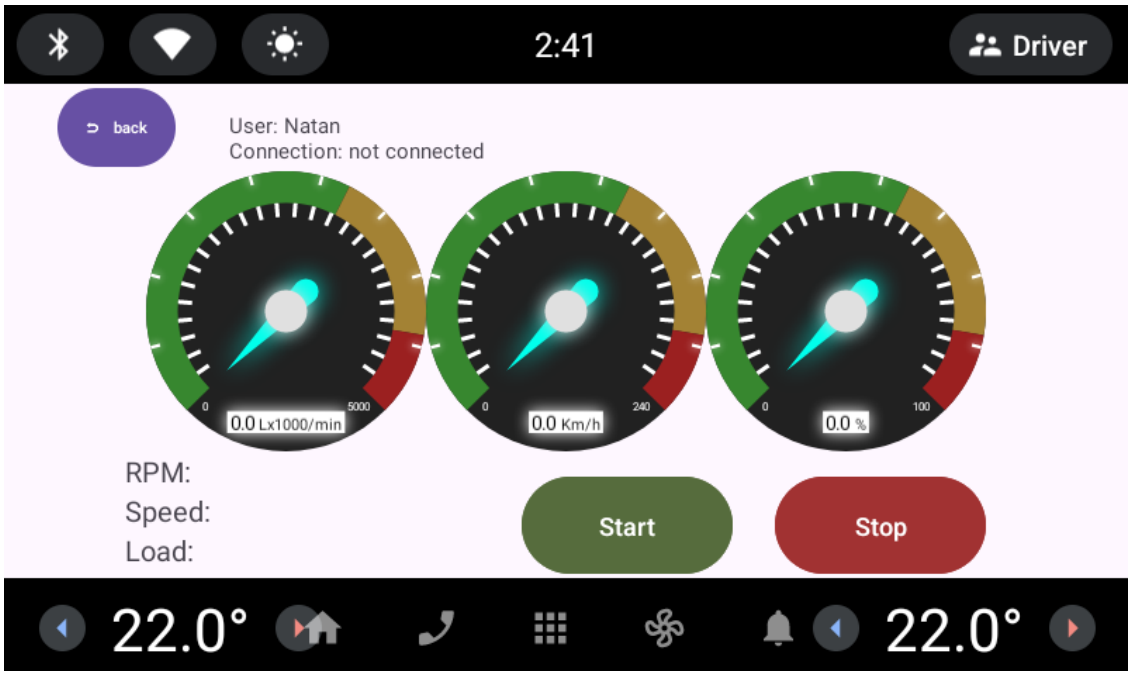
CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG



Eco-Score Tracker

Eco-score computation system for OBD-II equipped cars running on Android-based Raspberry Pi 4

Degree project report in Computer Engineering

EBRAHIM HAMDO
NÁTÁN DOBNER

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Degree project report in Computer Engineering 2024

Eco-Score Tracker

Eco-score computation system for OBD-II equipped cars running on Android-based Raspberry Pi 4

EBRAHIM HAMDO
NÁTÁN DOBNER

CHALMERS

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2024

Eco-Score Tracker
Eco-score computation system for OBD-II equipped cars running on Android-based Raspberry Pi 4
EBRAHIM HAMDO, NÁTÁN DOBNER

Supervisor: Alessio Cicero
Examiner: Lars Svensson

Degree project report in Computer Engineering 2024
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Eco-Score Tracker user interface

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2024

Eco-Score Tracker
Eco-score computation system for OBD-II equipped cars running on Android-based Raspberry Pi 4
EBRAHIM HAMDO
NÁTÁN DOBNER
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Vehicle electronic systems today have the computation power of small computers, but this technology has not always been available, as in the case of older cars. The aim of this project is to make modern technologies available for older vehicles by developing an Android application that utilizes the OBD-II protocol, and run it on a Raspberry Pi with Android Automotive OS to mimic a modern car UI. The application reads and saves vehicle data on the user's trips and provides a score based on the driving style and statistics of each trip. The application is developed in Android Studio and is tested throughout the development phase in real cars, assessing whether the concept is technically feasible. The results show that the key limitation is the OBD-II protocol itself, because, although the protocol is available in most cars, the supported commands vary drastically between car manufacturers. Regardless, the data acquisition itself from the OBD-II works well in the application, providing a score which allows the user to optimize their driving style in order to reduce their environmental impact.

Keywords: Android, OBD-II, Raspberry Pi, On-Board Diagnostics, vehicle data

Acknowledgements

This rapport is a Bachelor thesis in Computer Engineering at Chalmers University of Technology, in Gothenburg, Sweden. The project was carried out by Ebrahim Hamdo and Nátán Dobner during the spring semester in 2024, and is equivalent to 15 ECTS.

We want to thank Agoshi for sponsoring our thesis work and helping us during the project. We especially want to thank Claes Arkhult and Carl Cedergren, our supervisors at Agoshi and Alessio Cicero, our supervisor at Chalmers.

Ebrahim Hamdo & Nátán Dobner, Gothenburg, May 2024

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis, listed in alphabetical order:

AOSP	Android Open-Source Project
DAO	Data Access Object
GPIO	General-Purpose Input/Output
IDE	Integrated Development Environment
OBD	On-Board Diagnostics
OS	Operating System
SDK	Software Development Kit
UI	User Interface
USB	Universal Serial Bus
VHAL	Vehicle Hardware Abstraction Layer

Nomenclature

Below is the nomenclature of variables that have been used throughout this thesis.

Variables

S_T	Total score
F_{EC}	Eco factor
F_{EF}	Efficiency factor
F_S	Sub factor
F_{RPM}	RPM factor
F_{SP}	Speed factor
F_L	Engine load factor
C_L	Fuel consumption in liter
C_P	Fuel consumption in percent
C_R	Fuel consumption rate
C_{RO}	Optimal fuel consumption rate
D	Distance
T	Tank volume
V_A	Average value
V_T	Target value

Contents

List of Acronyms **ix**

Nomenclature **xi**

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**
1.1 Background . . . 1
1.2 Purpose . . . 1
1.3 Objectives . . . 2
1.4 Limitations . . . 2

2 Technical background **3**
2.1 Raspberry Pi . . . 3
2.2 OBD-II . . . 4
2.3 Android Automotive . . . 4
2.4 Android Studio IDE . . . 5
2.5 Kotlin . . . 5
2.6 Android application components . . . 6
2.7 Kotlin OBD API . . . 6
2.8 SQLite and Room . . . 7

3 Methods **9**
3.1 Technical prerequisites . . . 9
3.2 Project approach . . . 9
3.3 Git . . . 9
3.4 Development phase . . . 9
3.5 Testing and verifying . . . 10

4 Implementation **11**
4.1 Building and installing Android . . . 11
4.2 Visualizing the application system . . . 11
4.3 Database structure . . . 11
4.4 Bluetooth connection . . . 12
4.5 OBD-II Data service . . . 13

4.6 Trip Activity . . . 14
4.7 Sequence Diagram . . . 15
4.8 User interface . . . 15
4.9 Database Management . . . 16

5 Results **19**
5.1 Android on the Raspberry Pi . . . 19
5.2 Home page . . . 19
5.3 Connecting OBD-II adapter . . . 20
5.4 Set target values . . . 20
5.5 Statistics page . . . 21
5.6 Trip page . . . 21
5.7 Testing the application . . . 22
5.8 User tests . . . 22

6 Conclusion **27**
6.1 Bluetooth vs. USB . . . 27
6.2 Results and objectives . . . 27
6.2.1 Reflecting on the application . . . 27
6.2.2 Database and fuel measurement challenges . . . 28
6.2.2.1 Database management issue . . . 28
6.2.2.2 Fuel measuring challenges . . . 28
6.2.3 Score system assessment . . . 29
6.3 Improvement areas . . . 30
6.3.1 Saving data . . . 30
6.3.2 Data update frequency . . . 31
6.3.3 Fuel level measurement discussion . . . 31
6.3.4 Updating data on the UI . . . 31
6.3.5 Concurrency . . . 32
6.4 Future work . . . 32
6.4.1 Other approaches . . . 32
6.4.2 Alternative score system . . . 32
6.5 Summary . . . 33

List of Figures

2.1 Raspberry Pi 4b 3
2.2 OBD-II adapter 5
2.3 OBD-II pin layout 6

4.1 UML 12
4.2 Entity relationship diagram. Squares in the diagram represent entities, oval shapes are for attributes of these entities, and rhombus shapes stand for the type of relations between entities 13
4.3 Sequence Diagram of logging in and starting a trip 16

5.1 Android Automotive home page 20
5.2 Home page 21
5.3 Set target values page 22
5.4 Statistics page 23
5.5 Trip page 24

List of Tables

2.1 OBD command and response 4

5.1 Table for user Ebrahim 24
5.2 Table for user Natan 24
5.3 Comparing trip 1 and 5 in Table 5.2 25

1
Introduction

This chapter introduces the context and purpose of the project alongside with the objectives aimed to be achieved. Also presents limitations that the project has with the software and hardware components.

1.1 Background

In modern vehicles electronics are getting more and more complicated changing the way how people interact and what they expect to have in their cars. The interaction with the convenience features in a car have changed from knobs and physical buttons to large touchscreens that handle both information and entertainment purposes, today commonly referred to as infotainment systems. These systems can have numerous features compared to older cars, such as integrated control of the car's functionalities and giving better feedback on vehicle's usage. However, these advanced computing capabilities are not necessarily utilized, nor are they available in older cars.

The aim of this project is to make modern technologies available for older vehicles by developing an Android application that can monitor data from the vehicle using the OBD-II protocol. By utilizing the OBD-II the application is accessible for cars manufactured long before today's modern infotainment systems, going back as far as 2003, when the OBD-II protocol became mandatory for all vehicles in the EU [1]. The app aims to open up possibilities for the driver to get live information about the state of the car, as well as to give a score from an environmental perspective on how the driver's driving style affected each trip they took.

