Infrastructure for Test Car Display System

Part B – Spring 2019

Project report

Developed by

Daher Daher daher928@campus.technion.ac.il

Vivian Lawin vivlawin@campus.technion.ac.il

Supervisor

Boaz Mizrachi

Table of Contents

1 Preface

In order to enhance and improve the driving experience in cars in general (and autonomous cars in particular), drivers and car testers must understand and feel not only the condition of the environment and the traffic, but also the condition of the road, e.g. fraction of the road, humidity, temperature and angle.

Feeling and experiencing the interaction between the car and road is important for optimizing car's safety and efficiency both for human drivers and autonomous driving.

Our project aims to visualize the conditions of the road and the real-time data collected by the car's sensors and provide a comfortable and elegant user interface to graphically display the results on the vehicle's dashboard screen display.

2 Introduction

2.1 Is it necessary?

Car testing is very essential, and the more the testing procedure is efficient the more the results are reliable, and time is saved. both advantages are highly desired in the era of new technology, and especially in the emerging field of Artificial Intelligence and autonomous cars, enabling a transformation and technological rise in the mobility industry.

When mentioning AI and autonomous cars, the term “Safety” pops up as a key goal. Car testing – and especially when intended to be driven autonomously – is a necessary process for assuring and guaranteeing the safety of the passengers and the vehicle’s surrounding environment and pedestrians.

2.2 Project Goals

The main goal of this project is to provide car drivers and testers with a friendly and easy-to-use application with the ability to graphically display the real-time data received by the car's sensor.

Each of the following features of the application will add to the users experience much convenience, starting with choosing the configurations of each graph, to getting previous results, which are very useful for comparing results and investigating the changes in the results according to changes in the surrounding circumstances.

Full features list:

- Support multiple graphs (Up to 3 graphs at a time)
- Support logging and data recording
- Cloud storage of test results
- User authentication and user space
- Choose different configurations (Resolution, Max value, Color…)
- Time axis scaling
- Different appearance theme
- Caching previous configurations set for reuse

3 Technology

The following chapter describes the different technologies used for implementing the project, such like: display screen, data transfer solutions, development environments and developing tools.

3.1 Display screen

The device that is used for running the application and displaying the Realtime results and graphs is GINI Tab V7. Following are the specifications and features of the device:

Native Platform	Android Nougat 7
Release Year	2017
Device RAM	2 GB
Battery Capacity	2800 mAh
Max Internal Storage	16 GB
Expansion Slot Type	MicroSD
Expansion Slot Max Size	32 GB
Supported I/O	3.5mm Jack, Micro-USB
Supported Charger Types	Wire
Supported Bearers	Bluetooth, WiFi
Has NFC	true
Screen Type	IPS
Screen Inches Diagonal	7.0 inch
Screen Pixels Width	1280 pixels
Screen Pixels Height	800 pixels
CPU Cores	4
CPU Maximum Frequency	1.3 GHz
Screen Inches Square	22 inch
Screen Inches Width	5.94 inch
Screen Inches Height	3.71 inch

3.2 Software and IDEs

The transmitter is a Java program developed as a Java application, which runs on windows laptop. The IDE used for developing the transmitter is Eclipse, using common Java libraries for TCP networking and sockets.

The receiver is built-in the Android Application for the display system, developed in Android Studio IDE in Java language for android.

- Android Studio IDE 3.3

  - Java over Windows OS

- Eclipse Oxygen 4.7

  - Java over Windows OS

- Arduino 1.8.12

3.3 Graphics and UI components

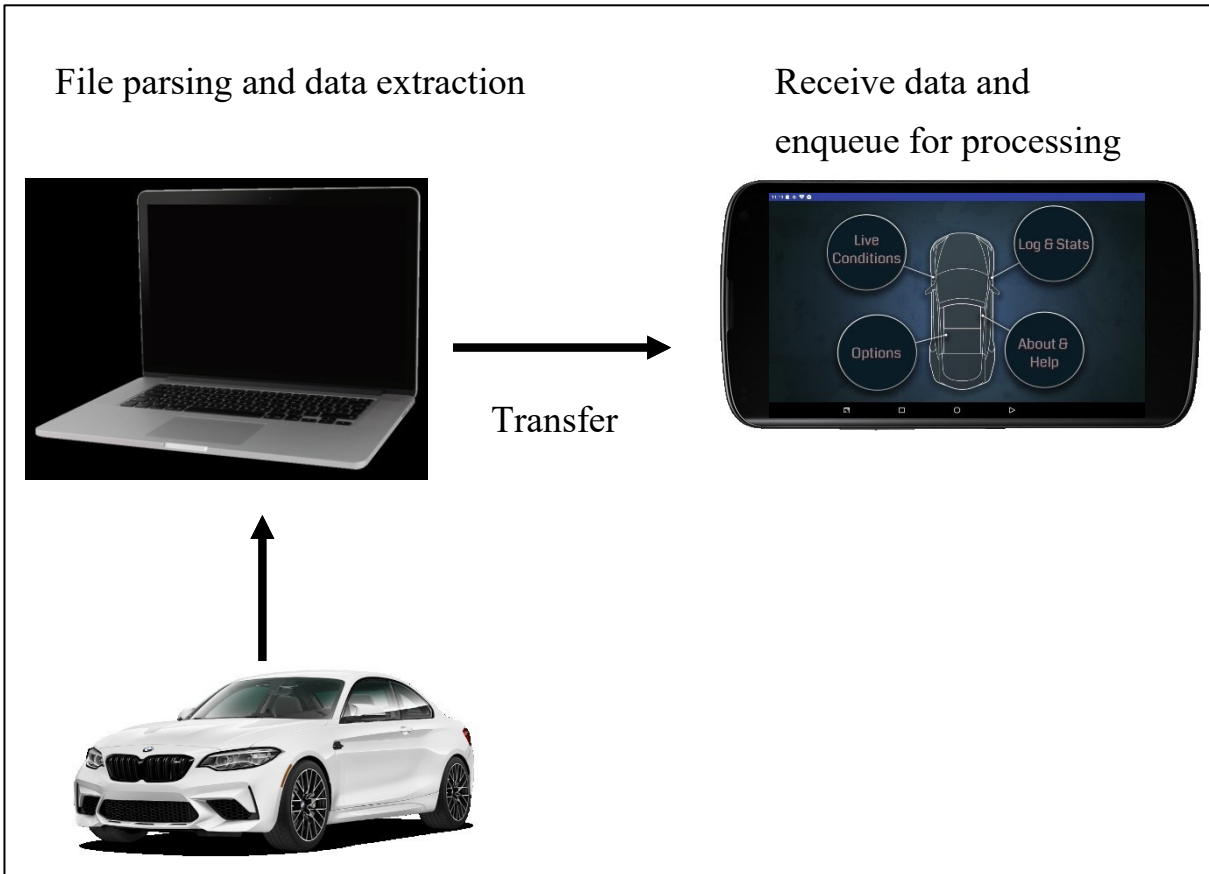
Most of the graphic components in the Android application are designed using Adobe Illustrator and Adobe Photoshop:

- Adobe Photoshop CC 2015
- Adobe Illustrator CC 2015

4 Part A

4.1 Recap

In part A, we had two independent parts communicate one another, in order to transfer data (figure 1). The roles are a *Transmitter* and a *Receiver* (client-server). The Transmitter is a laptop which receives file of sensors data streams, parsing it and transmitting the data to the receiver. The receiver is a part of the application installed in the car's dashboard display system. On receiving data, the receiver parses, filters and enqueues the data for further processing by the application (e.g. graph updates, logging, etc.…). Each of the above roles are independent and can be replaces and improved independently.

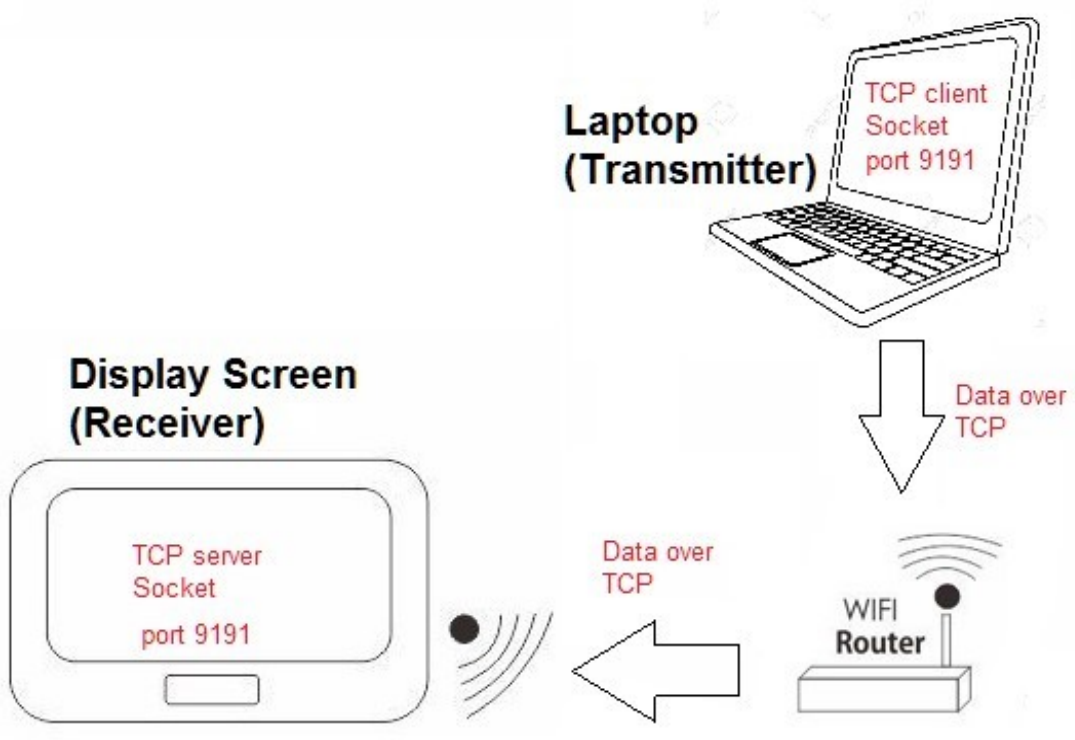


(figure 1)

4.2 Data Transfer Solution

(Part A)

As mentioned, there are two independent parts in part A which aim to transfer data between them (figure 2); the transmitter (laptop), and the receiver (display system application). In this project, we achieve this type of communication using **TCP networking protocol** and **TCP Sockets**. Both the transmitter and receiver must be connected to the same WI-FI network, then open a TCP socket on the same port allowing them to communicate. The receiver acts as a server and listens to incoming connections from the transmitter (client).



(figure 2)

4.3 High Level Design

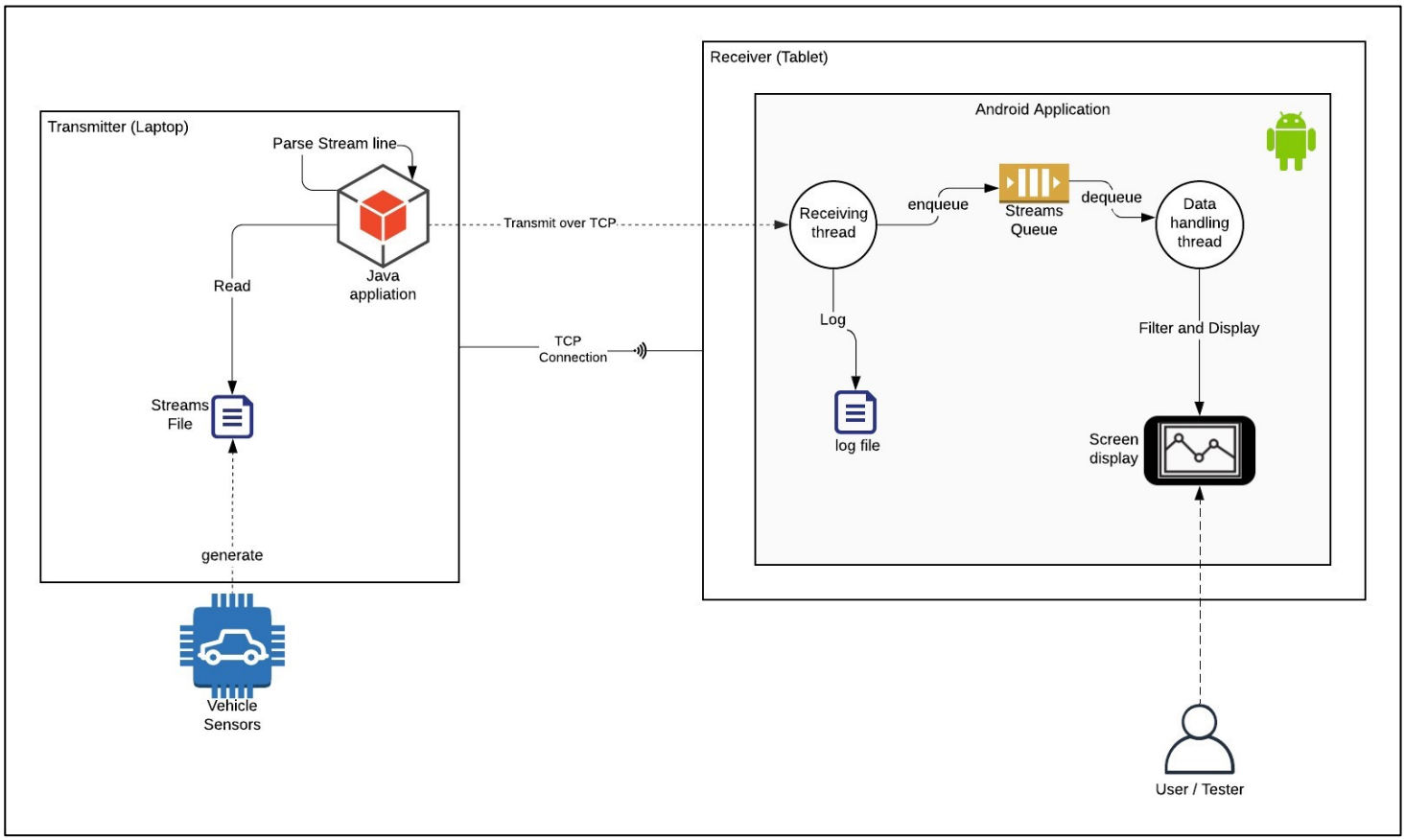
(Part A)

As mentioned earlier, the main components are a transmitter, and a receiver.

