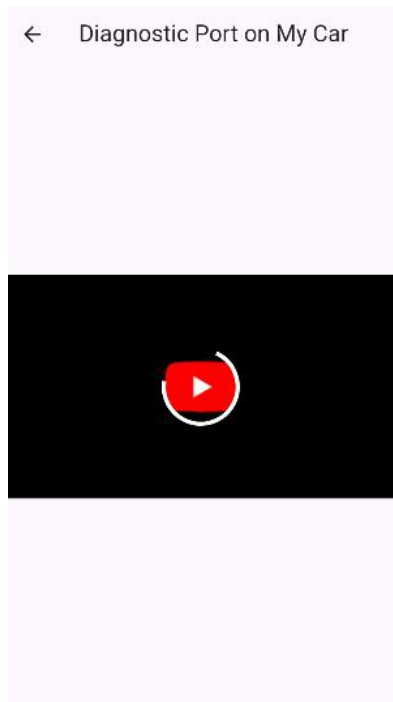


When you click on Autre-Other, it takes you to the page below where the information is stored in the database

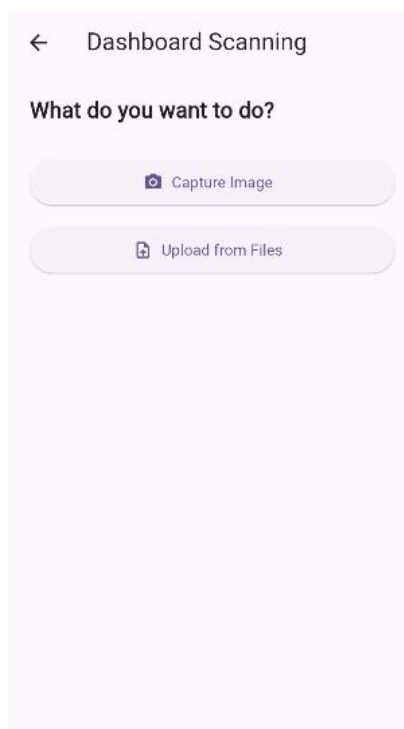
The screenshot shows a form titled "Update your car" with a back arrow icon on the left. The form contains three input fields: "Autre - Other (Brand)" with a dropdown arrow, "Autre - Other (Model)", and "Email (optional)". At the bottom of the form, there are two buttons: a red "Cancel" button and a grey "Continue" button.

From the homepage, when you click on where is the diagnostic port on my car?

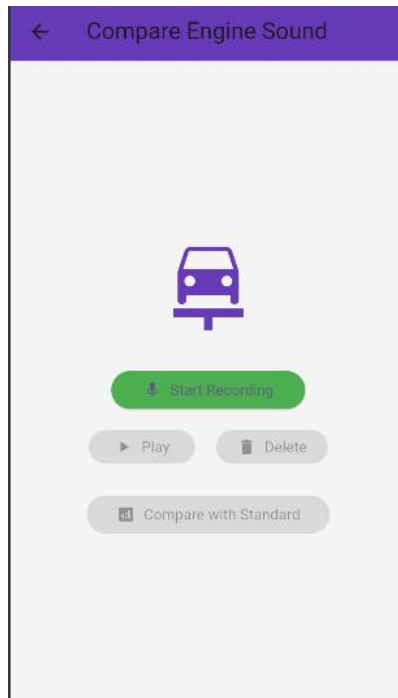
It takes you to a youtube video as shown below



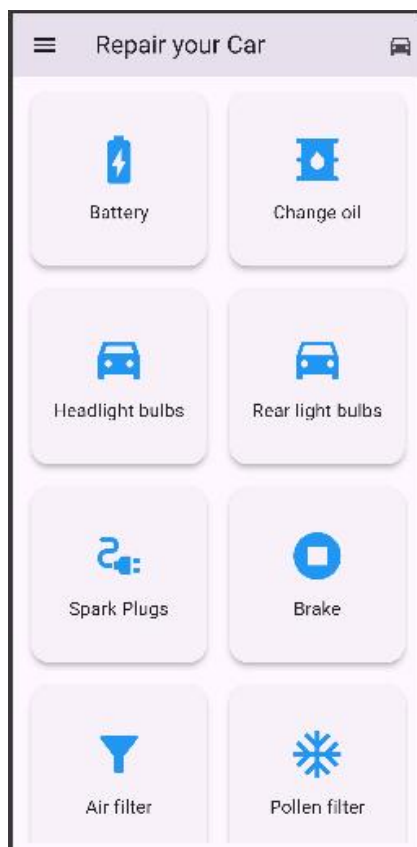
From the homepage, when you click on Dashboard scanning, it takes you to the page below, where you can take an image of the dashboard warning light for analysis. You can also upload from files.