The project was inspired and supported by Agoshi [2], which is one of four sister companies of the Cilbuper group [3]. Agoshi works in the IT and technology industry with the aim of developing businesses and preparing employees for the next step in their career. The company works in several service areas such as DevOps and Android consulting for digital transformation and technology solutions.

1.2 Purpose

The aim of the project is to develop a proof of concept for an Android application that allows car users to read detailed statistics about their driving style on their trips such as fuel consumption, engine Revolutions Per Minute (RPM) and more. The

Android application is used to find a correlation between these values and provide feedback to the users in the form of a score system. The concept includes the use of a Raspberry Pi running Android Automotive and a touchscreen as the main hardware modules for the application. This represents a use case where the user's car either has an Android infotainment system capable of running the application, or where the user installs the same system as used in this project and runs the application on it. Since the project requires only a few simple components like an OBD-II adapter and a Raspberry Pi with a screen, it can be easily applied in older cars as well, essentially all cars that have an OBD-II port. This gives an environmental friendly aspect to the project by making modern technologies available for old cars.

1.3 Objectives

The main goal for this project is to reach these objectives:

1. Develop an Android application which can run on Android Automotive OS in order to determine if the concept is technically feasible.
2. Provide statistics in the application, including a score system, to compare data between drivers and their trips.
3. Test the application with real user tests in order to determine whether the score system is working as expected, distinguishing between different driving styles.

1.4 Limitations

The application's features are limited based on the time constrains of the development phase and are as follows:

- The project is limited to developing an application that runs on Android devices, and the User Interface (UI) is specifically adjusted to fit the screen used in the project.
- The hardware is limited to a vehicle with an OBD-II port.
- The project uses a Bluetooth adapter to connect the Raspberry Pi to the vehicle's OBD-II port.
- The car used in the project should have the ability to give fuel level data via its OBD-II port. The fuel level is necessary when calculating the score, without it the score does not represent a comprehensive value.
- The application does not have the feature to read error codes from the vehicle's OBD-II port it only retrieves live-data.
- The UI is limited to practical standards and is not intended to focus on aesthetics or interaction design.

2
Technical background

This chapter presents the technologies, hardware, and software used during the project.

2.1 Raspberry Pi

The Raspberry Pi [4] is a microcomputer with the size of a bank card that was developed by the Raspberry Pi Foundation, and the exact model used for the project can be seen in Figure 2.1. The computer has a built-in Arm processor, and it has a pair of USB 2.0 and 3.0 ports, two micro HDMI ports, an Ethernet port and a 3.5 mm audio jack. Along with these standard ports, a 32 pin General-Purpose Input/Output (GPIO) interface can also be found, as well as ribbon cable connection for both displays and cameras, or other sensors. Additionally, it offers Wi-Fi and Bluetooth connections for wireless interfacing. These features make the Raspberry Pi a good option for the project because of its small form factor it offers compatible peripherals and is relatively powerful not to mention its affordable price point. As an interface the project used a touch screen with direct support for connecting and mounting the Raspberry Pi on the back of the screen.

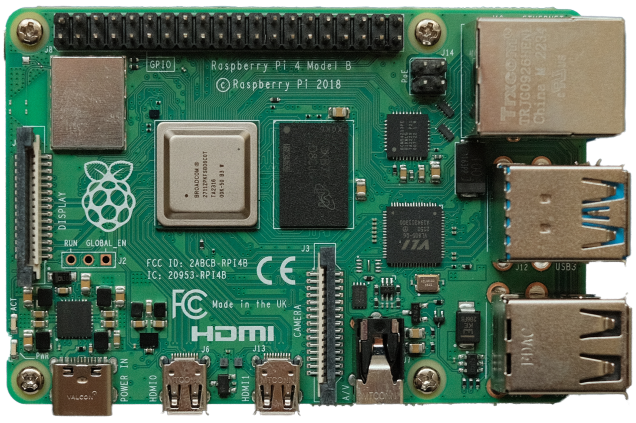


Figure 2.1: Raspberry Pi 4b

2.2 OBD-II

OBD-II stands for the second version of the On-Board Diagnostics protocol and was originally developed in the USA with the intention to detect vehicle engine issues that potentially lead to increased emission levels. It is built into almost all types of cars. Today, its main purpose is to quickly read engine diagnostics and error codes [5].

To interface with the OBD-II a Bluetooth adapter can be used, such as the one shown in Figure 2.2. This plugs in to the car's female port which is usually located under the steering wheel. The adapter has the 16-pin standard plug as shown in Figure 2.3.

Retrieving data from the OBD-II adapter can be done by sending a command from the software connecting to the adapter and getting back the response for this command. Commands and responses are OBD messages represented as hexadecimal code, like the following example for retrieving the vehicle speed: 01 0D 55 55 55 55 55. The first element, 01 is the program mode, in this case "current data" mode, 0D is the parameter ID (PID), in this case for the vehicle speed, and 55 is unused data. The response for this command would look like this: 41 0D 32 AA AA AA AA. Here 41 represents the mode chosen in the request, 0D is again the desired parameter repeated, 32 is the actual response value represented as a hexadecimal, vehicle speed in this case when parsed represents 50 km/h. The remaining AA values are unused in this example but can represent response values for other parameters which have longer responses, such as engine RPM. To give a better overview, the command, and response is illustrated in Table 2.1.

Message type	Raw message	Mode	PID	Result (Hex)	Parsed result
Command	01 0D 55 55 55 55 55	01	0D	-	-
Response	41 0D 32 AA AA AA	41	0D	32	50

Table 2.1: OBD command and response

2.3 Android Automotive

Android Automotive is a variant of the Android operating system, with the UI designed to be used in vehicles. It also offers some extra OS layers like Vehicle Hardware Abstraction Layer (VHAL) where the car software can be connected to work seamlessly with Android [6]. To give an example, proper connection between the VHAL and the car controllers gives the ability to manage the air condition from the Android UI, which in turn allows the developer to customize the infotainment system. The operating system kernel is the same as for the Android common in smartphones or tablets; therefore applications developed for these platforms also work seamlessly with Android Automotive [7].

Figure 2.2: OBD-II adapter

2.4 Android Studio IDE

Android Studio is an integrated development environment (IDE) endorsed by Google specifically for Android development. The software was developed by JetBrains [8]. It offers built in virtual device management for debugging, and also preloaded compilers for both Java and Kotlin. Furthermore, a very useful feature is the live monitoring of Logcat records, which are a dump of system messages valuable while debugging Android applications [9].

2.5 Kotlin

Kotlin [10] is a modern programming language that is interoperable with Java. Kotlin allows the developer to reuse the code across platforms such as Android, iOS, and Windows, making it a truly versatile tool for developing applications. It is also an object-oriented language, where it uses reusable programming objects, which make the code more structured and reusable in the same program or in others. It can give the developer fast results with minimum amount of code, making the language easy to use, and because of this Google named Kotlin as their official first recommendation for Android development [11].

The other advantage is that Kotlin is very succinct: it requires much less code to do the job compared to Java, making it much less error-prone because less code generally means less error. Although the Android's Software Development Kit (SDK) is written in Java, the interoperability of Java and Kotlin makes this modern language an effective combination with Android [12].

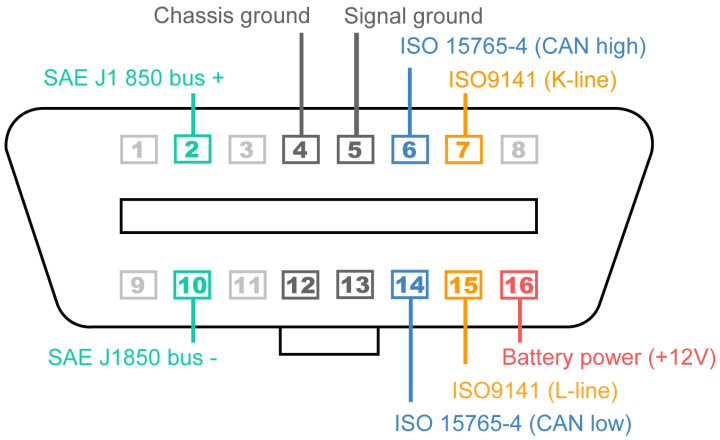


Figure 2.3: OBD-II pin layout

2.6 Android application components

Here are components used in the project, necessary to understand the structure of an Android application:

Activity In an Android application, activities are classes that primarily handle user interaction, they are the main entry points for users. Each activity is usually linked with a layout, which is an XML file incorporating the UI elements themselves [13].

Service In Android, services are components in the application performing operations in the background, executing continuously until stopped, even if the app UI is closed. For example, playing music in the background is a typical case of using a service in Android [14].

Coroutines In Kotlin, coroutines are a way to handle asynchronous operations on top of traditional threads.

2.7 Kotlin OBD API

The Kotlin OBD API is a community developed open source library on GitHub. It was started by the user *eltonvs* and implements a range of OBD-II commands, also handling the sending and receiving of OBD-II messages. The developer only needs to get hold of the input and output streams for an OBD-II Bluetooth adapter and call the appropriate function based on the desired OBD-II command. The library takes care of formatting and sending the OBD-II request and parsing the response. If the response does not contain any data or the data is not formatted correctly, an exception is thrown [15].

2.8 SQLite and Room

A flexible way to store data within an application is using databases, and for Android applications there are diverse database engines such as MongoDB, MariaDB, and SQLite. This project uses SQLite, an open-source database engine developed in C programming language, commonly used in Android development. It is used to implement file-based SQL database for storing small amount of data like the list of contacts or/and short messages. This database engine executes all operations and syntax for tables and views which are standard in SQL [16].