The transmitter (Java application), reads the streams file which the vehicle system (multiple sensors) produce, and parses each streamline according to the agreed structure. Afterwards, the application transmits data streams over TCP to the receiver application.

The receiver (Android application) contains a process for receiving the data sent over TCP, and enqueues relevant parts of the stream to a ProducerConsumer queue. The receiving thread also logs the data received into a log file located in the tablets file system, allowing the screen to display the loggings of the data. Another process in the receiver is responsible for consuming the data from the queue, filtering relevant data according to the user's desire, and updating the real-time graphs.

(figure 3)



4.4 Features

(Part A)

Supported features of Part A

- Support multiple graphs (Up to 3 graphs at a time)
- Support logging and data recording (Internal Storage)
- Choose different configurations (Resolution, Max value, Color…)
- Time axis scaling
- Different appearance theme

5 Part B

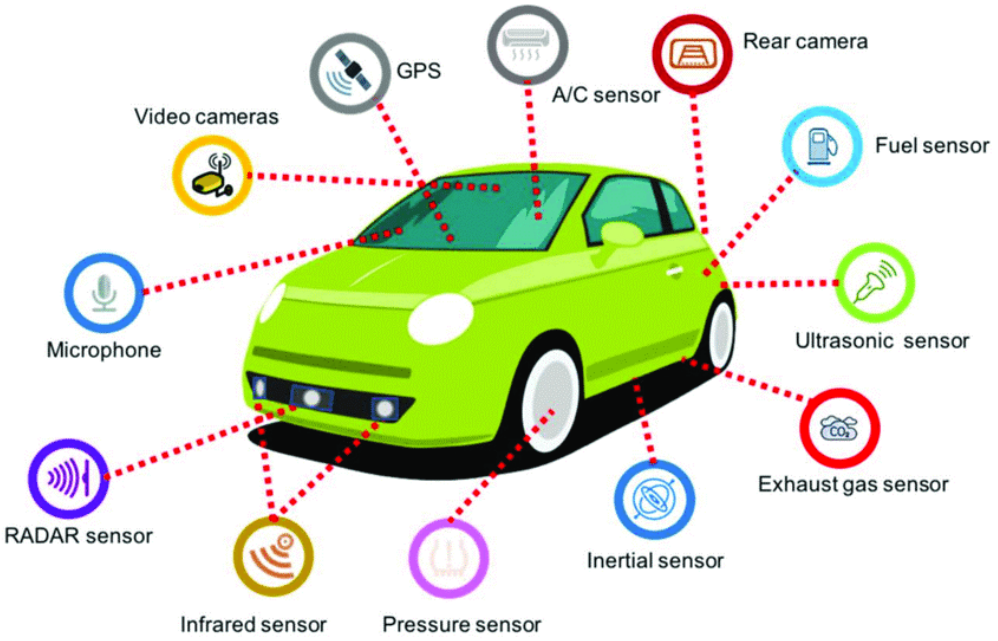
5.1 Overview and Incentive

While in part A we imitated the vehicle's operation of transferring sensors data via a Java PC application communicating over TCP connection with the Android application, in Part B we aim to receive sensors data collected by the vehicle, directly from the vehicle without any middle-man.

Such goal requires connecting the Android device directly to the vehicle's computer using USB connection. In order to do that, the Android device must act as a serial host, and to achieve this we use an **OTG** cable.

OTG (on-the-go) cable allows USB devices, such as tablets or smartphones, to act as a host, allowing other USB peripheral devices to be attached to them. Use of USB OTG allows devices to switch back and forth between the roles of host and device.

In addition, in part B we aim to migrate logs and data logging from the device's internal storage into cloud storage/database in order to support larger data and an ability to inspect the logs not only from the Android screen, but also in the cloud storage UI.



5.2 Features

Additional features supported in Part B

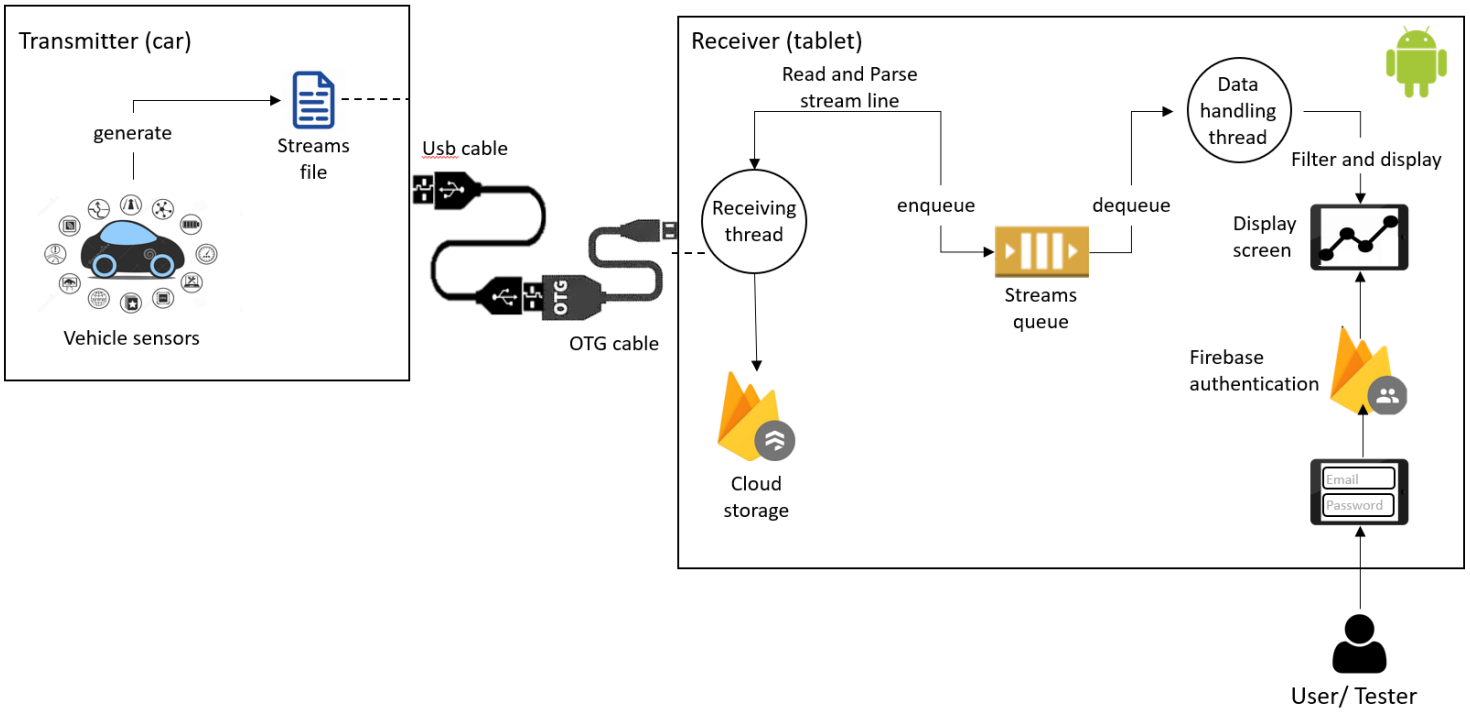
- Replacing TCP connection (Between PC & Android device) with USB connection, enabling data receiving from vehicle's sensors.
- Cloud storage logging of test results (Using Firebase Database and Functions)
- Log-in and User registration ability. (Using Firebase Authentication)
- UI improvements

5.3 High Level Design

As described in 5.1, we are replacing the Java application and TCP connection with direct USB connection, using 1 USB cable and 1 OTG cable which enables the Android device to act as host.

Upon establishing the serial communication, the receiver (Android device) handles the data by parsing the stream line received and enqueues the parsed object producer-consumer queue. This thread also logs the data received into a cloud-based database, allowing the screen to display the loggings of the data.

The flow proceeds with another process responsible for consuming the data from the queue, filtering relevant data according to the user's desire, and updating the real-time graphs.



5.4 Development Process

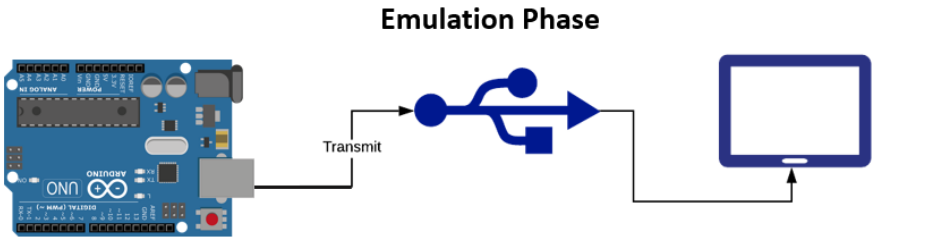
Since developing and debugging the application while being inside the vehicle can be challenging and makes complicated the developing process, we split the implementation and developing process into two parts:

1. **Emulation phase**
   In this phase, we develop a transmitter as similar as possible to the operation of the vehicle and the data transfer, including USB serial connection to the Android device, sending the exact same streams format and in high rates.

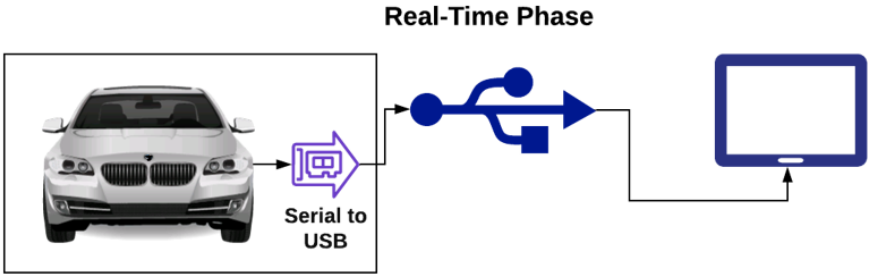
   Earlier, in the projects preliminary design, we designed the emulation phase to use PC as the transmitter, sending USB serial streams to the Android device. However, after further research, we realize that for the PC to act as a USB client and not host, it required hardware modifications of the PCs motherboard and serial interface. Therefore, we changed the design to replace PC with **Arduino** serial communication, which not also minimizes hardware modifications but also minimizes complexity of the implementation.

   * Further Explanation about the Arduino in the next chapters.

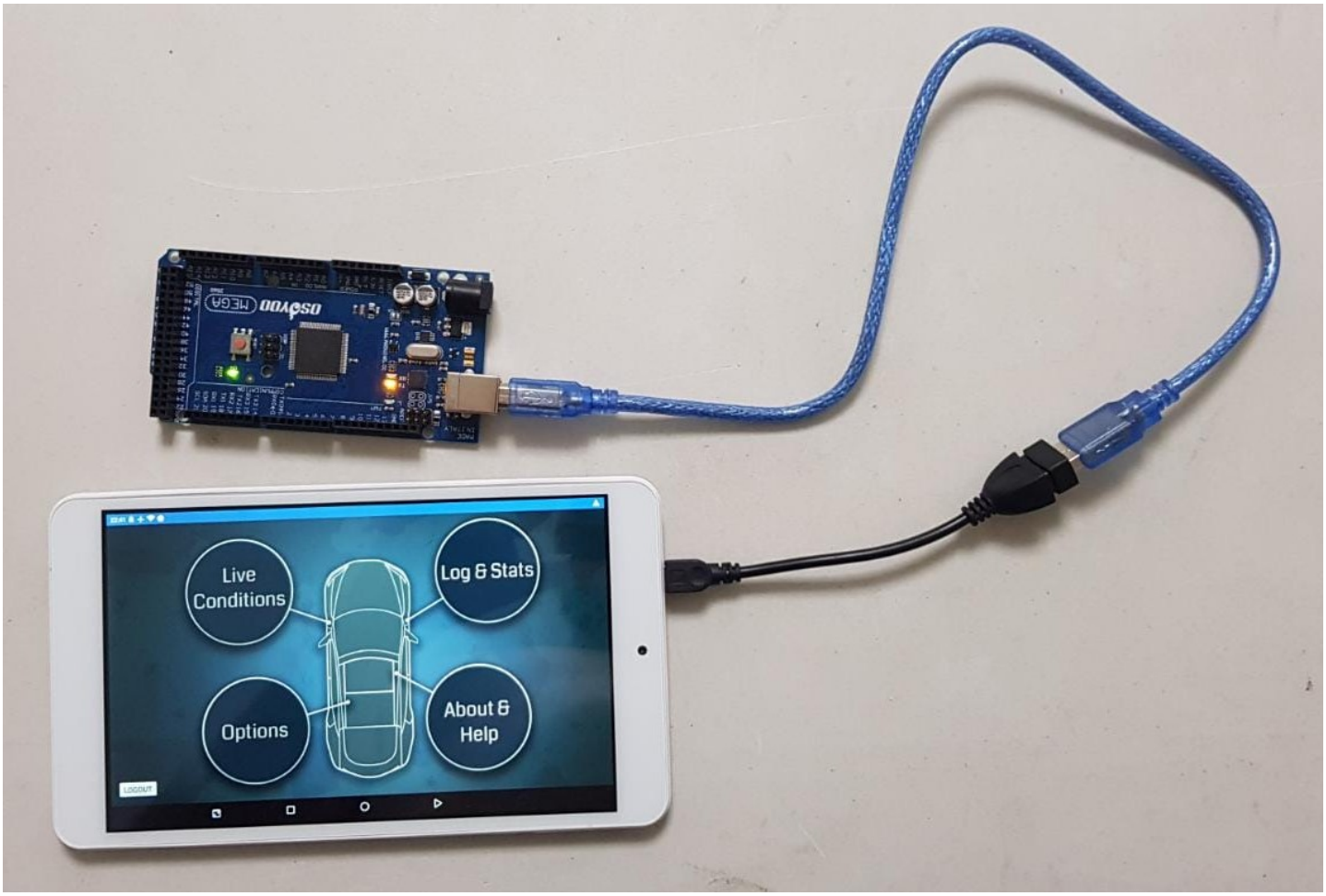
2. **Realtime phase**
   Assuming phase 1 works properly, connecting the vehicles USB cable to the Android device should work perfectly, without further modifications, as we emulated the operation of the vehicle exactly.



If the above works, then below will work without software changes



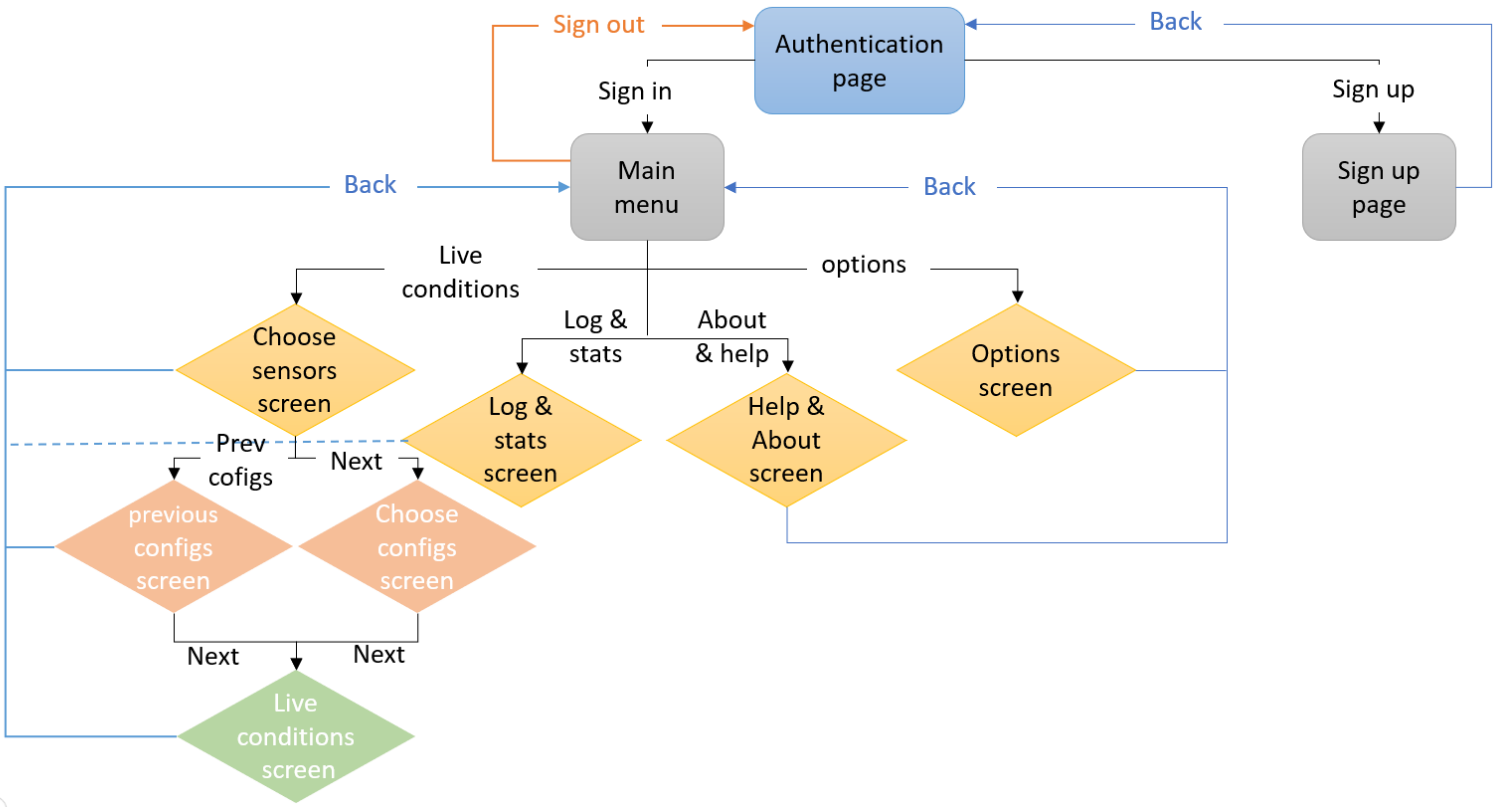
The following picture demonstrates the connections of the emulation phase:



5.5 User Interface

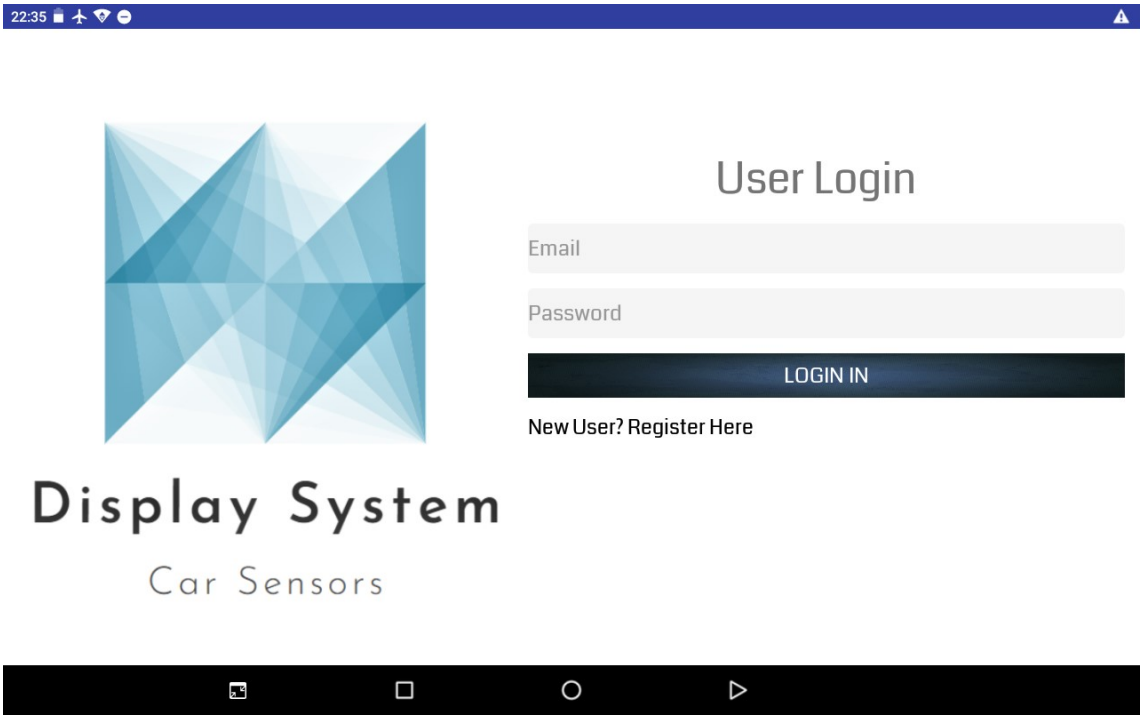
5.5.1 Flowchart

The following figure, demonstrates all possible flows and use-cases of the application.

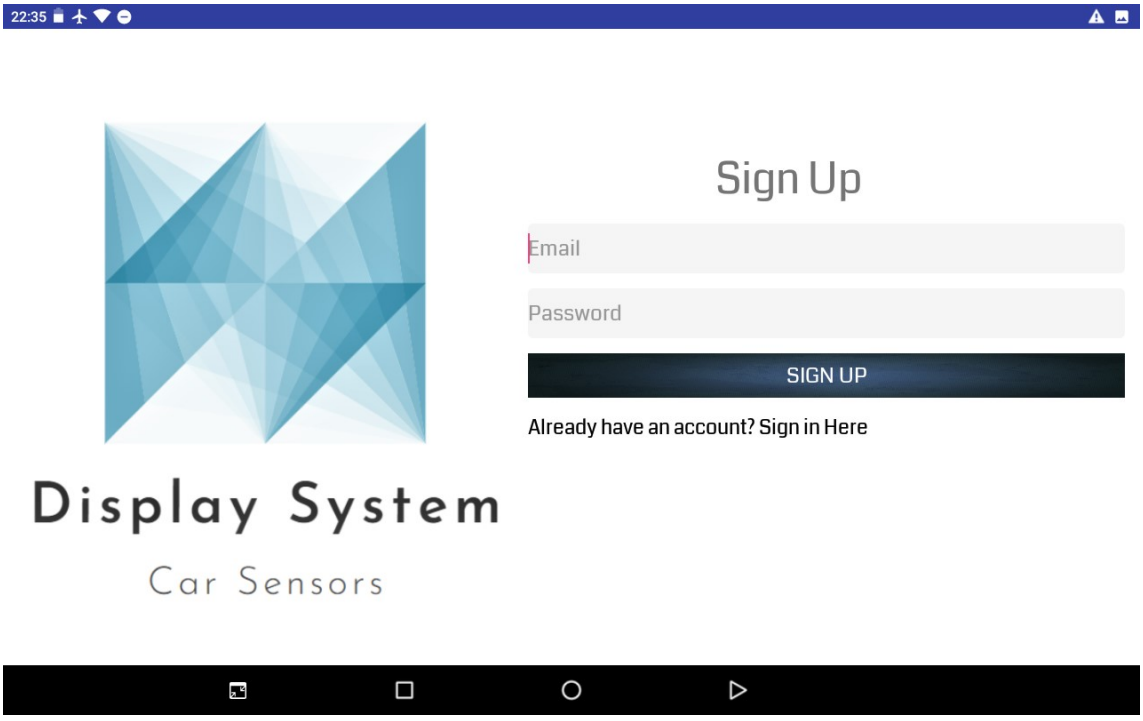


5.5.1 Screens and Usage

5.5.1.1 Authentication page – Sign in



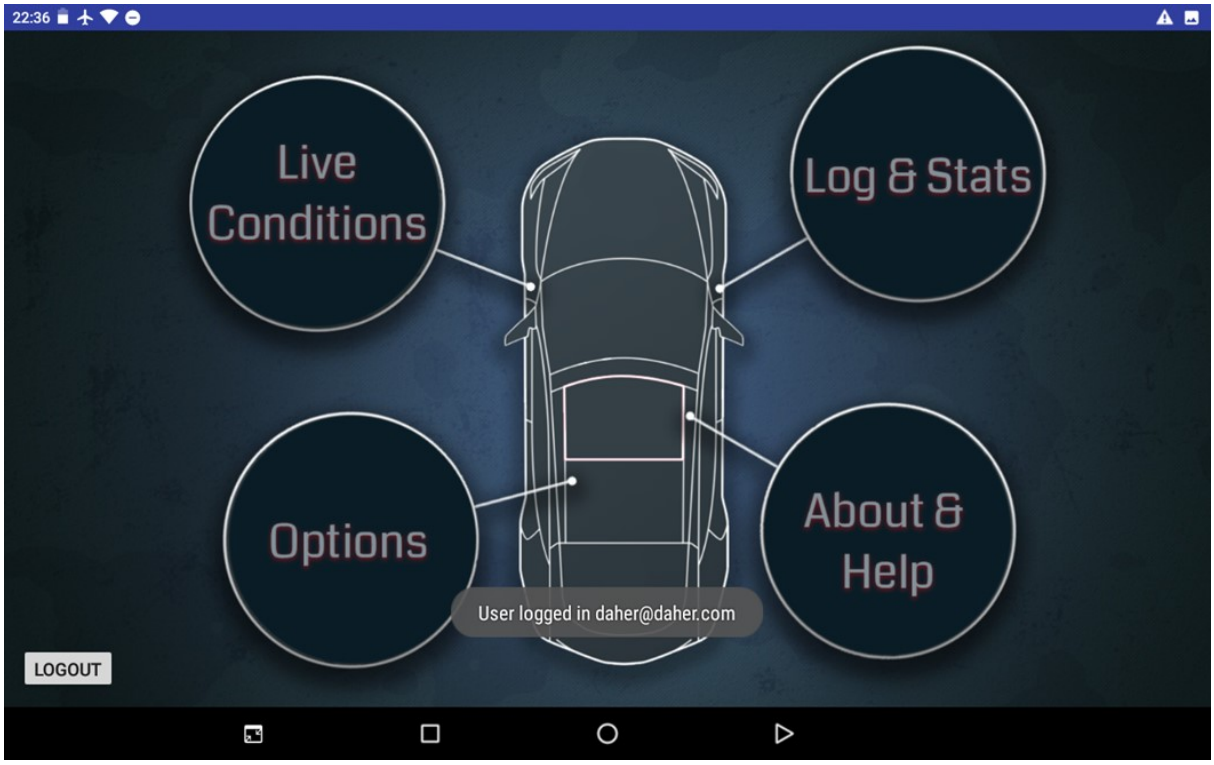
5.5.1.2 Authentication page – Sign Up



5.5.1.3 Main menu

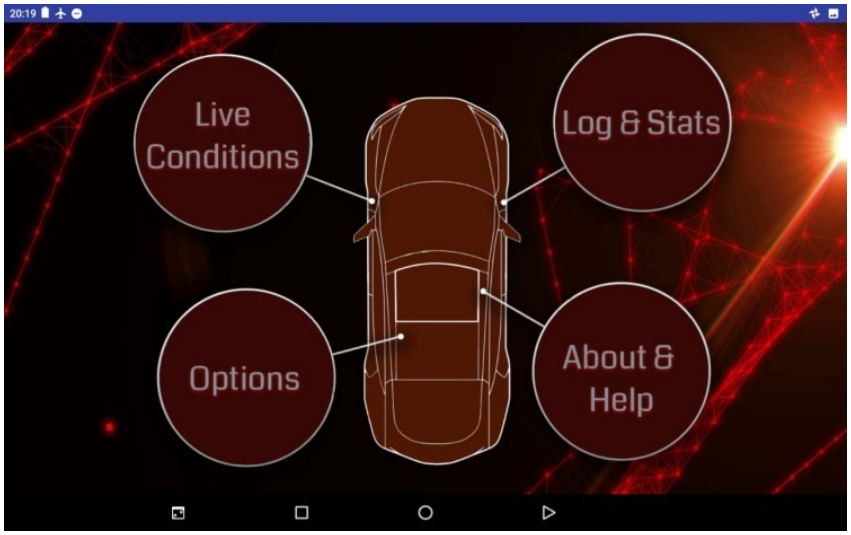
This screen is the homepage of the application and the first screen that the user sees upon logging in.

In the main page the user can choose one of the following:



Main Page

User can in addition change the theme of the application according to their preferences. This application supports two themes: blue and red).



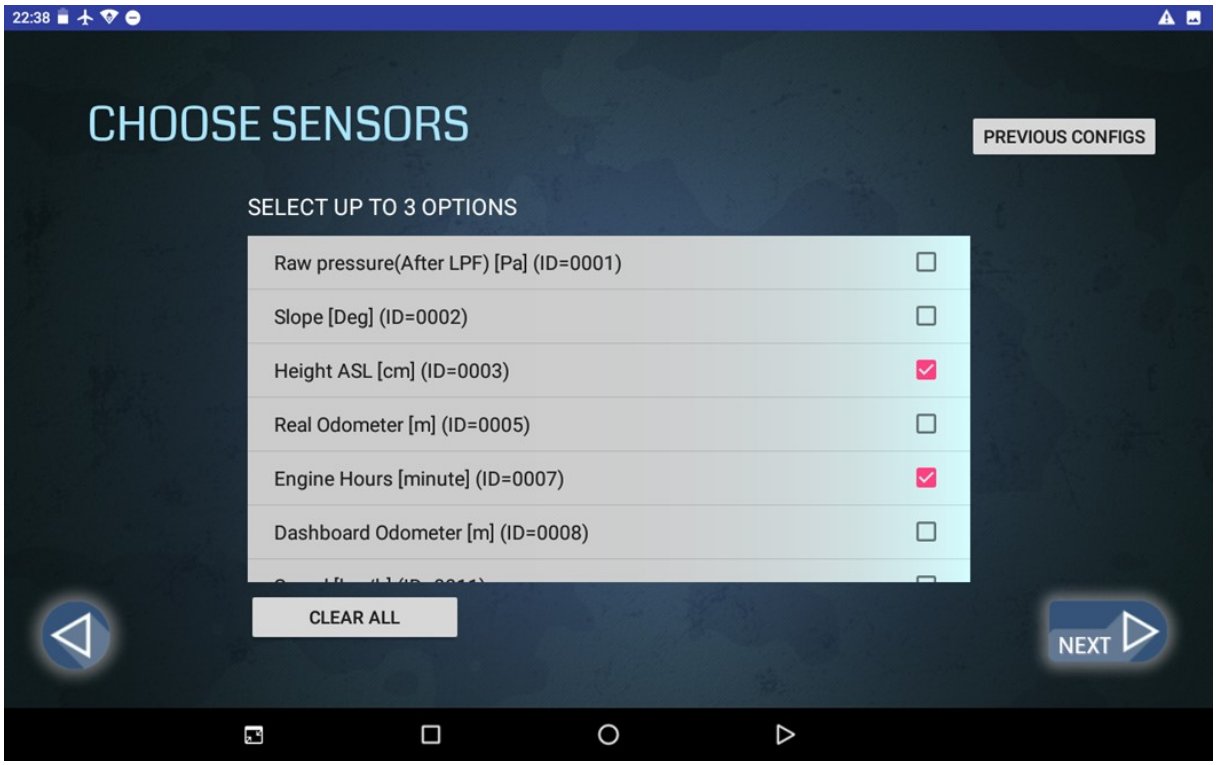
5.5.1.4 Live Conditions – Choose Sensors

In this page the user will be asked to choose up to 3 of the sensors to be shown in the graphs of real time measurements.

The user can choose to clear all their selections by pressing CLEAR ALL button as seen below. Users can also go back to the previous page or the next one, after they have chosen all the options they desire to be shown, by clicking the appropriate button.

Users cannot proceed to the next screen (using the next button) unless at least one option was chosen.

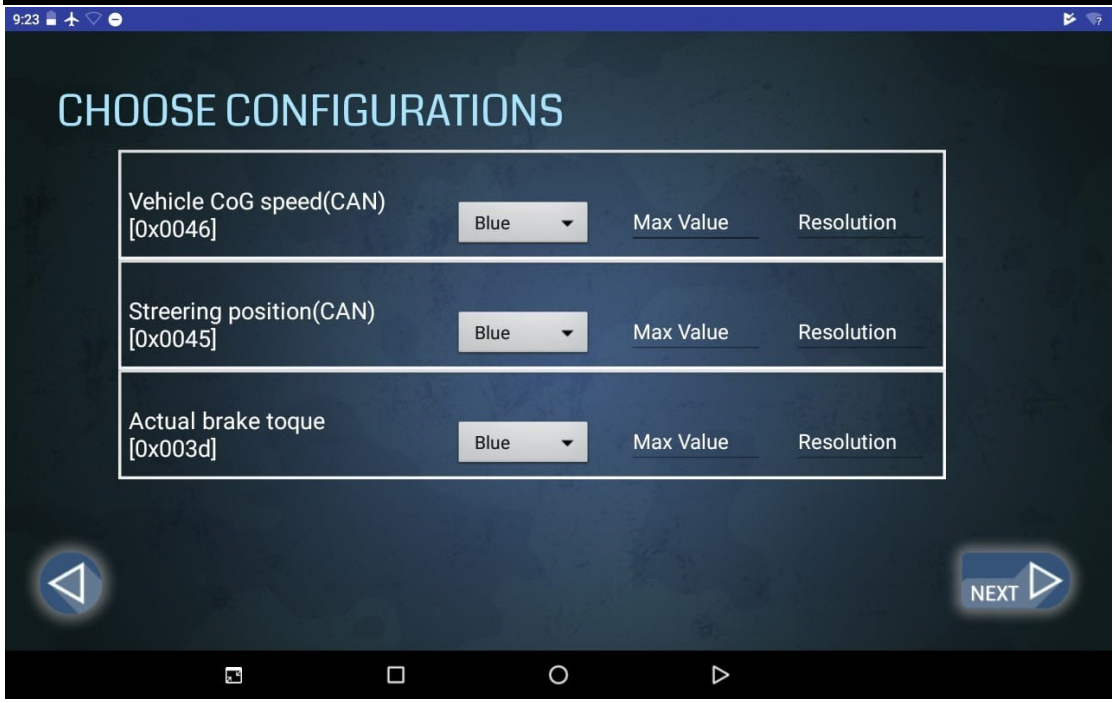
Besides the “next” button, another flow scenario is choosing the top right corner’s *previous configs* option.



5.1.1.5 Live Conditions – Choose Configurations

After choosing the sensors the user desires to be shown, the next step is to choose the configurations. When clicking the NEXT button, the user is redirected to the next page:

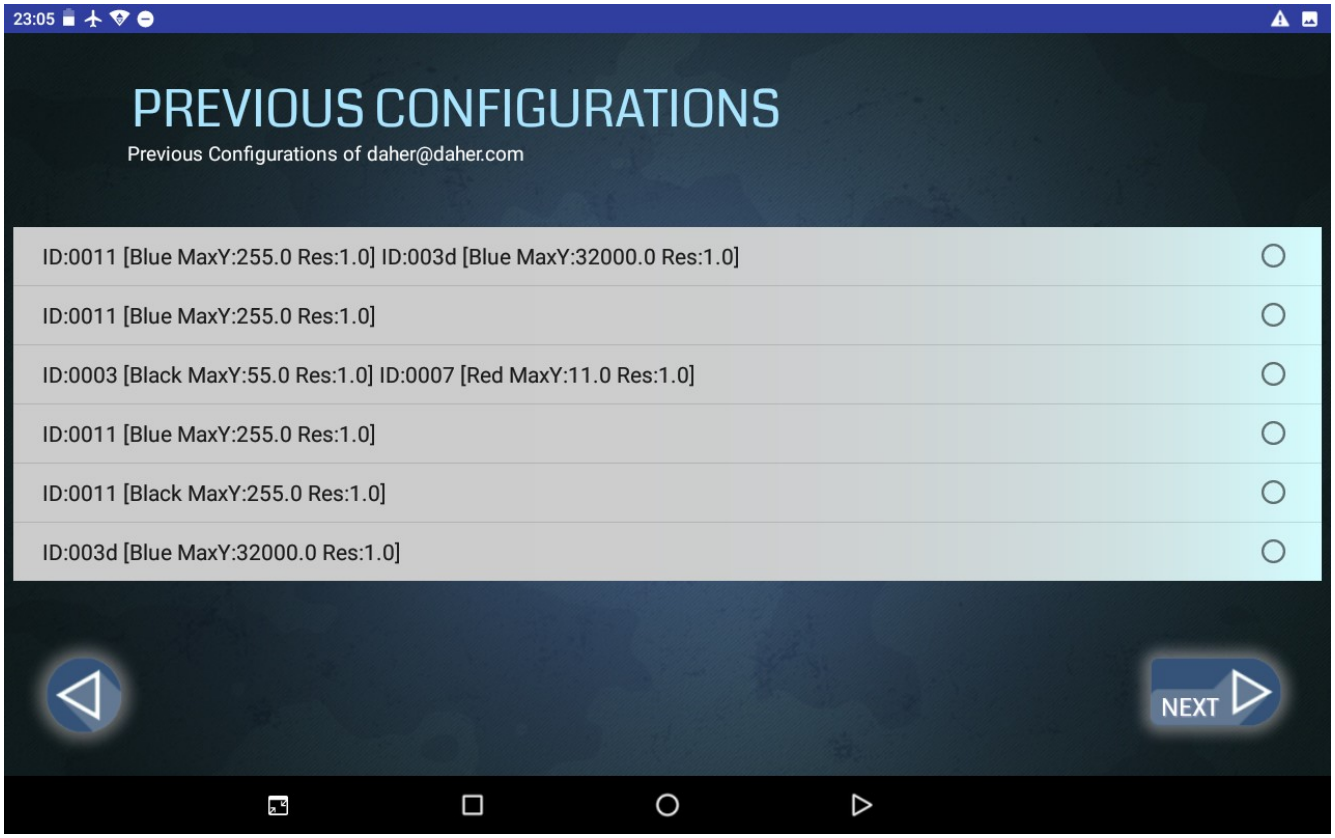
If no specific configurations were chosen, the default configurations (read by sensors configurations file) is used.



5.1.1.6 Previous Configurations

If the user chose in the Choose Conditions screen the *previous configurations* option, this screen is next to be presented.

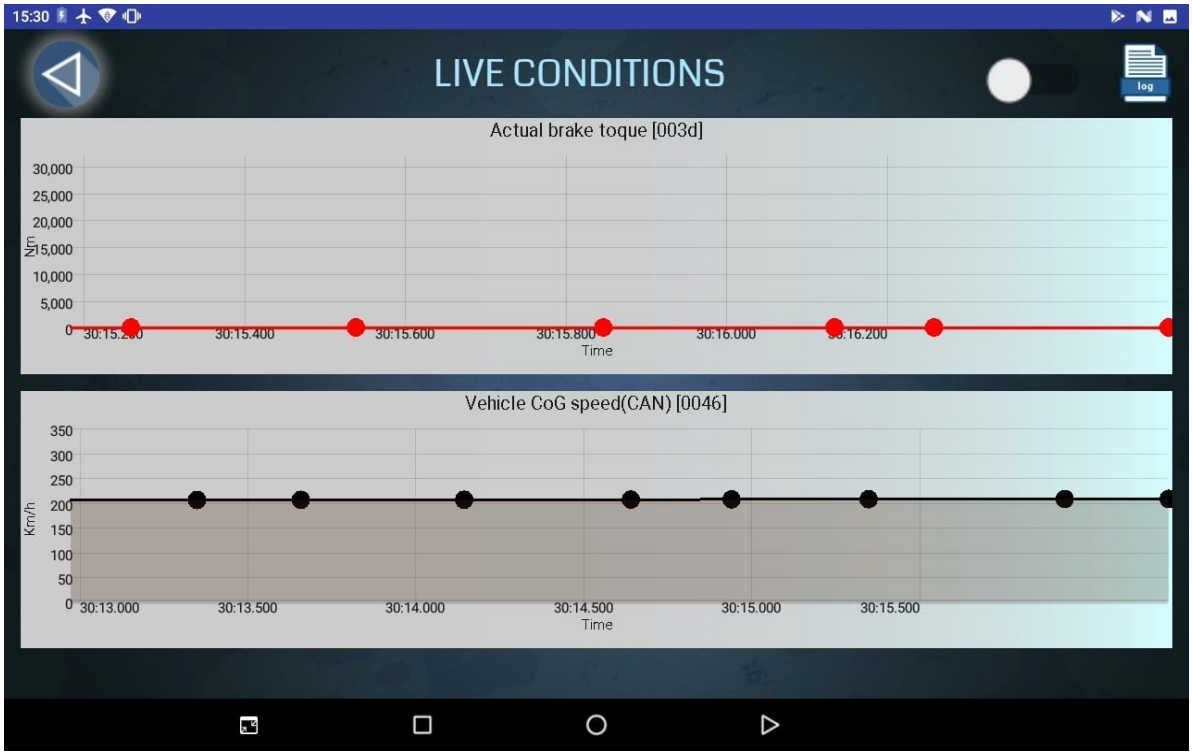
All these configuration sets are cached configurations that the user chose previously, and in order to make it easier for the user to use the same configurations desired, this screen enables configurations reuse.



5.1.1.7 Live Conditions

The most significant page – the actual live conditions and graphs, according to the configurations chosen by the user (of previously cached configs)

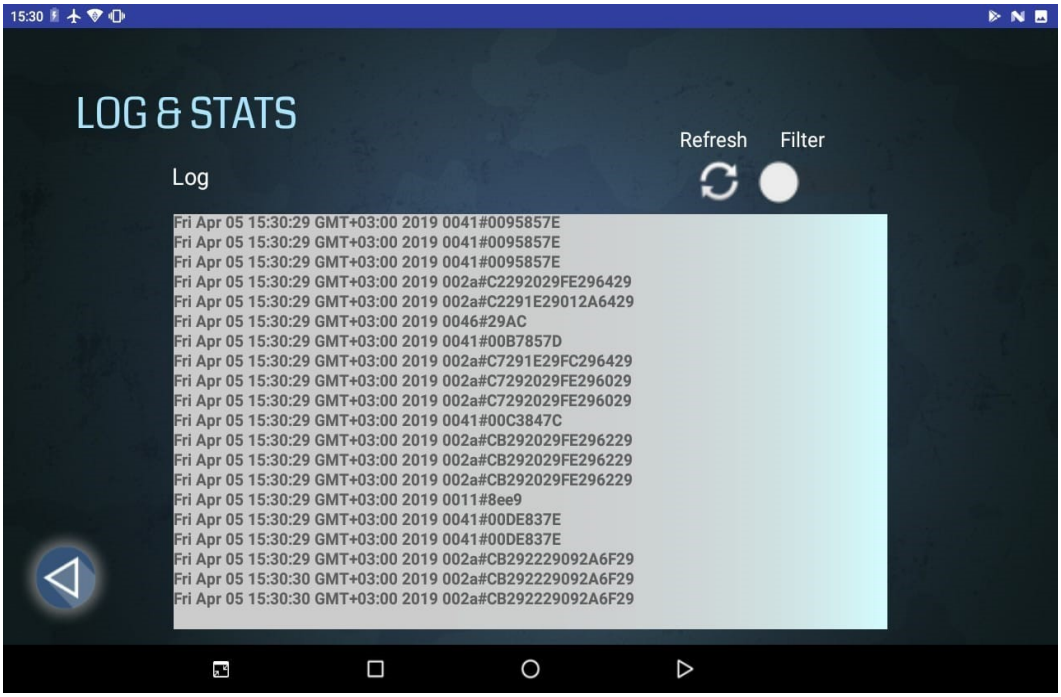
On the top right-hand corner, the user can switch the logging option on/off. When the option is On, the log file is displayed in the Log & Stats page.

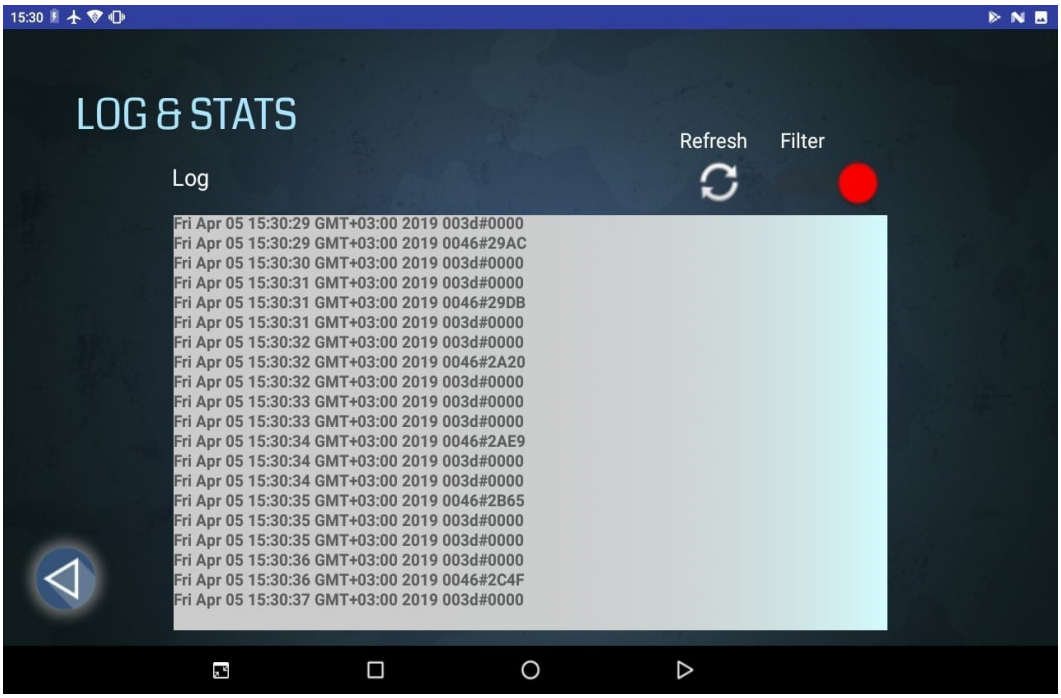


5.1.1.8 Log & Stats

In Logs & Stats the user can find log that were recorded. It helps the user compare between results and save previous results which makes this application more efficient.

The application provides logs with and without filtering according to the configurations (sensors) chosen.





5.1.1.9 Options

In options the user can change the theme of the application.

The application supports two themes: red and blue.

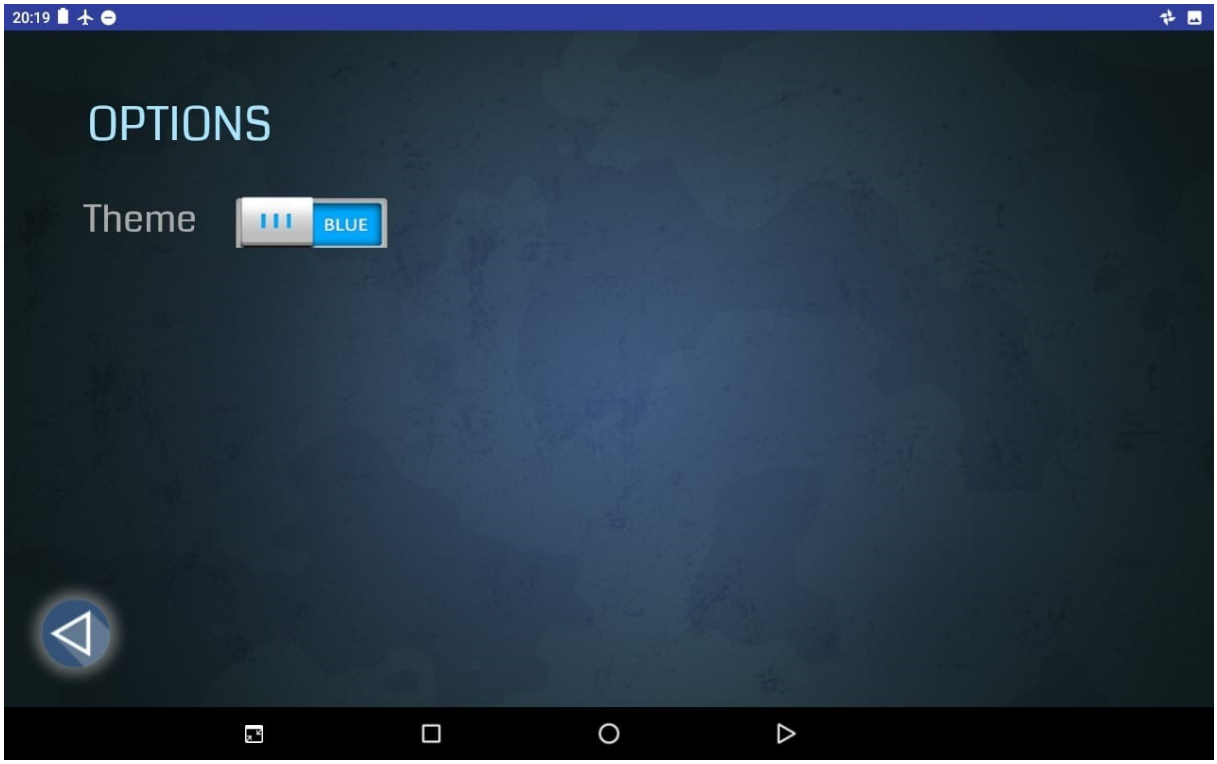


Figure 8

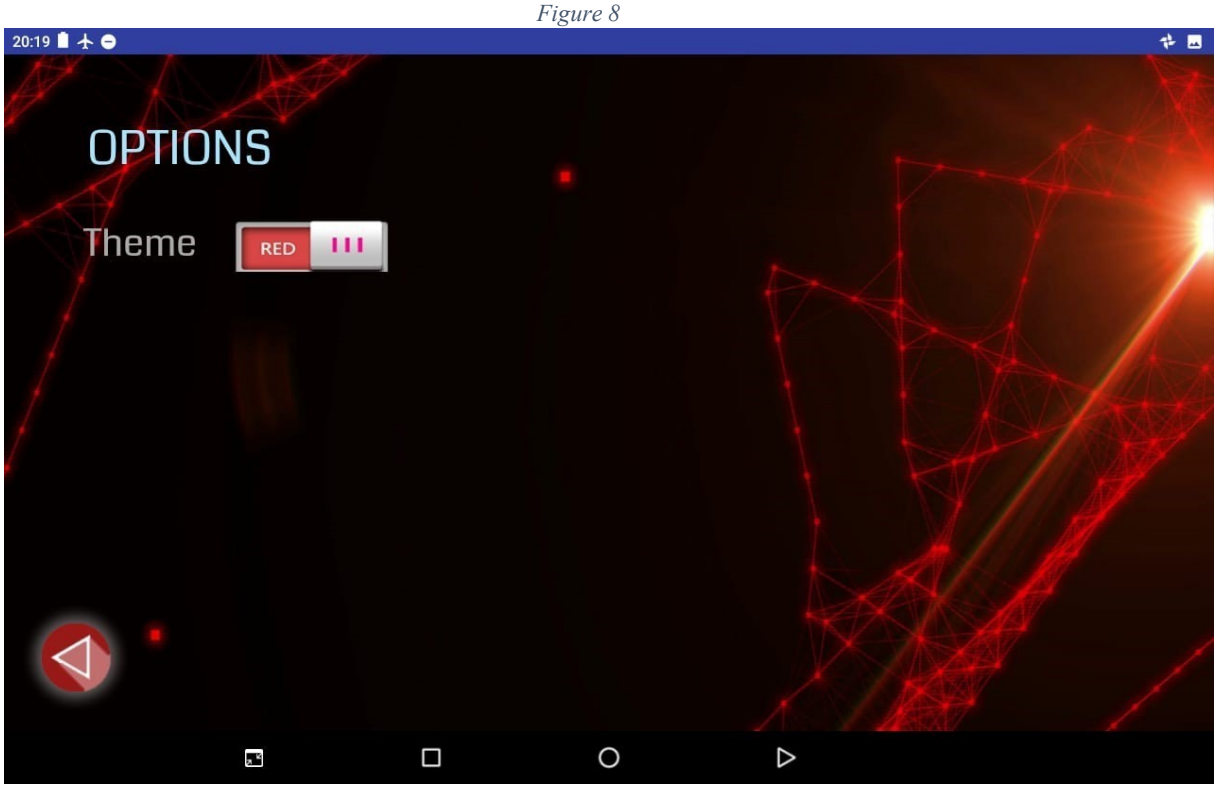
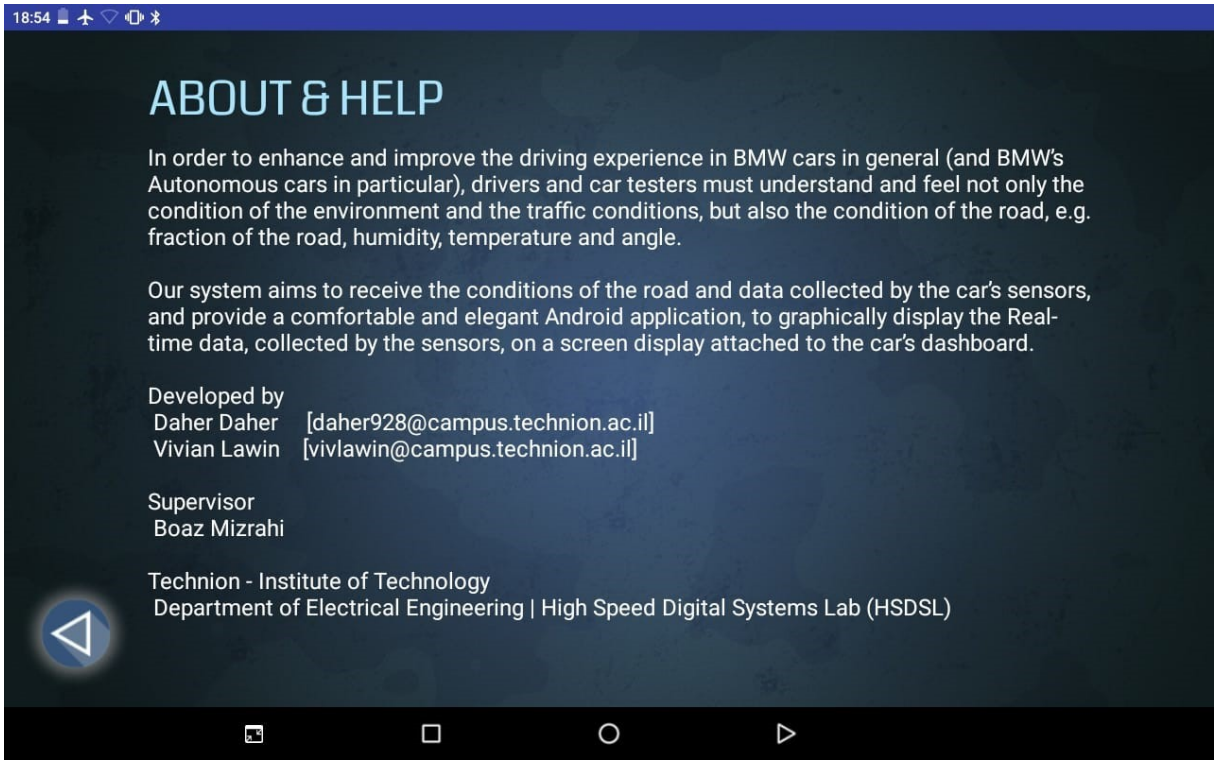


Figure 9

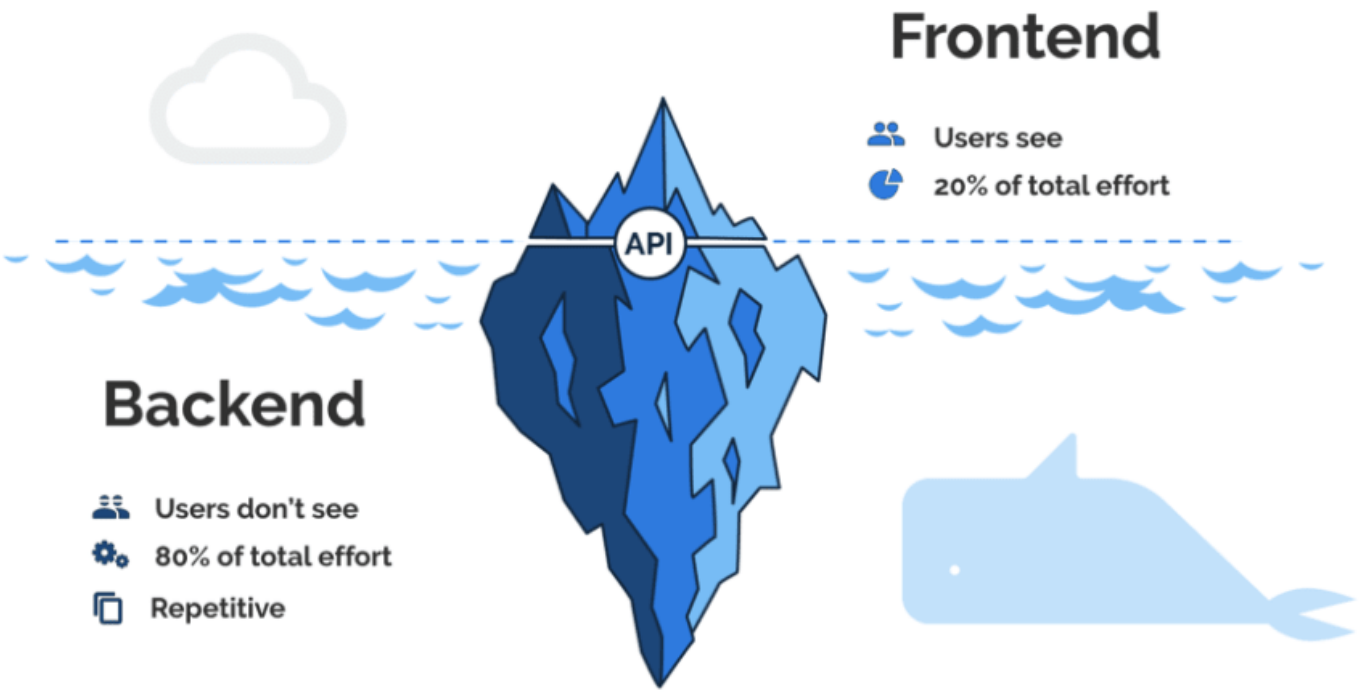
5.1.1.10 About & Help

In this page you can find a brief summary about the application.



5.6 Back-end

In this section, you will find a description of both the transmitter (in our case the Arduino device) and receiver at a more low-level and parts of the implementation code of the operations they perform, as well as a description of the library used for managing graphs, and all the cloud services used (Authentication and database).



5.6.1 Arduino

As mentioned earlier, we emulate the real time operation of vehicles by a code embedded into the Arduino device.

For this purpose, we are using Arduino Mega shown in the picture below.

Following is a snippet of the Arduino code, which transmits over serial USB connection streams of data.

The full Arduino code will be provided in the **Appendix**.

```
//9600 Baud rate is used.
void setup()
{
 Serial.begin(9600);
}

void loop()
{
  // wait for serial port to be ready. Needed for native USB
  while (!Serial) {
        ;
   }
   // start sending streams
   for (int i =0; i< ARRAYSIZE; i++) {
     Serial.println(stream_sample[i]);
     delay(2); // 500 streams/sec
   }
```

The 2 milliseconds delay sets transmitting rate of 500 streams/sec.

5.6.2 Serial Receiver

The serial receiver actually the Android application, which is composed of 2 threads:

1. Listener thread, which establish the serial connection with the serial transmitter (Arduino for instance), and upon receiving data, this thread parses the data line, creating a well-defined parsed object which is enqueued into a producer-consumer queue for further processing.
   This thread also logs the data (if logging option is enabled) into the cloud storage (explanation in the coming chapters).

The following code snippet show the serial connection establishing:

```
connection = usbManager.openDevice(device);
serialPort = UsbSerialDevice.createUsbSerialDevice(device, connection);
if (serialPort != null) {
    if (serialPort.open()) { //Set Serial Connection Parameters.
        serialPort.setBaudRate(SERIAL_BAUD_RATE);
        serialPort.setDataBits(UsbSerialInterface.DATA_BITS_8);
        serialPort.setStopBits(UsbSerialInterface.STOP_BITS_1);
        serialPort.setParity(UsbSerialInterface.PARITY_NONE);
        serialPort.setFlowControl(UsbSerialInterface.FLOW_CONTROL_OFF);
        serialPort.read(mCallback);

        Toast.makeText(context, text: "Serial Connection Opened!", Toast.LENGTH_SHORT);
    } else {
        Log.d( tag: "SERIAL", msg: "PORT NOT OPEN");
    }
} else {
    Log.d( tag: "SERIAL", msg: "PORT IS NULL");
}
```

The callback which is responsible for data receiving and parsing:

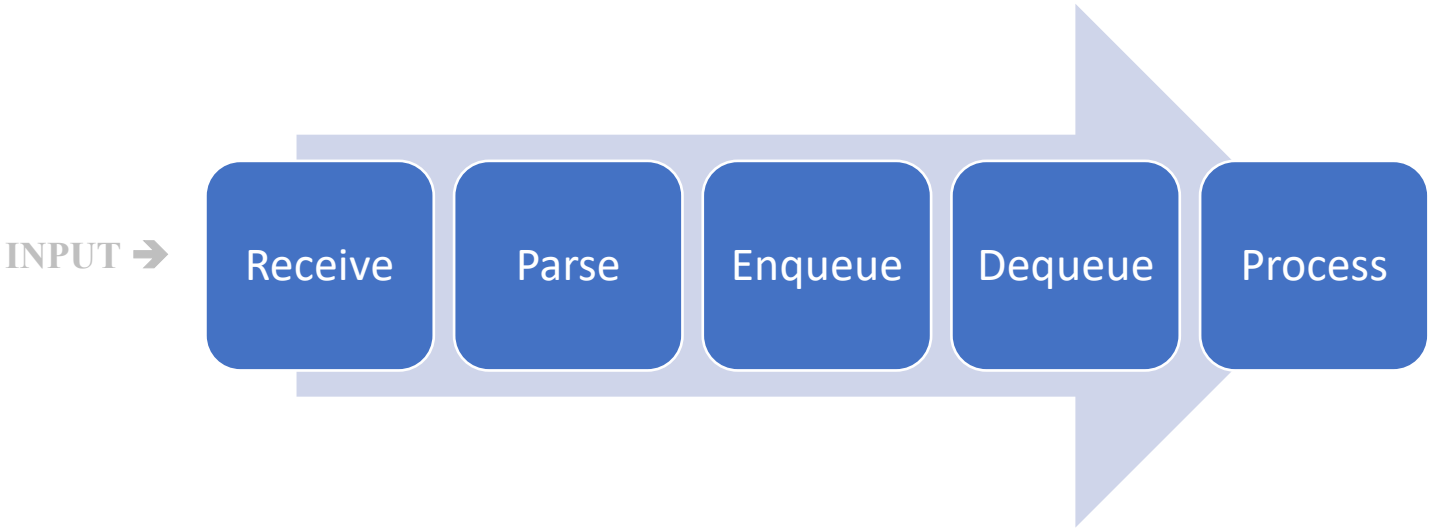
```
UsbSerialInterface.UsbReadCallback mCallback = new UsbSerialInterface.UsbReadCallback() {
    @Override
    public void onReceivedData(byte[] arg0) {
        //Here we receive the data
        try {
            String data = new String(arg0);
            streamLine += data;
            if (streamLine.contains(END_OF_LINE)) {
                String[] split = streamLine.split(END_OF_LINE);
                final String singleStream = split[0];
                streamLine = split.length > 0 ? split[1] : "";
                StreamLine parsedStream = StreamUtil.parse(singleStream);
                if (parsedStream != null) {
                    AppState.queue.add(parsedStream.toString());
                }
```

2. Processing thread, which dequeues the parsed data from the producer-consumer queue, and updates graphs accordingly, taken into consideration the different configurations and chosen sensors (filtering)

   Code snippet of the processing part:

```
while (!AppState.queue.isEmpty() && running) {
    final String s = AppState.queue.poll();
    h.post(new Runnable() {
        @Override
        public void run() {
            final String timestampStr = s.split( regex: "#")[0];
            final long timestampLong = Long.parseLong(timestampStr);
            Timestamp ts = new Timestamp(timestampLong);
            Date d1 = new Date(ts.getTime());
            final String id = s.split( regex: "#")[1];
            final String val = s.split( regex: "#")[2];
            if (!AppState.selectedIds.contains(id)){
                return;
            } else {
                Log.d( tag: "New point will be displayed", s);
                try {
                    final double double_val = Integer.valueOf(val,  radix: 16);
                    int graph_idx = AppState.selectedIds.indexOf(id);
                    Sensor currSensor = AppState.getSensorFromId(AppState.selectedIds.get(graph_idx));
                    double resolution = currSensor.getConfig().getResolution();
                    double offset = currSensor.getOffset();
                    double final_val = double_val * resolution + offset;
```

Altogether, the Android application is the following pipeline:



5.6.3 Authentication

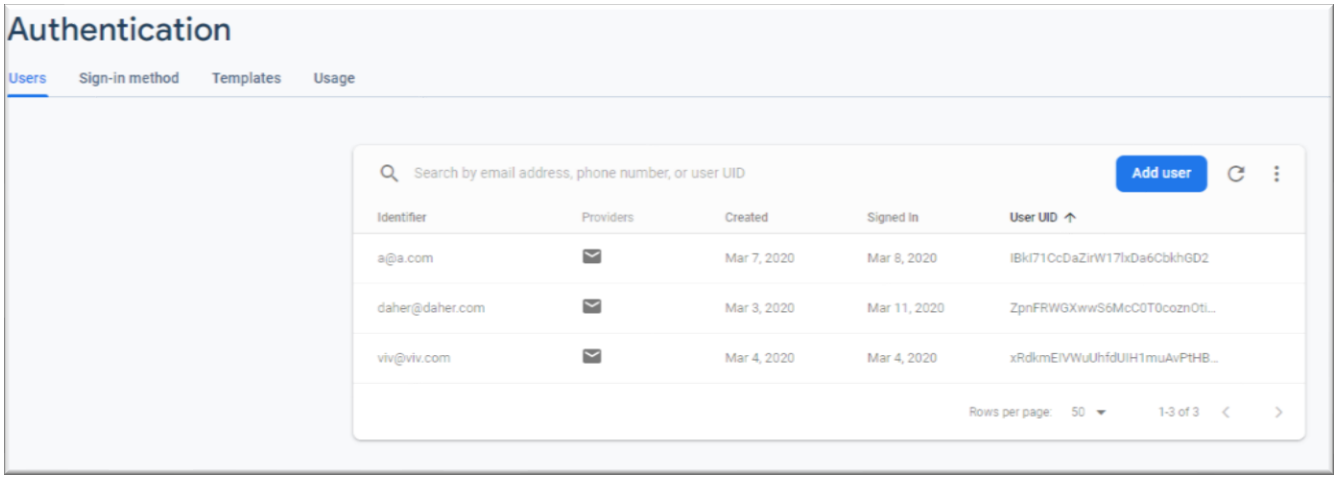
Knowing a user's identity allows an app to securely save user data in the cloud.

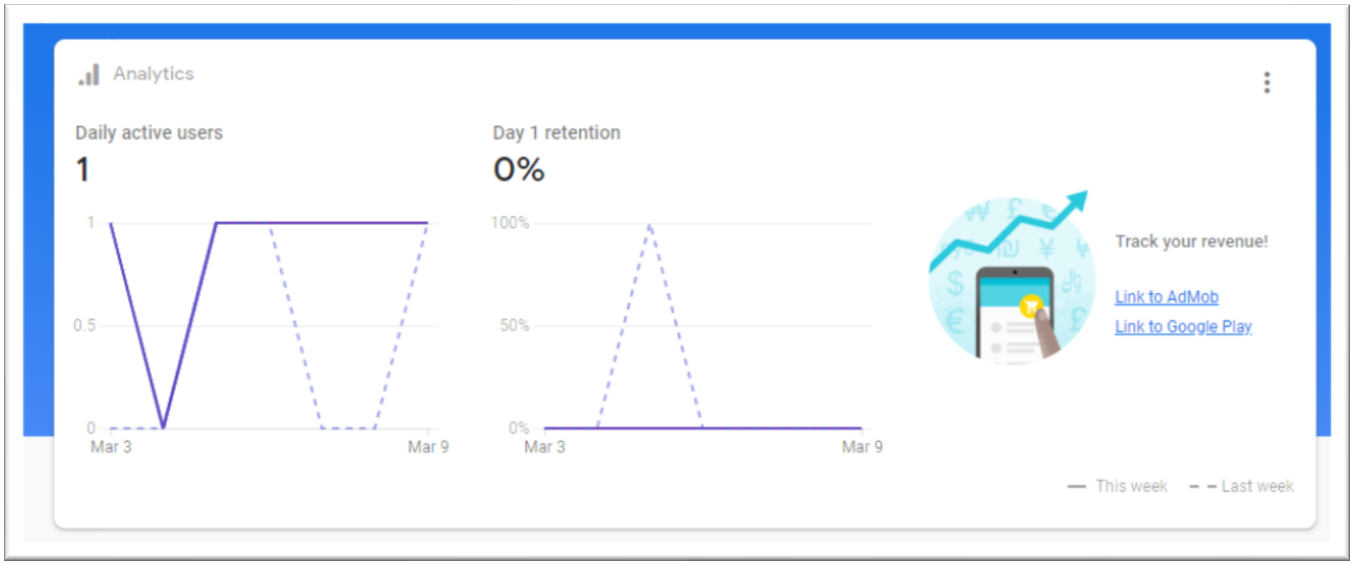
Therefore, and in order to give the user his own space, his own logging capabilities and history, we implemented an authentication mechanism using **Firebase Authentication.**

Firebase Authentication provides backend services, easy-to-use SDKs, and ready-made UI libraries to authenticate users to your app. It supports authentication using passwords, phone numbers, popular federated identity providers like Google, Facebook and Twitter, and more.

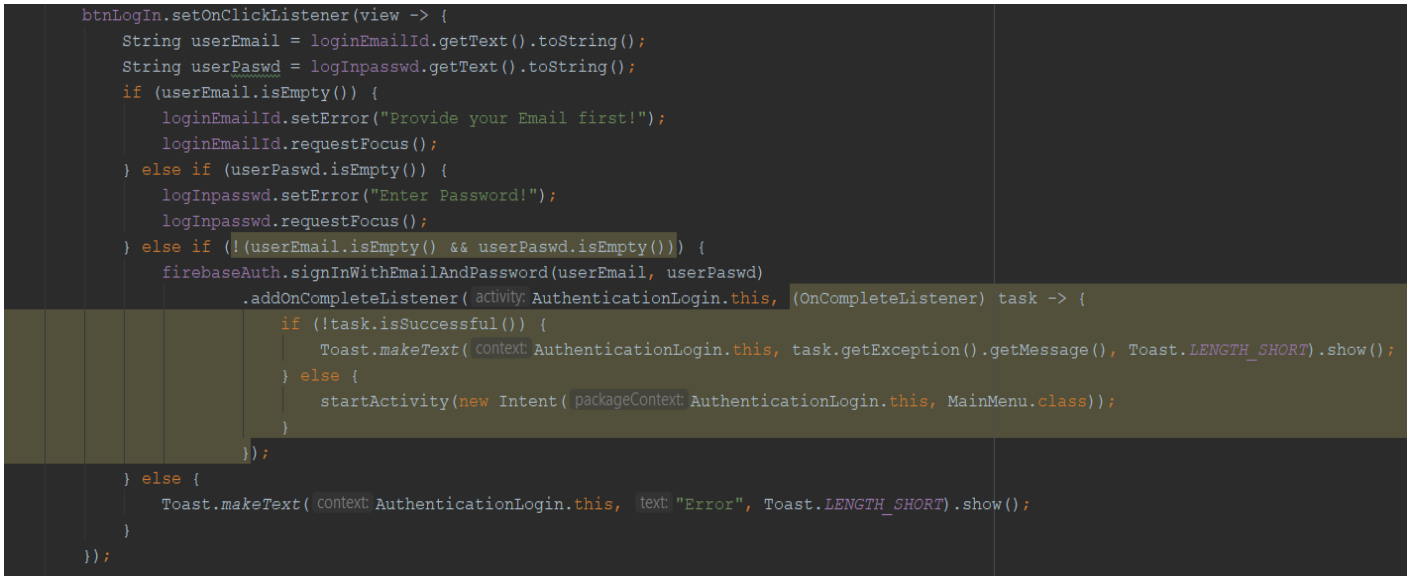
We chose authentication using Email-Password with an ability to create a new user (Sign in / Signup screens)

In the firebase console (which can be accessed in the web browser, mobile or any other supported platform) we can inspect all users and track their activity statuses:





The authentication backend is as follows:



```
btnLogIn.setOnClickListener(view -> {
    String userEmail = loginEmailId.getText().toString();
    String userPaswd = logInpasswd.getText().toString();
    if (userEmail.isEmpty()) {
        loginEmailId.setError("Provide your Email first!");
        loginEmailId.requestFocus();
    } else if (userPaswd.isEmpty()) {
        logInpasswd.setError("Enter Password!");
        logInpasswd.requestFocus();
    } else if (!(userEmail.isEmpty() && userPaswd.isEmpty())) {
        firebaseAuth.signInWithEmailAndPassword(userEmail, userPaswd)
                .addOnCompleteListener( activity: AuthenticationLogin.this, (OnCompleteListener) task -> {
                    if (!task.isSuccessful()) {
                        Toast.makeText( context: AuthenticationLogin.this, task.getException().getMessage(), Toast.LENGTH_SHORT).show();
                    } else {
                        startActivity(new Intent( packageContext: AuthenticationLogin.this, MainMenu.class));
                    }
                });
    } else {
        Toast.makeText( context: AuthenticationLogin.this,  text: "Error", Toast.LENGTH_SHORT).show();
    }
});
```

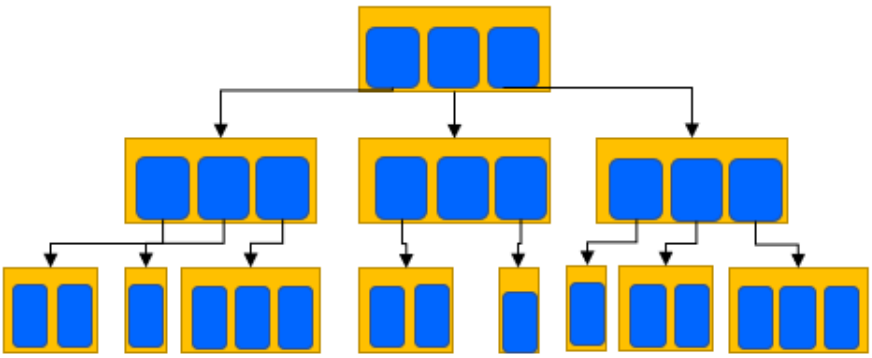
When the user inserts the user name and password, the application receives them, run validations in order to check if the email is a valid email format, and if the password meets the password security requirements.

When the user successfully logs in or signs up, the app will start the Main Menu screen, enabling the user to start the experience flow.

5.6.4 Cloud Storage

In order to record and save user logs of streams received (filtered or not), we use **Cloud Firestore** which is another firebase service.

Cloud Firestore is a real-time NoSQL database, in which data are arranged in a hierarchical structure of documents and collections



Cloud Firestore uses data synchronization to update data on any connected device, and lets developers easily store, sync, and query data for mobile and web apps - at global scale.

Using the Cloud Firestore, we log the received streams of data which contain the sensorId and sensorData, along with a timestamp and the users identification, and also in another document we log all previous configurations sets (see Previous Configurations).

Using Cloud Firestore's API, the following code implements the logging and the write operation of data to the database:

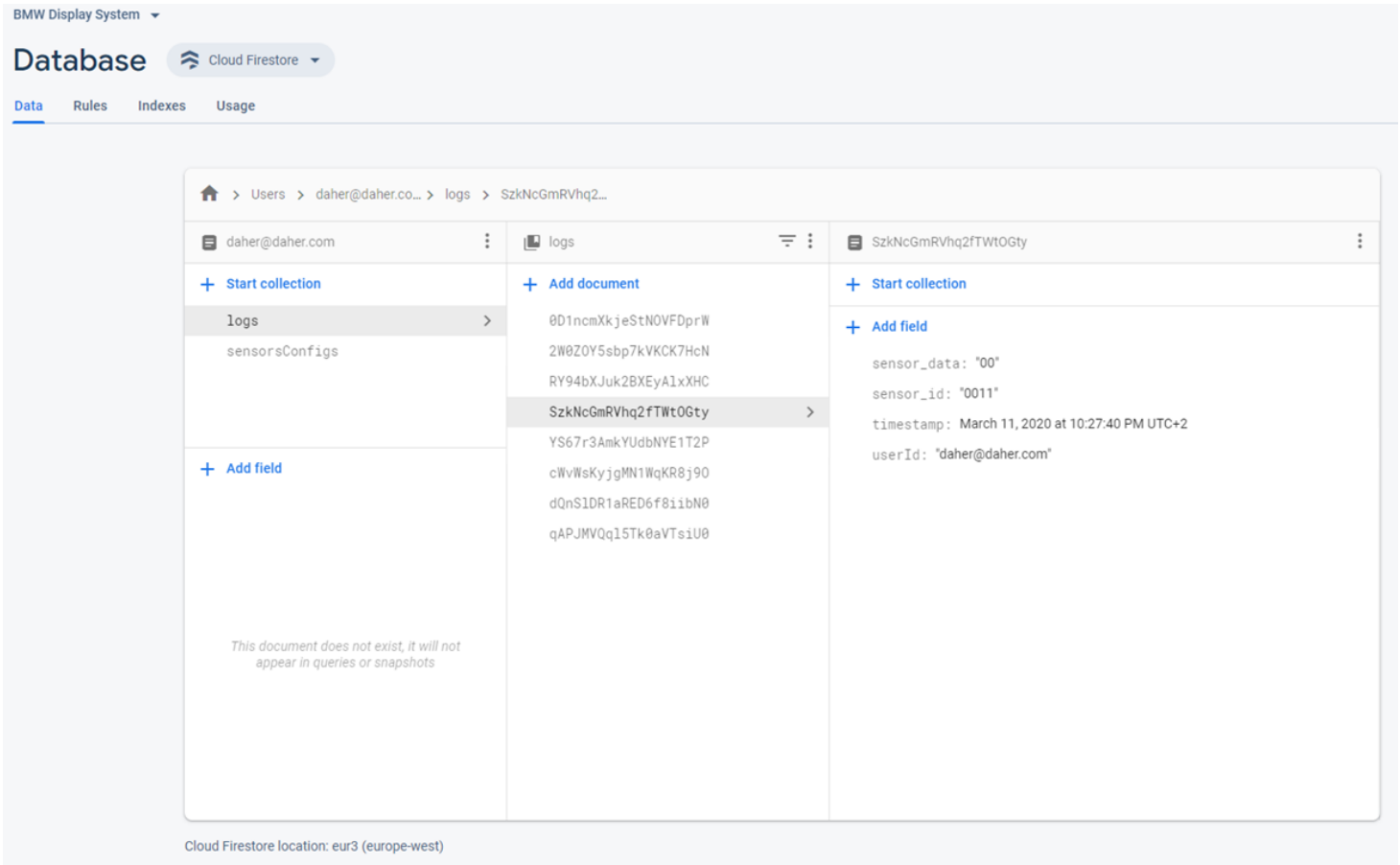
```
Map<String, Object> new_data = new HashMap<>();
String sensorId = parsedStream.getSensorId();
String sensorData = parsedStream.getSensorData();
long timestamp = parsedStream.getTimeStamp();
String userId = FirebaseAuth.getInstance().getCurrentUser().getEmail();
Date date = new Date(timestamp);

new_data.put(SENSOR_ID_DOCUMENT_PROPERTY, sensorId);
new_data.put(SENSOR_DATA_DOCUMENT_PROPERTY, sensorData);
new_data.put(DATE_DOCUMENT_PROPERTY, date);
new_data.put(USER_EMAIL_PROPERTY, userId);

FirebaseFirestore.getInstance().collection( collectionPath: "Users") CollectionReference
        .document(userId) DocumentReference
        .collection( collectionPath: "logs") CollectionReference
        .add(new_data) Task<DocumentReference>
        .addOnCompleteListener(task -> {
            if (!task.isSuccessful()) {
                Log.d(TAG, msg: "Get failed with ", task.getException());
            }
        });
```

While the next code snippet demonstrated the implementation of logs retrieving (querying the database):

```
firestore.collection( collectionPath: "Users") CollectionReference
    .document(userId) DocumentReference
    .collection( collectionPath: "logs") CollectionReference
    .whereIn(SENSOR_ID_DOCUMENT_PROPERTY, AppState.selectedIds) Query
    .whereEqualTo(USER_EMAIL_PROPERTY, FirebaseAuth.getInstance().getCurrentUser().getEmail()) Query
    .orderBy(DATE_DOCUMENT_PROPERTY, Query.Direction.ASCENDING) Query
    .get() Task<QuerySnapshot>
    .addOnCompleteListener(task1 -> {
            if (task1.isSuccessful()) {
                task1.getResult().getDocuments().forEach(
                        doc -> completeLogView.append(
                                doc.get(DATE_DOCUMENT_PROPERTY) + " ID=" +
                                        doc.get(SENSOR_ID_DOCUMENT_PROPERTY) + " Value=" +
                                        doc.get(SENSOR_DATA_DOCUMENT_PROPERTY) + "\n"));
            } else {
                Log.d( tag: "ERROR", msg: "get failed with ", task1.getException());
            }
        }
    );
```

In addition to being able to view the logs in the Android application itself (see logs and stats), users can inspect the database in the Firebase console.

For example, viewing the users logs:



In the above screenshot, we can see that we traversed over the required documents/collections to get to the desired data:

Users → daher@daher.com → logs → <required log>

6 Appendix

6.1 Arduino Code

```
#define ARRAYSIZE  500
 String results[ARRAYSIZE] = {"1004180008080",
                              "1002A1027102710271027",
                              "1002A1027102710271027",
                              "1003D0064",
                              "1002A1027102710271027",
                              "1002A1027102710271027",
                              "1002A1027102710271027",
                              "1002A1027102710271027",
                              "1002A1027102710271027",
                              "1002A1027102710271027",
                              "1003D0064",
                              "1002A1027102710271027",
                              "1002A1027102710271027",
                              "1002A1027102710271027",
                              "1002A1027102710271027",
                              "1002A1027102710271027",
                              "1002A1027102710271027",
                              "1003D0064",
                              "1002A1027102710271027",
                              "1002A1027102710271027",
                              "1002A1027102710271027",
                              "1002A1027102710271027",

                              ... etc
  };

 //9600 Baud rate is used.
 void setup()
 {
  Serial.begin(9600);
 }

 void loop()
 {
   // wait for serial port to be ready. Needed for native USB
   while (!Serial) {
        ;
    }
    // start sending streams
    for (int i =0; i< ARRAYSIZE; i++) {
      Serial.println(results[i]);
      delay(2); // 500 streams/sec
    }
}
```

6.2 Sensors Configurations File

Sensors configurations file is a CSV file that the Android application (receiver) reads on startup to learn and get information about the available sensors, and their default configurations.

File name: sensors.csv Content:

```
0001,Raw pressure(After LPF),Pa,0,1048575,0.01,0
0002,Slope,Deg,0,65535,0.1,500
0003,Height ASL,cm,0,1048575,1,0
0005,Real Odometer,m,0,4294967295,1,0
0007,Engine Hours,minute,0,4294967295,1,0
0008,Dashboard Odometer,m,0,4294967295,1,0
0011,Speed,km/h,0,255,1,0
0012,RPM,revolutions/minute,0,65535,1,0
0013,Pedal/Throttle,0~255,0,255,1,0
0014,Fuel Consumption Rate,L/1000km,0,65535,1,0
0015,Fuel Consumption Rate,L/1000hour,0,65535,1,0
0016,Engine Coolant Temperature,Deg,0,255,1,40
0017,Engine Running,boolean,0,1,1,0
0018,Engine Calculated Load,%,0,125,1,0
0019,Vehicle Weight(CAN),10kg,0,65535,1,0
001A,Vehicle Weight(MWProc),10kg,0,65535,1,0
001B,Vehicle Weight Percentage,%,0,100,1,0
001C,PTO State,boolean,0,1,1,0
001D,Cruise Control State,boolean,0,1,1,0
001E,Break Pedal State,boolean,0,1,1,0
0021,Gear state,0-neutral 251-parking,0,253,1,0
0022,Current retarder torque,No Unit,0,127,1,0
0028,Tank fuel level(CAN),%,0,255,0.4,0
0029,Raw Pressure(Before LPF),Pa,0,1048575,0.01,0
002F,Rough weight estimation,1-empty 3-full,0,3,1,0
0030,Low Resolution Fuelometer,mL,0,4294967295,0.5,0
0032,High Resolution Fuelometer,mL,0,4294967295,1,0
0038,Ambient temperature,0.01Deg °C +4000 offset,0,65535,0.01,4000
003A,Coarse weight estimation,No Unit,0,5,1,0
003D,Actual brake toque,Nm,0,32000,1,0
003F,Tire health,1-worst 10-best,0,10,1,0
0043,Supported OBD-II PIDs,No Unit,256,287,1,0
0044,Torque transfer percentage,%,0,100,0.4,0
0045,Streering position(CAN),Deg,-90,90.024645,0.002747,-90
0046,Vehicle CoG speed(CAN),Km/h,0,350,0.01,100
0047,Average rain intensity,%,0,100,1,0
0049,Supported OBD-II PIDs,No Unit,287,318,1,0
```

Each line is parsed as follows:

<SensorId>,<SensorName>,<Units>,<minVal>,<maxVal>,<Resolution>,<Offset>

6.3 GraphView Library

GraphView is a library for Android to programmatically create graphs and diagrams.

The GraphView library enables programmers to create Line Graphs, Bar Graphs, Point Graphs or create their custom graph.

Key Features

- Different plotting types: Line Chart, Bar Chart and Points Chart and they can be plotted together as a combination.
- Draw multiple series of data
- Let the diagram show more than one series in a graph.
- Realtime / Live Chart - Append new data live or reset the whole data.
- Secondary Scale.
- Tap Listener
- Handle tap events on specific data points.
- Show legend.
- Custom label formatter
- Handle incomplete data
- Viewport
- Scrolling and Scaling / Zooming
- XML Integration
- Optional Axis Titles
- Set vertical and horizontal axis titles.
- Customizable - color and thickness, label font size/color and more

XML Layout file:

```
<com.jjoe64.graphview.GraphView
android:layout_width="match_parent"
android:layout_height="200dip"          android:id="@+id/graph"
/>
```

Java code:

```
GraphView graph = (GraphView) findViewById(R.id.graph);
LineGraphSeries<DataPoint> series = new LineGraphSeries<>(new
DataPoint[] {
          new DataPoint(0, 1),
new DataPoint(1, 5),           new
DataPoint(2, 3)
});
graph.addSeries(series);
```

Reference: https://github.com/jjoe64/GraphView

6.4 Firebase References

Firebase: https://firebase.google.com/

Firebase Authentication: https://firebase.google.com/docs/auth/

Cloud Firestore: https://firebase.google.com/docs/firestore/