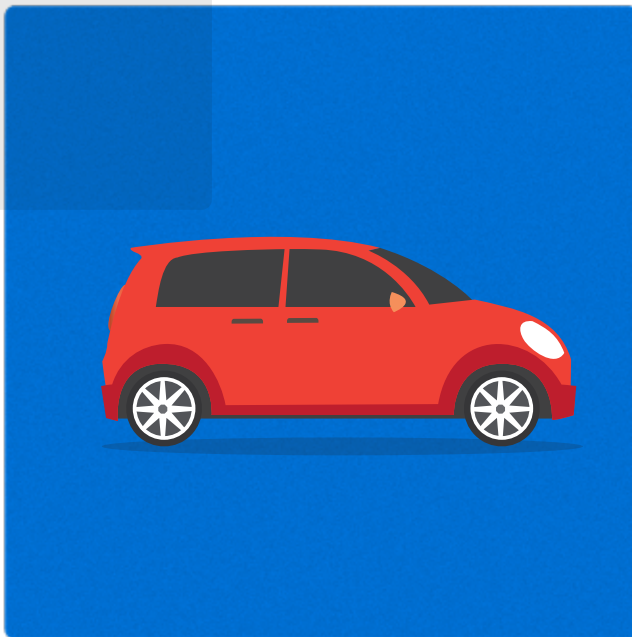


Telematics

Vehicle Data Monitoring and Intelligent Speed Assistance



4th Semester Project Group:

Lasse M. H. Traberg-Larsen

Bjarke Tobiesen

Dennis T. Petersen

Rickie Andersen

Michael N. H. Ho

English title: Vehicle Data Monitoring and Intelligent Speed Assistant

Module: I-ITI4

Educational programme: BEng in Software Technology

Educational institution: University of Southern Denmark

Institute: The Maersk Mc-Kinney Moller Institute

Project period: Spring semester 2019

Authors:

Lasse Marco Holst Traberg-Larsen, *455855*

Bjarke Tobiesen, *455142*

Rickie Mark Hedengran Andersen, *455549*

Michael N. H. Ho, *438829*

Dennis Tandrup Petersen, *113352371*

Supervisor:

Anooshmita Das, adas@mmmi.sdu.dk

Glossary

The table below acts as a brief dictionary of terms with explanations relating to the context of this report. Use the list of words as a tool to understand the meaning of selected terms and how they should be understood.

Table 2: Report Glossary

Term	Description
API:	Application Programming Interface is an interface that provides services for other programs. Can be HTTP based or a language library.
Arduino:	Arduino is an open software electronics platform and Arduino Board is a small electronic hardware with a microchip that can read input and write output. Arduino Board refers to Arduino.
CRUD:	Stands for Create, Read, Update, and Delete. Refers to the basic functions of persistent storage that are applied to data, for example in a database.
IoT:	Acronym for Internet of Things. The concept of internet connectivity and communication between physical and everyday objects and a important modern technology for the current trend of automation and data exchange in Industry 4.0
ISA:	Acronym for Intelligent Speed Assistance. Vehicle safety technology proposed by the European Commission that uses GPS-linked speed limit data to help drivers keep to the current speed limit. The technology developed in this project.
JSON:	Acronym for JavaScript Object Notation. A lightweight data format humans can read and write. It is easy for machines to parse and generate JSON. Additionally, it is a subset of JavaScript.
UUID:	Acronym for Universally Unique Identifier. It is a 128-bit number used as identification similar to entries or social security numbers. Sometimes it is called GUID (Globally Unique Identifier).

Abstract

The essence of this report is to document the process, from the analysis to design to the development, of the fourth-semester's semester project module I-ITI4. The objective is to develop a web based user interface to an industrial process. The development of the project has happened alongside other courses which contributes to the crucial expertise and finesse for this project. The purpose of the system is to collect vehicle data, so the system use the velocity of the vehicle, location of the vehicle, and the speed limit of the road, to enhance the road safety, by alerting the user of the system when they are speeding.

The system is built with a software solution combined with pieces of hardware. The main hardware is an Arduino that contains a GPS component that calculates the GPS coordinates and reads the vehicle data for processing at the backend server. The backend saves the data in a database for historical reasons and sends it further to the website for clients to view. The user interface must be able to update the process of the industrial process and allows the user to interact with it to change relevant parameters or control the process. The features that clients of the system can make use of are registering, login, managing vehicles, view live and archived data, perform administrative tasks, and most importantly use Intelligent Speed Assistance (ISA).

Furthermore, the project course utilises Scrum as a foundation for an agile software development that is split into multiple sprints. The development began with finding the problem domain, requirements and creating use-cases for the system to define the scope that will later lead to the design phase.

It can be concluded that it is feasible to implement such a system, and that the group has gained new knowledge of dealing with embedded hardware, advanced software architecture, web development, and how they all can work together using web based APIs and message queuing technologies.

i Preface

This report has been written in the spring semester 2019 in a project at the University of Southern Denmark under the supervision of Anooshmita Das by a group of five members studying Bachelor of Engineering in Software Technology on fourth semester. During the project, we developed a Java backend using component-based software engineering, an embedded hardware solution using microcontroller technologies, and a modern web based graphical user interface for live data visualisation and control.

The software system works as Intelligent Speed Assistance that monitors vehicle data. Synthesising sensor data from the vehicular process allows the system to update the driver of the vehicle with live data, and most importantly intuitively notify the driver if speeding occurs.

This report is a technical report that describes the process and technical aspects of the work that has been carried out, including requirements, design, architecture, and implementation of the software and hardware system.

We refer the reader to the Glossary in the very start of this report for a list of technical terms and abbreviations with explanations. Familiarity with Unified Modeling Language is favourable for understanding diagrams. Lastly, we refer the reader to section The Final Product for a presentation and demo of the final system.

Odense, Denmark, 24th May 2019



Lasse M. H. Traberg-Larsen



Bjarke Tobiesen



Rickie Andersen



Michael N. H. Ho



Dennis T. Petersen

Contents

i	Preface	i
ii	List of Tables	iv
iii	List of Figures	iv
iv	Reading Guide	v
1	Introduction	1
1.1	Problem Analysis	2
1.2	Problem Statement	2
2	Methods & Approach	3
2.1	Project Methodology	3
3	Research and Initial Studies	4
3.1	Technology Analysis	4
3.1.1	On-Board Diagnostics	4
3.1.2	Controller Area Network System	5
3.1.3	Global Positioning System	7
3.1.4	The Arduino Platform	7
3.2	Technical Solution	8
4	Analysis	9
4.1	Requirement Specification	9
4.2	Use Case Diagram	10
4.3	Prioritising	11
4.4	Domain Model	12
5	Architecture	13
5.1	System Software Product Quality	13
5.2	Architectural Overview	13
5.3	Service Oriented Architecture	14
5.3.1	REST & SOAP	14
5.4	Component-based	15
6	Design	18
6.1	Backend	18
6.1.1	Repositories	18
6.1.2	Services	18
6.1.3	Controllers	18
6.1.4	Providers	19
6.1.5	Speed Assistant	19
6.2	Simulation	20
6.3	Hardware	20
6.4	Database	20

7	Implementation	22
7.1	Client Tier	22
7.1.1	Angular Framework	22
7.2	Application Tier	23
7.2.1	Server	23
7.2.2	Arduino	28
7.3	Data tier	29
7.3.1	Database	29
7.4	Docker	31
7.4.1	MQTT	34
8	Test	36
8.1	Unit testing	36
8.1.1	Mocking	37
8.1.2	Example of unit test	37
8.2	Integration testing	38
8.2.1	Examples of integration testing	38
8.3	System testing	39
8.4	Passing tests in the project	39
9	Discussion	40
9.1	The Process	40
9.2	Technology	41
9.3	Requirements	41
9.4	Known Limitations & Proposed Solutions	42
9.5	Future Improvements	42
9.6	Summary	42
10	Conclusion	43
11	References	44
12	The Final Product	46
12.1	Source code	48
	Appendices	I
A	Internal Appendices	I
A.1	Project Plan	I

ii List of Tables

3	Prioritised system requirements	9
4	Prioritised use cases	11

iii List of Figures

1	Simplified concept of IoT and distributed hardware.	1
2	Intelligent Speed Assistance (ISA) explanation[9].	2
3	OBD-II diagram.	4
4	OBD-II message structure[7].	5
5	The layered ISO 11898 CAN standard architecture[22].	6
6	Bit stuffing for synchronisation[22].	6
7	CAN bus arbitration mechanism[22].	6
8	The underlying idea of positioning via Global Positioning System[11, p. 9].	7
9	The Arduino Mega 2560[2].	7
10	Illustration of the proposed solution and employed technology.	8
11	The scope of the software system system (Use Case Diagram).	10
12	Extract of the Backlog and the decomposition of requirements into classified tasks.	11
13	Domain for the system.	12
14	Deployment diagram	13
15	SOA Model. Clients connect to a server via the internet.	14
16	The flow for the HTTP calls for webserver, backend server and REST.	15
17	Monolithic (left) and Component-based (right)	16
18	Excerpt of full design class diagram for the backend	19
19	E/R diagram of database	21
20	The Angular Structure in a nested folder format. The placeholder <i>[module]</i> can be any module in angular and [pageX] can be any page in module.	23
21	Whiteboard Component Model (top) & Inversion of Control (bottom)	24
22	All tests	39
23	Desktop view of ISA	46
24	Desktop view of the user page	46

iv Reading Guide

The primary material of this report is organised into eight sections. Following the introductory part, section 2 recaps key questions the group set out to answer and describes the means by which the group will come to its conclusion in the form of rationale for the group's designed methodology.

Section 3 features technology analysis of domain specific technologies and presents the suggested technical solution to solve the problem.

Section 4 documents the analysis work and presents requirements for the software system, including prioritising. This section also documents the system scope and boundary via UML artifacts such as use cases and domain modeling.

Section 5 dives into software architecture design choices and provides an essential architectural overview, including description of applied architectural strategies and patterns and which software quality attributes that have been achieved.

Section 6 follows up on the designed architecture and presents high-level ideas about the structure of the software as well as the design of the components comprised by the system.

Section 7 encompasses technical implementation details and the very realisation of the design process and system features, including actual code examples, technologies and protocols used, and deployment practicalities.

Section 8 documents the various kind of testing techniques and practices used in validation of the software that has been built.

Finally, section 9 discusses the project results and implementation choices and section 10 presents the conclusion.

1 Introduction

As computerisation advances in nearly every aspect of our lives, industrial organisations and corporations are using automation increasingly to boost process efficiency and minimise the use of human labour and resources. Automation includes the use of computer processing power in control systems for operating and monitoring equipment, machinery, and processes. Technological advancements in the size and processing power of embedded technology as well as improved network capabilities has allowed the extension of internet connectivity into physical devices, the concept often coined as the Internet of Things (IoT). In recent years, the traditional centralised physical infrastructure of embedded systems and control systems, such as SCADA systems, has transitioned to a modern digitised cloud approach decentralising the traditional physical constraints. The IoT approach liberates sensor data from previously centralised or even cabled boundaries and allows sensors, actuators, and hardware to be treated as independent nodes in the cloud accessible via devices in the ecosystem, such as smartphones. The concept is illustrated on Figure 1. This report investigates the use of this technology and automation, in this case not to merely minimise human labour but to potentially save human lives.

In this project, the group dives into field of vehicle telematics and vehicle diagnostics, which is the convergence of telecommunications and information processing and services. Telematics, also related to vehicular automation, is a field that embodies the very process of vehicular transportation, wireless communications, internet, sensor data, and vehicular technologies such as road safety. Synthesising information concerned with positioning of a vehicle and data from the sensors already incorporated in modern cars, known as on-board diagnostics, illuminates numerous useful applications such as smart driving assistance, fleet management, analysis of driving behaviour, vehicle health reporting, and much more. However, the motivation for this project is to develop a smart, intuitive and accessible IT system for vehicle telematics in the context of human safety. This foundation and case of this project are built around new safety standards by the European Parliament, thus the project targets the lawmakers of EU as well as stakeholders including automobile manufacturers. The EU wishes to increase road safety for its citizens - this report will examine the case's problem and provide feasible design decisions and implementation of a software system that solves the problem. The problem is elaborated in section Problem Analysis.

In short, the goals of the project encompass (1) development of a distributed field control unit and acquisition, transmission, and processing of relevant sensor data, (2) development of a web-based GUI to visualise data of the observed process, and (3) exploration of telematics and domain specific technologies such as CAN bus and OBD-II. The overall aim is to design a state-of-the-art IT system to increase road safety.

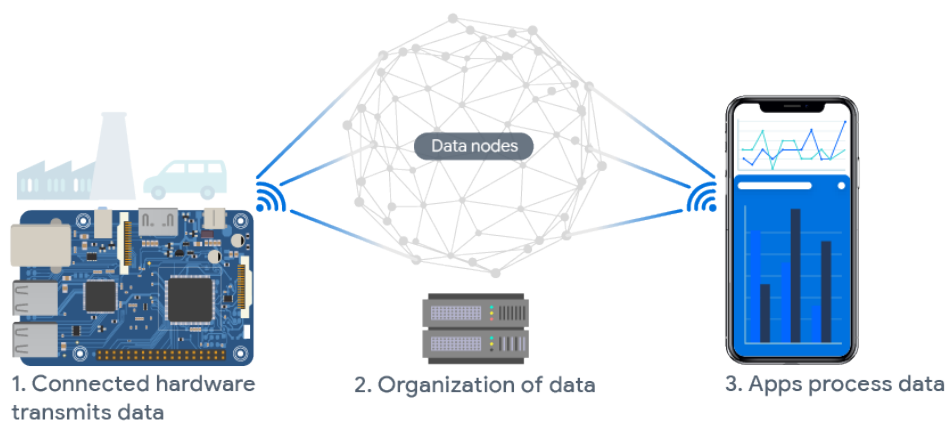


Figure 1: Simplified concept of IoT and distributed hardware.

1.1 Problem Analysis

This section is concerned with underlying problem that the group has set out to solve. According to the European Transport Safety Council, vehicle speed has steadily been recognised as one of three main contributing factors to deaths on the roads[9]. To battle this problem and save lives, a key committee of members of the European Parliament has voted to approve new vehicle safety standards proposed by the European Commission in 2018[10]. Executive Director of the European Transport Safety Council states that the safety standards proposed could reduce the number of collisions by 30% and bring down the number of fatal accidents and deaths by more than 25,000 each year during the next 15 years. The proposed mandatory safety features include Intelligent Speed Assistance (ISA), which has been found to be the most effective in saving lives by a 2014 Norwegian study[13]. In the introductory phase, a such system is supposed to monitor vehicle speed and notify the driver with beeps/flushes if the speed limit of a given road is exceeded.

The Intelligent Speed Assistance is supposed to be mandatory in all new vehicles. However, present vehicles may not have these live-saving capabilities. Therefore, the primary problem addressed is that a third-party solution is needed to implement Intelligent Speed Assistance and automated speed monitoring in present and older vehicles. Furthermore, the coming safety standard requirements also include Event Data Recorders; The purpose of these is to continuously analyse, report and archive vehicle status and health which allows investigators to understand causes of collisions. This is also highly relevant with the increasing interest of fully autonomous vehicles.

Analysing the required core functionality for the system as a whole brings to light how the requirements can be met. Three main areas of responsibility can be identified. First and foremost, a connected hardware device capable of reading data from the vehicle and receiving commands from anywhere is needed to enable communication with a backend server. Next is the server itself which must be able to organise, process and store the data. Finally, a frontend application is needed to visualise data from the vehicle and warn the driver of speeding.

In short, an Arduino hardware device will help connect to relevant sensors and publish vehicle data to a topic or data node on a publish/subscribe server. In extension, the backend will organise and store data in a database while being subscribed to the same topic to which the Arduino transmits data.

1.2 Problem Statement

The problem derived translates into the following problem statement:

A hardware and software solution for Intelligent Speed Assistance is needed to monitor the process of vehicular transportation and warn drivers of speeding, so that road safety can be increased and lives can be saved.

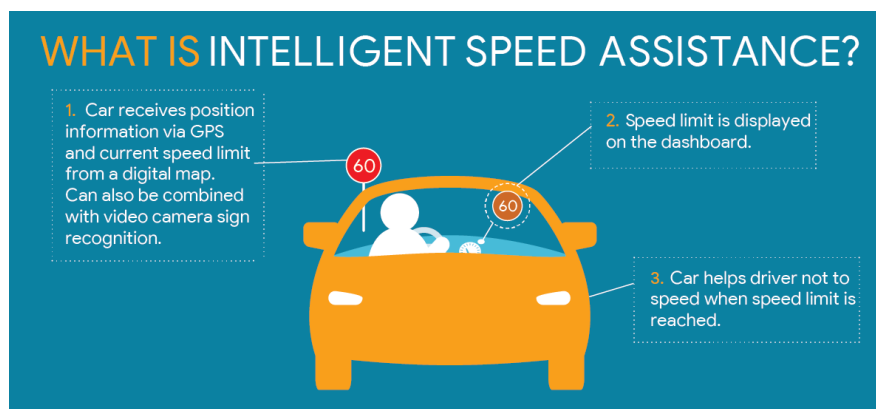


Figure 2: Intelligent Speed Assistance (ISA) explanation[9].