Room library [17] is an interface on top of SQLite which simplifies the creation, use, and maintenance of databases. Creating a database using Room library involves implementing three principal classes:

The Entity class which includes the *@Entity* tag for each table included in the database, including the column names and their types for the table.

Data Access Object (DAO) class which is an abstract interface to define all operations needed for the database table and maps method calls to it. The DAO uses annotations like *@Insert*, *@Query*, *@PrimaryKey*, and *@ColumnInfo* and interprets standard SQL queries.

Abstract database class which extends the Room database class and contains an abstract method which has DAO return type.

3
Methods

This chapter presents the methodological plans and techniques followed when starting the project planning and under the development process.

3.1 Technical prerequisites

The hardware was not an arbitrary choice, as provided directly by the company. The programming language that the company recommended, and the project uses, is Kotlin. In addition, the Linux operating system is used to build the Android Automotive OS.

3.2 Project approach

Meetings with the supervisors at the company and the university are booked depending on need and the amount of progress reached. The members of the group work in parallel by creating tasks weekly and planning for future weeks. The project loosely followed an agile working method, by using a Kanban board to track tasks and having a working version of the application at the end of every week.

3.3 Git

The version control system Git is used with GitHub to collaborate in a shared repository where both project members are given access to the code base, being able to clone it to their local machines and upload new changes to the cloud. Git not only helps with minimizing the need for physical meetings but also gives the ability to go back to previous versions of the code base in case any inconsistencies occur. Another useful feature of Git is the ability to create branches, which makes it possible to work individually on the same code base without conflicts and when time comes merge branches together.

3.4 Development phase

The firs step will be installing the operating system, after which the OBD-II adapter to the Raspberry Pi and retrieve data from the car. The development of the application takes places entirely in the IDE Android Studio and is written in the

programming language Kotlin. The main reason for choosing this IDE was that it is the official environment for Android development endorsed by Google, and thus it has extensive features specific for Android development. Another convenient reason for this choice was that Android Studio is very similar to other products of the publisher, such as IntelliJ, with which the project members had experience.

The first and most central part of the implementation is aimed to establish a connection between the application and the car's OBD-II interface via Bluetooth using an OBD-II Bluetooth adapter. To achieve and showcase this, the following must be done:

1. Establish a Bluetooth connection between the application and the OBD-II adapter.
2. Send correctly formatted OBD-II commands, then receive and parse the responses.
3. Create a view where the parsed responses update in real time.

User profiles are created to log in with multiple drivers and to save their data, a local database is established using SQLite. With the drivers' data stored in the database, the individual scores can be calculated for each trip and then visualized to the users comparatively.

3.5 Testing and verifying

The application will be tested continuously by adding terminal logs in different parts of the code to see how the code behaves in certain scenarios and to help find eventual bugs. The testing will be conducted by running the application on both the Raspberry Pi and an Android smartphone; some parts of the app are also tested on Android device emulators built into Android studio.

To test the Bluetooth connection and OBD-II communication, it is necessary to do a "hands on" test in a real car because the OBD-II Bluetooth adapter receives power and data only when plugged into the car via its proprietary pins. Testing the database however can be done at a workstation either with a physical Android device or emulator. To do this, random OBD-II responses will be generated and inserted into the database to observe its behavior when handling continuously incoming updates.

To test the application as a whole, user test will be conducted after finishing the implementation. In order to get a broad variety of results, multiple drivers shall drive the same car, taking long trips with different road conditions and simulating different driving styles. The results will be used to discuss the accuracy of the score system and determine possible miscalculations within the implementation. The discussions should help raise alternative solutions for the project, possibly improving the application and the score system in future iterations.

4

Implementation

The implementation chapter introduces the detailed design of the application, including details of the frontend and backend components.

4.1 Building and installing Android

The project scope included the installation of Android Automotive on the Raspberry Pi. The operating system is open source and as such publicly available as the Android Open Source Project (AOSP) on Google's Git repositories using the Git client Repo. To compile the AOSP, a Linux machine with the build environment established is required with a relatively large amount of storage because the source code can take up as much as 400 GB, depending on which features are downloaded [18]. To configure what version and what features of the OS is downloaded, a local manifest file [19] should be declared, which is an XML file configuring the download.

Compiling the source code can take up to multiple hours using a home PC: in the case of this project a 6 core machine was used with 32 GB of RAM, and the OS build took about 2 hours. The output file will be the OS image file, which then needs to be written onto an SD card and then the OS can be booted on the Raspberry Pi.

4.2 Visualizing the application system

The application contains a variety of classes. The UML diagram in Figure 4.1 illustrates how these classes are connected to each other and which interfaces they implement. It also shows which of the classes interact with the database.

4.3 Database structure

Figure 4.2 shows an entity relationship diagram for the database implemented in the project. This diagram shows the relations between tables, establishing the connection in the database, and relating the users with their trips, and the trips with their details. Primary keys are also indicated in the diagram as underlined attributes, for example the username in the user table is marked as a primary key to make each username unique, the same applies for the time stamp and the trip number and username as a compound key.

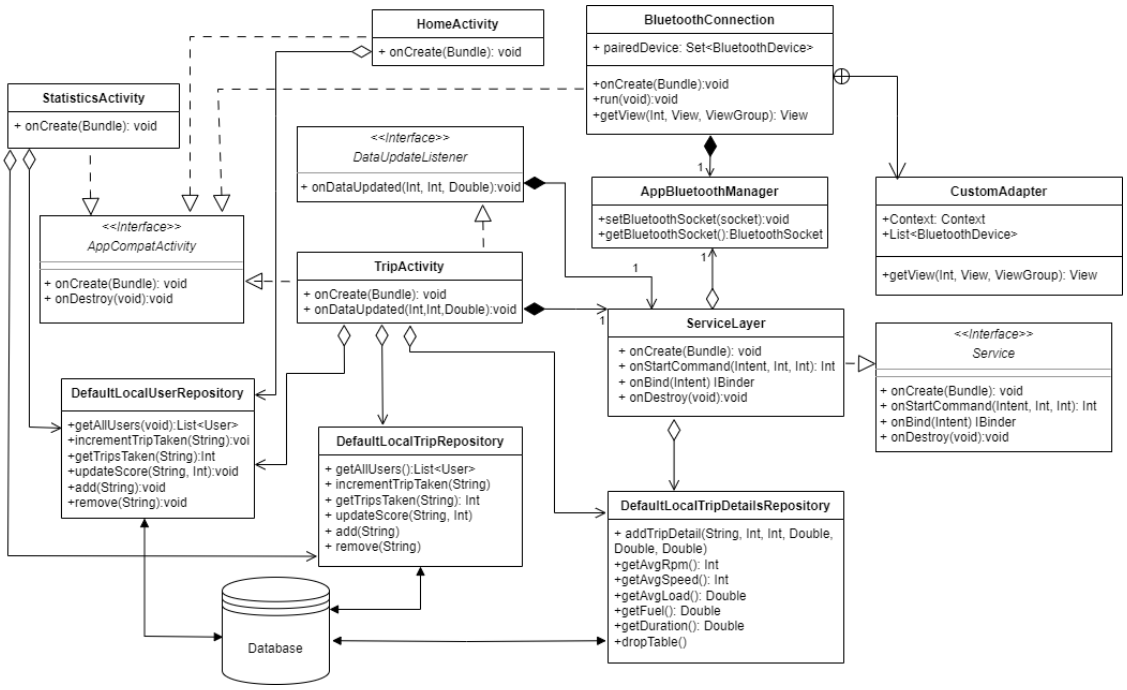


Figure 4.1: UML

4.4 Bluetooth connection

In the project, Bluetooth connection is used to communicate between the OBD-II adapter and the Raspberry Pi 4 or other Android devices running the application. The design and implementation of the Connection activity consists of the AppBluetoothManager and BluetoothConnection classes.

The AppBluetoothManager class is implemented as a singleton class and includes the management of the Bluetooth socket in the application. Implementing the singleton pattern in this class means that only one instance exists through the application's life cycle, ensuring that every application component has access to the same Bluetooth socket. This class contains only two methods, a *getter* and *setter* to manage the Bluetooth socket easily and enables the communication between the application's components, ensuring that the Bluetooth socket is the same throughout the application.

The BluetoothConnection class constitutes the principal functionality for building and managing the Bluetooth connection in the application. It handles permission requests to get hold of the device's Bluetooth adapter, and it lists all Bluetooth devices that were previously paired with the Android device. The user then (after pairing the OBD-II Bluetooth adapter) can select the desired Bluetooth device from the list, acquiring its adapter. After choosing the device to connect with, the ConnectThread inner-class takes care of establishing the connection, and handles potential connectivity problems. Once the connection is established successfully, the singleton instance of the AppBluetoothManager class is created, setting the Bluetooth socket to the newly acquired device's socket.