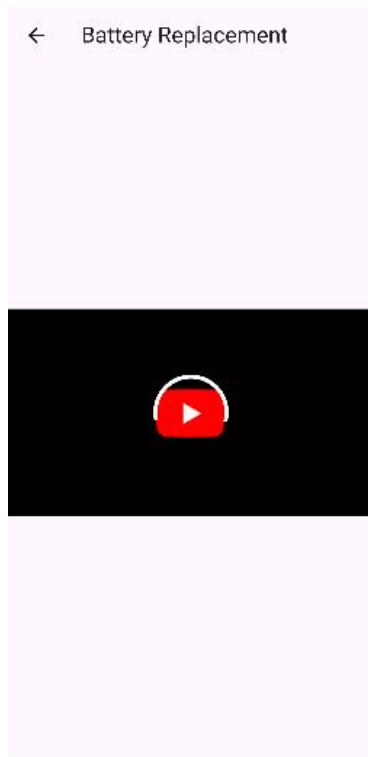
From the homepage, when you click on the record engine sound, it takes you to this page where you can record your engine sound, and get the analysis of the sound.



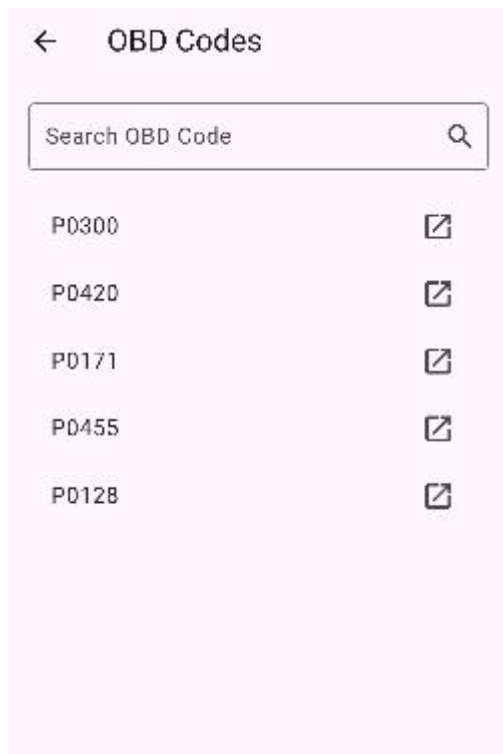
From the homepage, when you click on Repair your car, it takes you to the page below



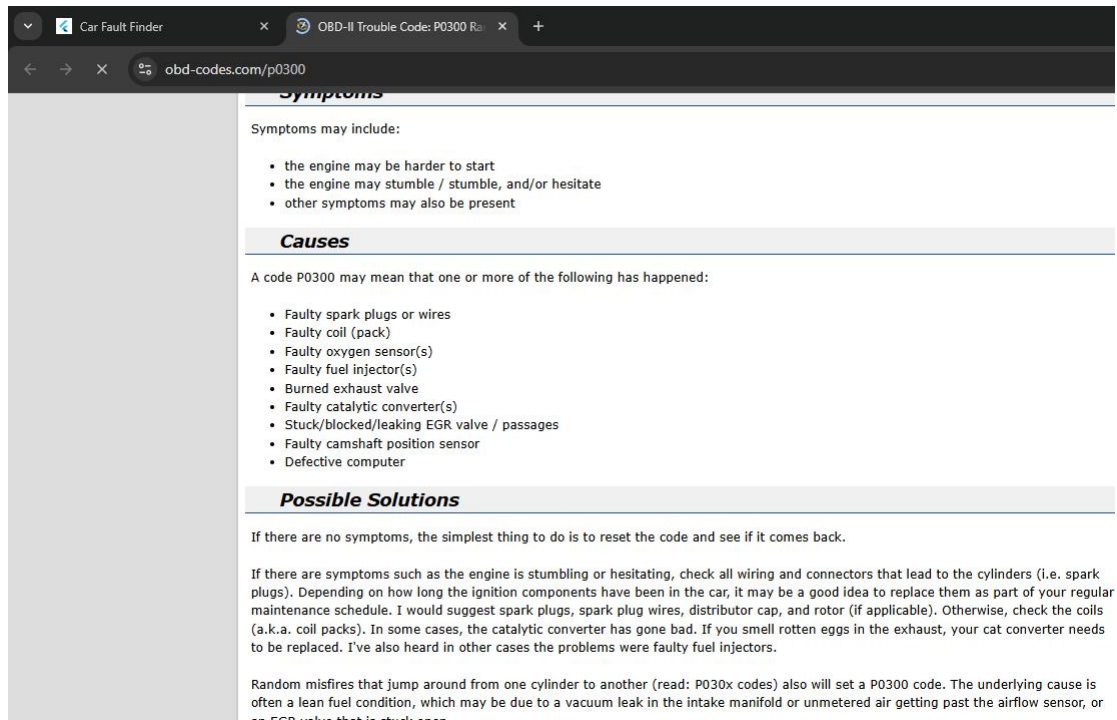
And when you click on any button, it takes you to the youtube tutorial as shown below