2 Methods & Approach

This section presents the application of methods and the adapted project delivery methodology in this project. Proceeding from the Problem Statement, the group outlined key project and research questions to answer:

1. How to design a third-party hardware solution that brings Intelligent Speed Assistance (ISA) to cars in correspondence with the EU safety standards?
2. How to design an underlying IT infrastructure that scales and enables live communication with the vehicular process?
3. How to construct a maintainable and usable web-based application that provides the means for ISA?

In order to answer the questions above while satisfying software quality attributes the group practised a notably agile software and system development methodology. This will be explained in the following section.

2.1 Project Methodology

The group applied a set of values from the methodology of the agile software development life cycle. More a philosophy than an actual process, the agile principles from the agile manifesto has been practised, including valuing (1) individuals and interactions over processes and tools, (2) working software over comprehensive documentation, (3) responding to change over following a plan[5][20, pp. 75-76]. In addition, the group has employed an architecture centric approach in order to cut down the risk of additional needed labour. Core values and principles practised from the agile philosophy include an iterative and adaptable work-flow and an aim for simplicity yet not inadequacy in regard to design and code[20, p. 76].

Even though changing requirements was welcomed and expected, the group has operated in a time-boxed iterative work scheme as a part of the Scrum method hand-in-hand with the prioritising technique MoSCoW. Thus, the process became a bit more prescriptive yet also more predictable by ensuring inspection and adaptation of progress towards the end of each sprint. The group has worked in sprints with a 2-week horizon too reduce complexity and risks. The group has used a Scrum Board as the focal point for the project to provide visibility of the Product Backlog, Sprint Backlog, and task responsibilities. In contrast to a plan-driven approach, working agile and iteratively has enabled the group to significantly reduce overhead[20, p. 75]. As a result, the group has been able to easily adapt to changing requirements and deploy essential new software quickly.

By its nature, the agile process reduces process bureaucracy and heavyweight, unessential documentation. As a supplement, the group has employed Agile Modeling (AM) for modeling and documenting, including core practices such as Model Storming, Architecture Envisioning and Just Barely Good Enough Models[1]. This practice-based methodology has been used together with Unified Modeling Language (UML) to effectively create meaningful models - in other words, UML artifacts have been created when the group believed it proved to be beneficial.

The group also integrated selected practices from the methodology Extreme Programming (XP), including, but not limited to, continuous integration and pair programming. The latter was used in conjunction with Scrum when appropriate, adding quality benefits and reducing risks, albeit a small overhead might have been added[20, pp. 83-84].

The rationale for the chosen project methodology comes from the many mentioned advantages of the agile philosophy, particularly when both the size of the development team, the size of the software system and the environmental uncertainty of this project are taken into account.

3 Research and Initial Studies

This section presents relevant technological and theoretical aspects of this project's suggested solution.

3.1 Technology Analysis

In order to gain insight into the problem domain the group continuously researched technology solutions, problems, and risks. The process of technology analysis is used to develop the requirement specification and find the most appropriate solution. This section will explore the technologies of on-board diagnostics (OBD), Controller Area Network System (CAN bus), the Arduino platform, and Global Positioning System. The different technologies provide various ways of solving the problems of this project. The following sections will dive into each technology and assess benefits and possible usages in this project as well as possible future applications.

3.1.1 On-Board Diagnostics

The automotive term on-board diagnostics (OBD) refers to fairly sophisticated systems traditionally used to control engine functions. OBD systems are composed of different parts, including sensors, actuators and an Electronic Control Unit (ECU). Together, the system monitors vehicle health and emission levels in cars.

OBD were originally put in cars to meet new emission standards and requirements by the Environmental Protection Agency (EPA) during the '70s and early 1980's[4]. Since then, OBD has undergone great development. In the mid-'90s the new standard OBD-II was introduced. Today, OBD-II is found in virtually every car and offers engine control, access to vehicle sensors, and diagnostic tools. When vehicle error occurs, a vehicle specialist can use the OBD-II protocol and interface to read vehicle data and pinpoint the exact location of the error. The physical connector is often located under the dashboard or in the passenger compartment. OBD-II enables access to the ECU in a constellation shown in Figure 3.

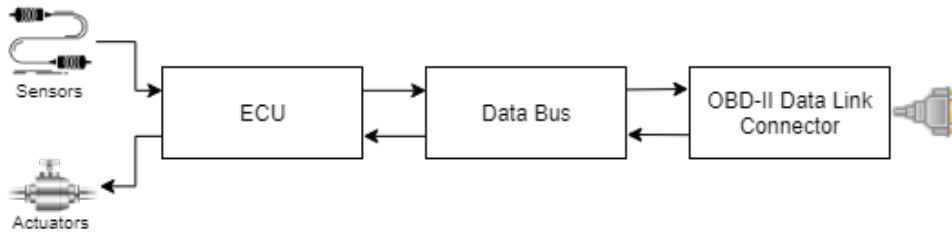


Figure 3: OBD-II diagram.

The OBD-II communication protocol defines ten diagnostic services, also referred to as modes, according to ISO 15031[15]. However, it is important to note that the available readings are dependent on vehicle manufacturers and which data points they support besides the basic OBD-II signals. Manufacturers are not obligated to support all modes[4]. Services accept parameter IDs (PIDs) to identify a data request and respond accordingly. To mention a few, Service 01 accepts a range of PIDs and enables access to current vehicle data. Service 02 is similar, except it only returns freeze frame data from a given point in time set in the request message. Service 09 can be used to request vehicle information, such as which PIDs are supported.

To receive or transmit a message, the message has to follow a certain structure. The OBD-II message structure is comprised of a 11-bit identifier block and a 64-bit data block. Figure 4 visualises the message structure. The identifier block is used to distinguish between request messages and response messages. Specifically, request messages are identified with ID 7DF and response messages are identified with ID 7E8 to ID 7EF, using hexadecimal numeral system. The data block is further divided into segments. The first part of the data block is used to

reflect the length in number of bytes of the remaining data. The second segment defines which diagnostic service that is requested. The third segment is the PID, which is the ID for a certain data set. The remaining parts are the actual data bytes.

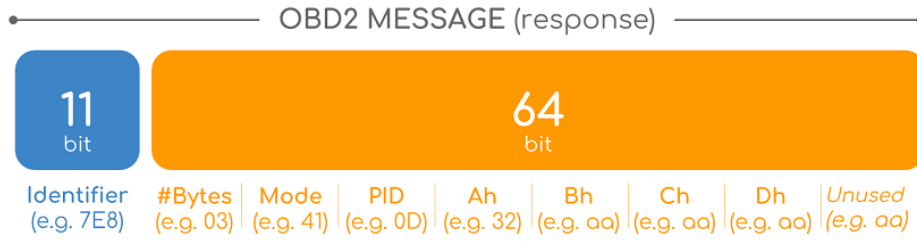


Figure 4: OBD-II message structure[7]

Deciphering the example values in Figure 4 manually reveals the following: **Identifier:** 7E9 → it is a response message. **#Bytes:** 03 → three data bytes are contained in the message. **Mode:** 41 → the data is real time from service 01. **PID:** 0D → the data is about vehicle speed. **Data byte Ah:** 32 → the vehicle speed is 50 $\frac{km}{h}$.

In regard to Intelligent Speed Assistance, OBD-II provides a way of accessing vehicle speed. It should be considered that OBD-II interestingly enables access to a range of vehicle data besides speed - even though speed is the essential data point. However since implementation of OBD-II may vary between vehicle manufacturers, additional needed labour might be added in order to support a range of vehicles - and still, vehicles are not obligated to enable access to PID 0D: **Vehicle speed**. Thus, manufacturer independence can not be fully guaranteed as of yet, possibly reducing the scalability of the ISA system. OBD-II functions as a higher-level protocol on top of well-known industrial bus systems. Many of the OBD-II systems in modern vehicles use CAN bus as the underlying protocol. This technology is elaborated in the following section for a deeper understanding of OBD.

3.1.2 Controller Area Network System

The Controller Area Network system, also referred to as CAN bus, is a serial communications bus developed for the automotive industry. The CAN bus can be described as the nervous system or central networking system of modern vehicles. Traditionally, devices such as switches, contacts, airbags, and all other sensors and actuators required complex, dedicated wiring in-between. CAN enables any ECU to communicate with the entire system via a single CAN interface[6]. Today, the CAN communications protocol is an ISO defined industry standard[14] also widely used in many other contexts than vehicles, especially in factory automation and distributed control of machines and hardware. CAN bus communication is beneficial in numerous ways. First and foremost, implementation of CAN is low cost due to simple digital communication and not analogue signal lines, reducing weight and errors. Secondly, the bus system enables centralised diagnosis and configuration. Thirdly, CAN is both robust, efficient, and flexible as a result of the fundamental concepts of the protocol's design and sophisticated error handling.

Looking at CAN in the perspective of the conceptual OSI model, the CAN communications protocol stack only consists of the lowest two layers of OSI, the data-link layer and physical layer, reducing processing delay and promoting simple communications software. The physical layer deals with the electrical aspects such as voltage levels, bit-timing, and synchronisation. The data-link layer has two sublayers, media access control (MAC) and logical link control (LLC), managing bus arbitration, frame coding, and error checking and communication service mechanisms, respectively. Figure 5 visualises the protocol stack.

The fieldbus can be characterised as a multi-master bus following the carrier-sense-multiple-access (CSMA) approach[22]. Additionally, CAN employs a distributed nondestructive arbitration mechanism, also simply known as collision avoidance (CA), making the access protocol CSMA/CA. CAN uses Non Return to Zero (NRZ) bit encoding together with a technique

called bit stuffing for synchronisation and to keep track of data bits. This means that binary signals transmitted to the bus are reflected directly in the voltage levels, resistive "1" to a high level and dominant "0" to a low level, and sequences of bits with same polarity show no voltage transitions.

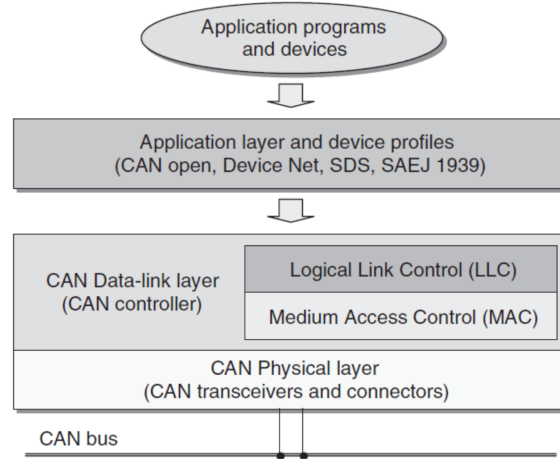


Figure 5: The layered ISO 11898 CAN standard architecture[22].

NRZ requires less bandwidth than other encoding schemes, however bit stuffing as an explicit synchronisation mechanism is required because otherwise it can be difficult for fieldbus receivers to data mine the number of bits in the incoming sequence. Figure 6 shows how a complementary bit is inserted after a sequence five bits of same polarity to force voltage change. The inserted bit is removed by the receiver when decoded:

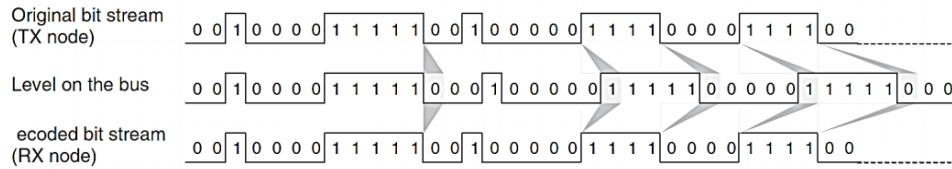


Figure 6: Bit stuffing for synchronisation[22].

Following the CSMA approach, a transmitter on the fieldbus will only transmit data if the bus is not occupied by another ongoing and more important transmission. The 11-bit identifier segment of the data frame sets the priority of a message. The lowest number has the highest priority. Figure 7 shows how three nodes transmit to the fieldbus, yet the arbitration mechanism ensures only one message wins access due to its lower identifier number. The other nodes will sense the importance of the winning node, go into receive mode, and try to retransmit later.

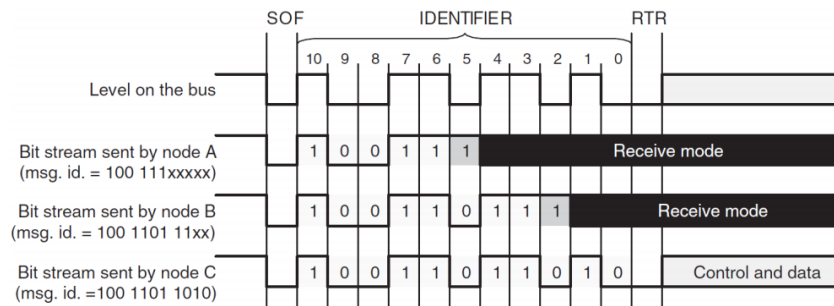


Figure 7: CAN bus arbitration mechanism[22].

3.1.3 Global Positioning System

An crucial feature for Intelligent Speed Assistance is to precisely track the vehicle's geolocation. The United States Department of Defence provides the Global Positioning System, which is a satellite-based radio navigation system designed for military use[11]. The US Military operates a fleet of satellites in orbit for this purpose. Civil use of this technology has grown very fast and the technology now commonly known as GPS is indeed very accessible and present in many modern devices, including virtually all smartphones and other consumer hardware. Even a entry-level GPS receiver is able determine its geographic coordinates by processing its acquired GPS signal and are accurate to within a few meters[17], making it more than capable of calculating vehicle location with street-level accuracy.

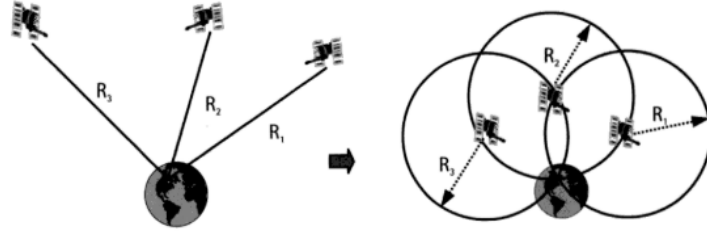


Figure 8: The underlying idea of positioning via Global Positioning System[11, p. 9].

GPS is a necessary technology to include in the ISA system for this project and can be used for both vehicle geolocation as well as determining vehicle speed.

3.1.4 The Arduino Platform

A field control unit is needed to acquire, process, and transmit sensor data, such as GPS data. Arduino is an open-source electronics platform, and Arduino boards are capable electronic boards with a microcontroller able to connect to sensors and control actuators[2]. The Arduino platform offers advantages amongst others because (1) it is inexpensive yet powerful, (2) the Arduino software is cross-platform reducing limitations, and (3) the platform is modular and many external hardware modules are available to equip the microcontroller with necessary functionality.



Figure 9: The Arduino Mega 2560[2].

The Arduino can be equipped with a GPS module and a GSM/WiFi module so the unit can transmit the necessary data. The Arduino can also connect to the vehicle via OBD-II. Additionally, the Arduino can be equipped with LEDs and/or speakers to warn the driver of speeding. Thus, it functions as a feasible hardware solution in order to bring ISA to vehicles.

3.2 Technical Solution

Based on the researched technologies the group outlined its idea for a technical solution for the Intelligent Speed Assistance system. Figure 10 illustrates the proposed solution:

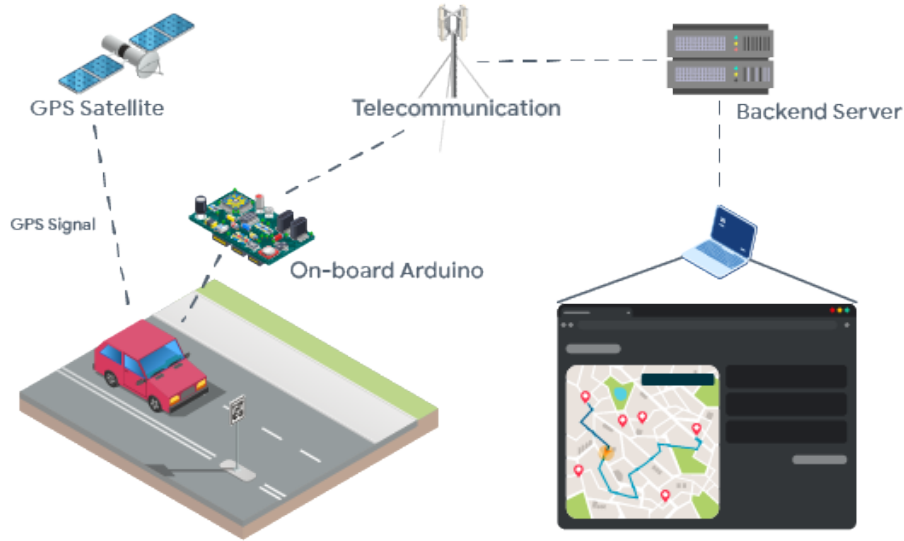


Figure 10: Illustration of the proposed solution and employed technology.

In terms of hardware, the technical solution consists of an Arduino board equipped with a GPS module and a GSM/WiFi module for internet connectivity. The GPS data can be processed to determine vehicle geolocation. The GPS coordinates can then be used to collect the current speed limit. The geolocation together with time itself can also be used to determine if the vehicles speed and if the driver is speeding. Alternatively, the vehicle's OBD-II interface can be used to collect vehicle speed. The on-board Arduino will publish the data over the internet. The data will be processed on a back-end server and further be visible on a web-based user application.

After achieving consensus regarding the technical solution the group continued to define the actual requirements for the system.