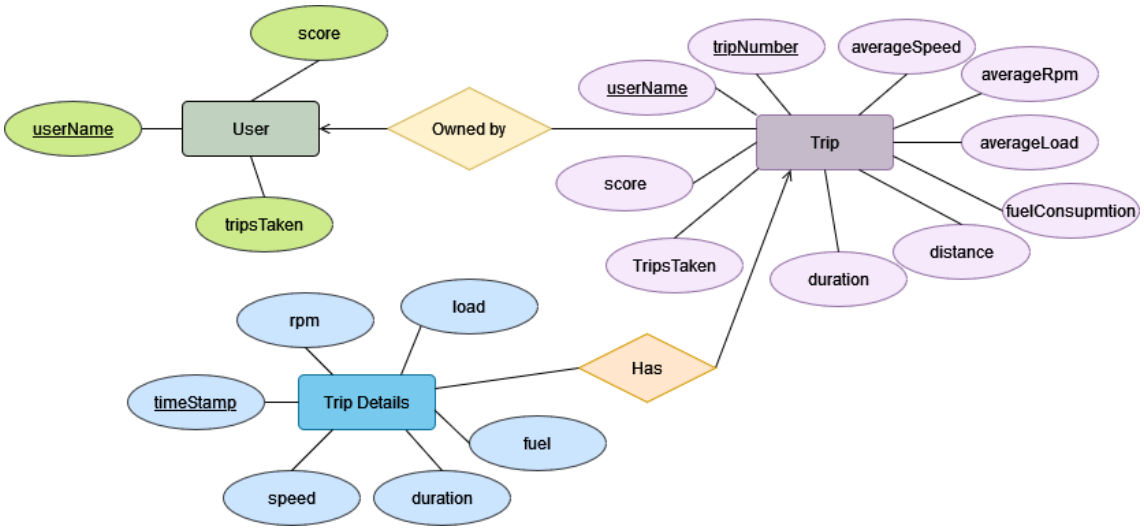


Figure 4.2: Entity relationship diagram. Squares in the diagram represent entities, oval shapes are for attributes of these entities, and rhombus shapes stand for the type of relations between entities

4.5 OBD-II Data service

To be able to send OBD-II commands and receive the responses continuously, the service layer is implemented. It runs in the background of the application even when none of the activities are active. The service layer takes hold of the Bluetooth socket for the connected adapter and initiates all communication utilizing the Kotlin OBD API. The API uses the socket's input- and output streams to send and receive information. The service starts coroutines to update the data, these coroutines then run until the service is stopped. There are three coroutines responsible for updating the values RPM, speed, and engine load. Additionally, there are two more coroutines, one informs the Trip Activity to update the UI with the new values, and the other one saves the three data points, RPM, speed, and engine load, to the Trip Details table in the database every ten seconds while the service is running.

When the service starts, it also retrieves the current fuel level of the vehicle as a percentage of the total tank capacity. The fuel level at the start and the end of a trip is measured by taking the average of 5 measurements for each one to get a more accurate estimate. In this way it is possible to compute the total fuel consumption for the trip, which is the difference between the start and stop fuel values.

Calculating the trip duration is done when service is stopped by calculating the time difference between the beginning and the end of the trip. At the end of the trip, the last insert in the Trip Details table contains, in addition to the other three parameters, the fuel consumption value and the trip duration. All these values received by the service are sent to the database and help to calculate the score later on in the Trip Activity.

4.6 Trip Activity

The Trip Activity class is the primary interaction point for the user, visualizing the Trip page of the application. It is where the service can be started and stopped (i.e., a trip can be started and stopped) with the help of a start and stop button. This activity also guarantees that the UI is continuously updated with live data of RPM, speed, and engine load from the service layer. The data is visualized on the screen both as text of the actual values and also displaying them on three different gauges using an open source library [20].

When the service is stopped, the Trip Activity gets data from the Trip Details table for the current trip and calculates the score for the whole trip based on the data. The calculations also take into account vehicle specific values such as the fuel tank volume and optimal fuel consumption rate of the car, but also target values for average engine RPM, speed, engine load set by the user. All these values set by the user are referred to as *target* values. If the user does not change these values manually, the application uses hard-coded values, referred to as *default* values. However, since every vehicle has different characteristics, the user should edit these values depending on their car. When calculating the score, these target values are compared to the trip average values to determine the points for the trip between 0 and 100.

There are two factors considered in the final score value: the *eco factor* and the *efficiency factor*. As shown in Equation 4.6 the *eco factor* represents 75% and the *efficiency factor* 25% of the total score. Within the *eco factor*, as shown in Equation 4.2, the *RPM factor* represents 40%, *speed factor* is 30%, and *engine load factor* is 30%. The efficiency factor is solely based on the fuel consumption rate of the current trip.

To calculate the *eco factor*, first three *sub factors* need to be calculated using Equation 4.1. The values used for these, are the *average values* for the trip and the *target values* inputted by the user.

$$F_S = 100 - \frac{(|V_A - V_T|)}{V_T} * 100 \tag{4.1}$$

Thereby *sub factor* is calculated for RPM, speed, and engine load, and so the *eco factor* in equation 4.2 is the sum of these sub factors multiplied by their weight in percent. If the trip *average values* derive so drastically from the *target values* that the *eco factor* gets a negative value, then its value defaults to zero, and also maxes out at 75.

$$F_{EC} = 0.4 \times F_{RPM} + 0.3 \times F_{SP} + 0.3 \times F_L \tag{4.2}$$

It is important to note that if the fuel measurement results in a negative value or zero, then the total score is only based on the *eco factor* (*efficiency factor* is ignored), which then does not have to be scaled down to 75%.

To calculate the *efficiency factor*, the values needed are the trip *fuel consumption*, trip distance, the *optimal fuel consumption rate* given by the manufacturer, and the *tank volume* of the car. Knowing that the *fuel consumption* is given in percent from the OBD-II, it has to be converted to liter by using equation 4.3.

$$C_L = \frac{C_P}{100} \times T \tag{4.3}$$

After which the average *fuel consumption rate* of the trip, given in $\frac{liter}{100km}$, can be calculated based on the trip *distance* using equation 4.4.

$$C_R = \frac{C_L \times 100}{D} \tag{4.4}$$

The *efficiency factor* represents 25% of the total score, with the maximum value of 25, and is calculated using equation 4.5.

$$F_{EF} = \frac{100 \times C_{RO}}{C_R} \tag{4.5}$$

Thereby, the total score is calculated as shown in equation 4.6.

$$S_T = 0.74 \times F_{EC} + 0.25 \times F_{EF} \tag{4.6}$$

After calculating the score, the current trip with its details is inserted in the Trip table for the current user, and the score in the User table is updated with the average score of the last five trips taken by the user.

4.7 Sequence Diagram

The sequence diagram shown in Figure 4.3 illustrates the sequence of connecting to the Bluetooth adapter and starting a trip within the application. The user presses the “Connect adapter” button on the Home activity and is redirected to the BlutoothConnection activity, where all paired Bluetooth devices are listed. Once the desired adapter is chosen, this class will retrieve the Bluetooth socket and with that initialize the only instance of the singleton class AppBluetoothManager. The user is then directed back to the Home activity, where he or she can log into his or her profile and start tracking the trip by pressing the “Start” button in the Trip Activity. This will initiate and start the service layer in the background, which in turn sends reset commands to initialize the OBD-II Bluetooth adapter and start the continuous tracking of vehicle data with the help of the Kotlin OBD API. The service, while running, continuously informs the Trip Activity to update the UI with the new values which are displayed to the user on the screen.

4.8 User interface

The project did not place great emphasis on the UI of the applications, the interface uses standard android XML components to visualize text, buttons, etc. The only special UI elements used are the gauges displaying the live RPM, speed, and engine

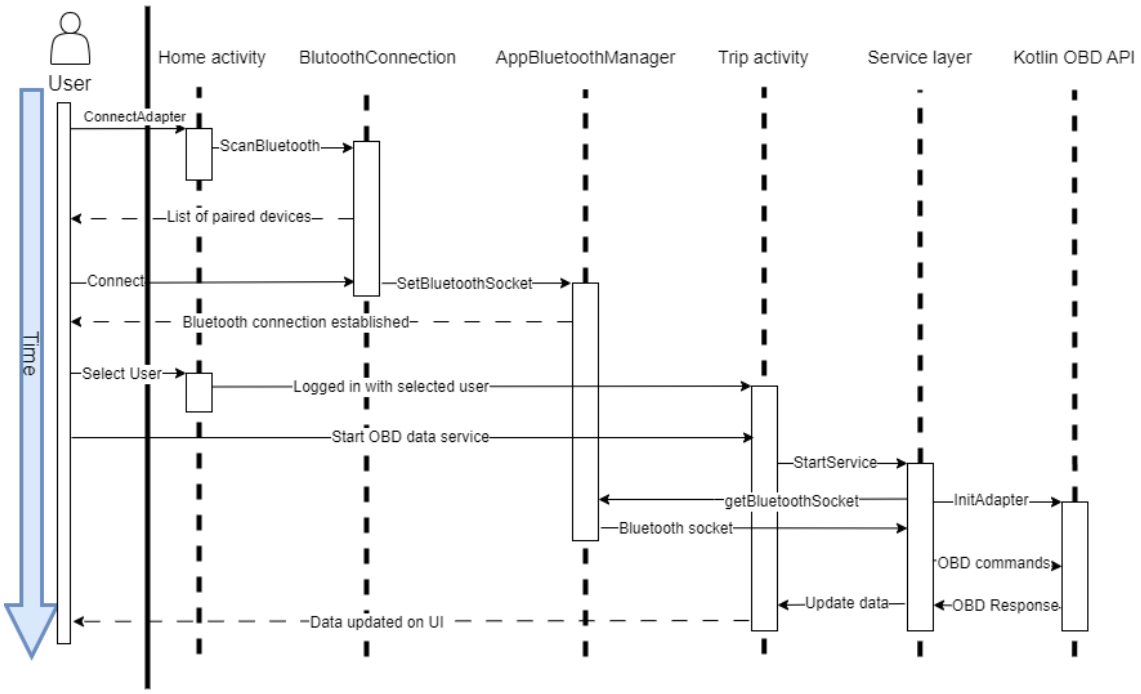


Figure 4.3: Sequence Diagram of logging in and starting a trip

load data on the Trip Activity. The gauge UI elements are imported from an open source library called Speed View [20].