From the home screen, when you click on the OBD-2 code, it takes you to the page below



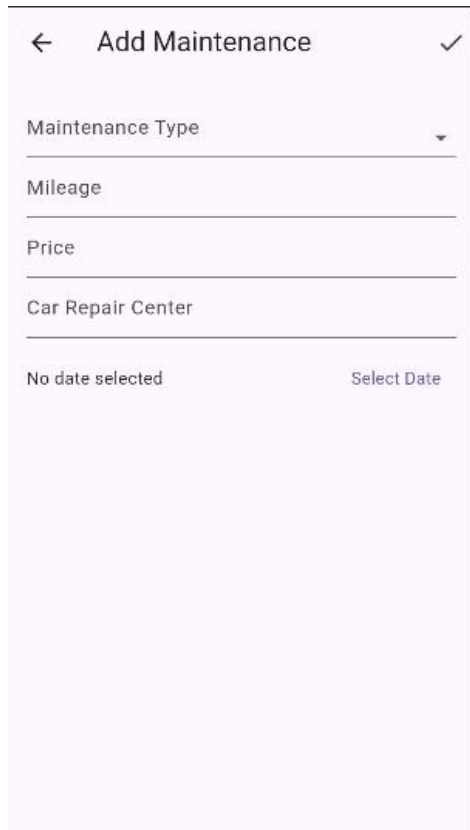
And when you search for the code, it takes you to a website where you can see and get information about the code



From the home screen, when you click on the maintenance book, it takes you to the page below. Where a user can add a maintenance

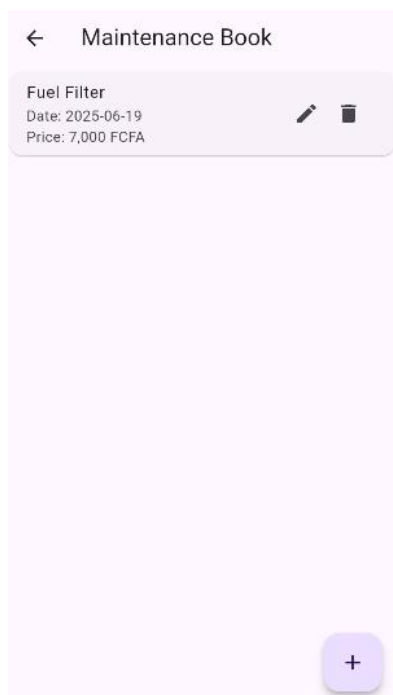


When you click on the plus sign, it takes you to this page, where you can fill in your maintenance info.



The screenshot shows a mobile application screen titled "Add Maintenance". At the top left is a back arrow icon, and at the top right is a checkmark icon. Below the title, there are five input fields: "Maintenance Type" (with a dropdown arrow), "Mileage", "Price", and "Car Repair Center". At the bottom, there is a date selection area with the text "No date selected" and a "Select Date" button.

When you click save, it is saved in the database and shown below, where you can choose to edit or delete your maintenance.



The screenshot shows a mobile application screen titled "Maintenance Book". At the top left is a back arrow icon. Below the title, there is a list item for "Fuel Filter" with the details "Date: 2025-06-19" and "Price: 7,000 FCFA". To the right of the details are two icons: a pencil (edit) and a trash can (delete). At the bottom right corner, there is a purple circular button with a white plus sign (+).



## 7.0 Conclusion

The *Car Fault Finder* mobile application represents a comprehensive solution for diagnosing vehicle faults using an intelligent, user-friendly interface. Through careful conceptual design, the application identifies and organizes key data elements such as vehicle information, fault categories, warning indicators, diagnostic results, and maintenance history. An Entity-Relationship (ER) diagram was developed to clearly define relationships among these elements, forming the foundation for a well-structured database.

The database implementation was carried out using Firebase, ensuring scalability, real-time data access, and secure storage of user and vehicle data. Backend logic was implemented using modern frameworks to manage data processing, perform fault predictions using integrated machine learning models, and provide dynamic communication between the application frontend and the database.

The successful connection of the backend to the database enables seamless data retrieval, updates, and user interactions. Together, these components ensure the *Car Fault Finder* app delivers accurate diagnostics, enhances user experience, and supports vehicle owners in maintaining their cars efficiently. This project demonstrates how mobile and cloud technologies can be harnessed to create a practical, intelligent tool for automotive health management.