4 Analysis

The analysis process is all about understanding the domain and the requirements that outlines system scope and shape the project boundary.

4.1 Requirement Specification

During this phase the group worked together to outline and define the system requirements, taking into account the pre-handled requirements for the system, stakeholder requirements, and the group's own derived requirements to solve the problem. Table 3 presents the derived requirements.

Table 3: Prioritised system requirements

ID	Requirement	Priority
RQ1	The system must continuously monitor vehicle speed.	Must Have
RQ2	The system should be able to position the vehicle via Global Positioning System.	Must Have
RQ3	The system should alarm the operator if the vehicle exceeds the legal speed limit.	Must Have
RQ4	The system shall visualise vehicle status in a web-based GUI.	Must Have
RQ5	The system should log Event Data Records in form of snapshots of vehicle data according to legislation.	Should Have
RQ6	Investigators should be able to investigate and read comprehensible historical vehicle data.	Should Have
RQ7	The system should employ a heartbeat to ensure normal operation.	Should Have
RQ8	The system should be able to read velocity from the vehicle's on-board diagnostics using OBD-II.	Could Have
RQ9	It should be possible for an operator remotely to set an alarm on custom parameters for any of the acquired sensor data.	Could Have
RQ10	It should be possible to analyse driving behaviour, possibly based on characteristics such as acceleration, braking, speed during turns, distance driven, frequency of driving, and time of driving.	Could Have
RQ11	The system should employ an authorisation system to allow users to log in and access device data.	Could Have

Most of these requirements have been implemented, although RQ9 and RQ10 unfortunately did not due to time constraints. The software technologies used to realise the project are MQTT, Docker, HTTP, and Angular. MQTT is used communicate between the different subsystems that runs on our backend server using different Docker containers. Angular is also running in a Docker container, but is used to provide a GUI for the end users using HTTP.

4.2 Use Case Diagram

The entirety of the system is modelled by using the UML use case diagram and it visualises the behaviour of the system from the user's point of view. The use case diagram can be seen at Figure 11 and it echos the functional requirements.

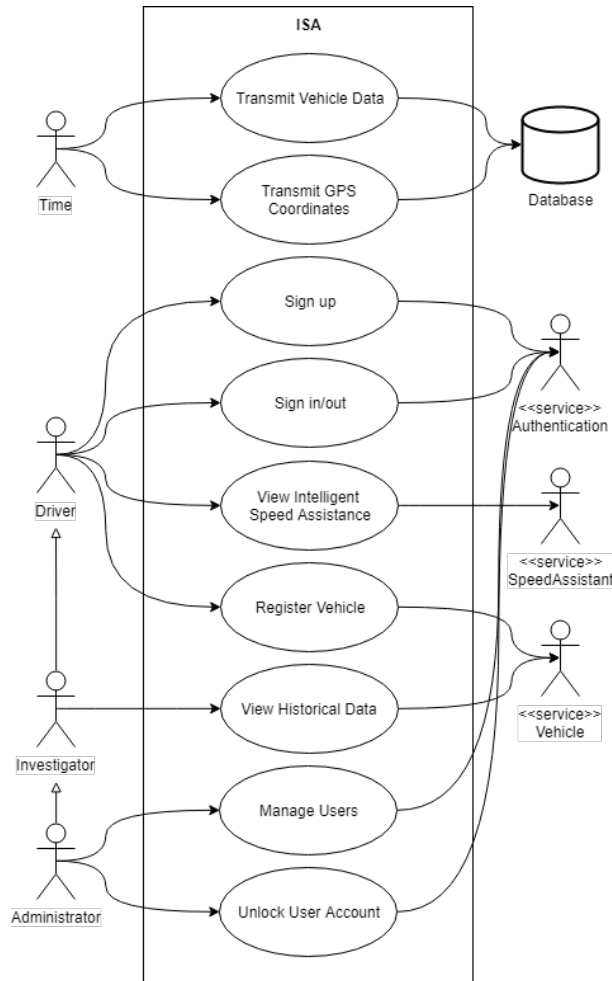


Figure 11: The scope of the software system system (Use Case Diagram).

The use case diagram contains four actors. First and foremost, *Time* is a primary actor and is responsible for uploading information regarding the vehicle and/or GPS coordinates in a looping interval to a secondary actor, the Database, as visualised by its two use cases. Another primary actor is The Driver which is associated with four use cases or scenarios for interaction with the system. The Driver is able to view data as the database receives it, resulting in live data from the storage itself. The third actor, The Investigator, can interact with the system in the same manner as the Driver, however the Investigator is extended by being able to view historical data. The Administrator further extends the Investigator and is able to perform administrative tasks and manage the users of the system. Lastly, the diagram also contains three other secondary actors in the form of services Authentication, SpeedAssistant, and Vehicle. These secondary actors are affected by the interactions of the primary actors as per the associated use cases.

Combined, the use case diagram effectively mirrors the problem domain and gives a good indication of the scope for the system and will help the group to avoid or reduce scope creep¹. The diagram clearly displays the needed features in order to satisfy the system's requirements. After defining the scope of the system the features were prioritised.

¹Scope creep refers to uncontrolled changes in requirements, or the scope of the project.

4.3 Prioritising

This section dives into prioritisation of the requirements and features. Firstly, the group has been using the MoSCoW method to prioritise the 11 initial requirements to determine if they are Must, Should, Could or Would requirements. The priorities can be seen on Table 3. Requirement 1, 2, 3, and 4 are the must haves because they represent the core system and fundamental functionality for Intelligent Speed Assistance (ISA). Requirement 5, 6, and 7 have a slightly lower priority due to the fact that they are not needed for ISA but they still provide great value for the user. Requirement 8, 9, 10, and 11 are could have these are the set to lowest priority for the product since they are requirements that comes with extra functionality.

Secondly, the use cases shown in Figure 11 were prioritised similarly using MoSCoW and largely based on the prioritisation of the initial project requirements. The prioritised use cases can be seen in Table 4:

Table 4: Prioritised use cases

ID	Actor	Use Case	Priority
UC01	Time	Transmit Vehicle Data	Must Have
UC02	Time	Transmit GPS Coordinates	Must Have
UC03	Driver	Sign up	Should Have
UC04	Driver	Sign in/out	Should Have
UC05	Driver	View Intelligent Speed Assistance	Must Have
UC06	Driver	Register Vehicle	Must Have
UC07	Investigator	View Historical Data	Should Have
UC08	Administrator	Manage Users	Could Have
UC09	Administrator	Unlock User Account	Could Have

The features were prioritised for each sprint in a manner that made each subsystem easier to complete, so each subsystem could be worked with independently. Requirements, features, and tasks were put in the Product Backlog and subsequently each new Sprint Backlog would be fed with tasks from the Product Backlog. As shown in Figure 12, project requirements and use cases were continuously classified and decomposed into smaller and more manageable tasks.

Speed Assistance komponent (websockets) ▾	Development ▾	Must have ▾	Doing ▾
Authentication interceptor (jwt) i GUI ▾	Development ▾	Should have ▾	Done ▾
Opsætning af MQTT server ▾	Development ▾	Must have ▾	Done ▾
MotorAPI.dk implementation (REST) ▾	Development ▾	Could have ▾	Done ▾
Vehicle komponent (registrering af device/v ▾	Development ▾	Must have ▾	Done ▾
Kort til visualisering af GPS-koordinater ▾	Development ▾	Could have ▾	Doing ▾
MQTT på Arduino ▾	Development ▾	Must have ▾	Doing ▾
Arduino OBD-ii simulator ▾	Development ▾	Could have ▾	Done ▾
E/R Model / Diagram ▾	E/R ▾	Must have ▾	Done ▾
Rapport: Analysis ▾	Documentatic ▾	Must have ▾	Done ▾

Figure 12: Extract of the Backlog and the decomposition of requirements into classified tasks.

4.4 Domain Model

The domain model for the project is made to aid the stakeholders of the project to communicate what's covered in the project and to provide a conceptual model of the problem domain and the real world phenomena covered by the system. The domain model is developed using techniques such as noun analysis and the already presented use case model. This helps visualising areas within the project, and its help both developers and other stakeholders to clear up any confusions that might arise during development, and keeps the project suffering of scope creep.

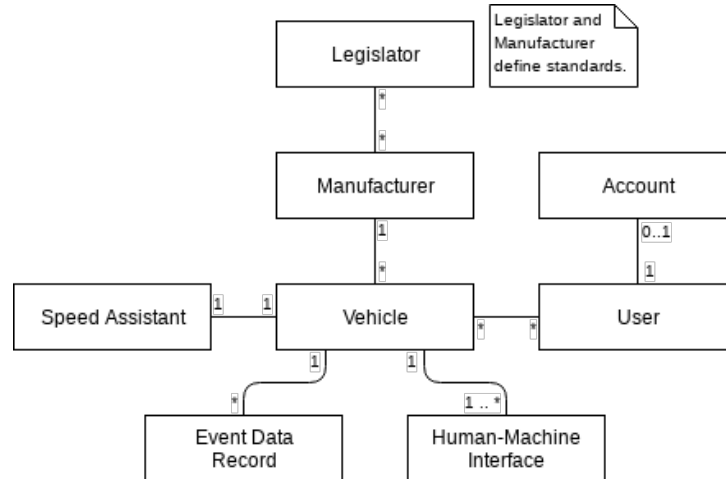


Figure 13: Domain for the system.

The domain model shown on figure 13 shows the fundamentals of the system. The model is composed of eight classes, effectively reflecting the problem domain. The classes include Legislator, Manufacturer, Vehicle, Speed Assistant, Event Data Record, Human-Machine Interface (HMI), User, and Account. The primary actors of the system such as Driver and Investigator will interact with the system through the vehicle's HMI, which will provide the information needed by the users.

The idea behind this model is to show the relationship between the real situation objects. The Vehicle object is associated with a Speed Assistant object. The data that is retrieved from the vehicle is referred to as the Event Data Record. The human-machine interface is a website that can allow the user to view historical data that the Arduino recorded. The Manufacturer and Legislator are on the domain model due to that they define the OBD-II standards available in vehicles.

5 Architecture

Architecture is an important part of a software system as it is one of the most expensive parts to change later in the software development life cycle. Because of this, the architecture is very important to get right from the start. In addition, the architecture can be influenced by design patterns all through the system and these patterns can affect the software quality attributes. Additional design principles were taken into consideration when it was developed:

1. **Separation of Concerns**

Distinct features or functionality must be encapsulated into their own tiers or components. Following this design principle provides modularity and maintainability.

2. **Single Responsibility Principle**

The responsibilities of each component, layer or tier must be cohesive. This principle increases robustness via information hiding and decreases impact of change.

3. **Principle of Least Knowledge**

Also called the Law of Demeter, the coupling of components, layers or tiers has to be minimal. This guideline increases adaptability as well as maintainability of the software system.

5.1 System Software Product Quality

Software product quality attributes are described within the "ISO/IEC FC 25010" standard[3]. To mention a few of the high importance attributes to this project are *maintainability*, *usability*, *compatibility*, *reliability*, and scalability. Together these attributes create the scope for the non-functional requirements to the system and how they will be achieved will be documented in throughout the architecture section.

5.2 Architectural Overview

One of the requirements given to the group from the project outline, is that the product must be have a web based user interface. This requirement makes a client-server based architecture very obvious, and that is what was chosen. Specifically, a multi-tier client-server architecture, as the product can be split into a client tier, an application tier, and a data tier.

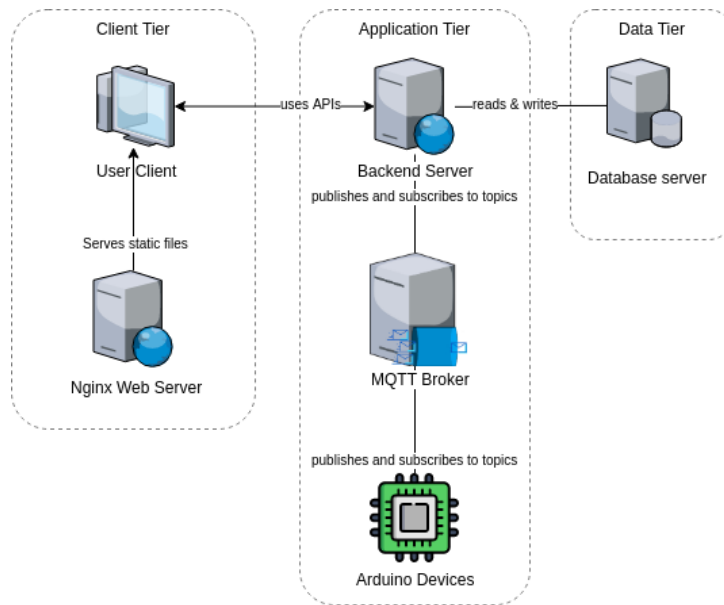


Figure 14: Deployment diagram

The client tier is what the user interacts with, to use the system. The application tier is where the functionality of the system is implemented, and the data tier is used for persistent storage of data, which includes vehicle data and user account information. The tiers communicate with each other through well defined interfaces, which contributes to the fulfilment of separation of concerns principle, and principle of least knowledge.

Within the application tier, a message queue broker (MQTT Broker on figure 14) is used to facilitate communication between the Arduino devices in the vehicles, and the backend server. By using a message queue broker, instead of letting the Arduino devices communicate directly with the backend server, the reliability and scalability of the system increases, as the message queue broker protocol is very lightweight, so it can be used by many devices, and it has some persistence, in case the backend server or message queue broker goes down.

5.3 Service Oriented Architecture

Interoperability is not a high priority quality attribute, but the system needs to support distributed systems for stakeholders to access the data. This problem can be resolved by employing a Service Oriented Architecture (SOA)[3, p. 246]. This way, other external systems or applications can access the services provided the system by consuming a communication protocol through a network. The only requirement for the consumers are the input parameters (and possibly authority). As a result, a very loosely-coupled architecture is achieved.

The client/server pattern is similar to SOA, however it is more strict in coupling and clients. The server exists for a specific client without re-use of the system. The connection between these two types is that SOA is a client/server, but with more clients, see figure 15.

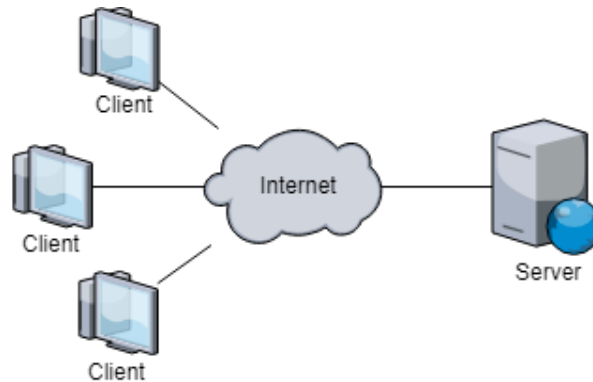


Figure 15: SOA Model. Clients connect to a server via the internet.

5.3.1 REST & SOAP

Communication between SOA and its clients can be established by two basic connectors: Representational State Transfer (REST) and Simple Object Access Protocol (SOAP). SOAP is a RPC-based approach, where the API reflects an object. The object contains public methods, these methods is the accessible API and to invoke any function the SOAP envelope has to be built[21]. The markup language the envelope uses is XML. REST is an architecture style with limitations and similar to the public methods, REST contains targetable endpoints to call the API. REST surpasses at building create, read, update and delete (CRUD) for a specific resource on the server's domain. The markup language for REST is it does not have an exact return type. The developer can choose to use XML, JSON, or a third markup language.

The REST-based approach is chosen for the SOA and client/server communication, since the

system involves a lot of basic CRUD operations, where REST is preferred over SOAP. In addition, the system sends a lot of data every second, so a low verbosity is important. SOAP contains an envelope with unnecessary data. By implementing the architecture style, the constraints it provides must be kept intact: [3][21]

1. **Focus on Uniformity**

The domain's resources must be uniquely allocated using HTTP methods.

2. **Stateless**

The state of the client cannot be stored on the server.

3. **Coupling and Cacheability**

Consumer and provider has no coupling and data caching when possible

The RESTful architecture provides the following qualities: *scalability* and *compatibility*. The stateless limitation is a further improvement of the scalability attribute of the system since servers do not have a state - hosting a new server is easy. The call flow of the API calls can be viewed on figure 16.

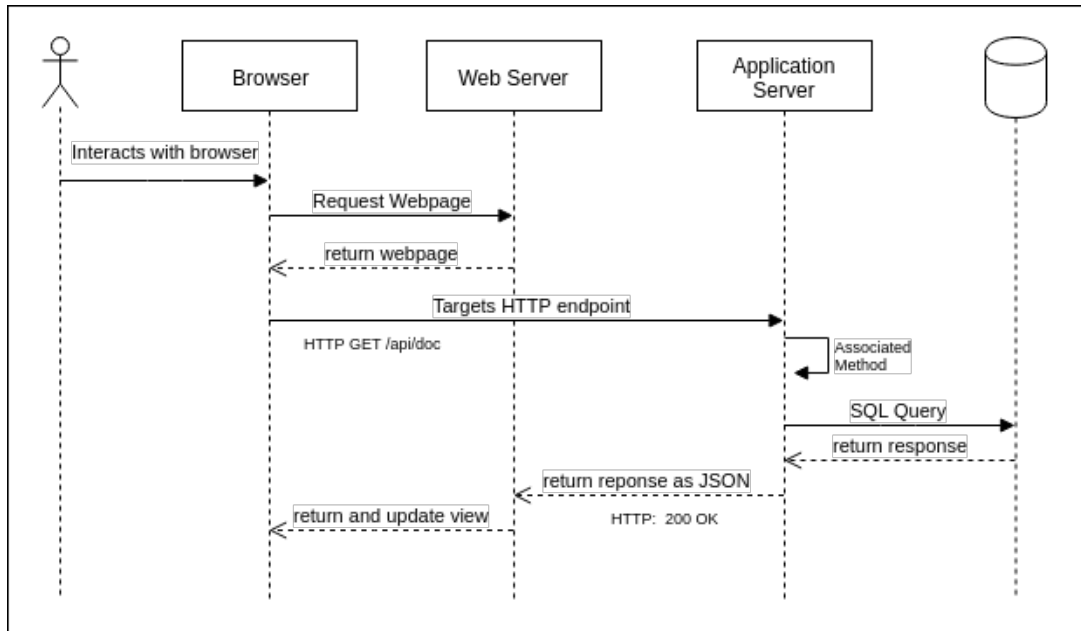


Figure 16: The flow for the HTTP calls for webserver, backend server and REST.

5.4 Component-based

The backend of the system uses a component-based architecture, and there are several reasons behind this choice, which will be presented in this section. The words "component" and "module" are used interchangeably.

Component-based systems accentuate separation of concerns, by splitting features into a component, for example a authentication feature which contains registration and login functionality is its own component. Building a system by composing modules together, will also make the system more maintainable[16], as it is harder to create "spaghetti code", code that is tangled and interweaved into an incohesive and unmaintainable mess, because the functionality is each component is clearly separated from the rest.

Splitting by feature in the context of this system, refers to going from a monolithic, non component-based, software system, to a component-based software system, by creating modules

and grouping code, based on the feature they implement, instead of the layer of they belong to. Inside the component, the code is divided into data, business and presentation layers, as one would expect of a monolithic application. The authentication component is split into a data layer that communicates with some form of persistent storage, a business layer that handles the business logic, and a presentation layer that in this case returns API data, formatted as JSON.

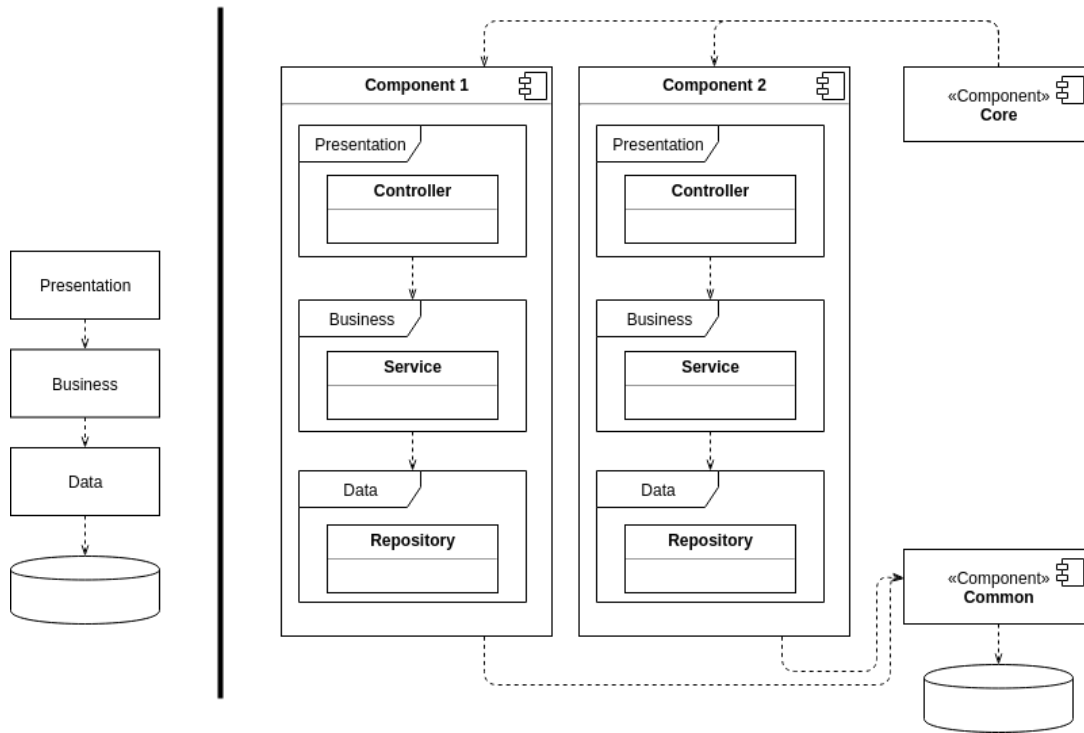


Figure 17: Monolithic (left) and Component-based (right)

The difference between monolithic and component-based is displayed visually on figure 17. Here the components depend on the Common module, which provides an interface for doing database operations, and the Core module depends on the components, so it can instantiate implementations of the interfaces provided by the Common module. This is in contrast to a monolithic approach (left side of figure 17) where the functionality provided by Component 1 and Component 2, is spread across the three layers, and can potentially depend on each other, even though they logically should not. Compared to a monolithic architecture, the component-based architecture has a greater upfront complexity when development starts, but this pays off, the further you get into the project.

Another reason for choosing to use a component-based architecture, is that it is easier for multiple developers to develop components parallel to each other, as long as the service provided interfaces are well defined, and do not change. These interfaces are often placed in a "Common" module, that all other modules have access to, and can request an implementation of via the component framework, without having a compile time dependency on the component that implements it.

In theory it is also possible to dynamically load and unload components, while the system is running. This is often used as a plugin system in other software systems. Although potentially useful, it was decided to not be necessary in this system. The additional complexity of using a component framework that supports that functionality is not worth the feature, because dynamically loading and unloading components did not make sense in the context of

this project, which is web based, with multiple clients connecting to a single backend. If the application tier was running on the client device, instead of on a remote server controlled by the developers, having a plugin system would be an easy way for the user to extend their system.

To facilitate a component-based system, the components need a component framework to hook into, otherwise the components just exist, without actually being usable. The exact way this is done, is explained in the implementation section on page 22.

6 Design

After the architecture has been decided, the next step is to design the components that make up the system. The design presents abstract ideas about the structure of the software, without going into implementation details.

Based on the groups knowledge of object-oriented principles and the responsibilities of the backend application, several components were identified. Three main components, which provide features that realises use cases: Authentication, SpeedAssistant, and Vehicle. Two components that will facilitate the component structure, and two components that allow for code reuse. Besides the components identified for the backend system, there was also written a simulation program, to help during testing and development.

Aside from the backend, the system also needs something to collect data from, which will be Arduino Devices. The collected data must be stored somewhere, which will be in a relational database. Finally, the data must be presented in a way, that the user finds pleasant, which will be done via a web interface.

All these subjects will be elaborated in the following sections.

6.1 Backend

The central classes within each component are Repositories, Services, Controllers, and Providers. This structure is inspired by the Real World² example applications.

Figure 18 shows an excerpt of the design class diagram for the backend. It shows the folder structure in the Vehicle component, and how it depends on the shared components, CommonVehicle and CommonAuthentication. Notice how VehicleRestProvider only depends on interfaces from the shared components, even though VehicleRestProvider is responsible for glue code in this component. This is because the implementations are provided to the component, by the component framework.

6.1.1 Repositories

The repository classes in each component has the responsibility of making queries against the database. Some repositories exists in shared components, so multiple components can access the database, without re-implementing the SQL queries, and duplicating code. Each repository is associated with a model class, that approximately maps to a database table. Repositories are equivalent to the Data Layer in a 3 layered application.

6.1.2 Services

Service classes perform the business logic of the application. Most services have a reference to one or more repositories, if they need to persist data in the database. By using a service class, the code is more testable, because they can take multiple arguments and return values, which controllers do not. Services are also used as an abstraction over third party API's. They are equivalent to the Business Layer.

6.1.3 Controllers

Controllers are the main entry point into the main components, as they are the methods associated with the API routes. Controllers can have one or more references to services, and their responsibility is extracting data from the HTTP request, passing the data onto a service, and returning the objects from the service, as JSON.

²<https://github.com/Rudge/kotlin-javalin-realworld-example-app>

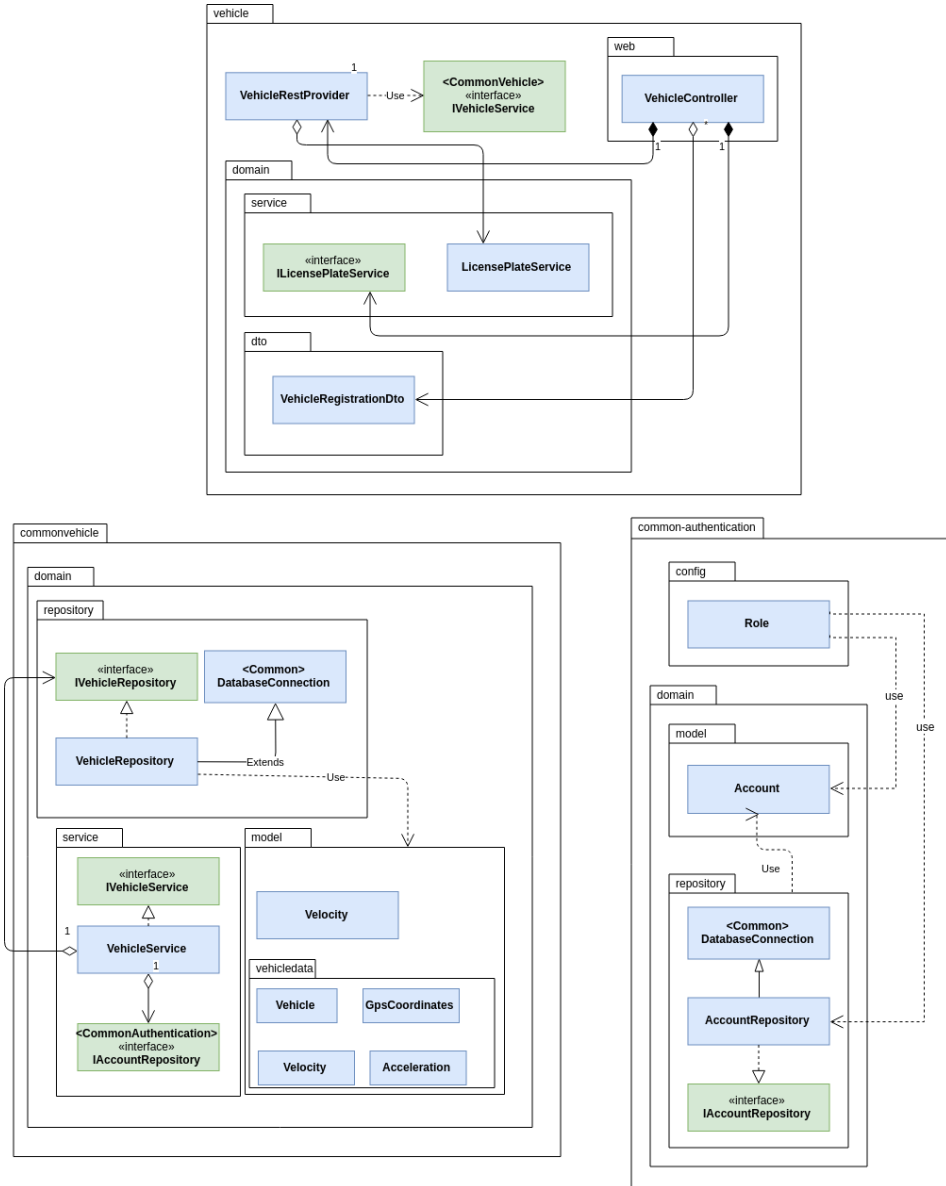


Figure 18: Excerpt of full design class diagram for the backend

6.1.4 Providers

Provider classes are the glue code of components. In the provider classes, the repositories, services, and controllers are instantiated. The provider classes implement interfaces, such that the Core module of the application can access them. The provider classes define the API routes, and the level of authentication required to access them.

6.1.5 Speed Assistant

The speed assistant component is structured slightly differently than the rest of the components, because it does not provide a REST API used for CRUD operations. The responsibility of the Speed Assistant is to subscribe to message queue topics, so it receives vehicle data, and to pass that data along to other entities that needs it.

These entities are called Speed Assistant Communication Services, and the system has three: One for storing the data in the database, one for sending data back via the message queue, and one for handling web socket connections. These services has methods for each data type, that can be passed around the system, such as GPS or velocity. The methods are called, each time a new piece of data comes into the system.

6.2 Simulation

A simulation application should be developed, to make testing and development of the application easier. The simulation module is not a part of the backend, as it is a self contained application, though it depends on the Common module, to avoid code duplication. The structure of the application is split into sensors, actuators, state, and a glue code class. The sensors return values, that the glue code sends to the message queue. Actuators act on the data. State contains the current state of the simulation. Glue code connects to the message queue, instantiates sensors and actuators, and makes sure the application runs for a set amount of time.

6.3 Hardware

The hardware used in this project is an Arduino Mega, because of its many pins, and because a member of the group already had it, so it saved time. The hardware components needed for the Arduino is a GPS, OBD-2 connector, and a WiFi shield. The GPS component is used to get the position of the vehicle, using the Global Positioning System. The WiFi shield is used to connect to the internet, so the data that is being collected, can be transmitted to the backend system. OBD-2 is used to read data about the vehicle, such as its current velocity. One of the requirements given to the group, is that it should be possible to retrieve data, but also control processes. The process that was decided to be controlled, was the velocity. The backend system detects, if the vehicle is exceeding the current speed limit, based on the speed limit of the vehicles current approximate location, and the current velocity that is being read from OBD-2. It would not be safe to attempt to control the brakes of the car, so instead, a LED changes colour when the car is exceeding the speed limit.

6.4 Database

This section will dive into the design of the database. For the purpose of designing a relational database, an E/R diagram has been developed for the database as seen on figure 19. A few choices were made while designing the database for the system such as the database should fulfil the requirements for Third normal form and if that the license plate attribute should be on the device table or on the vehicle table, the group decided to put licence plate on the device, due to if it were to on the vehicle table it would have a lot of rows, with duplicate data.

The database is structured as shown on the E/R diagram where it is divided into 3 tables device, vehicle, and account. The device table has 3 attributes which is device id, last active time, and licence plate where device id is the primary key and is always unique to identify the different devices in the database. The vehicle table is a weak entity set of device, which means it uses the same primary key as the device but provides more data to the device table. The vehicle table contains speed, timestamp, acceleration, speed limit, longitude and latitude. These values, except for timestamp and speed limit are data that there will be received from the OBD-II. The account table has the attributes username, password, last login attempt, number of login attempts, created, and last login. The account table is connected to the device table, via the connects table. Similar to the device table the account table also has a primary key which is account id.

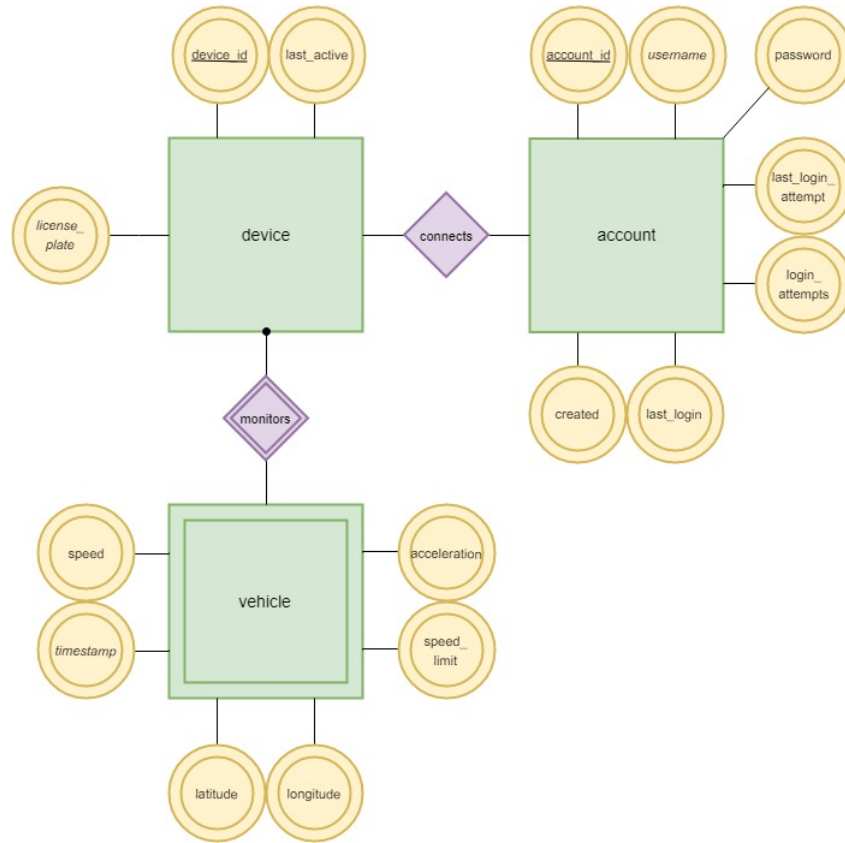


Figure 19: E/R diagram of database

The relationship between **account** and **device** is a many-to-many relation, which is allowed by the **connects** table, that will hold the primary keys of **device** and **account**. The reason for this type of relationship is, that it should be possible for an account to have multiple devices, and to allow for one device being used by multiple accounts.