4.9 Database Management

The database used in the project is implemented with SQLite using the Room interface for Android applications. The database has three tables which are User, Trip Details, and Trip as shown in Figure 4.2. Each user has a username, and every trip also has this attribute to hold the relationship between the tables. The tripsTaken attribute in the User table is used to keep track of how many trips the user has taken, and by using a database function it also helps to limit the amount of trips saved for each user to five in the table Trip. To be able to use the database queries across multiple classes, the application implements a Local Repository class for each table implementing a Data Access Object (DAO) interface.

The database is used in almost all classes of the application for different reasons. The Home activity communicates with the database for retrieving the list of users saved, and adding a new user in the Home activity adds it to the database with an initial value of 0 for score and tripsTaken.

The service layer inserts data into the Trip Details table with two different purposes, one of them is used every 10 seconds when running the service, saving RPM, speed, and engine load as a new row for the current trip. The other is used when stopping the service, saving the last row for the trip containing the fuel consumption and trip duration in addition to the other three values. The reason for making this special insert at the end of the trip is to be able to access the data in the Trip Activity.

The Trip Activity communicates with the database to get the average of engine RPM, speed, and engine load from all the rows in Trip Details inserted by the service layer and also retrieve the fuel and duration from the last row. Afterward, the Trip Activity obtains all necessary data from Trip Details and calculates the trip distance and score. Finally, it inserts a row into the Trip table, creating a new trip for the user currently logged in. The trips listed in the Trip table are used in the Statistics activity to retrieve the last five trips from the database for every user and display them in the form of tables.

5
Results

This chapter introduces the final results for different parts of the application, and also presents the user tests conducted after the implementation.

5.1 Android on the Raspberry Pi

Due to Android being designed to suit high resolution displays, on the screen used for this project visual elements are proportionally larger, some are even forced on top of together. This can be seen for example in the bottom left corner of Figure 5.1 where the home button and temperature controls overlap. This could normally be solved by setting the "Smallest width" value higher in Android's settings, but given the size of the screen (7 inch) and the resolution (800×480) Android does not allow it to be set above a certain value. Despite this, the user interface is still usable and does not hinder any functionality used for this project.

The operating system can successfully manage the Raspberry Pi's Bluetooth, Wi-Fi and USB ports. The only limitation is that to be able to connect the Pi to a PC and successfully detect it as an Android device (thus being able to install and debug the application) the PC must be connected to the Raspberry Pi's USB-C port. Due to the fact that the Raspberry Pi also uses its USB-C port as its power intake, there is a risk of under-powering it; given that PC USB port power delivery varies between devices.

5.2 Home page

On the Home page, as shown in Figure 5.2, the user is able to create new profiles and log in. The first time running the application on the Raspberry Pi, the Home page shows in the top left corner a green button called "Add User". The application cannot be used without creating a profile, and to create a profile the user needs to press the "Add User" button, which in turn shows an input field to write the username. This creates a purple button with the username on it, placing the new button on the left-hand side of the "Add User" button. This way, every user who has a profile on the application have their own "log-in" button. This means that logging in is as simple as pressing the button with the right username (no password required) and the user is directed to the main page of the application. The Home page contains three additional buttons: "Connect adapter", "Set target values", and "Statistics".

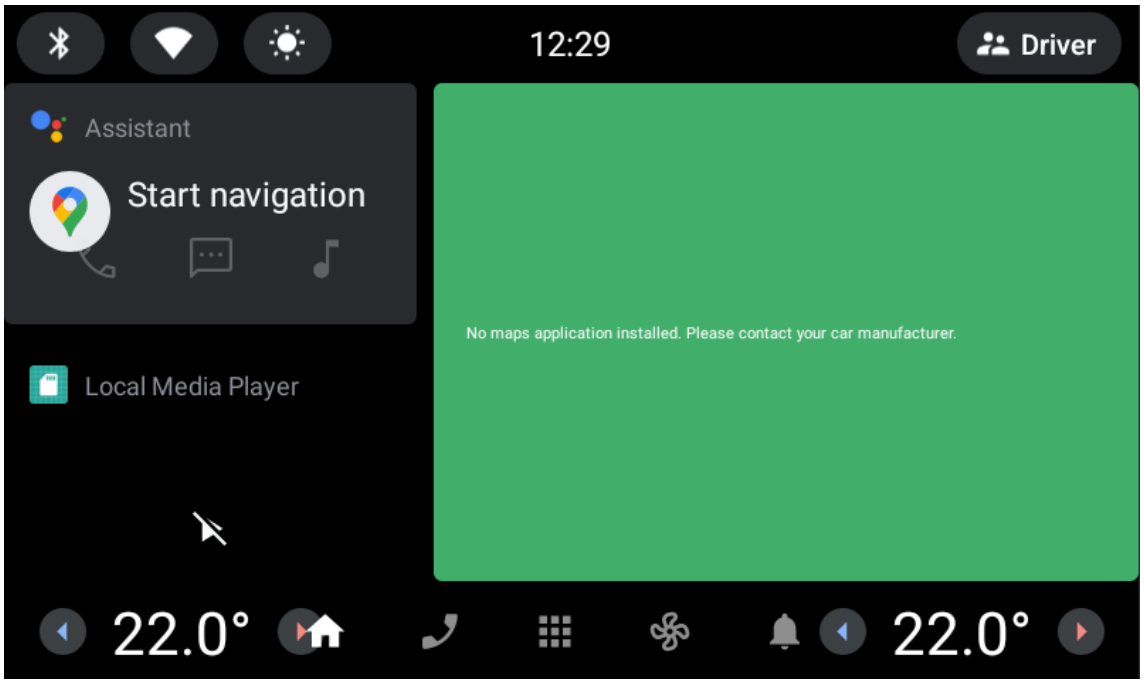


Figure 5.1: Android Automotive home page

5.3 Connecting OBD-II adapter

Pressing the "Connect adapter" button on the Home page opens a page containing a list of all Bluetooth devices previously paired with the Raspberry Pi. To connect with the desired adapter, the user needs to press on the device's name from the list, and then if the connection is established, a message is shown on the screen saying: "Connected successfully to device: *Device's name*" and afterward the user is directed back to the Home page.

5.4 Set target values

Pressing the "Set target values" button on the Home page opens an activity shown in Figure 5.3 where the user can set the target values depending on the characteristics of their vehicle. These values include the fuel tank volume in liter, the target engine RPM, the target speed in km/h, the target engine load in percent, and the optimal consumption rate in liter/100 km given by the manufacturer. After setting the new target values, the user can press on the submit button to be redirected to the Home page. If the user does not change these values, the application starts with default values, but if these are not optimized to the vehicle characteristics it can result in relatively inaccurate scores. The values are reset every time the application starts, therefore the user has to set them manually every time.

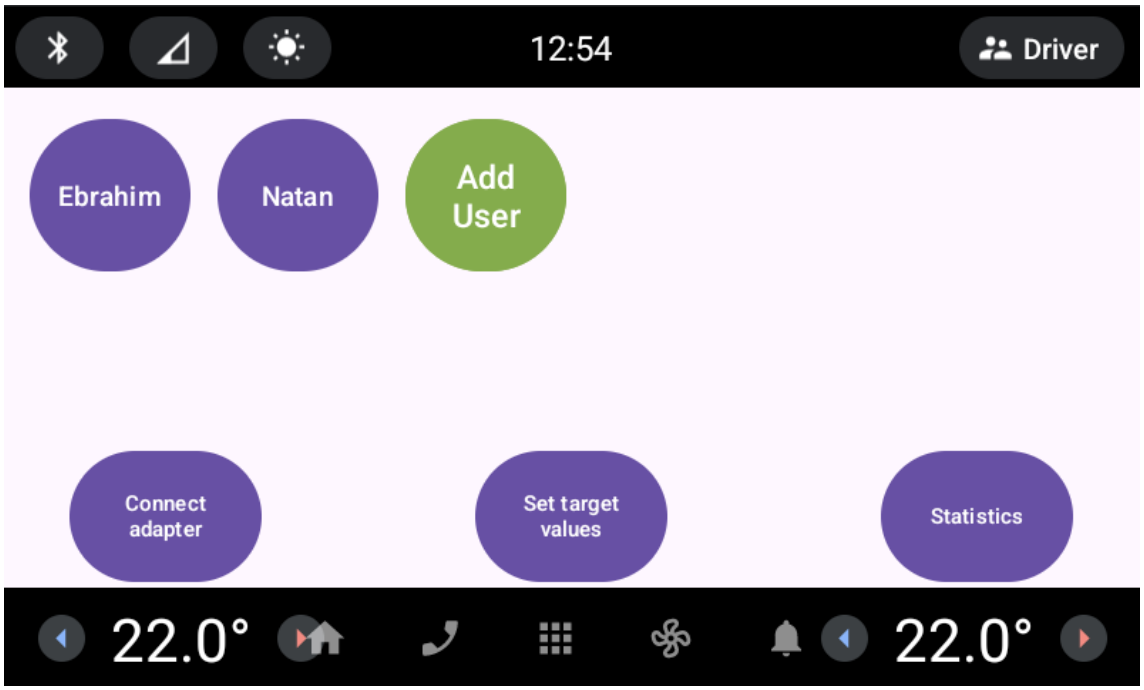


Figure 5.2: Home page

5.5 Statistics page

Pressing the "Statistics" button on the Home page takes the user to a page where the last five trips of every driver are listed in tables. As seen in Figure 5.4 only four trips are visible, the user has to scroll down to see all their trips and the trips of other users. The table shows each trip average rpm, speed (km/h), engine load (%), the percentage of fuel consumed, tip duration (hour), trip distance (km), and the trip score. The average score and the number of trips taken by each user is also visible next to the username.

