Android GPS Data Logger

A. T. Coetzee

Project Report

Supervisor:	**Prof. W. Venter**
Date:	**October 16, 2021**
Student Number:	**29982995**

Abstract

Trucking companies require a means to track their employees, ensuring they perform their work as expected. In order to achieve this, a tracking solution is considered making use of ubiquitous smartphone devices with built-in GPS capability. Using sensor data retrieved from smartphones, a solution is considered which intends to generate detailed reports for employers of trucking companies. Such a report is proposed to be displayed to the manager through a web application.

A draft high-level architecture for such a system is postulated and discussed, along with technical requirements, scope definition and deliverables.

CONTENTS

LIST OF FIGURES

ACRONYMS

API   Application Programming Interface, 19 f., 25, 39

APK   Android Package, 39

app   application, 9

CA   Certificate Authority, 39

CAN   Control Area Network, 6 f.

CoAP   Constrained Application Protocol, 6

CSS   Cascading Style Sheets, 24, 30

DAO   Data Access Object, 19

DI   Dependency Injection, 8 f., 19

ECU   Electronic Control Unit, 6

GPS   Global Positioning System, 1 ff., 6, 21, 34

HTML   HyperText Markup Language, 24 f., 30, 32, 37

HTTP   Hypertext Transfer Protocol, 12, 25, 37

HTTPS   Hypertext Transfer Protocol Secure, 12

I/O   Input/Output, 2, 4 f., 9 ff., 14 f., 17, 19–23, 34, 39 f.

ID   Identity, 17, 23, 25, 32–35

iOS   iPhone Operating System, 8

IP   internet protocol, 39

JSON   JavaScript Object Notation, 12, 23, 34 f.

JVM   Java Virtual Machine, 8

MQTT   Message Queuing Telemetry Transport, 6

MVC   Model-View-Controller, 11, 16, 24

MVVM   Model-View-ViewModel, 9, 18

NoSQL   Not only SQL, 11

OBD   on-board diagnostic, 7

OOP   object-oriented programming, 7

ORM   Object Relational Mapper, 9

RDBMS   Relational Database Management Systems, 11, 15, 27

REST   Representational State Transfer, 23

SAE   Society of Automotive Engineers, 7

1 INTRODUCTION

1.1 Purpose of document

This report documents the contextualization of a problem surrounding the tracking of truckers. The background and problem is considered, possible solutions with objectives are identified, along with an expected outcome for the potential solution, requirements and scope definition.

The design and implementation of the postulated solution is documented. Finally, this solution is evaluated and analyzed, and future recommendations are considered.

1.2 Background

Due to the nature of the trucking industry, it is difficult for company owners to keep track of their employees. Truckers carry out their shifts delivering cargo to various locations over far distances. As such, it is not possible for employers to track their whereabouts throughout their shifts.

Lack of supervision allows truckers the ability to behave undesirably while on the job. They can waste time by taking unnecessarily long stops. Some truckers may drive erratically, unsafely or illegally. Such employees are a liability to the reputation and profitability of their respective companies.

The ability to track truckers would provide a potential means to address this issue, by allowing employers to monitor their truckers' location, progress and behavior throughout their shifts. The ability to produce an audit trail detailing the truckers whereabouts during their shifts would allow managers to ensure that work is adequately executed. Such an audit trail would comprise of:

- **Global Positioning System (GPS) coordinates**

  GPS tracking will allow employers to ensure that truckers are traveling to required locations, and doing so via effective routes. This also allows employers to ensure no unnecessary detours occur.

- **Altitude**

  Altitude logging may be useful for identifying trends in routes traveled, especially where large altitude gradients occur. Trucks often struggle traversing steep gradients. Altitude analysis may offer insight for companies looking to perform optimizations.

- **Speed**

  Examining trucker speeds allows for managers to monitor how quickly truckers are able to transport

goods. Slower routes may be identified where traffic is more prevalent. This could allow for route optimization, or truck malfunctions.

- **Acceleration**

  Acceleration may be used for inferring any dangerous driving behavior. Erratic acceleration and deceleration is associated with dangerous driving behavior. Additionally, erratic driving can cause more strain and deterioration on the vehicle.

The ubiquitous nature of cheap, GPS-equipped smartphones provides a potential avenue for realizing a simple solution at low cost. In addition, nation-wide continuous access to the internet allows for live tracking to be utilized most of the time.

1.3 Problem Statement

A tracking system must be implemented to be used by trucking companies for tracking and logging their truckers' **GPS coordinates, altitude, speed and acceleration**. Logging should be performed with sensors available on most standard smartphones.

Logging data should be continuously (or periodically) made accessible to managers remotely. In addition, an interface is required for storing, processing and displaying logged data pertaining to the trucker's location and behavior.

1.4 Hypothesis

An low-cost anticipated architecture, as depicted in figure 1, involves the development of a smartphone application capable of interfacing with internal or external sensors, and transferring sensor data through some Input/Output (I/O) server into a data store. [1] For the purposes of information presentation, it is proposed that a web server process and serve inferred information to a user via a web application.
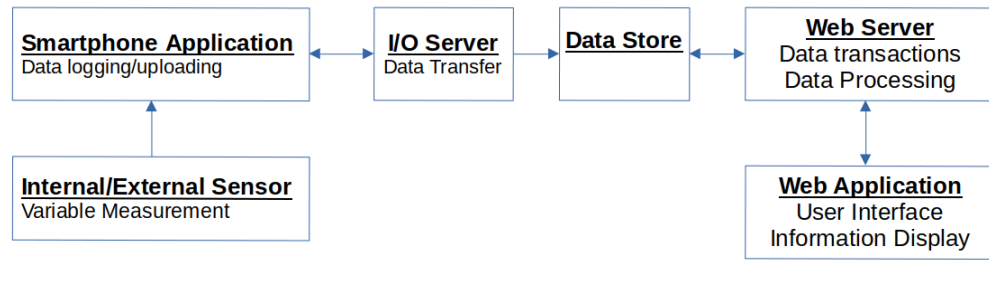
Fig. 1: Proposed High Level Architecture

1.5 Project Objective

1.5.1 Primary Objective: The primary objective in addressing the problem will be the development of detailed reports showcasing the trucker's whereabouts and behavior during their work shifts.

1.6 Anticipated Benefits of Solution

Managers will be able to ensure that their truckers conduct their work efficiently and responsibly. They will then be able to adequately handle truckers who fail to perform as expected.

Managers may also be able to analyze trucker behavior to perform optimizations, potentially allowing for increased efficiency.

1.7 Technical Requirements

A set of requirements are identified in realizing the hypothesized solution and the scope is identified.

1.7.1 Requirements:

1) **Smartphone Application**

   This will be a smartphone application used by the entities being tracked(i.e the truckers).
   a) Trucker identification control must be implemented to ensure that logs sent to the server correspond to a unique trucker. It must not be possible for multiple truckers to assume the same or no identity.

b) Every 2 minutes, sensor data capturing the **GPS coordinates, altitude, speed and acceleration** must be captured and stored internally on the android device. Data capacity for one continuous week of storage must be possible, to account for connectivity issues.

c) The application must be able to run in the background, allowing for multitasking.

d) Sensor data must be uploaded to a central data store, either continuously or on request. This communication must be encrypted for security purposes.

2) **I/O Server**

This server will facilitate the transfer of logged data from the Smartphone Application to a central data store, via an internet connection.

a) As a dedicated transfer server, it must exhibit high performance, handling multiple requests from the multiple smartphone clients asynchronously.

b) Trucker logs, received from smartphone clients, must be stored in a central database.

c) Information about the trucking company must also be sent to the smartphone client.

3) **Data Store**

The data store will be efficient, fast and capable of storing large volumes of data. It should also be capable of adequately interfacing with the I/O server and the Web Server. The web server is responsible for querying data from the data store, and serving requests to the web application.

4) **Web Server and Web Application**

The web server must implement back-end business logic and serve pages in the web application. The web application acts as an interface for managers to add truckers and view tracking information about their fleets.

a) Multiple trucking managers must be able to log in and use the application.

b) Managers must be able to add multiple truckers to their fleet, including trucker-specific information such as name, and vehicle number.

c) Managers must be able to view detailed trip information for any adjustable time period. Log data must be processed to determine starting and arrival times for locations traveled to. Statistical information about acceleration and speed should be displayed, including averages, maximums and percentiles.

1.7.2 Scope Definition: The scope of the problem considered will include

1) **Internal/External Sensor Interface**

The scope **does not** include the design of sensor circuitry meant to interface with the smartphone.

Only configurations capable of interfacing with the smartphone are considered. The smartphone app is not concerned with displaying user reports and statistics. That is left to the web application.

2) **Smartphone Application**

The smartphone application is purely responsible for logging the appropriate sensor data and transferring the sensor data on through the I/O server. Other measurable variables such as temperature, fuel and pressure are not considered.

3) **I/O Server**

The I/O server is purely responsible for facilitating the transfer of sensor data from the smartphone to the data store.

4) **Data Store**

The data store element is purely concerned with the storage of logs, user identity information and providing an interface for the I/O server to query and add records to the store. Existing data store providers will be considered.

1.8 Deliverables

The deliverables will require the entire project to function, from the smartphone logging implementation, to the detailed reports available in the web application.

- Smartphone Application and I/O server
- Web Application and web server

1.9 Conclusion

Basic contextualization of the problem has been performed. Low level details, however, have not been considered. Each aspect of the planned architecture may be realized in multiple ways on the low level. Further research and a feasibility analysis are necessary for adequate low level design.

2 LITERATURE REVIEW

This section tackles the investigation of components which make up the proposed high level system depicted in figure 1. There exists a variety of different tools available to realize each system. With the hardware pre-existing, most of the design exists in the software domain. Various software tools and methodology are considered.

2.1 Internal and External Sensors

Effective data logging of acceleration, altitude, location and speed all begin with the quality of measurements being made. Smartphones alone provide a wealth of options. However, external sensors available to the truck operators may also be considered.

2.1.1 Internal Sensors: Most smartphones come well-equipped with a wide variety of on-board sensors, such as GPS sensors, accelerometers, gyroscopes, magnetometers and ambient light sensors, among others [2]. As such, they are capable of inferring a wealth of information related to driving patterns. This includes dangerous driving behavior, for which algorithms have been developed [3].

The variety of on-board sensors provide an adequate means of measuring acceleration and location (and therefore altitude). However, no effective speed sensor exists for smartphone devices. GPS sensors may be used for inferring speed by computing location-time differentials, but with potentially fluctuating accuracy or possible performance reduction.

Battery life preservation and reduced performance are often concerns when running computationally heavy daemons (background operating system processes). Recent efforts in the development and standardization of new, lightweight sensor-probing protocols have been investigated. Namely, Message Queuing Telemetry Transport (MQTT) and Constrained Application Protocol (CoAP), which are targeted at achieving lightweight, low-power performance [4].

2.1.2 External Sensors: The most practical means of utilizing sensors external to the smartphone may be realized through the use of in-vehicle sensors. The Control Area Network (CAN) bus protocol is a centralized multiplex communication bus standard utilized in many modern vehicles, originally in an attempt to save on copper. The protocol allows for broadcast communication between various Electronic Control Unit (ECU)'s within a vehicle, all centrally connected to one bus. A priority-based scheme is utilized to ensure the most important units transmit their data packets first, while lower priority units are

delayed until a later time when transmission may be uninterrupted. Each packet contains an identifier designating what information is being transmitted, such as wheel speed, temperature, etc. [5]

Assuming that the vehicle has an on-board diagnostic (OBD) connector, communication with a smartphone requires some form of interfacing circuitry. Wireless CAN-to-smartphone interfaces can be most-practically realized via CAN-bus-to-Bluetooth implementations. Such an interface will allow for the smartphone to probe sensor data via the vehicle's CAN bus [6] [7]. The Society of Automotive Engineers (SAE) defines the J1939 standard for CAN-bus communication in the use of heavy-duty vehicles [8], which would be appropriate for the solution.

2.2 Software Architecture

Effective software architecture and design patterns are necessary for writing dynamic, modifiable and modular software.

2.2.1 Separation of Concerns (SoC) and SOLID principles: SoC addresses the need for software to be decomposed into different modular units. Each unit focuses on one main concern, such as data access, authentication, business logic and view rendering. Mixing multiple concerns within one unit leads to code which is less reusable and more difficult to modify. [9]

The 'SOLID' acronym defines a set of guidelines for software design, in object-oriented programming (OOP).

1) **Single Responsibility Principle**

   Classes should have single responsibilities. To achieve this, each responsibility must be implemented in a unique class.

2) **Open/Close Principle**

   Software components such as classes, modules and functions should be open for extension, but closed for modification. That is, classes implementing a modifiable functionality should be extended with interfaces instead of modifying code in the class.

3) **Liskov substitution Principle**

   Objects should be replaceable with derived sub-types without affecting the correctness of the program.

4) **Interface Segregation Principle**

It is better to implement many client-specific interfaces instead of one general-purpose interface. This ensures the interface being implemented only does the minimal that is required.

5) **Dependency Inversion Principle**

Where possible, it is better to depend on implementable abstractions instead of concretely defined objects. This can be realized by depending on implementable interfaces instead of base classes. This allows classes to be less tightly-bound to a base class, allowing for more modular code.

[10]

2.2.2 Dependency Injection (DI): Often classes require instances of other objects (or dependencies) to perform certain functions. It is wasteful to re-instantiate these objects especially if they are used by other classes. DI provides a means to *inject* an instance of a helper object into a class without explicitly recreating the dependency. [11]

Objects which exist for the lifetime of the application are known as singletons, and the use of singletons is often used with DI.

2.3 Smartphone Application

The smartphone application is responsible for extracting the acceleration, altitude, location and speed data from the sensors and relaying this information to the data store. Certain platform and development design decisions are investigated.

2.3.1 Platform Considerations: The two major mobile operating systems are Android (approximately 72.8 % market share) and iPhone Operating System (iOS) (approximately 27.4 % market share) [12]. Android's high market share makes it an attractive option as a target platform for the Smartphone application component of the system.

2.3.2 Development Technologies: Native Android development officially supports the Java, Kotlin, C and C++ programming languages. Kotlin, which compiles on the Java Virtual Machine (JVM), has been pushed by Google as their suggested language for app development. Kotlin aims to reduce the verbosity of traditional Java (which was the standard language used for app development), thereby reducing the prevalence of "bad coding practices." [13] It is noted that Java may still be preferable for programmers with prior Java experience, or in cases where more verbosity is preferred. A native C/C++ tool-chain offers finer control of system hardware for potential performance boosts [14].

Cross-platform development presents a popular option for developing applications for both major platforms. Several development frameworks such as Xamarin, Flutter and Apache Cordova allow for cross-platform development, among others. However, cross-platform development does impose potentially reduced performance, according to [15]. In an ecosystem where hardware used by truck drivers has potential to be slower, cross-platform development is undesirable.

2.3.3 Android - Model-View-ViewModel (MVVM) Design Pattern: Figure 2 depicts the MVVM architecture used in a typical android context. The view (typically activities or fragments in Android) represents the actual rendered output visible to the user. Data displayed by the view is accessed by the view model. The separation of views and view models is necessary for Android applications due to the temporary nature of views. That is, data stored purely in the view component is lost upon re-rendering of the view, while view models hold onto data for longer.[16]

The repository singleton acts as a central holder of application data, which is then accessed by the multiple views. It also interacts with I/O resources such as web resources and database access. Views request data through the repository, and as such shouldn't have direct handles to database connections. [16]

2.3.4 Android - DI with Hilt: Hilt is an android library used for easily implementing DI. It has support for common android components. [17]

2.3.5 Android - Running continuously in the background: Tracking application (app)s need to run continuosly, without forcing the user to keep the app view components open. This can be achieved by implementing the tracking component as a *foreground service*. In this way, the component runs continuously while allowing the user to use other applications.

Users must also be notified of continuously running services for clarity. It is therefore required display notifications about the service. [18]

2.3.6 Android - SQLite database with the Room Object Relational Mapper (ORM): The Room ORM library provides a neat database abstraction layer over SQLite useful for modeling data. SQLite is preferable for android due to its lightweight nature. [19]

The storage capacity of SQLite is basically unlimited. Storage capacity is, however, limited to the storage capability of the smartphone running the application. This makes the use of external storage desirable.
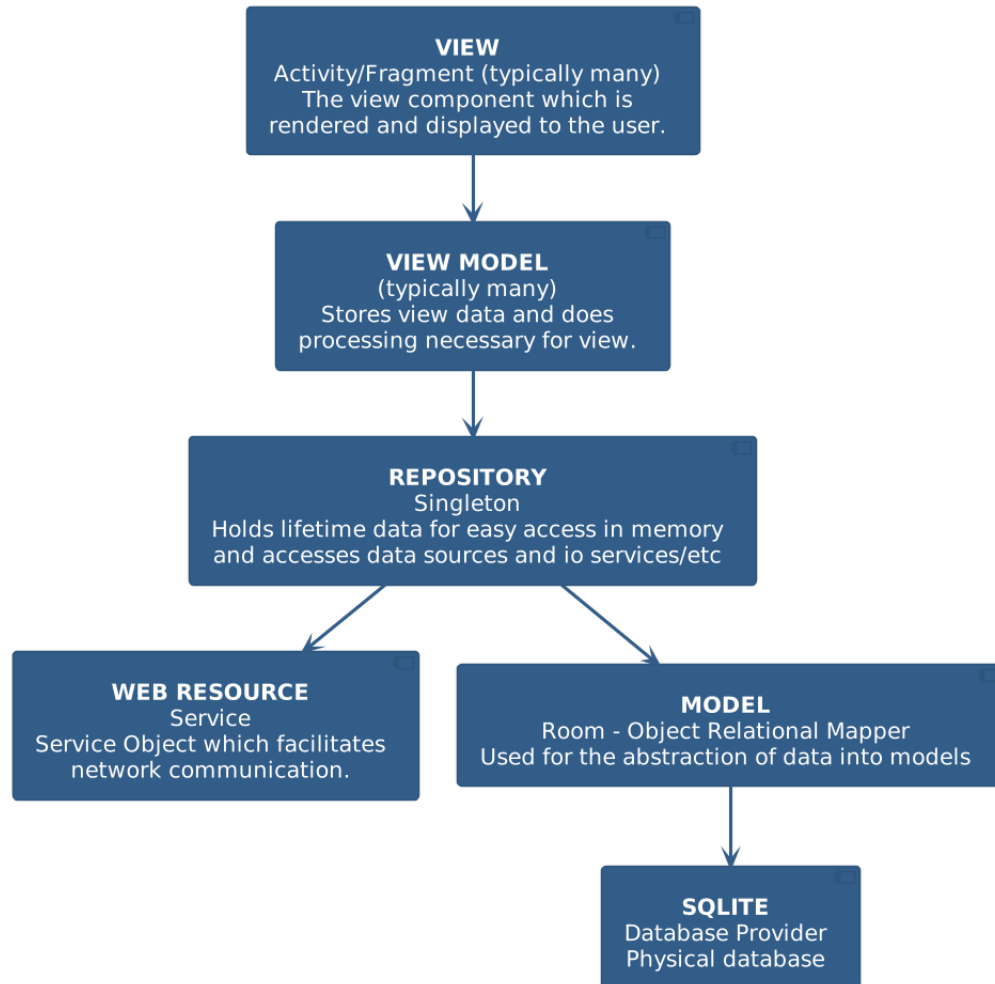
Fig. 2: Android - MVVM Architecture

2.4 I/O Server

The I/O server is required for relaying logged data from the smartphone application to the central data store. It must be many clients quickly and efficiently. This server plays a typical server role; In that it must await requests from clients attempting to establish connection for transmitting data.

Implementations for realizing such a server are possible in many programming languages, and almost all top popular programming languages. Generally, for performance-critical applications, C and/or C++ are considered most appropriate. [20]

2.4.1 Asynchronous I/O: Servers (and many other application) are required to run relatively slow operations; that is communicating over networks and writing to disk. Implementing such functionality

synchronously (using blocking calls) leaves functions essentially waiting for data streams to be read, transmitted and written to disk. This is slow and incapable of handling multiple simultaneous connections.

Asynchronous I/O operations enables other processing to continue before a slow I/O operation has completed. This is essential for servers which handle many simultaneous connections. A popular C++ library, *asio* provides asynchronous I/O functionality. [21]

2.5 Database Considerations

Relational Database Management Systems (RDBMS)s are commonly used in for data handling. Typically, for unnormalized complex data, conventional Structured Query Language (SQL) RDBMSs prove inefficient at scale, due to the tendency of modern data catalogues lacking in structure. In addition, relational databases also start to exhibit slower lookup times for immensely large data sets. The solution to this comes in the form of Not only SQL (NoSQL) database systems, which are scaleable, efficient and capable for storing large volumes of unnormalized data. [22] [23] [24]

However, due to the completely uniform structure of the data being stored, an RDBMS would suffice. Numerous high quality RDBMSs, such as MySQL, Microsoft SQL, PostgreSQL, and Oracle Database are available, among others. All options offer relatively efficient performance. [25]

A lightweight caching database is necessary on the client-side for the momentary storage of data which has yet to be transmitted to the server. To this end, SQLite offers a popular solution for smartphone applications [26].

2.6 Web Application

The web application will be used by managers to display daily reports highlighting their truckers' behavior throughout their shifts.

The web application may be easily realized by utilizing pre-existing web frameworks, such as Microsoft's ASP.NET Core and Oracle's Java Enterprise Edition (with comparable performance) [27].

2.6.1 MVC design pattern for web applications: A relatively popular design pattern in web development is the MVC architecture. As seen in figure 3, MVC attempts to achieve SoC by separating logic required for viewing, routing and data into separate components.
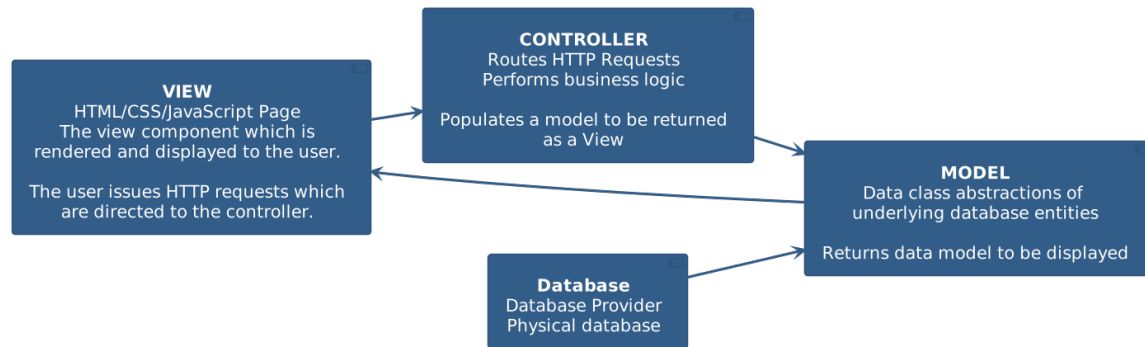
Fig. 3: Web Design Pattern - Model-View-Controller (MVC)

2.7 Secure communication with Secure Socket Layer (SSL)

The use of secure communication over the internet is a modern-day standard. And due to sensitive location data being transmitted, it is necessary to ensure that logs are adequately encrypted.

The SSL protocol is a de facto standard for encrypted communication on the internet. SSL itself is deprecated, and the current standard for encryption is Transport Layer Security (TLS). However, it is common to refer to refer to these related technologies interchangeably, when TLS is the protocol actually in use. [28]

2.8 Serialization and communication protocols

Facilitating communication between two devices requires both devices to use the same protocol. Regardless of this protocol, it is necessary for communication to be encrypted, therefore making use of SSL.

2.8.1 Hypertext Transfer Protocol Secure (HTTPS): HTTPS implements the de facto Hypertext Transfer Protocol (HTTP) protocol over the encrypted SSL protocol. HTTP is an application layer protocol which makes use of standard headers carrying a payload under formalized request types, of which *GET* and *POST* are common. HTTPS is commonly used for web services and websites. [29].

2.8.2 JavaScript Object Notation (JSON) and Extensible Markup Language (XML): XML is a strongly-typed text protocol which can be used for serialization. It follows a tight tagging schema.

JSON is a fast and simple text protocol for serializing objects carrying data. Support for arrays makes JSON reliable for the transmission of many logs. [30]

3 DESIGN

This section considers the context in which the problem exists and the design of each system and subsystem necessary to visualize and realize a possible solution to solve the problem.

The nature of the system exists primarily in the software domain. As such, a suitable design architecture is postulated by the C4 model. This model breaks down the system architecture into different layers of complexity, from a generic high-level system overview down to low-level software abstractions.[31]

Low-level abstractions are realized with Unified Modelling Language (UML) diagrams. UML diagrams detail the members and methods belonging to classes, and the relationships between those classes in an object-oriented codebase. [32]

3.1 System context and base requirements

Figure 4 depicts the system context in the problem domain. Project specifications have identified two parties expected to utilize the system - truck drivers and fleet managers. Identified requirements on the solution dictate that truck drivers will use an android application to log data on the system. In addition, fleet managers must view the logged data and manipulate their fleets via a web application running in a browser.

Fig. 4: System Context Diagram

The high-level life cycle view of the fleet-tracking system design is depicted in figure 5. This life cycle view gives a broad indication of how the system is expected to work for a user. Only front-end components of the system are considered to clarify exactly how users will interact with the system.

Managers are required to perform initial configuration, including adding trucker identity records to a data store. After this, truckers may connect to the system and perform their work while allowing their smartphone applications to track the required sensor data. This data is then relayed to the system, in which managers may analyze and inspect data logs.
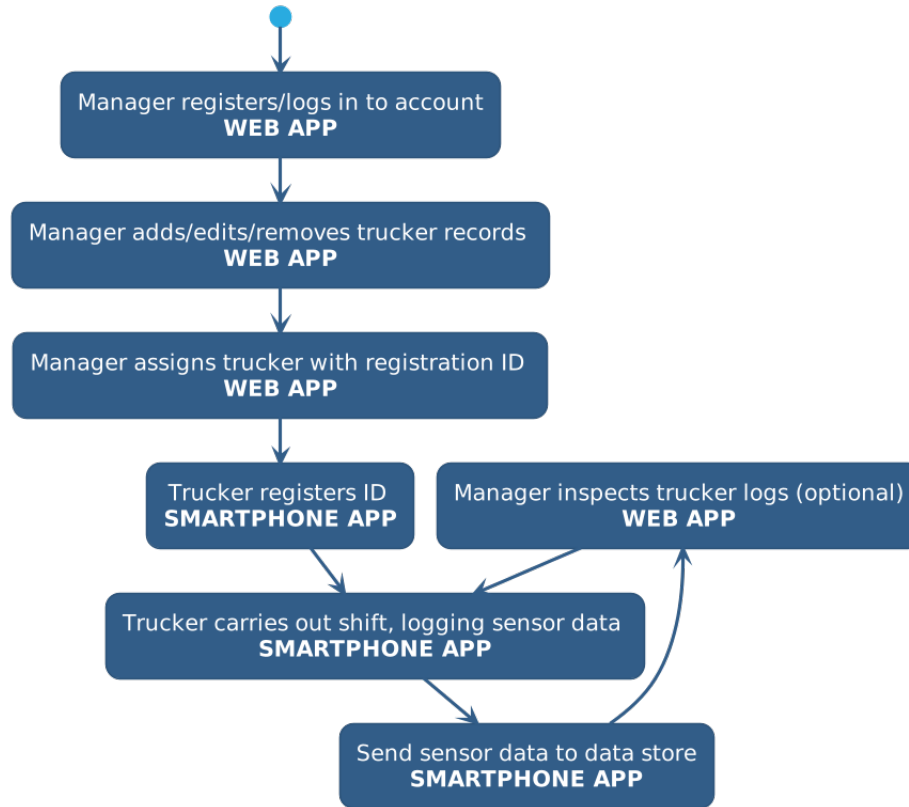
Fig. 5: System Lifecycle - High Level

3.2 Contained subsystems and choice of technologies

The second level of the C4 model identifies the choice of technologies to be utilized to realize the fleet tracking system. The fleet-tracking system is divided into mostly-independent containers, as depicted in figure 6. Each container is a standalone process which makes calls to other processes in the system. The main choice of software tools are identified for each container.

Truckers will make use of an android data-logging application to fetch the various sensor data, and securely transmit this data via an SSL connection. The I/O server, implemented in C++, will listen for multiple asynchronous connections from the android application and relay the data to a MySQL database. A web application, realized with Microsoft's ASP.NET framework fetches the data and allows the fleet manager to view the whereabouts of each member in his/her fleet.
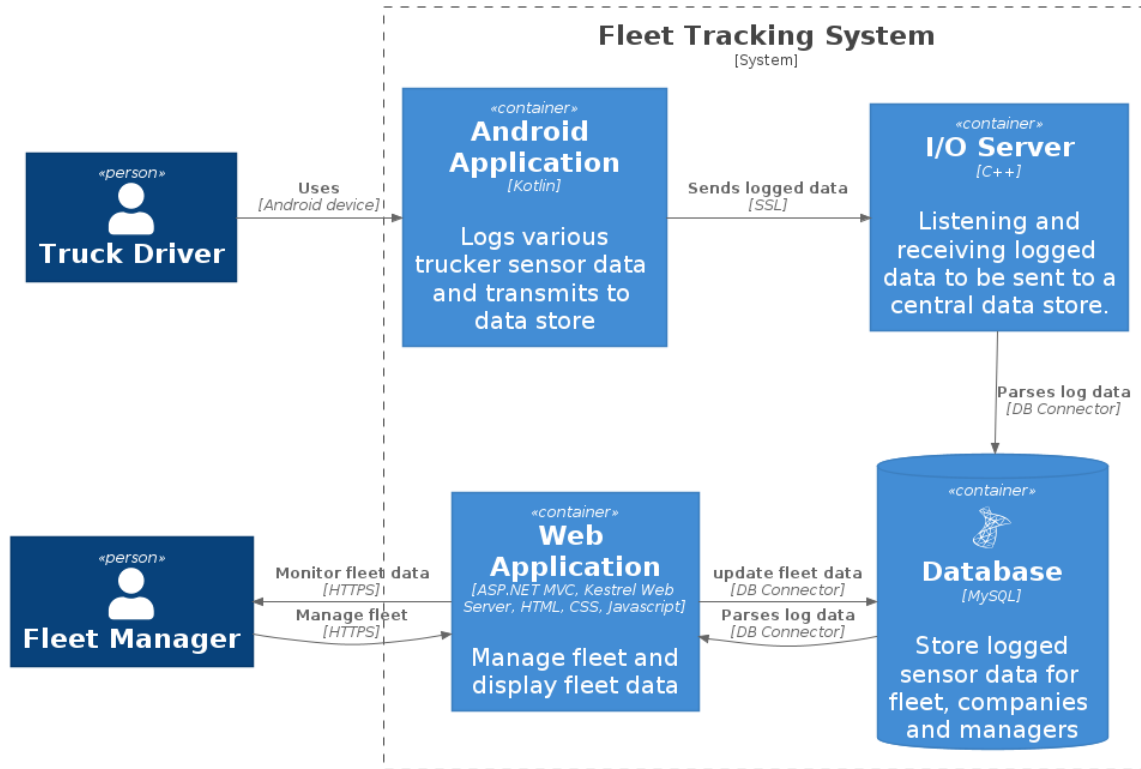
Fig. 6: Container Diagram - Fleet tracking system

3.2.1 Data Model: The entire system revolves around the effective abstraction and manipulation of logged fleet data. MySQL is chosen as the RDBMS to realize a relational database model, as it is high performance and reliable. Other RDBMSs (such as Microsoft SQL Server, PostgreSQL) offer comparable performance, but MySQL is chosen for familiarity.

The relational model is depicted in figure 13. The model is designed to allow one company to have many managers and truckers. Each trucker can have many logs.

3.2.2 Android Application: Kotlin is the language of choice to write the android application due to its simplicity and mainstream Google support.

Truckers must receive an initial code from their managers' to register their devices. Sensor readings are taken every two minutes, and stored into a lightweight database. Finally, a connection is attempted with an I/O server. If available, the database contents is emptied into via the I/O server to the central system database.

3.2.3 I/O Server: C++ is chosen for the I/O server, due to its high performance capabilities. The I/O server needs to listen and allow multiple asynchronous connections, during which log data is transmitted to the database.

3.2.4 Web Application: The MVC architecture will be realized with the Microsoft ASP.NET framework. This architecture allows for separation between business-logic, data models and viewing logic. This is necessary to ensure that code related to displaying data is not mixed with code used for core logic, thereby separating and modularizing the functionality of different components in the system.

3.3 Subsystem components and Design

Each container in figure 6 is subdivided into several core software components necessary to achieve the desired outcomes. This is depicted through container diagrams, which makes up the third level of the C4 model.
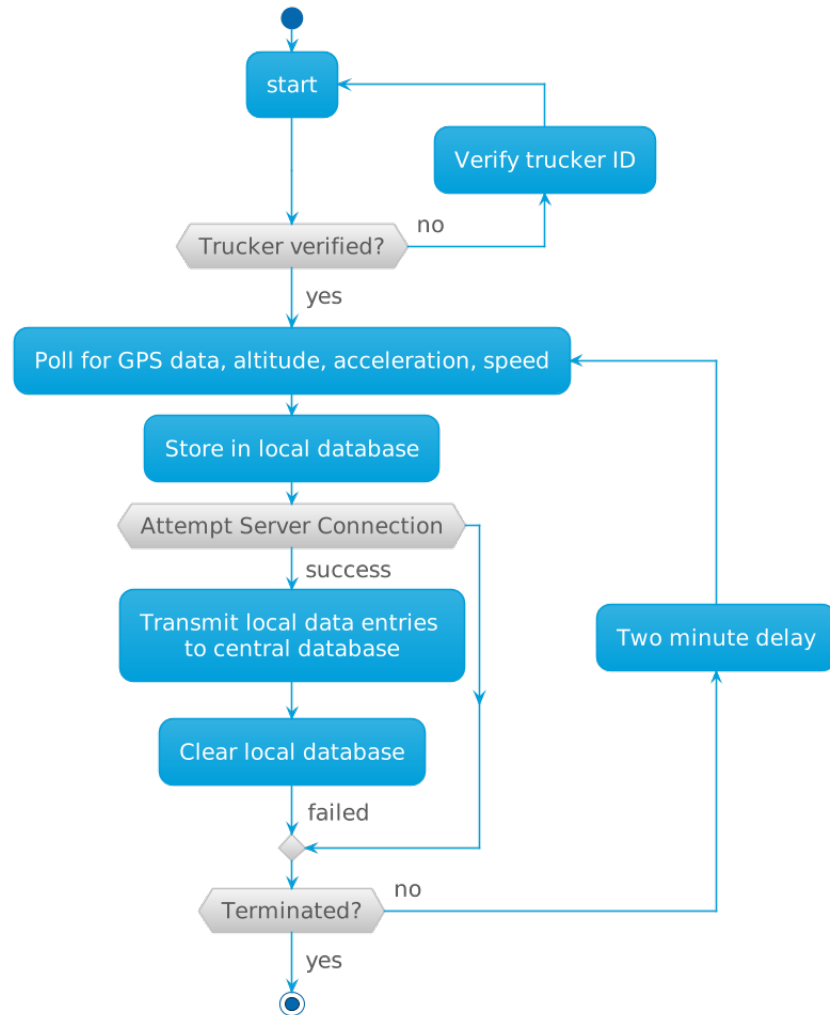

Fig. 7: Life-cycle - Android Application

3.3.1 Android Application - lifecycle and software abstractions: The life cycle of the Android application is depicted in figure 7. Initially, a check is performed to confirm that the trucker Identity (ID) is in the central database, and is not duplicated. If this ID is not valid, the trucker must request a valid ID from the fleet manager.

After this, the usual logging process is continued. Data is polled from the available sensors and stored in a
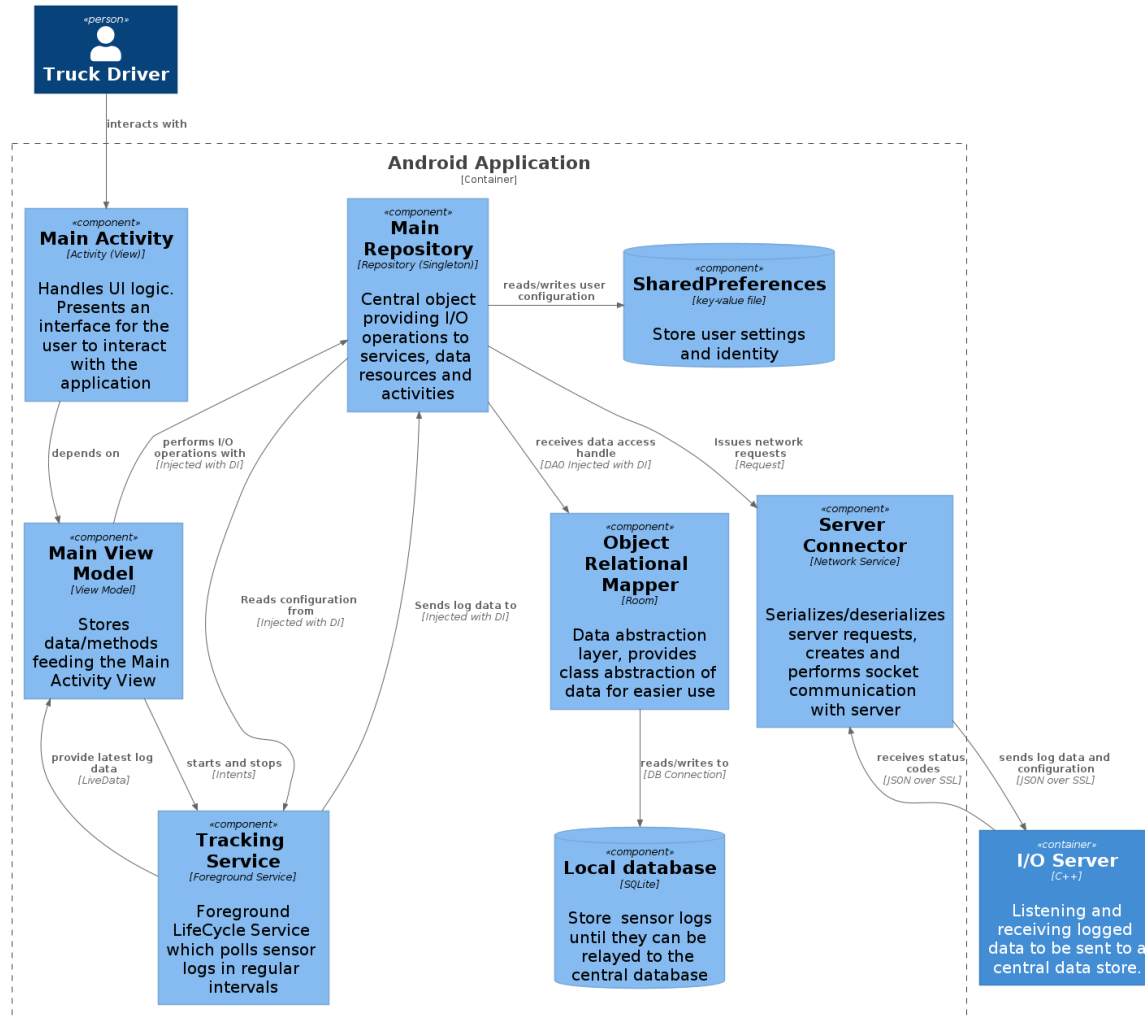
Fig. 8: Component Diagram - Android Application

local database. A connection is attempted with the I/O server and the local database entries are transmitted to the server. Upon successful transmission, the local database is cleared. However, if a connection fails, the local database is not cleared. This process loops continuously loops every two minutes.

The android application is designed with the MVVM design pattern (detailed in figure 2). Figure 8 details software components (classes) that are clearly represented in the source code. The application is targeted to support Android version 4.4 onward. This allows for the application to run on 99.6% of Android devices.

1) **Main Activity**

The main activity renders the application's main user interface to the user. This component mainly

implements User Interface (UI)-handling logic, with callbacks which are primarily event-driven (when users press buttons for example).

2) **Main View Model**

Activities have short lifetimes and are often recreated when users switch between applications or tilt their screens. Due to this, a manager class is necessary to ensure data is persisted - this is achieved by the view model.

3) **Tracking Service**

The tracking service is toggleable service which runs in the background as a foreground service. It polls acceleration and location data via interfaces made available in the Android Application Programming Interface (API). It runs without requiring the main activity to be open on the user's screen.

a) **Fused Location Provider API**

The Android API provides the *Fused Location Provider* for the purposes of accessing location data according to required settings. The API provides callbacks which can be hooked into for storing location data, at an adjustable interval.

b) **Sensor Manager**

The sensor manager provides callbacks for reading data from the various sensors (including accelerometers). The *Linear accelerometer* is a "composite" sensor which relies on magnetometers or gyroscropes, in addition to the accelerometer to "zero" out the acceleration due to gravity. This is provided by the Android API. Devices without Linear acceleromters require signal processing to remove the gravitational component. However, this processing is very limited in accuracy.

Acceleration will only be logged in devices with linear accelerometers.

4) **Main Repository**

Multiple components require performing I/O operations. To avoid repetition and prevent conflicts, the main repository performs these operations. It exists as a singleton and is injected into calling objects with DI.

5) **Room - Object Relational Mapper**

Android's room abstraction layer provides a data class abstraction of data stored in the SQLite database. This abstraction makes it easier to work with data in language-specific data structures. Room provides a Data Access Object (DAO) to the main repository for database operations.

6) **SQLite database**

SQLite is a lightweight go-to database provider for Android applications. It is ideal for storing medium to small sized volumes of data.

7) **Server Connector**

The server connector provides SSL socket communication with the central I/O server. Request objects are serialized into text data streams for transmission. Likewise, server responses are deserialized into response objects and handled appropriately.

8) **Shared Preferences**

Android's *SharedPreferences* library provides an API for the purposes of reading/writing key value data in a file on disk. This is used for storing user configuration, such as identity and upload preferences, which aren't appropriate for a database.

These components are necessary for realizing a modular, extendable application.
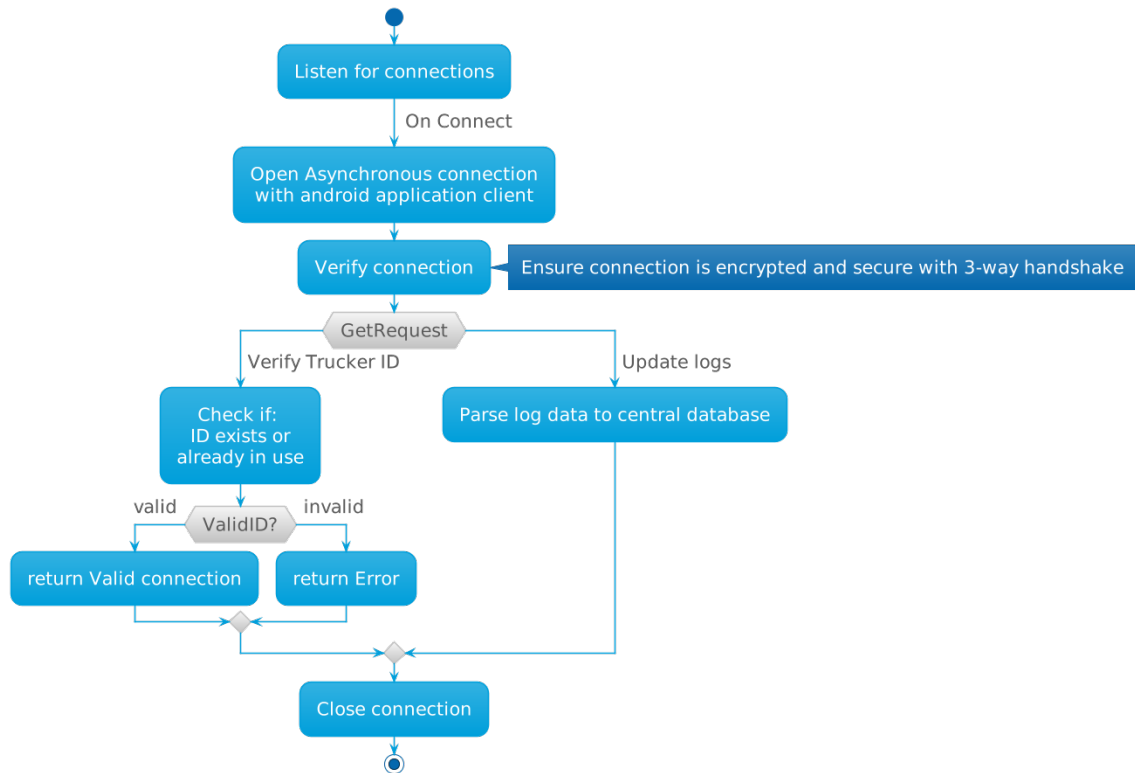
Fig. 9: Life cycle - I/O Server

3.3.2 I/O Server: The typical life cycle view of the I/O server is depicted in figure 9. A secure connection must be made due to the sensitive nature of GPS data.

- A session is assigned for the lifetime of the communication, which handles the three-way SSL handshake, ensuring the client trusts the server. The incoming payload is decrypted.
- A request handler parses (and deserializes) the decrypted payload, which queries the database to generate an appropriate serialized response.
- The response is sent back to the client and the session is terminated.

The popular C++ library, *asio* can implement the above workflow in an asynchronous manner.
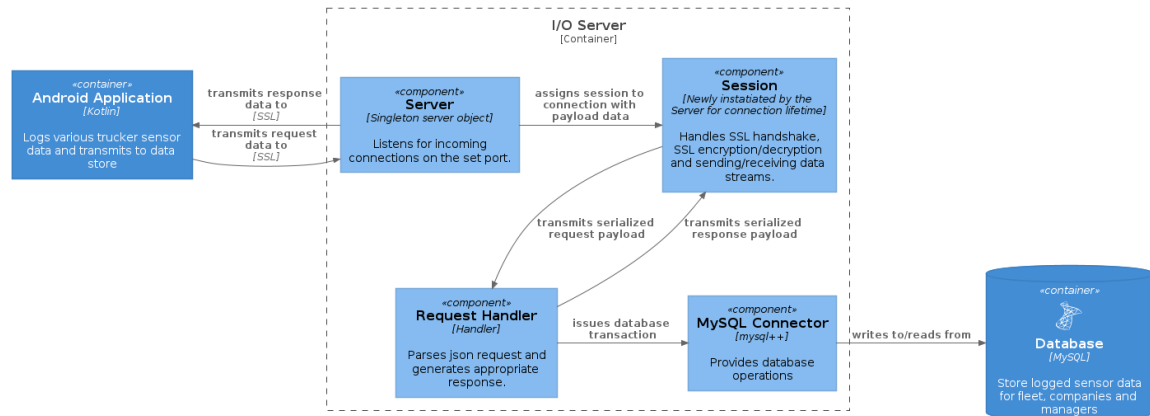
Fig. 10: Component Diagram - I/O Server

Figure 10 depicts the software abstractions and program structure used to realize the I/O server. The codebase clearly contains these low-level abstractions.

1) **Server**

   The server object listens for incoming Transfer Control Protocol (TCP) connections. Upon receiving a connection, a new session is instantiated to handle to communication.

2) **Session**

   The session performs the necessary encryption, decryption and three-way handshake required for the SSL protocol. The session reads in and writes out the serialized payload on the socket.

3) **Request Handler**

   The request handler performs serialization and deserialization. It processes the request and queries the database appropriately.

4) **MySQL Connector**

   An interfacing object to the MySQL database.

3.3.3 JSON protocol: Figure 11 depicts the structure of the protocol used for communication between the I/O server and the android client. The communication follows a Representational State Transfer (REST)ful structure, which is common for web communications. That is, communication requires no knowledge of intermediate state. One request is enough to complete the required transactions, after which an appropriate response is sent back to the client.

acc	0.18192993
alt	1376.5499
id	23312
lat	-26.70164
lon	27.086374
spd	0.09585135
tim	1633541355

data	
id	1
req	3
uuid	4fa0084c-fa7a-40a9-8f02-60e636766c90

acc	0.07141463
alt	1376.4137
id	23325
lat	-26.701616
lon	27.086409
spd	0.4677902
tim	1633541478

manager	adam@test.com
res	1
trucker	Adam Coetzee
veh	A_DAMN

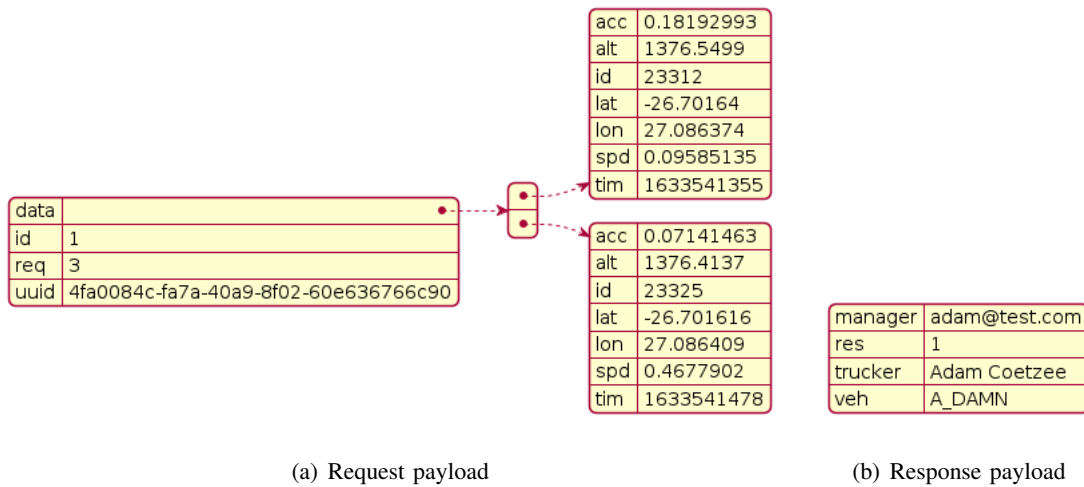(a) Request payload   (b) Response payload

Fig. 11: JSON protocol

- The client makes a request in JSON form, as shown in figure 11(a). It contains ID information about the client, a request code and any payload data (typically tracking data). Possible requests include verifying ID, sending log updates and registering new IDs.
- The server appropriately handles the request (based on request code) and generates an appropriate response. Usually the response will just contain the response code, but it may sometimes carry extra information (as seen in figure 11(b)). Responses can return a fail, ok, invalid credential, database connection error or parsing error status.

This communication is realized through SSL sockets over the network.

3.3.4 Web Application: The web application is modeled with the MVC design pattern, which allows for following SoC principles. Backend logic is realized in C# using Microsoft's *ASP.NET* framework. Web pages are generated with a combination of C#, JavaScript and HyperText Markup Language (HTML) styled with Cascading Style Sheets (CSS).
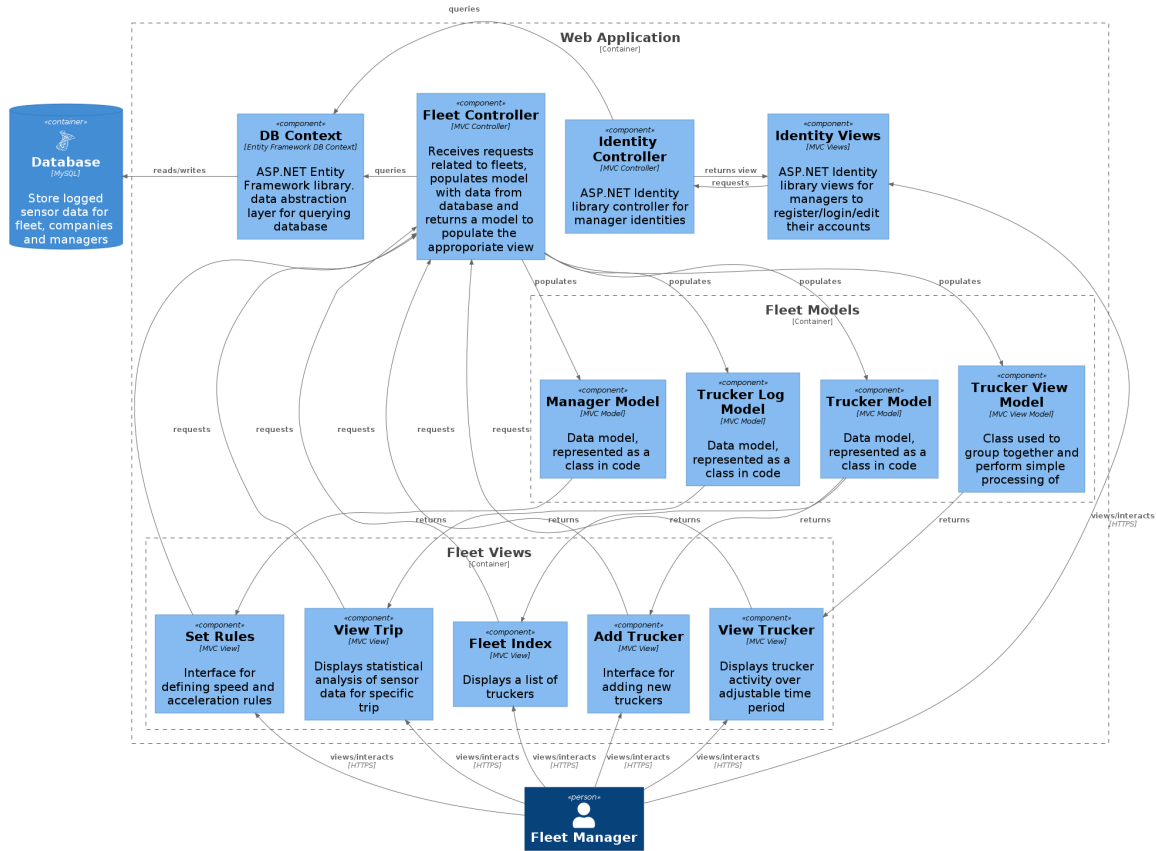

Fig. 12: Component Diagram - Web Application

Figure 12 depicts the architectural arrangement of the web application.

1) **ASP.NET Identity Views and Controller**

   Microsoft provides a professional library for handling user access, known as *ASP.NET Identity*. This library handles logic for user registration, signing in and account editing. In addition, there is support for email verification and two-factor authentication. Identity also implements logic for restricting access to pages, ensuring managers can only view data related to their fleets'.

2) **DB Context and MySQL database**

   Microsoft's *Entity Framework* provides the abstraction of data as 'entites' to be represented by models (data classes in code). This makes for easier interaction with data in code. The back-end

database in use is MySQL.

3) **Fleet Models**

The Fleet Models are a set of data classes used for the abstraction of data entities in code. They allow for the easy passing of data from controllers to views. An extra view model class is used for viewing truckers. Since viewing truckers requires extra processing of trucker information logs, an extra class is utilized to handle this processing.

4) **Fleet Controller**

User interactions from any of the fleet views results in HTTP requests issued to the Fleet Controller. The Controller has multiple methods for handling different HTTP requests. Upon receiving a routed request, it selects the appropriate method and queries the appropriate fleet data from the database abstraction layer. The results of the query are populated into one of the models, and the model is returned to the view.

5) **Fleet Views**

Multiple views are returned by the Fleet Controller depending on request, and act as the UI component visible to the user. Each view only handles the necessary logic required for displaying data returned as a model from the controller. The UI is rendered as HTML, with backend C# logic utilized to dynamically render view components, such as tables. Additional UI logic is realized with JavaScript.

a) **Index**

The Index view displays a list of all truckers registered by the manager. It provides an interface for resetting each trucker's Android ID.

b) **Add Trucker**

The "Add Trucker" page provides an HTML form for the purposes of adding trucker's to the fleet. Details about the trucker such as name and vehicle number can be added.

c) **Set Rules**

The "Set Rules" page allows the manager to define custom rules defining unacceptable driving behavior, including maximum speed and acceleration.

d) **View Trucker**

The "View Trucker" page displays details about the trucker's activities for an adjustable time period. A table is used to group location data into individual trips. This table is generated by means of a grouping algorithm to segment the driver's activities into separate trips, and includes information such as waiting time, average speed and indications of any rule breaks.

In addition, each trip is drawn in a map using the *Google Maps JavaScript API*. This provides a neat visual representation of Trucker activities.

e) **View Trip**

The "View Trip" page allows for detailed statistical analysis of individual trips. It provides percentile analysis and graphs showcasing the trucker's speed, acceleration and altitude.

3.3.5 MySQL Database and Entity relationships: The central backend RDBMS used is MySQL. A relational data structure is utilized, as shown in figure 13. Relational modeling allows for logical structuring and integrity of the data.
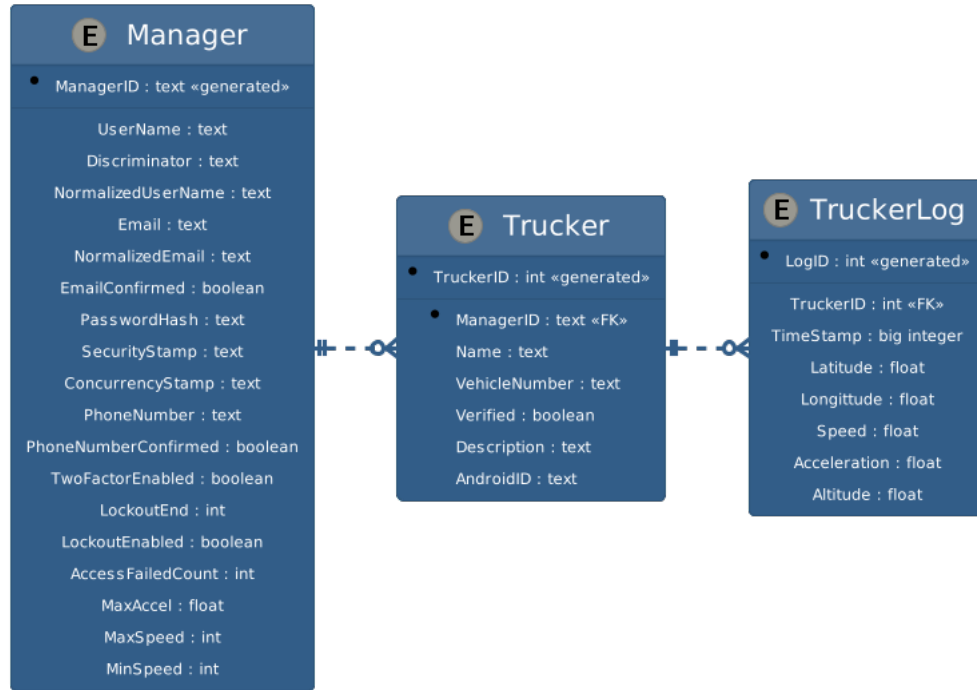
Fig. 13: Fleet Tracking System - Entity Relationship Diagram

The Manager entity represents the web application user, and contains various fields used for storing the manager's identity. In addition, the fields "MaxAccel", "MaxSpeed" and "MinSpeed" define rules for good trucker behavior.

One manager can manage multiple truckers (or none), therefore enforcing a zero-to-many relationship. Similarly, each trucker can have multiple (or no) logs. The Trucker entity stores information about each trucker. The TruckerLog entity stores the entries of each log in the database.

Unix timestamps are used for identifying the time of each log, which is convenient and saves on storage, as only 8 bytes are required. Additionally, single precision float precision provides location precision within 2.37 meters[33] in the worst case, making it adequate it for this application, while saving on storage. Double precision is more computationally expensive for little benefit.

3.4 Data Processing

The main focus and purpose of the system is to generate useful information for managers which can be used to optimize their fleets. The raw tracking data alone doesn't give the clearest indication of trucking behavior.

3.4.1 Aggregating nearby logs: Determining when trucker's have stopped is useful for segmenting trips. Grouping trips into different segments gives a clear idea of what truckers are doing. To this end, an algorithm is designed with this goal in mind.

It is first helpful to remove logs where an insignificant distance is traveled, or where the user is stationary. Algorithm 1 achieves this, by creating a new list where the distance between each log is some *MINDISTANCE* away from the previous log. A threshold of 150 meters is chosen.

> **input** : List of chronologically Sorted Logs
>
> **output:** Aggregated list of logs, where each log is a significant distance away from the next
>
> *Generate list of aggregated logs far enough away from each other*;
> $AggregatedLogs.Append \leftarrow LogsSortedByDate[0]$;
> $LastLog \leftarrow LogsSortedByDate[0]$;
> **for** $i \leftarrow 0$ **to** $LogsSortedByDate.Length$ **do**
> > **if** $Distance(LogsSortedByDate[i], LastLog) \geq MINDISTANCE$ **then**
> > > $AggregatedLogs.Append \leftarrow LogsSortedByDate[i]$;
> > > $LastLog \leftarrow LogsSortedByDate[i]$;
> >
> > **end**
>
> **end**

Algorithm 1: Aggregating logs close to each other

3.4.2 Defining trips between stop locations: The aggregated list of logs determined in algorithm 1 is then used to group together trips. A trip is defined as the logs between consecutive stops. A stop is defined using the time difference between two distance-aggregated logs, where the time between each log is greater than some threshold (*MINTIME*). A value of 5 minutes is chosen to designate a stop. Algorithm 2 shows the algorithm used to determine this.

input : List of chronologically, aggregated Logs, where consecutive logs are a minimum distance
away from each other

output: List of trips(segments) defined from some start log to some end log

Use aggregated list to determine individual trips separated by stopping points;

$startLogCount \leftarrow 0$;   /* The start of a trip */

for $i \leftarrow 0$ **to** $AggregatedLogs.Length - 1$ **do**

    if $TimeDifference(AggregatedLogs[i+1], AggregatedLogs[i]) \geq MINTIME$ **then**

        if $i == 0$ **then**

            If the first log is a stop, don't start the trip from here

            $startLogCount \leftarrow startLogCount + 1$;

        else

            Add a trip, starting from the previous start log till the current log;

            $Segments.Append \leftarrow AggregatedLogs[startLogCount], AggregatedLogs[i]$;

            the next trip will start from the next aggregated log;

            $startLogCount \leftarrow i + 1$;

        end

    end

end

Algorithm 2: Grouping logs into trips

The *Segments* determined in algorithm 2 can be tabulated to give details of each trip the trucker performed.

3.5 User Interface (UI) Design

The primary goal of UI design is to make the interface clear and intuitive for users.

3.5.1 Android Application: Figure 14 depicts the blueprint of Android application's UI. XML is used in Android to define the positioning of elements in the layout.

Fig. 14: Android Application - UI

To account for different screen sizes, a Constraint layout is used, in which elements are tied to edges or specific points in the layout's geometry. Elements are places relative to each other, and will dynamically adjust when tilting the screen. The constraint layout is placed within a scroll view to allow scrolling if the elements can not fit on the screen.

The UI allows the user to toggle the tracking service. It provides information about the trucker and his/her manager. An interface is provided for toggling upload frequency of logs. Finally, the current sensor readings are displayed.
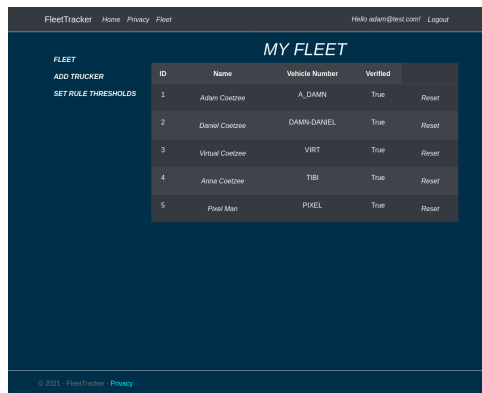
(a) Fleet index page
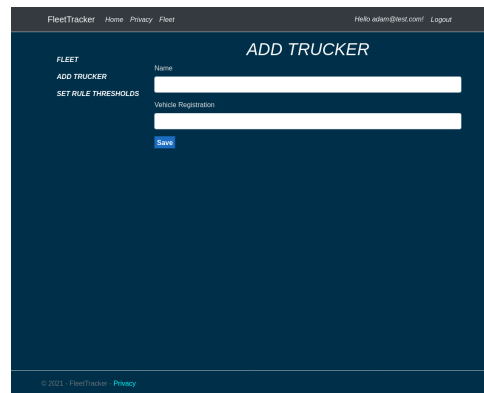
(b) Add Trucker
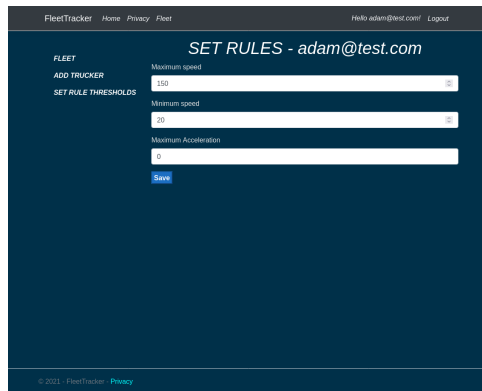
(c) Set rules
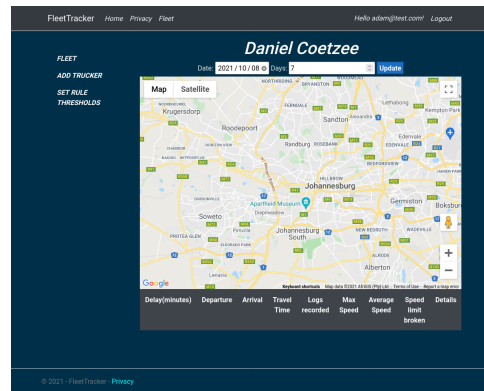
(d) View Trucker
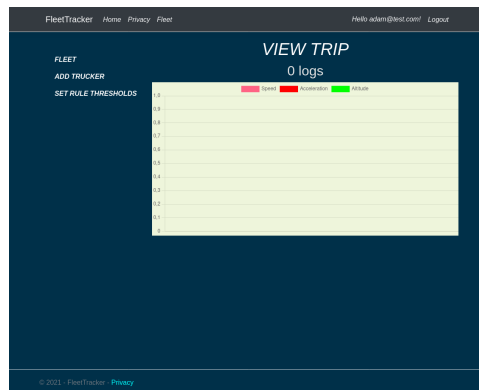
(e) View Trip

Fig. 15: Web application - Pages

3.5.2 Web Application: Figure 15 depicts the main pages used by managers for manipulating and viewing their fleets. The web pages are designed using HTML elements, which are style in CSS aided by the *Bootstrap* library which provides elegant CSS presets.

HTML elements are spaced using *div* containers in a grid layout.

- The index page, seen in figure 15(a) displays a list of truckers in the managers fleet. Managers can reset trucker IDs and navigate to individual trucker pages.
- The "Add Trucker" page, in figure 15(b) allows managers to add new trucker entries.
- The "Set Rules" page in figure 15(c) allows managers to set rule thresholds for defining good behavior.
- The "View Trucker" page, in figure 15(d) displays a map and table for displaying information about trips.
- The "View Trip" page, in figure 15(e) contains a graph for viewing statistics for trips.

4 IMPLEMENTATION

This section considers the implementation and realization of the design. Upon completion, the system is deployed to a Virtual Private Server (VPS), allowing for the service to be accessed on the internet.

4.1 Android Application

Figure 16 depicts the functionality and layout of the implemented android application. Screenshots are taken of the application running in an Android emulator.

(a) Idle state   (b) Running state

(c) Background operation   (d) Stashing logs locally
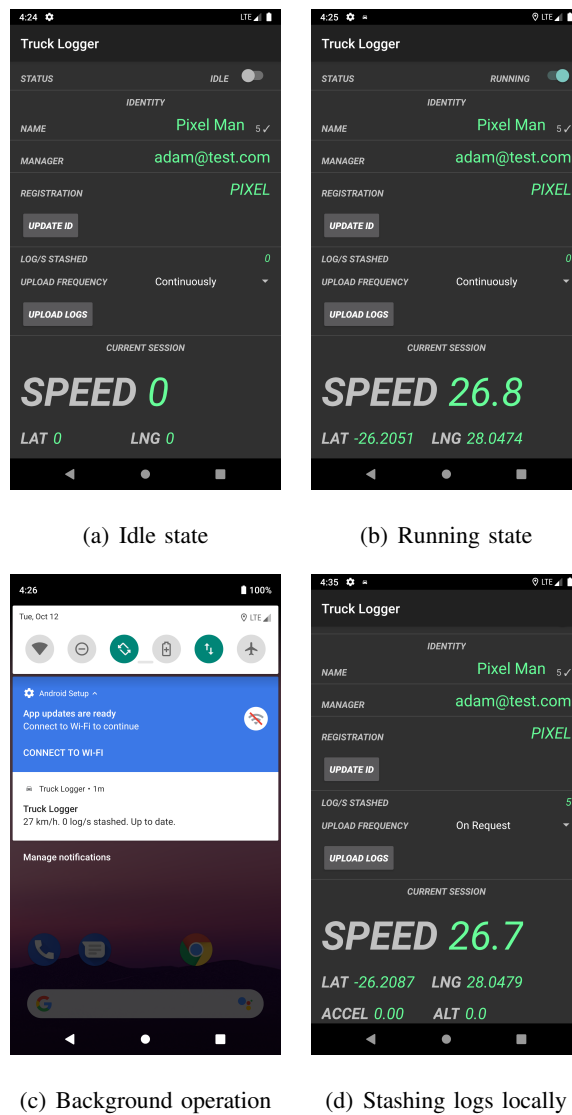
Fig. 16: Android application - Implemented layout

The UI provides an interface detailing ID information related to the trucker and manager. Users can update their ID as long as they have connectivity to the server. They can also upload all log data at any instance.

Figure 16(a) depicts the application in an idle state. The application performs no logging in this state.

Toggling the status check box allows the application to start logging data, as seen in figure 16(b). This activates the background process which polls for GPS, acceleration (provided the linear composite accelerometer if available) and altitude. While tracking, a constant notification is displayed, indicating speed and number of logs stashed.

Depending on the upload frequency, an attempt is made to upload logs to the central server. Otherwise logs are stashed in the SQLite database.

4.2 I/O Server

Figure 17 depicts the output of the I/O server, logging request information to standard output.

Fig. 17: IO Server - Request information logged to standard output

The I/O server handles SSL connections transmitting a serialized JSON payload consisting of the truckers ID, Universally Unique Identifier (UUID) and extra data (detailed in figure 11). The UUID is added to ensure that one trucker can be associated with a device. Depending on the request code provided, the server processes each request appropriately.

4.2.1 Requests: The I/O server handles various requests. Each request type is specified by a request code, as shown in figure 11.

1) **UPDATE ID**

   The trucker sends a request to update their ID with a newly generated UUID. As long as the *Verified* flag is set to false, the server will update its record of the UUID corresponding to that trucker ID. The manager must first set the flag to false if they need to reset the trucker's device or change the identity associated with a specific trucker. The *Verified* flag is then set to true. All future requests must provide the UUID.

2) **VERIFY ID**

   This request ensures that a truckers ID and UUID correspond in the database. If not, the server returns a *INVALID CREDENTIALS* response code. This mechanism ensures that only one device can send logs for a corresponding trucker ID.

3) **UPDATE LOGS**

   This request first performs logic for verifying IDs, associated with the *VERIFY ID* request. If the incoming ID is valid, the log records for the specific device are added to the database.

4.2.2 Responses: The server responds with an appropriate response code, to the android requests.

1) **FAIL**

   A fail code indicating the response was invalid.

2) **OK**

   A success code indicating the response was valid.

3) **INVALID CREDENTIALS**

   A fail code indicating that the client's ID and UUID do not correspond.

4) **DB CONN FAILED**

   A server error triggered by an exception when the server can't establish connection with the database.

5) **PARSE FAIL**

   The incoming JSON serialized payload was malformed and could not be deserialized.

4.3 Web application

The web application is implemented using the *ASP.NET Core* framework, running in Linux on the Kestrel web server.

4.3.1 User login and signup: Microsoft's *ASP.NET Core Identity* library provides an easy automated interface for handling manager identity. It provides user log in and sign up pages, as seen in figure 20.

(a) Sign up   (b) Log in

Fig. 18: Web application - Manager account handling

4.3.2 Fleet Index: The Fleet Controller is implemented in handling HTTP requests for serving web pages related to the manager's Fleet. Fleet Controller methods are called when the Uniform Resource Identifier (URI) in the address bar is appended with the text *Fleet*.

Fig. 19: Web application - Fleet Index

The Fleet Index page shown in figure 19 displays a list of all truckers in the manager's fleet. The *Verified* attribute indicates whether a trucker is paired to a specific device. This attribute can be reset to allow the pairing process to be performed again for the same or a new device.

(a) Add Trucker

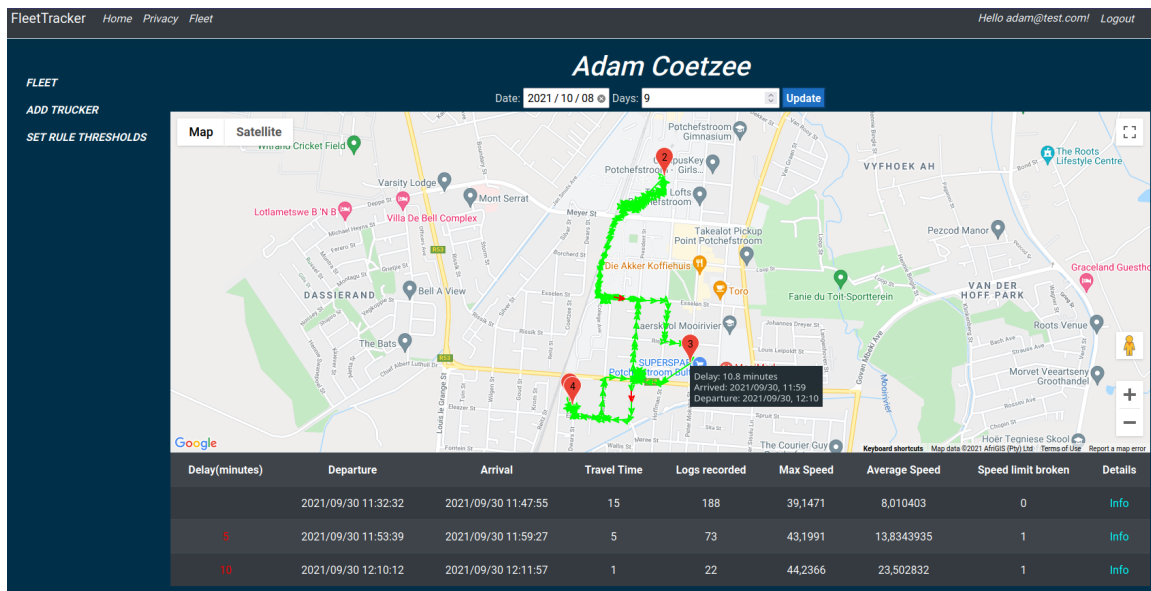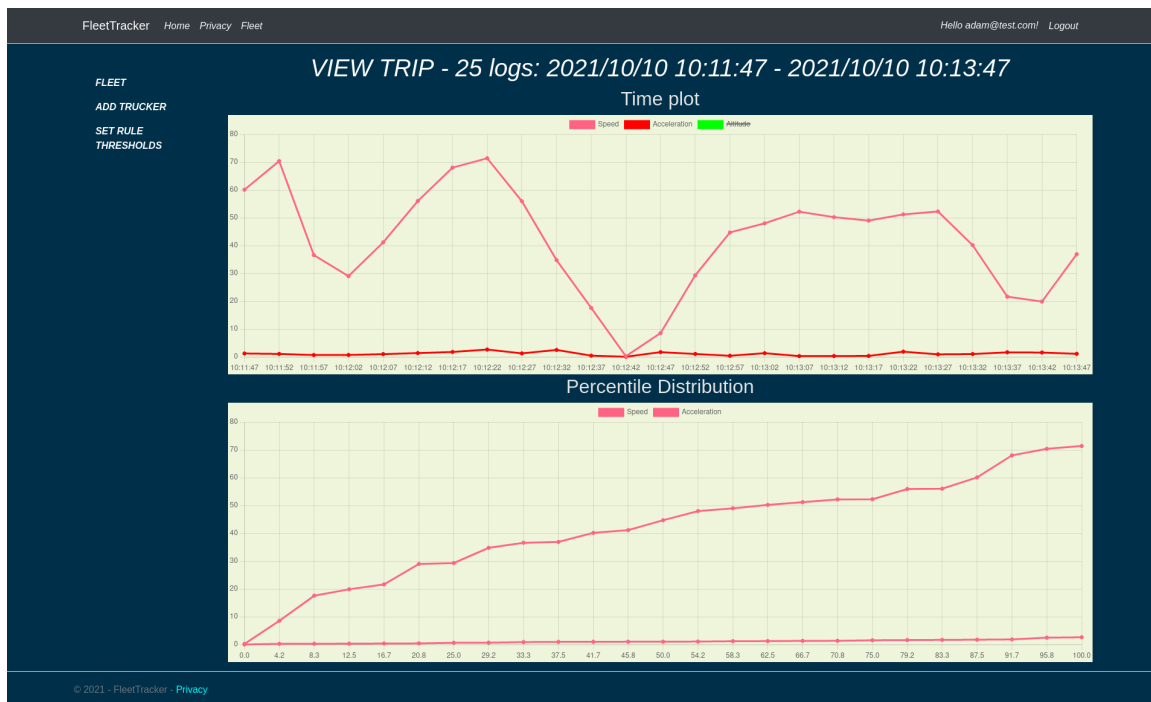(b) Set rules

Fig. 20: Web application - Fleet management

Figure 20 depicts pages implemented for adding truckers and setting rule thresholds. They both make

use of HTML forms bound to the Manager and Trucker model. HTTP requests are routed to the Fleet Controller and handled by an appropriate method.

(a) View Trucker

(b) View Trip

Fig. 21: Web application - Trip collection and information

Figure 21 details the interface for viewing trucker activity.

Figure 21(a) provides a time adjustable page which groups the logs into individual trips. A map is rendered (using the Google JavaScript API) with routes physically drawn in the map, using arrows. If the defined speed limit is broken, the arrows are drawn in red. Stopping points are indicated with markers labeled in chronological order. Hovering over a stop label indicates how long the trucker had stopped at a given location. A table is rendered displaying this information.

Figure 21(b) allows a more detailed view of an individual trip between two markers. Time plots indicate give speed, acceleration and altitude plots which vary with time as truckers carry out their trips. A percentile plot is included to visualize the spread of speed and acceleration. This allows managers to visualize what portions of the trip was driven with specific behavior.

4.4 Deployment

The web application and I/O server are deployed to a Linux VPS with access to a static internet protocol (IP) address and domain name. A non-profit Certificate Authority (CA) Lets Encrypt provides SSL certificates, which are required for Android applications which make use SSL communication.

Docker containers are created and used for used for deploying the MySQL database and I/O server. They are useful for managing dependencies and preventing unwanted changes to the host server.

A link is provided for downloading the Android application as an Android Package (APK) file.

5 EVALUATION

Evaluation of the system is performed by testing the deployed I/O server and web application with several Android devices which are available.

5.1 Android application

Four Android devices were readily available for testing purposes, which are highlighted in table II.

TABLE II: Android devices tested

Android Device	Price[R]	Major Android Version	Linear Accelerometer
Huawei P8 Lite 2017	R2900	8	Yes
Samsung Galaxy A01	R2180	10	No
Galaxy A3 Core	R1400	10	No
Huawei Nova 5T	R7700	10	Yes

5.1.1 Log accuracy:

5.1.2 Application reliability:

<div align="center">REFERENCES</div>

[1] M. Bertocco, F. Ferraris, C. Offelli, and M. Parvis, "A client-server architecture for distributed measurement systems," *IEEE transactions on instrumentation and measurement*, vol. 47, no. 5, pp. 1143–1148, 1998.

[2] S. Majumder and M. J. Deen, "Smartphone sensors for health monitoring and diagnosis," *Sensors*, vol. 19, no. 9, p. 2164, 2019.

[3] F. Li, H. Zhang, H. Che, and X. Qiu, "Dangerous driving behavior detection using smartphone sensors," in *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2016, pp. 1902–1907.

[4] N. De Caro, W. Colitti, K. Steenhaut, G. Mangino, and G. Reali, "Comparison of two lightweight protocols for smartphone-based sensing," in *2013 IEEE 20th Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*. IEEE, 2013, pp. 1–6.

[5] A. Van Herrewege, D. Singelee, and I. Verbauwhede, "Canauth-a simple, backward compatible broadcast authentication protocol for can bus," in *ECRYPT Workshop on Lightweight Cryptography*, vol. 2011, 2011, p. 20.

[6] C. Campolo, A. Iera, A. Molinaro, S. Y. Paratore, and G. Ruggeri, "Smartcar: An integrated smartphone-based platform to support traffic management applications," in *2012 first international workshop on vehicular traffic management for smart cities (VTM)*. IEEE, 2012, pp. 1–6.

[7] O. Walter, J. Schmalenstroeer, A. Engler, and R. Haeb-Umbach, "Smartphone-based sensor fusion for improved vehicular navigation," in *2013 10th Workshop on Positioning, Navigation and Communication (WPNC)*. IEEE, 2013, pp. 1–6.

[8] M. R. Stepper, "J1939 high speed serial communications, the next generation network for heavy duty vehicles," SAE Technical Paper, Tech. Rep., 1993.

[9] K. Gamage, "Separation of concerns for web engineering projects."

[10] E. Chebanyuk and K. Markov, "An approach to class diagrams verification according to solid design principles," in *2016 4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. IEEE, 2016, pp. 435–441.

[11] Z. A. Kocsis and J. Swan, "Dependency injection for programming by optimization," *arXiv preprint arXiv:1707.04016*, 2017.

[12] "Mobile operating system market share worldwide." [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide

[13] M. Flauzino, J. Veríssimo, R. Terra, E. Cirilo, V. H. Durelli, and R. S. Durelli, "Are you still smelling it? a comparative study between java and kotlin language," in *Proceedings of the VII Brazilian symposium on software components, architectures, and reuse*, 2018, pp. 23–32.

[14] D. Kwan, J. Yu, and B. Janakiraman, "Google's c/c++ toolchain for smart handheld devices," in *Proceedings of Technical Program of 2012 VLSI Technology, System and Application*. IEEE, 2012, pp. 1–4.

[15] A. Biørn-Hansen, C. Rieger, T.-M. Grønli, T. A. Majchrzak, and G. Ghinea, "An empirical investigation of performance overhead in cross-platform mobile development frameworks," *Empirical Software Engineering*, vol. 25, pp. 2997–3040, 2020.

[16] "Guide to app architecture." [Online]. Available: https://developer.android.com/jetpack/guide

[17] "Dependency injection with Hilt." [Online]. Available: https://developer.android.com/training/dependency-injection/hilt-android

[18] "Foreground services." [Online]. Available: https://developer.android.com/guide/components/foreground-services

[19] "Save data in a local database using Room." [Online]. Available: https://developer.android.com/training/data-storage/room

[20] J. O. Ogala and D. V. Ojie, "Comparative analysis of c, c++, c# and java programming languages," *GSJ*, vol. 8, no. 5, 2020.

[21] W. Anggoro and J. Torjo, *Boost. Asio C++ network programming*. Packt Publishing Ltd, 2015.

[22] A. Gupta, S. Tyagi, N. Panwar, S. Sachdeva, and U. Saxena, "Nosql databases: Critical analysis and comparison," in *2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN)*. IEEE, 2017, pp. 293–299.

[23] M. A. Qader, S. Cheng, and V. Hristidis, "A comparative study of secondary indexing techniques in lsm-based nosql databases," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 551–566.

[24] G. Ongo and G. P. Kusuma, "Hybrid database system of mysql and mongodb in web application development," in *2018 International Conference on Information Management and Technology (ICIMTech)*. IEEE, 2018, pp. 256–260.

[25] W. Truskowski, R. Klewek, and M. Skublewska-Paszkowska, "Comparison of mysql, mssql, postgresql, oracle databases performance, including virtualization," *Journal of Computer Sciences*

Institute, vol. 16, pp. 279–284, 2020.

[26] S. Bhosale, T. Patil, and P. Patil, "Sqlite: Light database system," *International Journal of Computer Science and Mobile Computing*, vol. 4, no. 4, pp. 882–885, 2015.

[27] K. Kronis and M. Uhanova, "Performance comparison of java ee and asp. net core technologies for web api development," *Applied Computer Systems*, vol. 23, no. 1, pp. 37–44, 2018.

[28] K. Hickman and T. Elgamal, "The ssl protocol," 1995.

[29] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," 1999.

[30] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, "Comparison of json and xml data interchange formats: a case study." *Caine*, vol. 9, pp. 157–162, 2009.

[31] A. Vázquez-Ingelmo, A. García-Holgado, and F. J. García-Peñalvo, "C4 model in a software engineering subject to ease the comprehension of uml and the software," in *2020 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, 2020, pp. 919–924.

[32] M. Petre, "Uml in practice," in *2013 35th international conference on software engineering (icse)*. IEEE, 2013, pp. 722–731.

[33] "Required Precision for GPS Calculations - TresCopter." [Online]. Available: http://www.trescopter.com/Home/concepts/required-precision-for-gps-calculations