The other relationship, that is represented by **monitors**, is between **device** and **vehicle**. This is an exactly one-to-one relationship, because a vehicle cannot exist in the system, without a device that monitors it.

7 Implementation

In this section, the implementation of the system will be explained in every aspect of the client tier, application tier and data tier. In addition, the usage of Docker will also be described.

7.1 Client Tier

The client tier is responsible for handling the connection between clients and backend. In this case, it is a web-server and it contains the frontend of the application. To implement the GUI, a framework was used.

Front End The frontend of the application is a web-based solution which means applying *HTML*, *CSS*, and *JavaScript* to structure, layout, and behaviour, (respectively) the site. Combined, they fuse to be the essence of a modern web-based interface and the cornerstone for web development and technologies. Furthermore, when a client uses a browser to request a webpage the web server sends files of these formats (.html, .css, .js) back to the browser to interpret.

Depending on the backend, the GUI is not always essential, and in this case it is not. It is primarily used to demonstrate the backend. However the GUI supports user interaction with inputs and events to control the flow of the application. Instead of building the client tier from scratch, the group chose a framework to achieve more structured code, scalability and take advantage of the time-saving feature a framework it provides.

7.1.1 Angular Framework

The chosen framework is the Angular Framework. Angular is a JavaScript-based framework that can be installed by using NodeJS. Angular itself contains additional features and further extensions to improve the ease of the implementation for a complex web application. Yet, Angular is just one of many frameworks available to use, but the reason the group chose this framework are: (1) few members of the group had already experience with it. (2) Angular is not small simple framework, it is maintained and published by Google, making it a solid choice for a safe environment. (3) Angular uses a superset of JavaScript, TypeScript, which is an open-source programming language developed by Microsoft. Additionally, this adds features from object-oriented languages, a well known method to structure code.

Structure In Angular there is not a default or standard way to structure the files. Therefore, the group has implemented their own unique way that made the most sense for everyone. The root folder of the Angular project is *src*. Below the *src* folder it is categorised into three domains, *core*, *modules* and *shared*.

core The core folder is a wrapper for the two folders, modules and shared. It contains the main routing to all the initial pages and the navigation (menu) of the page.

modules The modules folder has all the modules within the website. A module is an container for multiple components.

shared The shared folder, as the name imply, is a folder containing shared services, modules and components.

However, the modules do have their own structure. Every module consists of a parent folder, a module file and a folder called pages. The pages folder is for the different pages a module can have. An example would be the home module. It only has one page, home itself. In contrast, the auth module consists of two pages, sign in and sign out. The structure is supported by figure 20, which shows the folder tree that gives a glimpse of explained design.

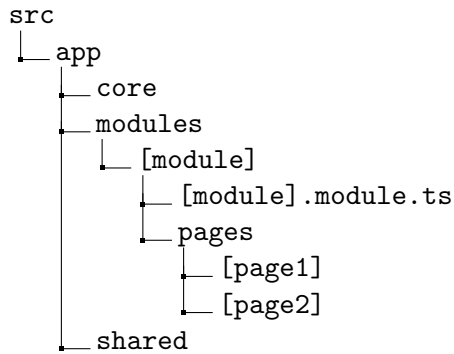


Figure 20: The Angular Structure in a nested folder format. The placeholder *[module]* can be any module in angular and *[pageX]* can be any page in module.

7.2 Application Tier

The application tier is the most important of all the tiers. This tier incorporates all the functionality and services of the product to make it an actual application. Therefore it is the business logic and essence of the application. This involves the server, MQTT and Arduino.

7.2.1 Server

In this section, everything related to the server will be discussed. The majority of the application relies on the server, since it is the service the client tier can use and other businesses (stakeholders) can invoke to achieve their intentions. The section will go into depth about how component-based is implemented and what tools/technologies have been used to accomplish this. In addition, smaller topics such as Javalin, authentication, and web sockets will be weaved into existing sections or standalone explanation.

Component-based

The component-based architecture of the backend does not use a component framework like OSGi or NetBeans Platform. Initially, Service Loader was used because it was what the authors of the project learned first. This was later replaced by the Spring Framework, and the annotation based configuration. The functionality of Service Loader was abstracted from the start, into its own class called SPILocator, so it was very easy to replace.

There are three main interfaces, that the core module depends on, to start the application. These are IRouterService, IAccessMangerService, and IWebSocketService. IRouterService is used to define HTTP API routes, IAccessManagerService is used to add role based authentication to HTTP API routes, and IWebSocketService is used to define a web socket route. By default web sockets cannot have authentication, so to work around this, the group made their own role based authentication for web sockets.

The type of component model used for this project is called a Whiteboard Component Model. The whiteboard component model works by components publishing an interface the to component registry, and allowing other components to query the registry for implementations of interfaces. The component registry holds a mapping of interface to implementation, and the assembler will create an instance for the component that needs it. The whiteboard component model is shown on figure 21. An alternative to this model is called dependency injection, where the component that requires an implementation of an interface does not need to retrieve it from the component registry, as it is injected into the component, either via constructor injection or setter injection. This is also called inversion of control, as it is the assembler that takes initiative to create and pass an implementation to the component.

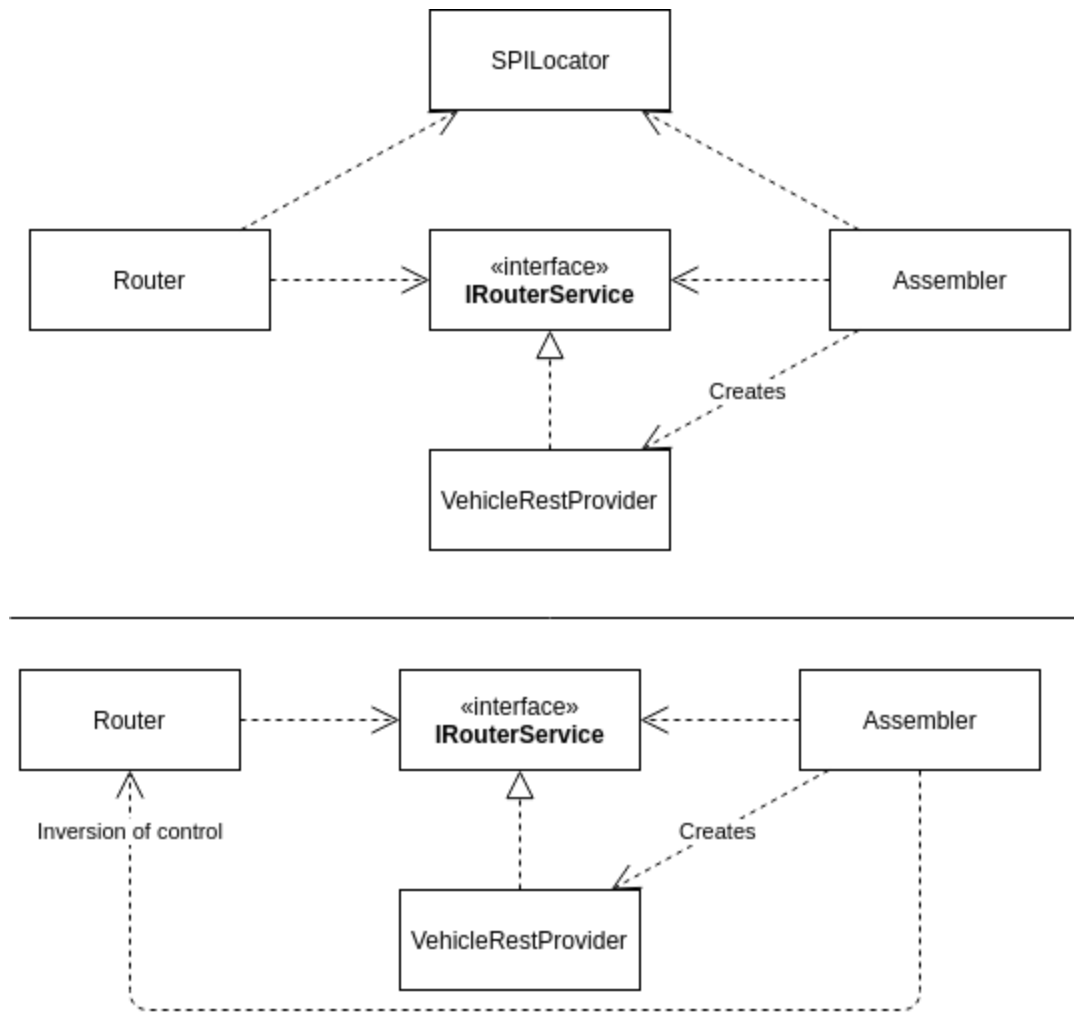


Figure 21: Whiteboard Component Model (top) & Inversion of Control (bottom)

A problem that occurred when compiling the project, when using Service Loader was that not all implementations of interfaces were found, so certain modules were completely unused. This is because of the flat class path problem that is associated with attempting to develop a component based application, without using an actual component framework like OSGi or NetBeans Platform. When all the modules are compiled, all their classes and resources folders are placed into one folder, and because Service Loader requires that the file that maps the interface to implementation, is placed in a folder called **services** in the resources directory, these files overwrite each other, when there are multiple implementations of the same interface. The end result is that, whatever module that gets compiled last, is the one that gets to stay, because it has overwritten all the modules before it. In the context of this project, this meant that only the HTTP routes of the Authentication module were available, because that is probably the last one that got compiled.

To resolve this problem, it is possible to use a **transformer** in the Maven pom.xml file. This handles the problem, by appending to the SPI file, instead of overwriting it. This worked, but in the end, the group decided to use Spring annotations, which completely avoids this problem.

Javalin

Javalin is the framework used to enable HTTP functionality. Javalin routes works by defining an HTTP method, a route, a function references which takes a context object as argument, and a set of roles that are allowed to access this route. This is a simple building block, that is used to abstract the registration of routes, to fit Javalin into the component based structure of the application. The core module is responsible for acquiring implementations of the interfaces that define HTTP routes, authentication management, and web socket routes. The interface used to define HTTP routes is `IRouterService`. Figure 21 shows how the Router class in the Core module gets access to all implementations of the interface. Codeblock 1 shows the interface, that other components use to define routes, and codeblock 2 shows an excerpt of the implementation of the interface, in this case from the Vehicle module.

```
1 public interface IRouterService {
2     EndpointGroup getRoutes();
3 }
```

Codeblock 1: IRouterService interface

```
1 @Override
2 public EndpointGroup getRoutes() {
3     return (() -> path("vehicle", () -> {
4         post(controller::registerVehicle, roles(Role.AUTHENTICATED));
5         get(controller::getVehicles, roles(Role.AUTHENTICATED, Role.ADMIN));
6         get("licenseplate/:license-plate", controller::
7             getVehicleLicensePlateData, roles(Role.AUTHENTICATED));
8     }
9     [...])
10 }
```

Codeblock 2: Excerpt of IRouterService implementation

Codeblock 2 shows three routes. All routes in the module as prefixed with `"/vehicle/"`, and that is the route that is used for the first two. Line six defines a route, with a variable part, which is denoted by the colon (`:`). The actual value of the variable part is available in the controller's method, using the context object, and calling the `pathParam("license-plate")` method.

Another thing that is handled by the Javalin framework, is exception handling. Specific exceptions are defined to be caught by the Javalin framework, and instead of simply logging the exception to the console of the application, an error is sent to the user. An example of this can be seen on codeblock 3.

```
1 app.exception(NotFoundResponse.class, (e, ctx) -> {
2     log.error(String.format("NotFoundResponse occurred for req -> %s (%s)", e,
3         ctx.path()));
4     String msg = !e.getMessage().isEmpty() ? e.getMessage() : "404 Not Found";
5     ErrorResponse err = new ErrorResponse(Collections.singletonMap("body",
6         Collections.singletonList(msg)));
7     ctx.json(err).status(HttpStatus.NOT_FOUND_404);
8 }
```

Codeblock 3: Excerpt of IRouterService implementation

In this example, `app` is a instance of the Javalin class. The `NotFoundResponse` is an exception provided by the Javalin framework. It is thrown by Service classes, when for example, a user requests vehicle information for a certain license plate, but no data is found.

Authentication

HTTP API authentication is implemented with JSON Web Tokens (JWT), which is a standard for transferring data between interested parties. Tokens can be digitally signed using symmetric key cryptography, to ensure that they have not been changed during transmission.

JWT consists of three parts, separated by a dot (`.`): a header, a payload, and a signature.

header Denotes what algorithm is used for signing, in this case HMAC256 is used.

payload The data that is being transferred between parties. This data can include such as the date the token was generated in Unix time format, the date which the token expires in Unix time format, and other data. These data are called claims, some of which are defined by the JWT standard, but it is also possible to add custom claims. In this project, a custom claim is used to add the role of the user.

signature The signature is used to verify, that the message was not changed during transmission. The signature is generated like this (pseudo code):

`HMACSHA256(base64UrlEncode(header) + "." base64UrlEncode(payload), secret-key)`

A concrete example of a JSON Web Token looks like this:

```

1  {
2    "alg": "HS256",
3    "typ": "JWT"
4  }
5
6
7  {
8    "sub": "bob",
9    "iat": 1557995887,
10   "exp": 1558082280,
11   "role": "authenticated"
12 }
```

Codeblock 4: JWT Header (first object) and JWT Payload (second object)

When these parts are transformed into the actual token, using base64 encoding, it looks like this

```

1  eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
   eyJzdWIiOiJib2IiLCJuYW11IjoIc2U1LCJpYXQiOiE1NTc5OTU0ODcsImV4cCI6MTU1
2  ODA4MjI4MCwicm9sZSI6ImF1dGh1bnRpY2F0ZWQifQ.
   Hj4btVwxyNAHkhmUK_mmnbtB8U51kEKjh4uHudcepsY
```

Codeblock 5: Actual token

This token can be parsed by the backend server and frontend client, to read the values defined in the payload.

Web Sockets

Web sockets are used to certain data from the backend to the frontend. Web sockets are used because it reverses the usual responsibilities of an HTTP request, by enabling the server to send new data when it is available, instead of having the client send a request in a timed interval.

By default, web sockets do not provide a way to implement authentication, because the protocol does not support the same HTTP headers, as a regular HTTP request does. To work around this, a ticketing system has been developed, where the first message a connecting client sends, is their JWT token. When the backend system receives this token, it can proceed as usual, as if it was a HTTP request.

To add a web socket route to the system, it must implement the `IWebSocketHandler` interface, shown on codeblock 6, which defines the route, the roles that are allowed to access it, and methods for several events, such as when a new message is received, and when a new connection is opened.

```

1 public interface IWebSocketHandler {
2     String getPath();
3     Set<Role> getPermittedRoles();
4     void onConnect(WsSession session);
5     void onMessage(WsSession session, String message);
6     void onClose(WsSession session, int statusCode, String reason);
7     void onError(WsSession session, Throwable throwable);
8 }

```

Codeblock 6: Web Socket interface

It is in the onMessage method, that authentication is handled. In this system, the clients does not have to send data via web sockets back to the client, even though web sockets are bi-directional. Therefore, the method onMessage is used only for authentication.

```

1 if (!webSocketAuthenticationService.doesUserHaveRole(getPermittedRoles(),
2     message)) {
3     websocketEndpoint.send(new Response(HttpStatus.UNAUTHORIZED_401, "Not
4         permitted"), session);
5     return;
6 }

```

Codeblock 7: Web socket authentication

Codeblock 7 shows how authentication is done in the onMessage method: If the user does not have one of the permitted roles, it returns an object with a message and a status code, telling the user what went wrong.

Third party API's

Two third party API's are used in this project, one of which is integral to the functionality of the project. The API's used are Overpass³, and MotorAPI⁴.

Overpass is a part of the OpenStreetMap project. It is used to get road data from a GPS coordinate, more specifically the speed limit. It is a public API, without any specific limitations, but they recommend doing less than 10,000 queries per day.

MotorAPI is used to retrieve data about vehicles, based on their license plate. This is only used to make the frontend web interface more exciting to interact with, because you can see the make, model and type, instead of just a license plate. This API is not public, and has a limit on the free tier of about 3000 requests per month.

To use these API's in the project, Java interfaces have been created and implemented, so it is easier to use, and to develop in accordance to principles like encapsulation.

```

1 // ILicensePlateService.java
2 public interface ILicensePlateService {
3     String getData(String licensePlate);
4 }
5
6 // ISpeedLimitService.java
7 public interface ISpeedLimitService {
8     short getSpeedLimit(GpsCoordinates gpsCoordinates);
9 }

```

Codeblock 8: Java interfaces for Overpass and MotorAPI

By using interfaces and providing implementations, instead of making direct HTTP calls when they are needed, it is possible to implement things such as rate limiting, and result caching. The implementation for Overpass API (ISpeedLimitService) implements rate limiting, so a request

³https://wiki.openstreetmap.org/wiki/Overpass_API

⁴<https://www.motorapi.dk/>

can only happen every 5 seconds, and if the user of the interface tried more often than that, the last value is used again. If this is not done, the API may throw an error.

The `ILicensePlateService` implementation has a map, that maps Strings to Strings, where the key is a license plate, and the value is the data associated with that license plate. When a user of the interface requests data, the map is checked first before a request is made, if the map has a value for the license plate, it returns that, otherwise it makes a request to the API, and stores the data in the map before returning it to the user. Using this method does come with a trade off, because if the application ran for a long time, its memory usage would increase.

A similar technique is used with the Overpass API, except the mapping is from a `GpsCoordinate` object to a `Short`, the speed limit.

Another advantage of this encapsulation is maintenance. API's may change version, make breaking changes, or even shutdown completely, but because all interactions with the API is done via the interface, the users of the interface do not have to think about this.

7.2.2 Arduino

The Arduino is the mediator for the user, the car and the application. It is the hardware responsible for fetching data from can-bus in the car and transmitting it to the server by the MQTT server. However, hardware tend to requires low-level language to program, thus making it difficult to program from scratch. Accordingly, libraries were used to carry out the necessary functionality of the Arduino. Every library delivers certain features. These features are implemented as abstractions in the Arduino's two default methods, `setup()` and `loop()`.

Function: `setup()` The `setup()` function is always the first method to be called when the Arduino starts up to initialised everything that only has to be done once. In the codeblok 9 shows the actually code from the Arduino. `Serial` is the monitor built-in the Arduino for developers to follow. The variable `ss` is a variable of `SoftwareSerial` type and reads data from the GPS on the Arduino. Furthermore, the rest are created methods to handle WiFi, MQTT and reconnect to MQTT.

```

1 | void setup() {
2 |     // Serial Monitor
3 |     Serial.begin(9600);
4 |     ss.begin(GPSBaud);
5 |
6 |     setupWifi();
7 |     setupMQTT(&mqttClient); // & is a pointer
8 |     reconnect(&mqttClient);
9 | }
```

Codeblock 9: Arduino code taken directly from `setup()` method. Variables are not showed.

Function: `loop()` The `loop()` function is invoked after the `setup()` method and is called multiple times every second until a restart happened. Codeblock 10 is the full code of the `loop()` method. The first call is the exact same reconnect call. It made sense as a precaution to increase usability and up-time. It checks if it is (still) connected to the MQTT server and if it is not, reconnect to it. Then it does a heartbeat to the MQTT server, so the MQTT knows it is alive. The if statement is a constant delay that has to be passed before it publishes the GPS coordinates, velocity, and acceleration to the specific topic based on the Arduino's identifier (ID) on the MQTT server. Every Arduino has a unique ID that is the type `UUID`⁵. Furthermore, the `blinkLED()` method lets the LED blink if server sends an error code to the Arduino. The last line in `loop()` function, `smartDelay(1000)`, sleeps the Arduino rather than

⁵See Glossaries for definition.

just delaying it. When it reaches the line, it will not continue until a second has passed. It ensures that the GPS is being fed with information about its location.

```

1 void loop() {
2   reconnect(&mqttClient);
3   mqttClient.loop();
4
5   unsigned long currentMillis = millis();
6
7   if (currentMillis - previousMillis >= interval) {
8     previousMillis = currentMillis;
9
10    sendGPS();
11    sendVelocity();
12    sendAcceleration();
13
14    blinkLED();
15    smartDelay(1000);
16  }
17 }

```

Codeblock 10: Arduino code taken directly from loop() method. Variables are not shown.

At the beginning of the section, the word libraries were mentioned to create the functionality of the Arduino, but what libraries were used? A brief description of what each library contribute to in a list (these libraries are also mentioned in the Arduino source code):

WiFi101 is a unique library that is compatible with the WiFi shield the group had and made the Arduino access the Internet. The creator is Arduino themselves.

pubsubclient is one of many libraries to publish and subscribe to a MQTT server. The creator is knolleary.

ArduinoJson is the only library found to generate JSON objects. The parse feature is unknown since it was not used. The creator is Benoit Jackson.

TinyGPS++ is the library for the GPS hardware that is extended alongside the Arduino. The creator is mikalhart and the library can be found on Github.

OBD2UART is the library for fetching data from car's CAN-BUS to the Arduino. The creator is stanleyhuangyc and the library can be found on Github.

7.3 Data tier

The data tier is small and is the source of persistence storage. Anything that needs to be store for use in future or archival use. The storage for the product is a SQL database and will be further explained in section Database, 7.3.1.

7.3.1 Database

The database is the storage of the application, hence the name. It is used to save anything related to the application or return data when an action is executed or the system needs it. A database can come in many variations, however, the one chosen for this system uses a language called Structured Query Language, also known as SQL. SQL is widely used and different manufacturers develops their own sub-type. The one being used is called Postgres from PostgreSQL and the logic behind it is features, architecture, experience and easy-to-use compared to use standard text files or excel files. Nevertheless, every SQL database consists of tables and every tables has a name and multiple columns. Whenever data is inserted it is a row or a tuple. The code used to create the account table for the system to use can be inspected on codeblok 11. The 'statement' word and 'query' word will be used interchangeably.

```

1 CREATE TABLE account (
2     account_id BIGSERIAL PRIMARY KEY,
3     username TEXT NOT NULL UNIQUE,
4     password TEXT NOT NULL,
5     created TIMESTAMP DEFAULT NOW(),
6     last_login TIMESTAMP,
7     login_attempts INT DEFAULT 0,
8     last_login_attempt TIMESTAMP,
9     role TEXT
10 );

```

Codeblock 11: The SQL code for creating the 'account' table in a database.

Explanation of Codeblok 11

Every action is called a statement. The first line in the statement tells Postgres to create a table with the name *account* and between the parenthesis the configuration for the table is described. The format for creating a column is `<name> <type> <extra>`. So the first word is the name of the column, the next is the type and the rest is extra, yet, these extra attributes contain functionality to the column and the outcome varies for every keyword. Every type or extra attribute being used in the statement is described in their own respective list: ⁶

Types:

- **VARCHAR(X)**, a maximum of X characters to be store.
- **INT**, (integer) that ranges from $\pm 2^{31}$.
- **BIGINT**, (integer) that ranges from $\pm 2^{63}$.
- **TIMESTAMP**, a combination of date and time e.g. 2019-05-24 23:59:59.

Extra:

- **UNIQUE**, self explanatory - only a unique value is allowed.
- **DEFAULT <value>**, default value will be placed instead, if not set when inserting.
- **NOW()**, current date and time.

The previous material explained how to construct a new table in the database. Inserting and retrieving are similar, but differs in the starting keywords of the statement. The first example, codeblock 12, captures the essence of inserting a tuple into a table; in this case the account table. For this specific statement the query is about inserting three values into three different columns, *username*, *password* and *role*. The '?' would be replaced with real data and has to match the same order as mentioned. Afterwards it returns columns to be processed or use by the system.

```

1 INSERT INTO account (username, password, role)
2 VALUES (?, ?, ?)
3 RETURNING account_id, username, password, created, last_login, login_attempts,
   last_login_attempt, role;

```

Codeblock 12: The SQL code for inserting into the 'account' table.

Another important action is selecting tuples. The codeblok 13 is about selecting a distinct user depending on a column and a value. This is directly taken from the system, hence the field variable and '?' sign. The **SELECT** keyword selects all the columns that comes afterwards and returns their values. The **FROM** keyword targets a specific table within the database the select should focus. The last keyword **WHERE** is an optional to specify conditions that rows have to match to be returned.

```

1 SELECT account_id, username, password, created, last_login, login_attempts,
   last_login_attempt, role FROM account WHERE " + field + " = ?;

```

Codeblock 13: The SQL code for selecting information from the 'account' table.

⁶<https://www.postgresql.org/docs/9.2/datatype.html>

SQL Injection

In every statements above, the '?' is a replacement for value. Being able to do SQL injections puts the system at risk from outsiders, decreases the security and allows them access to the database by exploiting the flaw in the application. Let us take the codeblok 13 as an example. If the attacker is aware that the system has no defence against SQL injection, the '?' can be changed to whatever they desire, which is generally to add a condition that is always true to receive all the information or end the statement the system is trying to execute. Afterwards the attacker can extend the statement to inject their own query. This depends on how the SQL is handled, so the attack(s) may vary. The factors on the attack can be seen in the list:

- Information received by column name or by index?
- Does the system only execute one statement or can it do multiple at once and return the latest?

The factor in this scenario is the second item, as the system executes only one statement; the column name/index does not matter. If the value has to be a string (text) and injection can be discovered below. The system adds the character ' before and after the text (codeblok 14). To exploit, the attacker adds the sentence ' OR 1 = 1; #. The character ' to end the condition, adds another condition that is always true, ';' ends the statement and '#' comments the rest of the query out. When the system executes, the attacker will receive all accounts from the database. If the attacker could inject their own statement, it would happened before the '#' letter (potential to delete tables).

```
1 || [...] WHERE " +field+ " = '';
```

Codeblok 14: The inserted value contains no text.

```
1 || [...] WHERE " +field+ " = '' OR 1 = 1;
   #;
```

Codeblok 15: The inserted value contains the attackers injection.

To secure the system and avoid injection, the technique is to implement prepared statements. Prepared statements replace values to be correct format before inserting it to the actual statement, hence the name prepared. The outcome format of the previous injection string in a prepared statement would be '\ OR 1 = 1; #'. The '\' character is added to escape the character in front of it. The injection no longer has an effect since the text is intact.

7.4 Docker

Docker technology is container-based and can virtualize down to the operating system level. It is not identical to a virtual machine since containerisation technology is not required to run on a hypervisor and the virtualisation method allows the containers to run on a shared host kernel instead a kernel of their own. The docker technology removes some of the overhead that the virtualizing operation systems are associated with[19]. This is the difference between using docker instead of virtual machines. However, they do not target the same problems. Docker can be used to isolate individual applications whereas virtual machines can isolate operation systems.

To create a docker container, a docker image has to be constructed. Docker images are layers built from a **Dockerfile** which basically is a file containing the instructions for that specific image. Every instruction is cached, so if an instruction is not changed, the cached instruction will be used. Using this technique requires more storage, however, it greatly increases the performance when building docker images. A simplified instruction change could be the instruction **COPY <file>**, and if the <file> specified has changed, compared to last time, the image would

be built.

In this project, all tiers from the multi-tier architecture can be virtualised by using containers as their physical server, but also the applications themselves. This means the web server runs in a container, which by itself is the client tier. In the application tier, server and MQTT server have their own separate container. At last the database in the data tier has also its own container. Each service mentioned is associated with a Dockerfile, which determines how the service is built and executed.

```

1  # Builder
2  FROM maven:3.6.0-jdk-8-alpine AS builder
3
4  WORKDIR /build
5  COPY ./ /build
6  RUN mvn clean package
7
8  # Runner
9  FROM openjdk:8-jre-alpine
10 EXPOSE 7000
11 COPY --from=builder /build/Core/target/Core-1.0-SNAPSHOT.jar /app/app.jar
12 CMD exec java -jar /app/app.jar

```

Codeblock 16: Backend (server) Dockerfile

The Dockerfile to build the image for the server can be seen in codeblock 16. The `FROM` instruction specifies the base image that the Dockerfile inherits from. Afterwards it states the working directory by the instruction `WORKDIR` and copies files from current directory to the working directory into the Docker image by using the `COPY` instruction. This is used to build the server. The next part is about running it. The `EXPOSE` instruction is to expose a port in the container. The last instruction is `CMD` and whenever an instance is being ran it will be executed.

Multi-stage builds

The Dockerfile in codeblock 16 is a multi-stage build. Multi-stage build means using more than one image to build the desire result. In the mentioned codeblock, one is use for building/-compiling and the other one for running. The beneficial of this is primarily the size of image, however, this also affect the processes within the image since two different base images can be involved. Furthermore, it is not only the server that is multi-stage built. The angular web server is multi-stage built and can be seen in codeblock 17. The reason for the size decrease is, that the build time dependencies are not part of the final image.

```

1  # Builder (STEP 1)
2  FROM node:alpine AS builder
3
4  RUN apk update && apk add --no-cache make git
5
6  WORKDIR /build
7
8  RUN npm install -g @angular/cli
9
10 COPY ./package.json ./package-lock.json ./
11 RUN npm install
12
13 COPY . .
14
15 RUN ng build
16
17 # Runner (STEP 2)
18 FROM nginx:alpine
19
20 RUN rm -rf /usr/share/nginx/html/*

```

```

21 |
22 | COPY --from=builder /build/dist/frontend /usr/share/nginx/html
23 | COPY --from=builder /build/nginx.conf /etc/nginx/nginx.conf
24 |
25 | CMD ["nginx", "-g", "daemon off;"]

```

Codeblock 17: Angular + web server Dockerfile

The building process of the angular project is in two parts. In *STEP 1*, it runs a few instructions to achieve the required system to build the angular project. To compile/build the angular project it uses commands from NodeJS and calls that image "builder". In *STEP 2*, **nginx:alpine** image is low size application to serve HTML, CSS and JavaScript files, separating the concerns of building and running. The result of building the angular is copied to the running environment and in this particular case of angular, the image building angular is almost 500 MB, whereas the final image, the run image, is below 20 MB and that is a significant decrease in size.

Docker Compose

Applications can have many docker containers to fulfil its purpose, just as this project. It has containers in all tiers, from web-server to database and it would take time to run each service or Dockerfile individually one by one. To accomplish this, a feature called *docker-compose* exists and to use docker-compose, services are defined within a YAML file. A service in the docker-compose.yml file is defined either by an image or Dockerfile, and additional configurations which are optional. In this project, the services are **frontend**, **backend**, **mqtt** and **db** (database).

- **build**, defines which image or Dockerfile to use alongside arguments for the build process.
- **ports**, defines which port the container to expose. Can also be written within the Dockerfile. The web service uses port 80 and Java uses port 7000.
- **environment**, if supported, defines variables that are accessible from the service at run time. Some variables may be specific depending on the image being used, like Postgres.
- **volumes** are used to map directories from the host machine to the docker container. This can be used to allow for persistent storage, because docker containers, by default, do not save information that has been written to the containers, when the container is deleted. In the case of the database service, it is also used to run the database migration file, by mapping the migrations directory to a specific directory inside the container, because the developers of the image implemented a mechanism that runs all SQL files inside the `/docker-entrypoint-initdb.d/` directory of the container.

An example of the docker-compose.yml file can be seen at codeblock 18. The example contains the configurations mentioned above.

```

1 | version: '3'
2 | services:
3 |   frontend:
4 |     image: sdugrp4/vehicle-telematics:frontend
5 |     build:
6 |       context: ./frontend
7 |     ports:
8 |       - "80:80"
9 |   backend:
10 |    [...]
11 |    environment:
12 |      - env=prod
13 |    [...]
14 |   mqtt:

```

```

15     [...]
16     db:
17         image: postgres
18         ports:
19             - "5432:5432"
20         environment:
21             - POSTGRES_USER=postgres
22             - POSTGRES_PASSWORD=password
23             - POSTGRES_DB=4semdb
24         volumes:
25             - "./migrations:/docker-entrypoint-initdb.d/"
26             - db_data:/var/lib/postgresql/data
27
28 volumes:
29     db_data:

```

Codeblock 18: Excerpt of docker-compose.yml and having at least one example of every configuration.

Now, how to call the `docker-compose`? Docker-compose is a command line followed by secondary arguments for it. Using the `build` argument, builds all the images while `up` builds and starts all containers with their image. The opposite is `down` and shuts the container down and deletes them; a fresh start. In addition, images can be pushed to the docker hub by the using `docker push` and the images are available on Docker Hub without being able to access the source code of the project.

By having each service in a separate container allows easy separation of the application and transferring of separations into physical servers.

7.4.1 MQTT

MQTT is a publish/subscribe-messaging protocol and it stands for *Message Queuing Telemetry Transport*⁷. By using Docker it is simply a docker-image that can be configured to the desired behaviour and authentication. Codeblock 19 shows the docker configuration to run a MQTT server (mosquitto is the type). The volumes are linked to various files containing the actual configurations such as username and password used to connect to the server. When it is up and running users can either publish or subscribe on a topic. Publishing means sending a message. Subscribing is about catching messages when they are published. The topic `/id/alarms/speeding` (id is substituted with an actual id) used in the application. Additionally, when subscribing, the character `'#'` can be added to catch everything, for example `/id/alarms/#` catches all messages no matter the type of an alarm.

```

1 | version: '3'
2 | services:
3 |     [...]
4 |     mqtt:
5 |         image: eclipse-mosquitto
6 |         volumes:
7 |             - "./mqtt/mosquitto.conf:/mosquitto/config/mosquitto.conf"
8 |             - "./mqtt/acl.conf:/mosquitto/config/acl.conf"
9 |             - "./mqtt/password.conf:/mosquitto/config/password.conf"
10 |         ports:
11 |             - "1883:1883"
12 |     [...]

```

Codeblock 19: Snapshot of the MQTT configuration in the docker-compose.yml.

The way the server handles MQTT is by having a wrapper class that wraps the default implementation of Java's MQTT library. The wrapper class is `MqttServiceProvider.java`

⁷Source: <https://mqtt.org/faq>

and implements an interface `IMqttService`, see codeblock 20. It contains numerous methods communicate with the MQTT server.

The `StaticMqttTopic` `VariableMqttTopic` enums were created, so users of the interface would not have to use strings, to specify topics. This restriction means there is full control over what topics are being used in the system, and mistakes like misspelling a topic are avoided. The difference between the two enums is that the `VariableMqttTopic` requires an argument, which is the device id, and the other does not. This is how multiple different devices can send messages to the MQTT server, without specifying their device id in the message payload. The backend server parses the topic, to see which device id the message originates from, and what type of message it is. The types of messages include what kind of data (velocity, GPS, acceleration), and alarms.