5.6 Trip page

This page is where a trip can be started and stopped, it also shows the live updates of the engine RPM, speed, and engine load of the vehicle. These updates are shown on three gauges for each one of these values and also as text under the gauges as shown in Figure 5.5.

On the top of the main page, there is information about which user is currently logged in and whether the connection with the adapter was successfully established. Trying to run the application without connecting to an OBD-II device gives the user the message "Device not connected" and the trip can not be started. There is also a back button to take the user back to the Home page if needed. Pressing the start button starts the live data updating on the screen, and pressing the stop button ends the trip.

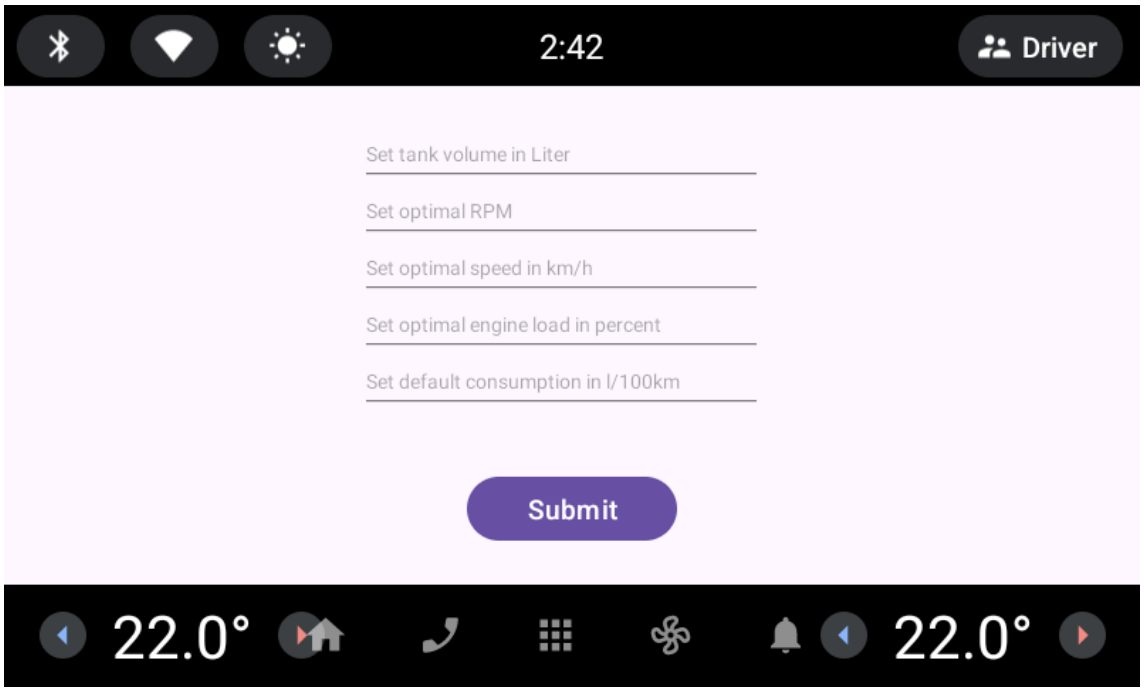


Figure 5.3: Set target values page

5.7 Testing the application

The application has been tested under the developments phase to be sure that establishing the connection works as intended. After connecting to the adapter, users can easily create their own profiles to save their trips. Setting the target values works well if the driver wants to change these values depending on the car model. Starting a trip shows live updating for values on the Trip page and starts saving data in the database. To verify the live data monitoring function, the RPM, and speed values shown in the application were compared to the vehicle's own indicators. This observation showed that, although delayed by about one second, the application can monitor the data accurately. When the trip is ended, statistics are updated with the current ended trip for the user.

To test the database under the development process, a simulator class was implemented which generates random values to simulate live data added to the database, and also updating the UI on the Trip page. The application is stable and works without interruptions even under longer trips. Due to time constrains, the longest trips the application was tested on were about 15 minutes. Starting and stopping trips takes a few seconds, but once the service is running in the background, the UI is updated continuously with the minor delay mentioned above.

5.8 User tests

As described before, the application components were tested separately before conducting the final user tests. After finishing the implementation, general user tests

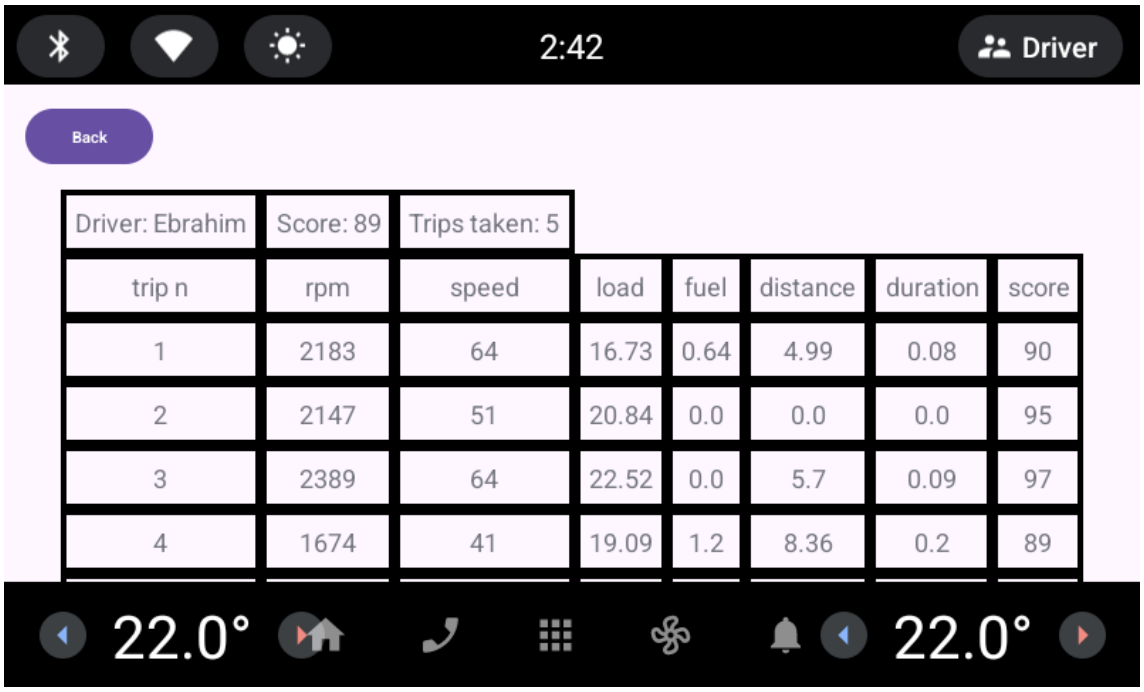


Figure 5.4: Statistics page

were carried out to analyze whether the functionality and score system meets the purpose. The application was tested by two different drivers on the same car to evaluate how different driving styles affect the score. Throughout the development, the application was tested in around 40 trips, with a distance in total about 100 km. In Tables 5.1 and 5.2 5 trips are shown for each user, enough to provide useful results for analysis. The car model used for the test is an Opel Astra H, 2006.

For this test, the target values were set to account for the specific vehicle model used for the test as follows:

- tank volume: 51 l
- target RPM: 2000
- target speed: 80 km/h
- target engine load: 25 %
- optimal fuel consumption rate: 6.1 l/100 km

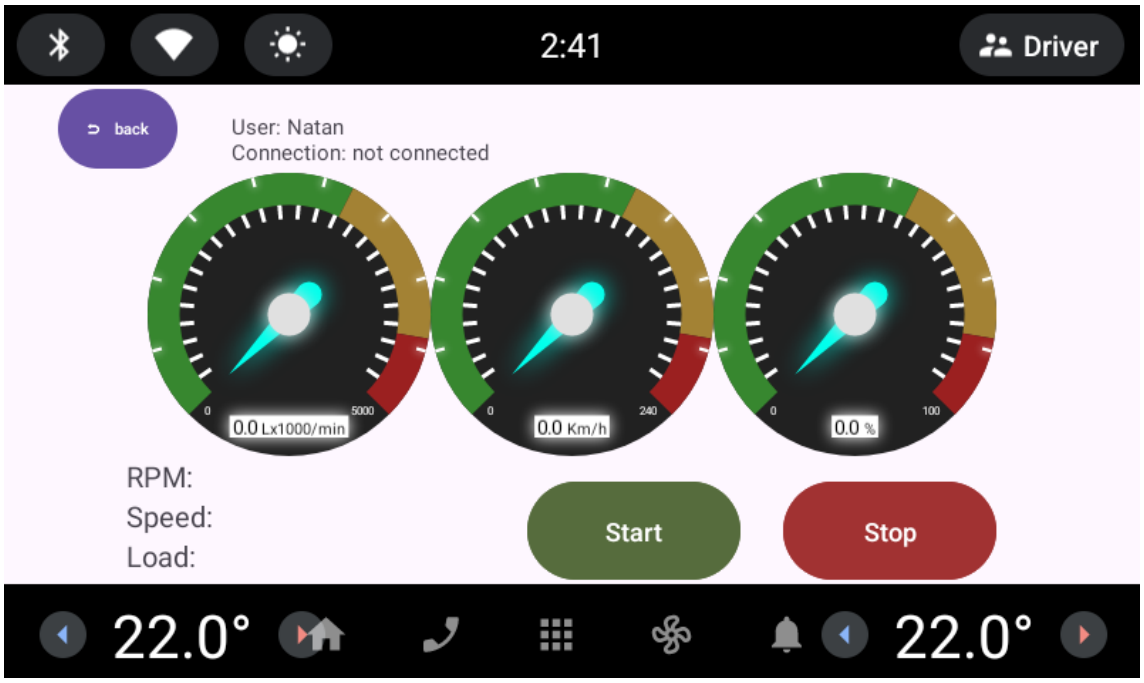


Figure 5.5: Trip page

User: Ebrahim	Score: 89	Trips taken: 5					
trip n	rpm	speed (km/h)	load (%)	fuel[1] (%)	distance (km)	duration (hour)	score
1	2183	64	16.73	0.64	4.99	0.08	90
2	2147	51	20.84	0.0	0.0	0.0	95
3	2389	64	22.52	0.0	5.7	0.09	97
4	1674	41	19.09	1.2	8.36	0.2	89
5	1586	30	16.86	1.2	3.33	0.11	75

Table 5.1: Table for user Ebrahim

User: Natan	Score: 88	Trips taken: 5					
trip n	rpm	speed (km/h)	load (%)	fuel (%)	distance (km)	duration (hour)	score
1	2026	43	17.84	1.52	5.98	0.14	79
2	2079	53	17.48	0.0	4.98	0.09	90
3	1910	45	16.52	0.08	10.67	0.24	92
4	1967	47	18.6	0.6	8.13	0.17	94
5	1804	46	17.7	0.8	5.52	0.12	88

Table 5.2: Table for user Natan

Both Tables 5.1 and 5.2 represent the same layout as the Statistics page (shown in Figure 5.4) which the user can access through the application. It can clearly be

[1]The fuel value represents the consumed fuel (in percent of the tank volume) during a trip.

	Trip 1	Trip 5
Eco factor	68	68
Efficiency factor	11	20
Total score	79	88

Table 5.3: Comparing trip 1 and 5 in Table 5.2

observed that the different values for the different parameters affect the score, and as seen for some trips the distance, duration, and the fuel consumption resulted in zero, for which an explanation is given later in the report in Section 6.2.2.2.

How the score system works can be observed clearly in Table 5.2 comparing trip one and five. The two trips have very similar distance and duration, and the average speed and load are also very close to each other. The average RPM in trip 1 is slightly closer to the used target value of 2000, but this does not affect the eco factor, as shown in Table 5.3 that both trips have the same eco factor. However, since the fuel consumption in trip one is nearly two times as much as in trip five, the efficiency factor is also almost twice as much for trip five, as seen in Table 5.3, because of the better fuel efficiency. This results in a higher total score for trip five, but only with a relatively small margin because the two trips have mostly similar results and the weight of the efficiency factor is only 25% of the total score.

A similar scenario can be observed for trip four in the Table 5.1 and trip four Table 5.2. These two trips were taken by two different drivers, and both have similar distance, around 8 km. The results for these trips show how the user Ebrahim has lower score than Natan because the fuel consumption for Natan's trip is lower than the fuel consumption for Ebrahim's trip. Considering that the average values in both trips are close to the target values, the eco factor for the two trips is similar or the same. Meaning that the higher efficiency factor for Natan's trip results in a higher score, and vice versa for Ebrahim's score.

6
Conclusion

The conclusion chapter represents discussions about methods and results reached at the end of the project, including problems and challenges encountered; it also includes other discussions about alternative solutions and future works that the project could be extended with.

6.1 Bluetooth vs. USB

This project uses an OBD-II adapter with a Bluetooth connection, but there are also OBD-II adapters with USB cables to connect with devices. This inspired the idea to give the user ability to choose between USB and Bluetooth connections when starting the application. As the Kotlin OBD API work using input and output streams, to take advantage of its ability to handle the sending and receiving of OBD-II commands, these streams need to be acquired for both Bluetooth and USB devices. After further investigating, the conclusion was made that the USB implementation should be abandoned due to several reasons:

- Accessing the streams for the USB device was proved to be a more demanding task than doing the same for the Bluetooth device. Without them, the Kotlin OBD API can not be used.
- Implementing a USB connection manager would also mean that the sending, receiving, and parsing of the OBD-II commands would have to be implemented, which was outside the scope of the project.
- In almost all European cars, the OBD-II port is placed under the steering wheel on the left side of the driver, and connecting the Raspberry Pi with an OBD-II adapter via a cable can disturb the driver.

6.2 Results and objectives

6.2.1 Reflecting on the application

Ideally and in theory, the application should work even if running in the background while other apps are open, but this scenario was never tested during this project.

The database is working as intended, all user data is saved locally on the device. However, there is no option within the application to delete a user or a user's trip from the database.

Although the application works well, the open source Android Automotive OS has very limited capabilities on its own. To utilize its capabilities, such as adjusting the car's air conditioner or audio system, a deeper integration with the vehicle is needed, which this project did not include in its scope. For this project, the only connection between the Android on the Raspberry Pi and the vehicle is though the Bluetooth adapter, interfacing only with the car's OBD-II port.

Because the application only saves data locally, in order to track the trips of multiple users the same device must be used. Ideally, the device should be permanently installed in the vehicle, which requires the user to either buy the same setup as used in this project or to use any other Android device, such as a mobile phone or tablet. However, the UI implementation in its current state is adjusted and only tested on the screen used in the project. Using another Android devices requires adjusting the UI for the device's screen, but this project did not include these modifications.

6.2.2 Database and fuel measurement challenges

The objective to provide statistics to the user is carried out by presenting every driver with a table of their last 5 trips, with the data for each parameter. The tables also show the summarized average score for each user and how many trips they have taken in total. The values for speed, trip distance, duration, and fuel consumption of each trip appear to be correct when comparing them to the equivalent values calculated by the car's own computer.

6.2.2.1 Database management issue

One major problem that the project faced can be seen in the Table 5.1 for trip 2 where the duration and distance have received the values zero. To explain why this can occur, here is how the program is intended to work:

1. When the service layer is stopped, it calculates the fuel consumption and duration of the trip and writes it to the Trip Details table in the database.
2. After this, the Trip activity reads these values from Trip Details and right after writes them to the Trip table, as a new entry for the current trip alongside with the other parameters.

The problem occurs because sometimes writing to the Trip Details table is not completed before reading from it, resulting in a value zero for both fuel consumption and duration, which in turn gives zero for the distance as well. To remedy this problem, a time delay was placed between writing to and reading from the database. This is a quick but in no means complete solution.

6.2.2.2 Fuel measuring challenges

Another issue that can occur is the fuel consumption measurement resulting in zero, as can be seen in Table 5.1 for trip number 3 and in Table 5.2 for trip number 2,

where only the fuel value is zero. This can occur if the fuel consumption measurement results in a negative value, in which case a mechanism protects the score by disregarding the fuel consumption from the score calculations.

The reason for a negative fuel consumption value (which the program defaults to zero) depends on the car's parking position and the stability of the liquid fuel in the tank. When the car is parked on a road with a slope, regardless of the parking direction, the start fuel level measurement will have an incorrect value. This fuel level value can be higher or lower than the actual value depending on how the liquid is placed in the tank, i.e., whether the fuel indicator takes an incorrect level measurement for the liquid inside the tank.

The same can happen when stopping the trip, the car's parking position affects the end fuel level measurement. As explained before, the start and end fuel level in the service layer are used to calculate the trip's fuel consumption by taking the difference between the start and the stop value. If one or both of the measurements are incorrect, the difference can result in a negative fuel consumption. It can also result in a fuel consumption higher than the actual value, but this is difficult to consider without knowing the actual fuel consumption. To try fixing this problem, when measuring fuel level, the application takes 5 measurements and then calculates the average of them to give a better result, as described previously. Although taking more than 5 measurements could improve the results, this would still only solve the problem when the car is parked on a relatively flat surface, and would inevitably give a faulty measurement if the car is parked on any slope.

A better solution for the fuel consumption issue would be to use the fuel consumption rate command to retrieve the value directly from the car for each trip. But unfortunately, this command is not available in all cars. As this was the case with the car used for this project, the implementation used only the fuel level command as described in previously. Because of the issues presented, the user should make sure when using the application that the car is parked on a flat surface at the start and at the end of the trip. They should also let the liquid fuel stabilize inside the tank before stopping the trip in the application. Furthermore, while a trip is ongoing the car should not be fueled, as this can obviously give faulty fuel consumption measurements.

6.2.3 Score system assessment

The score system implemented in the project works as a proof of concept, but has many weaknesses left for discussion. The results of testing the score system on real trips in a vehicle can be seen in Table 5.1 and Table 5.2 for the two drivers. As it can be observed clearly, the range of the given scores is quite small, with only a 22 points difference between the highest and lowest scores. On a scale of 0 to 100 this is relatively not a very drastic difference. To improve the score, it might be reasonable to scale down the point system, maybe as low as a scale from 0 to 15, because the difference between driving styles does not require such a wide distribution.

One of the most relevant results can be seen in Table 5.1 for the trips 4 and 5. Here the score difference between these trips is relatively high, and most parameter values are close to each other, so it can be clearly observed where the two trips differed. The target values for all trips were the same as mention in the Chapter 5 Section 5.8. We see that the RPM, speed, and engine load values for trip 4 are marginally closer to the target values than they are for trip 5. This in itself would produce a higher score for trip 4, but not by the margin seen between the two rows. The major difference between them is the distance and the duration. Trip 4 took 12 minutes and 8.36 km and was taken mostly on highway conditions, whereas trip 5 took 6.6 minutes and 3.33 km and was taken on small town roads. These values compared with the equal fuel consumption for both trips is what really sets apart the scores, because trip 4 was clearly more efficient than trip 5, by covering a longer distance, at a higher average speed, while consuming the same amount of fuel. Although as mentioned previously, there could be some error in the fuel measurements, but these values are reasonable when compared to the car's own fuel consumption meter. These results seem to be reasonable because driving in the city usually consumes more fuel than on the highway, due to the speed limitations and the need to accelerate and break more frequently.