The other enum, `StaticMqttTopic`, is used to subscribe to messages of a certain type, from all devices.

```
1 public interface IMqttService {
2     void publish(StaticMqttTopic topic, String message);
3     void publish(VariableMqttTopic topic, UUID deviceId, String message);
4
5     void subscribe(StaticMqttTopic topic, ThrowingBiConsumer<String, String>
6         callback);
7     void subscribe(VariableMqttTopic topic, UUID deviceId, ThrowingBiConsumer<
8         VariableMqttTopic, String> callback);
9
10    void unsubscribe(VariableMqttTopic topic, UUID deviceId);
11    void unsubscribe(StaticMqttTopic topic);
12
13    void connect();
14    void disconnect();
15    boolean isConnected();
16 }
```

Codeblock 20: Excerpt of `IMqttService.java` without any Java documentation.

8 Test

Testing is an important part of software development, and they can be used during most stages of the software development life cycle, for example during development, testing, and maintenance. Testing is part of the verification and validation process, where verification asks the question: *Are we building the product right?* and validation asks the question: *Are we building the right product?*[20]

In theory it should be enough to just make sure all requirements are met, but requirements are not always accurate, or if they are vaguely stated, may be interpreted incorrectly. This is one of the reasons why validation is also important, because one of its goals is to ensure, that the software meets the expectations of the customer.

The three kinds of tests that will be covered in this section are unit tests, integration tests, and system tests.

Tests are not enough to prove that the system is bug free, or that it will behave as expected in every possible scenario. Edsger Dijkstra, famous computer scientist, formulated it thusly: *"Testing can only show the presence of errors, not their absence"*[8]

8.1 Unit testing

Unit testing is the testing of small components of systems, such as objects or functions, depending on the language that is used. As previously mentioned in this report, XP is used during development, and unit testing is one of the elements of Extreme Programming. Having many unit tests is generally a good thing, but the collection of unit tests, also called a test suite, should run fast, so as to not burden the developer, causing them to skip tests, in favour of faster development. Unit tests, and testing in general, also improve maintenance, because developers can confidently make changes in a large code base, and quickly see if the change they did, made a test fail.

The testing framework JUnit 4 is used to automate testing. JUnit provides several annotations, some of which are used in this project:

BeforeClass is used to mark a method, that will run before the test class is instantiated by the testing framework. When used together with the **assumeTrue** method, it skips all tests in the class. This is useful for integration tests, which will be explained in the integration test section.

Before is used to mark a method, that will be run before each test method.

Test is used to mark a method as a test method, that will be run by the testing framework.

A recommend pattern to use, when creating tests, both unit and integration, is the Arrange, Act, Assert (AAA) pattern. AAA divides test code into three areas: the first, arrange, is used to *arrange all necessary preconditions and inputs*. The second, act, is used to *act on the object or method under test*. The third, assert, is used to *assert that the expected results have occurred*.

The benefit of structuring test code like this, is that it makes it clear for the developer, and potential future readers, what is going on in the test, as it separates setup from the actual test.[12]

The naming scheme used for unit test methods is as follows:

UnitOfWork.StateUnderTest.ExpectedBehavior. This naming scheme makes it easy for a reader to see what is being tested, what kind of input is being given to the test, and what the expected behaviour is. When running all the tests in a project, some of them might fail, and when using this naming scheme, it is easier to tell what has probably gone wrong in the test, just by reading the name of the test method, rather than having to look at the actual test code[18].

8.1.1 Mocking

Sometimes a class that is being tested, depends on something that is not relevant for logic itself that is being tested. This could be a connection to a database, which is only used to retrieve data. To work around this, mock objects are used. Mock objects are objects that behave according to the interface the class under test depends on, but it does not, in the case of databases, make an actual connection to a database. In the case of this project, this is commonly used to test service classes, that depend on repository classes. Using mocks is only possible, when the class under test is coded in a way, that makes it testable. This means, that the class under test should not be instantiating its own dependencies, they should be passed as a constructor argument, so it can be changed during testing.

The library used for mocking is called Mockito. The library provides certain annotations and methods, that are used to create mock objects. One of the most important methods is Mockito.when. This method is used to specify the return values of certain method calls, so when the class under test calls the method, its return value is known beforehand, and the expected logic resulting from that value can be tested.

8.1.2 Example of unit test

The purpose of this unit test, is to make sure, that when a user requests data from a device they do not own, the service class should throw a `NotFoundResponse` exception.

```

1 public class VehicleServiceTest {
2     @Mock
3     private IRepository vehicleRepository;
4     @Mock
5     private IAccountRepository accountRepository;
6
7     private IVehicleService vehicleService;
8
9     @Before
10    public void setUp() {
11        MockitoAnnotations.initMocks(this);
12
13        vehicleService = new VehicleService(vehicleRepository,
14            accountRepository);
15    }
16
17    @Test
18    public void getData_UserDoesntOwnDevice_ShouldThrowException() {
19        // Arrange
20        Mockito.when(vehicleRepository.isUserOwnDevice(anyString(), anyString(
21            )), thenReturn(false);
22        NotFoundResponse ex = null;
23
24        // Act
25        try {
26            vehicleService.getData(UUID.randomUUID(), null, null, null);
27        } catch (NotFoundResponse nfr) {
28            ex = nfr;
29        }
30
31        // Assert
32        Assert.assertNotNull(ex);
33    }
34 }
```

Codeblock 21: Unit test with mocks from VehicleService.

Line 1-4 declares variables that are to be mocked, and these are initialised in the `setUp` method. On line 13 the vehicle service class is instantiated, passing the two mocked instances as constructor arguments.

In the test method, the first thing that happens, is the mocked object is setup. When the `doesUserOwnDevice` method is called, with a string as the first argument, and a string as the second argument, it should return false. An exception of type `NotFoundResponse` is declared and assigned to null, this is used to store the thrown exception from the method under test.

Next step is to act on the method under test, `getData`. In this test, the arguments does not matter, so most of them are null. The exception is caught, and stored in a variable

Finally, it is asserted, that the exception is not null. If everything went as expected, the assertion is true, and the test passes.

8.2 Integration testing

Integration tests are used to test, that interaction between independent pieces of software work as expected, when connected together. The AAA pattern and naming scheme mentioned in the previous section also apply to integration tests. In this project, integration tests are used to test that the third party API's work as expected, and that the MQTT server is processing messages as expected.

8.2.1 Examples of integration testing

In codeblock 22 the `@BeforeClass` annotation is used to skip all tests in that class, in case the MQTT server is not running on the developer's computer. If the tests ran, while the MQTT server was down, they would all fail, but that fail would not be meaningful because it does not say anything about the system, or the code that interacts with the MQTT.

```

1 | @BeforeClass
2 | public static void checkMqttConnection() {
3 |     IMqttService mqttService = new MqttServiceProvider();
4 |     try {
5 |         mqttService.connect();
6 |     } catch (RuntimeException e) {
7 |         // do nothing
8 |     }
9 |     Assume.assumeTrue(mqttService.isConnected());
10| }

```

Codeblock 22: Use of BeforeClass

The test shown in codeblock 23, tests that the rate limit functionality works as expected.

```

1 | @Test
2 | public void getSpeedLimit_SlowRequests_NoRateLimit() throws
3 |     InterruptedException {
4 |     // Arrange
5 |     GpsCoordinates limit60Coordinate = new GpsCoordinates(55.375049, 10.425339)
6 |         ; // Niels Bohrs Alle, near Campusvej. Limit should be 60
7 |     GpsCoordinates limit110Coordinate = new GpsCoordinates(55.353179,
8 |         10.375056); // On motorway. Limit should be 110
9 |
10|    // Act
11|    short speedLimit60 = speedLimitService.getSpeedLimit(limit60Coordinate);
12|    TimeUnit.SECONDS.sleep(rateLimit + 1);
13|    short speedLimit110 = speedLimitService.getSpeedLimit(limit110Coordinate);
14|
15|    // Assert
16|    Assert.assertEquals(60, speedLimit60);
17|    Assert.assertEquals(110, speedLimit110);
18| }

```

Codeblock 23: Integration test from SpeedLimitService

Two different coordinates, that should resolve to two different speed limits, are setup in the arrange part. In the act part, the requests are made, but the program waits a number of seconds

between each request, to simulate slow requests. Finally, in the assertion part, the expected values are compared to the actual values.

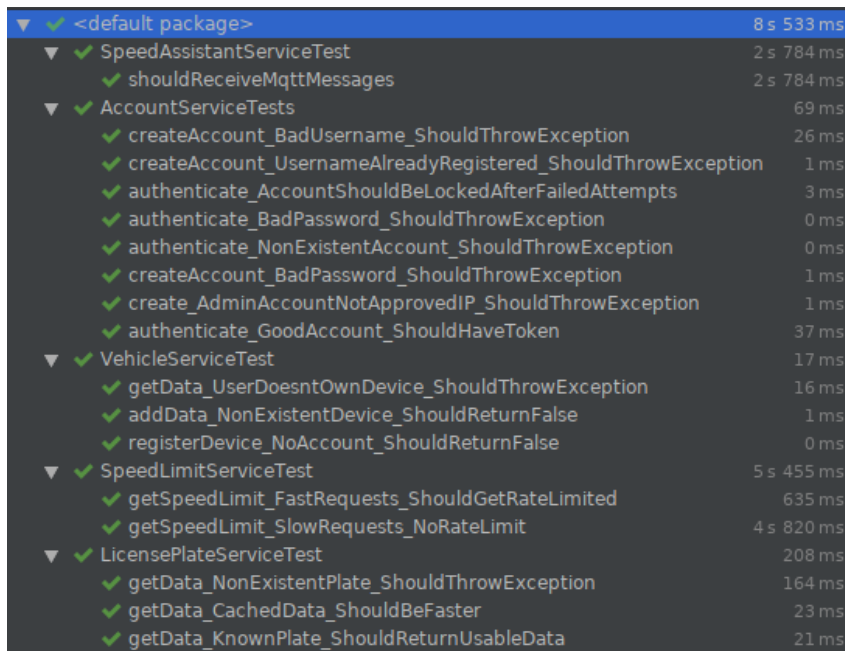
8.3 System testing

System testing is done on the completely integrated system, to make sure the entire system works as it should. In the context of this project, this means creating users, adding vehicles, using the Intelligent Speed Assistant, viewing past data, and performing administrative tasks. This is a manual test.

All these features work as expected. All the requirements, except RQ9 and RQ10, are met.

8.4 Passing tests in the project

This project has 17 tests of which 15 are unit tests, and 3 are integration tests. It takes about 8 seconds and 533 ms to run all the tests, where 6 seconds of that is artificial wait time, because two of the integration tests depend on the passage of time.



▼ ✓ <default package>	8 s 533 ms
▼ ✓ SpeedAssistantServiceTest	2 s 784 ms
✓ shouldReceiveMqttMessages	2 s 784 ms
▼ ✓ AccountServiceTests	69 ms
✓ createAccount_BadUsername_ShouldThrowException	26 ms
✓ createAccount_UsernameAlreadyRegistered_ShouldThrowException	1 ms
✓ authenticate_AccountShouldBeLockedAfterFailedAttempts	3 ms
✓ authenticate_BadPassword_ShouldThrowException	0 ms
✓ authenticate_NonExistentAccount_ShouldThrowException	0 ms
✓ createAccount_BadPassword_ShouldThrowException	1 ms
✓ create_AdminAccountNotApprovedIP_ShouldThrowException	1 ms
✓ authenticate_GoodAccount_ShouldHaveToken	37 ms
▼ ✓ VehicleServiceTest	17 ms
✓ getData_UserDoesntOwnDevice_ShouldThrowException	16 ms
✓ addData_NonExistentDevice_ShouldReturnFalse	1 ms
✓ registerDevice_NoAccount_ShouldReturnFalse	0 ms
▼ ✓ SpeedLimitServiceTest	5 s 455 ms
✓ getSpeedLimit_FastRequests_ShouldGetRateLimited	635 ms
✓ getSpeedLimit_SlowRequests_NoRateLimit	4 s 820 ms
▼ ✓ LicensePlateServiceTest	208 ms
✓ getData_NonExistentPlate_ShouldThrowException	164 ms
✓ getData_CachedData_ShouldBeFaster	23 ms
✓ getData_KnownPlate_ShouldReturnUsableData	21 ms

Figure 22: All tests

9 Discussion

Throughout the project the group has been facing different decisions, comparisons, problems, and various new areas, which this chapter will explore and explain the logic behind.

9.1 The Process

In this subsection, the benefits and consequences of the processes the group has choose to take and been through to develop the application will be discussed. The methods chosen are described in section Methods & Approach.

During the development of the product, the group used different software techniques, processes, and methods to communicate and plan the advancement of the project. The technique primarily used is the agile process whereas the scrum method as a tool to support and enhance the agile foundation. Yet, the full potential of scrum cannot be achieved, since not all the original concepts of scrum benefits the project, such as scrum-buts. These scrum-buts is usually not avoidable when working on a school project and cannot follow the scrum guide to the book. Additionally, the meeting everyday concept, which was been changed to meet every week. The concepts that do aid for example are scrum-master, product owner, divide into sprints, and the overridden rule, a meeting every week (instead of daily meetings). During the project the group has been using a backlog to distribute work and deadlines among the members of the group and give a overview of the process. The scrum-master changed between the group members. The product owner exists, as a result of the group itself was responsible for the development and had the most knowledge of the product.

The length chosen for sprint was two weeks. This sprint length was very appropriate, as the group never felt like they had too much, or too little work to do in each sprint. The only exception to the two weeks sprints, was the fourth and final sprint, because it was extended to include the last four days before the project had to be submitted.

Sprint reviews and sprint retrospectives were not used, instead the group thoroughly planned at the start of each sprint. During these planning sessions, the group decided what tasks should be done in the new sprint, and if necessary, move an uncompleted or long running task to the next sprint.

Furthermore, another technique used was Extreme Programming (XP) to assist the development of the software, but the consequences affected the groups work-flow in the development area. The group took not the full advantage of XP since not all the practises were pursued, but a few are worth mentioning.

Pair programming derives from XP and under the process of building the system, this practise was available. Not every task was developed in pairs since task size may vary and some practises are beneficially and few are not. When two people are doing a task together, it completely ensures that two people have knowledge of that specific task and the knowledge is not encapsulated by a single person. However, it is also more time-consuming than. and functionality to limit the project within a boundary. being alone, by extending the amount of man hours required to develop features; a single person could probably develop the feature. In contrary, the quality of the code will be better because written code goes through the mind of two programmers instead of one.

In the process of developing the software, UML was used to aid in communication of software design, architecture and general development decisions. The groups decision to only create the UML models and diagrams that were deemed beneficial was a good one, as it led to more time being available for other activities, rather than spending time creating extremely detailed models, that can quickly get outdated, or would not be looked at.

9.2 Technology

Today, tons of different technologies exist to achieve certain milestones, but could a technology not used in this project been particular better than those being used? At the hardware level the group used an Arduino to produce a mediator to connect between car and backend. This was done by sending messages between through a MQTT server in the application tier, however, an alternative solution would be using a web-server on the Arduino or a standalone web-server the Arduino can connect to which is identical to the backend. That way, similar to how the angular web-server and backend communicate with each other, requesting information and transmitting commands through an API could probably have accomplish the exact same end-goal, except it limits the work the Arduino can do while having a web-server running. In addition, the backend needs an IP address for every Arduino before it can send commands or error messages, and the other way around with the standalone concept. This alone is time consuming to examine on how to build and may not even be possible since the backend has to go through the internet, not locally in a network. Another technology, that was an option, is the Raspberry Pi instead of the using an Arduino. It will allow us to use an operation system and develop in a familiar programming language while running multiple programs at once. Nevertheless, it is not required to run multiple programs for this system to function because the Arduino only has to run one program and that is the one given. A lot of overhead to each of the alternatives mentioned because HTTP has more overhead compared to the protocol used by the message queue broker (MQTT).

The topic on the Arduino continues. The group discussed about whether the Arduino is in the right place of the multi-tier architecture. Currently, is it placed within the application tier, but the Arduino is an interface from the client side to access their vehicle's data. It was represented that the client tier could contain the Arduino as well, since it is a gateway for the user to connect the car to the system developed. Additionally, the logic in the Arduino is not directly related to business, but required information for the system to function and it is the client's choice if they want to contribute it and make it visible through a web-page.

Furthermore, the technology between the angular-web and backend is either the REST API or web socket. MQTT supports web sockets, so in theory frontend could directly connect to it, but imposes a security risk since the connection would not go through the authentication functionality on the backend, allowing anyone with the knowledge of the MQTT address and the credentials to connect to it, and read all values from all vehicles in the system. Therefore this is not available nor applied.

9.3 Requirements

The requirement *"It should be possible for an operator remotely to set an alarm on custom parameters for any of the acquired sensor data."* was not included in the project, because the group assessed that implementing the functionality in a satisfying manner, would take too much time. The requirement was marked with "Could Have" priority from the start, so it did not come as too big of a surprise to the group, that it would have to be discarded. The next best thing, which was implemented, is the speeding alarm. Parts of the speeding alarm is hard coded into the system, which is not great, but it is better than no alarm capability.

The other requirement that was not included is *"It should be possible to analyse driving behaviour, possibly based on characteristics such as acceleration, braking, speed during turns, distance driven, frequency of driving, and time of driving."* because the word analyse is a painful and challenging word to reach. Without any experience, analyse is a very abstract word and the word may vary depending on background. Analyse in this scenario means using simple mathematics to construct a function/diagram to view the data. Analyse in the real world may mean data optimisation, or detecting diversity behaviour from vehicle data in unique locations. The main reason it was not implemented is the lack of knowledge to accomplish a more complex

analysis and that the requirement was marked with "Could Have" priority from the beginning, so higher priority requirements come before an optional requirement.

9.4 Known Limitations & Proposed Solutions

This project is primarily based around tinkering with a car, and its OBD-II and internal CAN bus system. However, when the group was halfway through the time period, and done with developing the system regarding all the critical implementations except the connection between car and Arduino, a Point of Failure happened. Of all the members, only one had a car available to try the OBD-II link. The owner of that car went through multiple states of severe illness and was hospitalised. It was not possible to borrow a car since messing with OBD-II can lead to disabling the car or incapacitate the internal system which will result in errors, especially when expertise in practice is equal to none. A solution to this would have been to have multiple cars at the group's disposal, but that is not a possible statement.

When testing the application starting from register an account, adding a car, and see historical/live data, not a single limitation occurred.

9.5 Future Improvements

An improvement to the system, would be to implement the custom alarm parameter requirement. It should be implemented in such a way, that new parameters could easily be added. The group discussed one way of designing it, which included a type of command system, where the user could choose from a list of parameters, a compare function, and a set of values that the compare function must be true or false for. Using this system, the speeding alarm could be implemented by choosing the car's velocity from the parameter list, using a integer compare function, and making it so that the alarm must activate, when the velocity is greater than the speed limit.

A simple feature to improve the system is the feature to name the car you are adding to the car management system. Currently, it tells you what type of car it is, but it would be more appropriate to give it a name such as "Mom's Car" instead of remembering each and every car on the account. A convenient improvement to better User Experience.

9.6 Summary

Through proper software stages, such as analysis, design, and implementation, a minimal IT system was indeed realised. The system reflects the (upcoming) EU-laws in telemetry that contain a black-box with historical data, web-page that shows historical data or live data to a user, able to control hardware components from backend to send various signals to the customer, and at last, a service that allows external stakeholders, for example the insurance companies to access the backend API. Albeit a different approach could have changed the outcome and possibly improved it, yet the system fulfils many of the business requirements to a remarkable degree.

10 Conclusion

In this project the primary goal was to solve the (upcoming) laws in the European Union (EU) with an IT system, that all vehicles have to carry a telemetry device that records every vehicle data and monitors the velocity to increase security on the road and assisting at bringing the fatal number down. The project came with predefined requirements, which were mandatory, and they were elements that the final product must comply with. The given requirements were that the product must contain a web based interface, the interface should present data from an industrial process, and control the process. Adjacent to the mandatory requirements, other requirements were added to define the scope of the project.

The system has a role-based authentication, where clients can register and login. When a user is logged into the system, they can add vehicles to their account, using the vehicle's license plate, and the device id of the Arduino.

When a vehicle has been added to the account, the user can view live data, if the device and vehicle are active. When using the vehicle, and the ISA is active, the device will activate a LED on the device. The live data, that the user can view, is the latest received velocity of the vehicle, the latest received speed limit based on their GPS location, and a view of a map, with a marker placed on their location. This LED is the industrial process that is being controlled. After live data has been recorded, the user can view the historical data records.

If an account has the role of administrator, they have access to the admin dashboard of the system. On this page, the administrator can access historical data of all clients in the system, and access the account management, where they can see all accounts.

It can be concluded that, based on the documentation and development of the project, the group has amassed new information about the interplay between hardware and software, in an industrial setting. The solution produced in this project addresses the pre-handled requirements, stakeholder requirements, and the group's own requirements, and can be regarded as achieved.

11 References

Scientific References

- [3] Bass, Len; Clements, Paul, and Kazman, Rick. *Software architecture in practice*. 3rd ed. SEI series in software engineering. Addison-Wesley Professional, 2012.
- [6] Corrigan, Steve. *Introduction to the Controller Area Network (CAN)*. Texas Instruments, 2016.
- [8] Dijkstra, Edsger W. *The Humble Programmer*. Vol. 15. 10. New York, NY, USA: ACM, Oct. 1972, pp. 859–866. DOI: 10.1145/355604.361591. URL: <http://doi.acm.org/10.1145/355604.361591>.
- [11] Gopi, Satheesh. *Global Positioning System: Principles and Applications*. Tata McGraw-Hill, 2005.
- [16] Manickam, Piram; Sangeetha, S., and Subrahmanya, S. V. *Component-Oriented Development and Assembly*. CRC Press, 2014. ISBN: 978-1-4665-8100-5.
- [20] Sommerville, Ian. *Software Engineering Global Edition*. 10th ed. Pearson Education Limited, 2016.
- [22] Zurawski, Richard. *The Industrial Information Technology Handbook*. CRC Press, 2004.

Technical Papers

- [1] Ambler, Scott W. *Agile Modeling (AM): Effective Practices for Modeling and Documentation*. 2019. URL: <http://agilemodeling.com/>.
- [2] Arduino. *What is Arduino?* 2019. URL: <https://www.arduino.cc/en/Guide/Introduction>.
- [4] B&B Electronics. *OBD-II Background*. 2019. URL: <http://www.obdii.com/background.html>.
- [5] Beck, Kent et al. *Manifesto for Agile Software Development*. URL: <https://agilemanifesto.org/>.
- [7] CSS Electronics. *OBD2*. 2019. URL: <https://www.csselectronics.com/screen/page/simple-intro-obd2-explained/language/en>.
- [9] European Transport Safety Council. *Intelligent Speed Assistance (ISA)*. URL: <https://etsc.eu/intelligent-speed-assistance-isa/>.
- [10] European Transport Safety Council. *MEPs back life-saving vehicle safety standards in key vote*. URL: <https://etsc.eu/meps-back-life-saving-vehicle-safety-standards-in-key-vote/>.
- [12] Grigg, Jeff. *Arrange Act Assert*. 2012. URL: <http://wiki.c2.com/?ArrangeActAssert>.
- [13] Institute of Transport Economics, Norwegian Centre for Transport Research. *Driver Support Systems: Estimating road safety effects at varying levels of implementation*. URL: <https://www.toi.no/getfile.php/1335912/Publikasjoner/T%5C%C3%5C%98I%5C%20rapporter/2014/1304-2014/1304-2014-elektronisk.pdf>.
- [14] International Organization for Standardization. *Road vehicles - Controller area network (CAN)*. 2015. URL: <https://www.iso.org/standard/63648.html>.

- [15] International Organization for Standardization. *Road vehicles - Communication between vehicle and external equipment for emissions-related diagnostics*. 2016.
URL: <https://www.iso.org/standard/51828.html>.
- [17] National Coordination Office for Space-Based Positioning, Navigation, and Timing. *GPS Accuracy*. 2017.
URL: <https://www.gps.gov/systems/gps/performance/accuracy/>.
- [18] Osherove, Roy. *Naming standards for unit tests*. Apr. 3, 2005. URL:
<https://osherove.com/blog/2005/4/3/naming-standards-for-unit-tests.html>.
- [19] Red Hat. *What is Docker?* 2018.
URL: <https://www.redhat.com/en/topics/containers/what-is-docker>.
- [21] Sturgeon, Phil. *Understanding RPC Vs REST For HTTP APIs*. 2016.
URL: <https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis>.

12 The Final Product

This section shows the final iteration of the project's web interface. Since the project depends on the Arduino hardware, or a simulation program, a video has been made that shows the project in action. The video can be viewed at the following link:

https://drive.google.com/file/d/17R05BcV-bTYEcrPCtsVIhjZTTwPzW10_/view

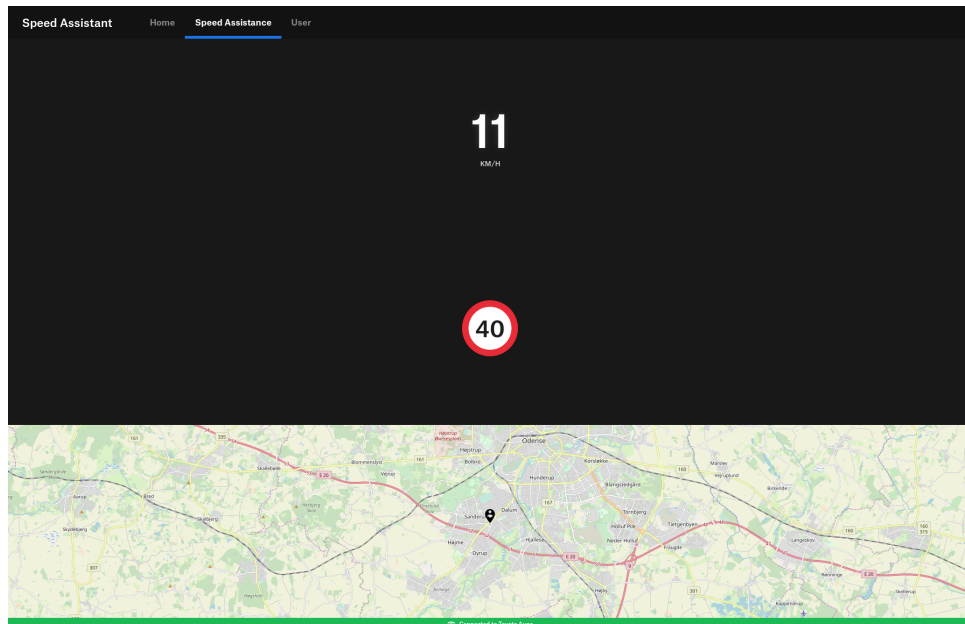


Figure 23: Desktop view of ISA

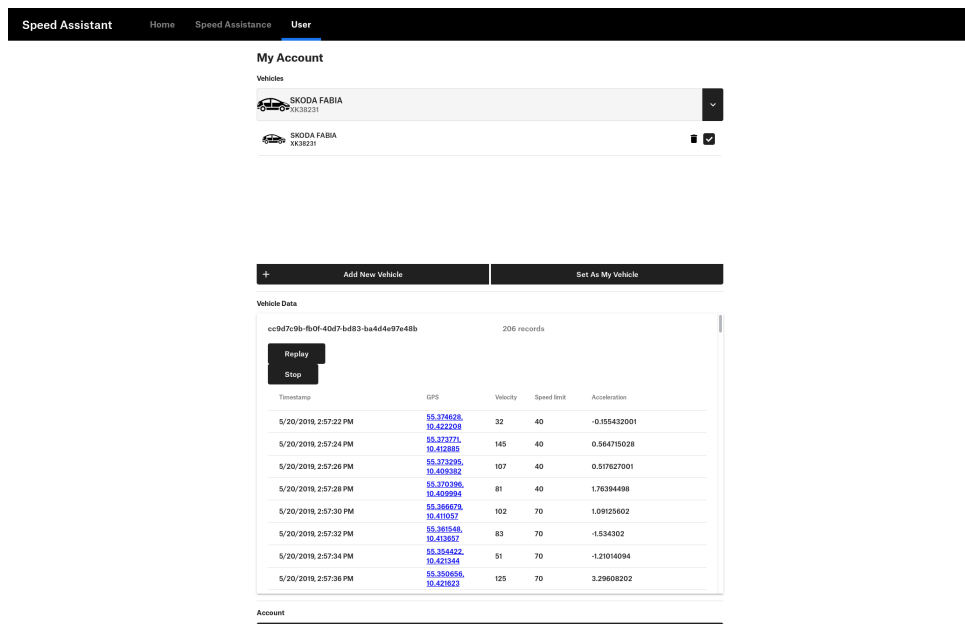
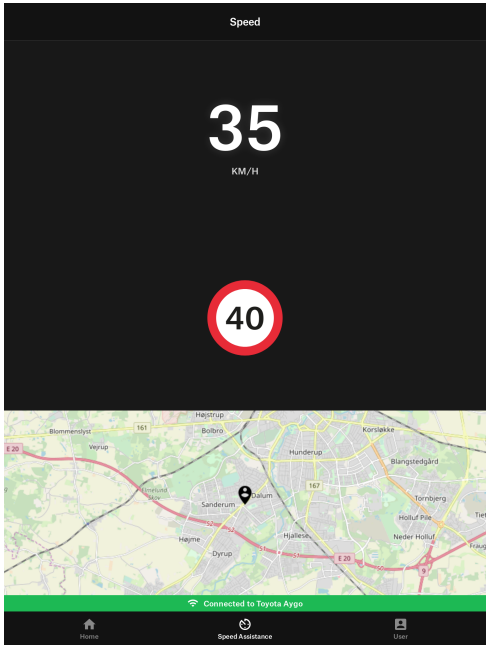
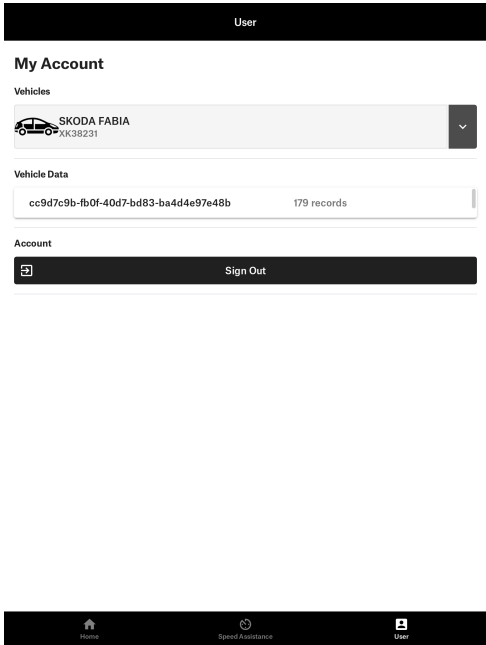


Figure 24: Desktop view of the user page



(a) Mobile view of ISA



(b) Mobile view of the user page

12.1 Source code

The source code of the project can be found on GitHub:

<https://github.com/lassestraberg/4sem>

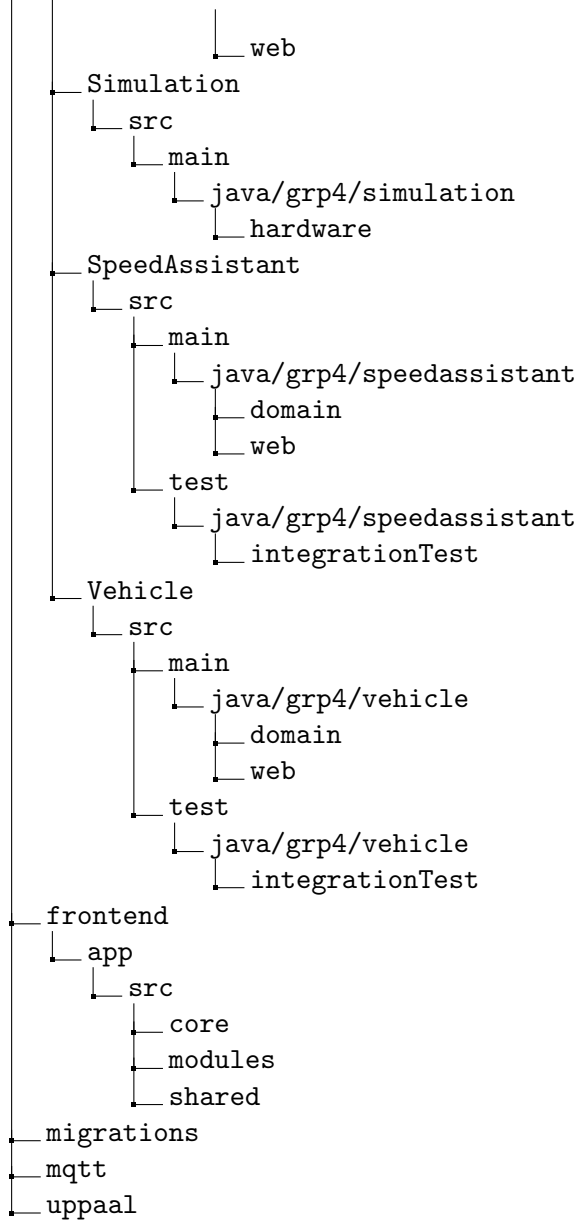
Source code tree

The source code tree shows the directory structure of all the source code.

```

4sem
├── arduino
├── backend
│   ├── Authentication
│   │   ├── src
│   │   │   ├── main
│   │   │   │   ├── java/grp4/authentication
│   │   │   │   │   ├── config
│   │   │   │   │   ├── domain
│   │   │   │   │   ├── util
│   │   │   │   │   └── web
│   │   │   └── test
│   │   │       ├── java/grp4/authentication
│   │   │       └── unitTest
│   ├── Common
│   │   ├── src
│   │   │   ├── main
│   │   │   │   ├── java/grp4/common
│   │   │   │   │   ├── config
│   │   │   │   │   ├── data
│   │   │   │   │   ├── domain
│   │   │   │   │   ├── lambda
│   │   │   │   │   ├── spi
│   │   │   │   │   ├── util
│   │   │   │   │   └── web
│   ├── CommonAuthentication
│   │   ├── src
│   │   │   ├── main
│   │   │   │   ├── java/grp4/commonauthentication
│   │   │   │   │   ├── config
│   │   │   │   │   └── domain
│   ├── CommonVehicle
│   │   ├── src
│   │   │   ├── main
│   │   │   │   ├── java/grp4/commonvehicle
│   │   │   │   │   └── domain
│   │   │   └── test
│   │   │       ├── java/grp4/commonvehicle
│   │   │       └── unitTest
│   └── Core
│       ├── src
│       │   ├── main
│       │   │   ├── java/grp4/core
│       │   │   └── config

```



Appendices

A Internal Appendices

A.1 Project Plan