After closely examining the results, a flaw was discovered in the implementation. When calculating the sub factors for speed and rpm, as shown in Equation 4.1, there is a datatype mismatch in the code causing the fraction in the equation to result in an integer division rather than a float division. This causes that the sub factor of speed and rpm results in either extremely low or high values. The load sub factor calculation is not effected because the load value is a float, resulting in a correct division. Due to time constrains, the tests could not been repeated, however the results are still useful to compare between trips. The load factor and efficiency factor works as intended, and the speed and rpm factor would still indicate drastic discrepancies compared to the target values.

Another issue which affects the score system is when the distance and fuel consumption of a trip results in zero, because of the issue described in the previous section. The score system is designed to ignore the fuel consumption when its value is equal or less than zero, making the final trip score a combination of only the other three factors: RPM, speed, engine load. Because of this potential fuel measurement error, the efficiency factor weights only 25% of the trip score. This decision was made so that a score can be given even for those trips where the issue occurs, but knowing that comparison between trips with and without fuel measurement is not entirely comprehensive. This can be seen taking place in Table 5.1 for trips 2 and 3, and in Table 5.2 for trip 2, where fuel consumption is not counted into the score.

6.3 Improvement areas

6.3.1 Saving data

As explained before, the database has three tables: User, Trip Details and Trip. To reach the project's objectives, these tables are necessary in order to save each user,

the details of their current trip and the data for their last five trips. All information from the database are used to create the statistics. More tables can be used, but three are sufficient to reach the goal.

The implementation does not include saving the target values inputted by the user to the database. This results in that each time the application is restarted, these values are reset to the default ones hard coded, requiring the user to input their desired values each time when starting the application. Solving this issue would improve user-friendliness and is relatively simple to implement as it only needs a new table in the database.

6.3.2 Data update frequency

The application saves the current trip's data every 10 seconds to the database. Saving these values less frequently, for example only every 30 seconds, would risk potentially missing out on important data. At the same time, updating more frequently would not necessarily give more details but would use up more resources. Car trips are usually not that short in average, for example a short trip can take 10 minutes, when the data is saved every 10 seconds that means it gives about 60 measurements and a 60 minutes trip would give 360 measurements, which is enough to calculate a reasonable average for each trip.

6.3.3 Fuel level measurement discussion

The fuel level is an important part for calculating the score, because it is used to calculate the fuel consumption rate. There is a direct OBD-II command to access the car's fuel consumption rate, but none of the cars tested in this project supported this command. The fuel level readings from the car are drastically effected by how the vehicle is currently positioned, and whether the liquid fuel inside the tank is still in motion. As described in the *Implementation* chapter, to counteract faulty measurements, the application takes five samples of the fuel level to give a more accurate estimate.This mechanism gives more reliable data but does not guarantee fail safety, but the application ensures that a negative fuel consumption is not used in the score calculation. The application also has a mechanism to protect the score when the fuel consumption is zero by only taking the eco factor into the consideration. This is necessary because not all car models/brands supports the fuel level command, and some trips may be so short that the fuel consumption can not be measured.

6.3.4 Updating data on the UI

When updating the UI elements in the application, there could be a more suitable solution. In the current state of the implementation, the service layer updates the Trip activity through a listener. A perhaps more elegant way to achieve this would be to implement the database using flows, to which the Trip activity can subscribe to. This way, every time the service layer updates the data and writes it to the database,

the Trip activity could instantly observe the changes. This implementation would also solve the issue presented in section 6.2.2.1.

6.3.5 Concurrency

Regarding how the OBD-II data acquisition uses coroutines, there could be an alternative solution. The service layer starts a coroutine (with the *launch* command) for each data retrieval (RPM, speed, load). Due to that fact that they all communicate with the same Bluetooth device, the Bluetooth socket is made mutually exclusive for each coroutine using a locking mechanism. This approach could be changed by only starting one coroutine (with the *launch* command) which in itself launches three coroutines with the *async* command for each data retrieval. The *async* command launches a coroutine and can return the result to the caller using the *await* function. This way the use of locks could be avoided.

6.4 Future work

6.4.1 Other approaches

Part of the solution in the projects could have been solved with another approach, namely how the data acquisition happens from the OBD-II adapter to the Android application. As discussed above, in this project the application communicates directly with the Bluetooth adapter connected to the OBD-II port, accesses the data directly. Another solution to this would have been to configure the Android operating system itself to connect to the OBD-II and store data in the Vehicle Hardware Abstraction Layer (VHAL). This way, the parameters like speed and engine's RPM would be stored within the OS and any application could access them without having to build a connection for the application itself via Bluetooth on a higher level. But this solution would also require deep knowledge on the inner workings of the Android operating systems.

The idea was a wish from the company in order to develop a custom version of Android Automotive that can be used as a reference for similar projects. Working on the configuration of the VHAL is an advanced project which needs more resources and time, exceeding the scope of this bachelor thesis project. Building a custom Android Automotive OS with these features is a very niche area, pursued by only some OEMs in the vehicle industry. These realizations, together with the very limited amount of public documentation for the configuration of Android Automotive's VHAL, resulted in quickly abandoning this approach for this project.

6.4.2 Alternative score system

The target values for RPM, speed, and engine load vary between different road conditions, such as speed limits and the slope of the road, some of which conditions may cause higher RPM, speed, and engine load. To solve this, it would be possible to implement a solution where there are pre-determined target values depending on

the road condition, for example, distinguishing between urban and highway driving conditions. Implementing this solution would also improve the statistics by allowing to compare drivers' trips within the same driving condition.

With this approach, the score given would be relative to the driving conditions, meaning that two trips could give the same results even if the objective measurements are completely different. This could be claimed as a more fair score calculation, if fairness means that different driving conditions should not be compared to one another directly. However, if fairness is determined solely on objectiveness, then different driving conditions should not be distinguished.

6.5 Summary

The project reaches its goal and proves that it is technically feasible to create an application to provide a score based on driving style, by keeping track of live vehicle data. To use a Bluetooth OBD-II adapter instead of a USB one proved to be a more seamless because Bluetooth connections are relatively easy to establish in Android. Similar open source projects [21] are also implemented with Bluetooth connection, making it easier to problem-solve when compared to USB.

The score, to some degree, represents the combination of engine usage and fuel-efficiency of the driver, but needs more refining to make it more accurate for comparing between driving styles. Retrieving the fuel level from the car was more challenging than expected, but as it is an important part of the score system, it can not be easily ignored. The limited time for the project did not allow for further improvement to the score system and to correct its flaws, but could be refined further with more planning and analysis. Keeping in mind the project's results, the conclusion can be made that the biggest limitation for this system is the fact that the OBD-II protocol, although standard in every vehicle today, is not guaranteed to support the same functions for retrieving data in every car model.

Bibliography

[1] the European Parliament and the European Council, "Relating to measures to be taken against air pollution by emissions from motor vehicles and amending," 1998. [Online]. Available: https://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CONSLEG:1998L0069:19981228:EN:PDF

[2] Agoshi, "Vårt erbjudande," Available at https://www.agoshi.se (2024/01/17).

[3] Cilbuper, "Cilbuper Group - vi hjälper er att växa," Available at https://www.cilbuper.se (2024/01/17).

[4] "Raspberry Pi 4b," Available at https://www.raspberrypi.com (2024/01/17).

[5] C. ARD, "On-board diagnostic ii," Available at https://ww2.arb.ca.gov/resources/fact-sheets/board-diagnostic-ii-obd-ii-systems-fact-sheet (2024/04/29).

[6] "Vehicle Hardware Abstraction Layer," Available at https://source.android.com/docs/automotive/vhal (2024/03/22).

[7] "What is Android Automotive?" Available at https://source.android.com/docs/automotive/start/what_automotive (2024/01/17).

[8] "Android Studio," Available at https://developer.android.com/studio (2024/04/21).

[9] "Logcat," Available at https://developer.android.com/tools/logcat (2024/03/22).

[10] "Get started with Kotlin," Available at https://kotlinlang.org/docs/getting-started.html (2024/01/17).

[11] M. Flauzino, J. Veríssimo, R. Terra, E. Cirilo, V. Durelli, and R. Durelli, "Are you still smelling it?: A comparative study between Java and Kotlin language," pp. 23–32, 09 2018. [Online]. Available: https://doi.org/10.1145/3267183.3267186

[12] J. Horton, *Android Programming with Kotlin for Beginners: Build Android apps starting from zero programming experience with the new Kotlin programming language*. Packt Publishing, 2019. [Online]. Available: https://books.google.se/books?id=CzCWDwAAQBAJ

[13] "Activity," Available at https://developer.android.com/reference/android/app/Activity (2024/04/15).

[14] "Services overview," Available at https://developer.android.com/develop/background-work/services (2024/04/15).

[15] "Kotlin OBD API," Available at https://github.com/eltonvs/kotlin-obd-api (2024/04/15).

[16] G. Allen and M. Owens, *The Definitive Guide to SQLite*, ser. Books for professionals by professionals. Apress, 2011. [Online]. Available: https://books.google.se/books?id=-5zk12NiBBQC

[17] A-R.M.Yunis, *Android SQLite, Firebase, and Room Databases*. Springer, Cham, 2022. [Online]. Available: https://doi.org/10.1007/978-3-030-87459-9_11

[18] "Set up for AOSP development," Available at https://source.android.com/docs/setup/start/requirements (2024/04/15).

[19] "Android local manifest," Available at https://github.com/raspberry-vanilla/android_local_manifest (2024/04/15).

[20] "Speed View," Available at https://github.com/anastr/SpeedView (2024/04/14).

[21] "Bluetooth-OBD-II-Diagnostic-Tool," Available at https://github.com/fussek/Bluetooth-OBD-II-Diagnostic-Tool (2024/04/23).

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY