

NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY



FACULTY OF ENGINEERING

DEPARTMENT OF ELECTRONIC ENGINEERING

PROJECT REPORT

Name:	Mqaphelisi Mathonsi
Student Number:	N0188255Y
Course Name:	Design Project
Course Code:	TEE5003
Project Title:	Fuel Truck Anti-Tempering System
Supervisor:	S. Nhema
Year:	2023/2024



2 DECLARATION

I, Mqaphelisi Mathonsi, hereby declare that the project work shown in this report is my original work and that all the information that is not my work is properly referenced. Submission of the project report is a partial fulfillment of the requirements of Part V of the bachelor's degree in Electronic Engineering.



ACKNOWLEDGMENTS

Completing this project would not have been possible without the support and contributions of several individuals and groups. I am grateful for their help and guidance throughout the process.

Firstly, I would like to thank Miss S. Nhema, my project supervisor, for her unwavering support and guidance throughout the entire project. Her insights and feedbacks were instrumental in shaping my ideas and molding me into the researcher I am today.

I would also like to express my appreciation to my colleagues and friends - Pardon Ndlovu, Juba Nqobizitha and Munashe Madzudzo. Their support and their valuable inputs have been a great source of motivation and inspiration, not just for this project but also for my growth as an engineer. Their creativity and perspectives contributed to making this report informative and insightful.

My appreciation also goes to my family members, I am forever grateful for their unwavering support, encouragement, love and their sacrifices during the entire project duration.

I would like to express my heartfelt gratitude to the entire staff of the engineering department of this great institution. The availability of resources and expertise that they provided made the research journey more comfortable and fulfilling.

Last but not least, I appreciate the contributions of every individual who participated in the project in one way or the other. Your contributions, no matter how small, played a significant role in shaping the outcome of this research project.



3 ABSTRACT

This research project aimed to develop an innovative fuel truck anti-tempering system to mitigate safety risks and enhance logistical efficiency. The primary motivation was to address the pressing issues of fuel theft, spillage, and accidents during transportation. The system's design and development focused on real-time tracking of vital parameters, including location, fuel level, outlet valve state, tank weight, and pressure.

A bespoke system was designed and prototyped using cutting-edge technologies, including ESP32 microcontrollers, Node.js server, and MongoDB database. Sensors collected data, which was transmitted to the server for storage and analysis. A user-friendly React-based dashboard displayed the data in real-time, enabling prompt response to deviations from set values.

The results demonstrated the system's efficacy in real-time fuel truck monitoring, enabling swift response to anomalies and enhancing logistical efficiency. This research contributes to the development of intelligent transportation systems, offering a robust solution to fuel transportation challenges.



4 TABLE OF CONTENTS

2	DECLARATION	1
	ACKNOWLEDGMENTS	2
3	ABSTRACT.....	3
4	TABLE OF CONTENTS.....	4
	LIST OF FIGURES	8
	LIST OF TABLES	10
	LIST OF ABBREVIATIONS.....	11
5	INTRODUCTION	12
5.1	Introduction.....	12
5.2	Background.....	12
5.3	Problem Statement.....	12
5.4	Solution.....	12
5.5	Aim	13
5.6	Objectives	13
5.7	Justification.....	13
5.8	Conclusion	13
6	LITERATURE REVIEW	14
6.1	Introduction.....	14
6.2	Fuel Transportation Safety	14
6.3	IoT In Fuel Tank Monitoring.....	15
6.4	Weight Monitoring	16
6.5	Location Tracking.....	16
6.6	Fuel Level Monitoring.....	16
6.7	Valve Status Monitoring.....	17
6.8	Pressure Monitoring.....	17
6.9	Inter-Device Communication	17
6.9.1	Message Query Telemetry Transport (MQTT).....	17
6.9.2	Mosquitto	18



6.9.3	HTTP.....	19
6.10	Web Technologies	19
6.10.1	HTML And CSS	19
6.10.2	JavaScript.....	20
6.10.3	The Document Object Model.....	21
6.10.4	Node.js	22
6.10.5	Libraries and Frameworks	22
6.10.6	Database Technologies	23
6.10.7	The MERN Stack.....	26
6.10.8	The MERN Stack in IoT	26
6.11	Existing Systems.....	28
6.11.1	The Weighbridge	28
6.11.2	Fleet Management Systems (FMS).....	28
6.11.3	IoT-based Fuel Monitoring Systems.....	29
6.11.4	GPS-based Tracking Systems	29
6.11.5	Cloud-based Fuel Management Systems	29
6.12	Proposed Model.....	29
6.13	Literature Review of Components.....	30
6.13.1	ESP32.....	30
6.13.2	Ultrasonic Sensor	31
6.13.3	GPS Module.....	31
6.13.4	Load Cell.....	32
6.13.5	Pressure Sensor	32
6.13.6	Valve.....	32
6.14	Conclusion	33
7	METHODOLOGY	34
7.1	Introduction.....	34
7.2	Research Design and Approach.....	34
7.2.1	System Requirements Analysis:	34
7.2.2	System Design and Development	34
7.3	Hardware Components Selection	35



7.3.1	Sensors	35
7.3.2	Microcontroller	37
7.4	Software Components Selection	37
7.4.1	Microcontroller Firmware.....	37
7.4.2	Communication Protocol	37
7.4.3	Server-Side Application.....	37
7.4.4	Database	38
7.4.5	Front-End Application	38
7.4.6	Additional Considerations	38
7.5	System Design	38
7.5.1	Hardware Circuit Design	38
7.5.2	Firmware Algorithm Design	39
7.5.3	Database Design.....	40
7.5.4	Frontend Application Design.....	41
7.6	System Development and Integration.....	41
7.6.1	Hardware Integration	41
7.6.2	Microcontroller Firmware Development	42
7.6.3	Firmware – Hardware Integration.....	45
7.6.4	Database Schema Development.....	45
7.6.5	Server API Development	46
7.6.6	Front End Application Development	48
7.7	Testing and Validation Procedures	52
7.7.1	Hardware Unit Tests	52
7.7.2	Firmware Test	54
7.7.3	Firmware – Hardware Integration Test.....	54
7.7.4	Server API Tests	55
7.7.5	Front End Component Tests	56
7.7.6	System Integration Tests.....	57
7.8	Data Analysis Techniques	58
7.8.1	Descriptive Statistics.....	58
7.8.2	Statistical Anomaly Detection:	58



7.8.3	Visualization Techniques.....	58
8	RESULTS AND DISCUSSION	59
8.1	Hardware Unit Tests Results	59
8.1.1	Pressure Sensor	59
8.2	Firmware Test Results	60
8.3	Firmware – Hardware Integration Test Results	60
8.4	Server API Tests Results	61
8.5	Front End Component Tests Results	63
8.6	System Integration Tests Results	63
8.7	CONCLUSION AND RECOMMENDATIONS	76
8.7.1	Summary of Findings.....	76
8.7.2	Challenges Faced	76
8.7.3	Recommendations.....	77
9	REFERENCES AND BIBLIOGTAPHY	78
	APPENDICES	82
	Appendix A: Complete ESP32 Firmware Program	82
	Appendix B: Dashboard Software License.....	91
	Appendix C: System Users Data Schema.....	92
	Appendix D: Notifications Data Schema.....	93
	Appendix E: Jobs Data Schema.....	94
	Appendix F: Drivers Data Schema	95
	Appendix G: Trucks Data Schema	96
	Appendix H: System Users Router	97
	Appendix I: Trucks Router	101
	Appendix J: Drivers Router	105



LIST OF FIGURES

Figure 6.1IoT block diagram	15
Figure 6.2 HTML document and corresponding DOM representation	21
Figure 6.3: A relational database	23
Figure 6.4: Mongoose schema definition.....	25
Figure 6.5: MongoDB documents from schema in Figure 2.7.6.2	26
Figure 6.6: The MERN stack architecture	26
Figure 6.7: The MERN stack in IoT simple block diagram	27
Figure 6.8: Weighbridge	28
Figure 6.9: ESP32 pinout [image credits: RandomNerdTutorials.com]	30
Figure 6.10: Ultrasonic sensor working principle.....	31
Figure 6.11: GPS module.....	32
Figure 7.1 Research Design and Approach flow chart	35
Figure 7.2 Hardware circuit schematic	39
Figure 7.3 ESP32 Firmware Algorithm	40
Figure 7.4 Truck Database Design.....	40
Figure 7.5 Power supply voltage regulation	42
Figure 7.6 Imported libraries	42
Figure 7.7 Global variable declaration.....	43
Figure 7.8 Loop function	43
Figure 7.9 Setup function.....	44
Figure 7.10 HTTP request to send sensor data	45
Figure 7.11 Mongoose schema	45
Figure 7.12 Installed dependencies.....	46
Figure 7.13 Back-end folder structure	47
Figure 7.14 Server API entry point.....	48
Figure 7.15 Front-end application project structure	48
Figure 7.16 Application routes	49
Figure 7.17 Fetching truck data from the server.....	50
Figure 7.18 Detecting metric changes and raising alerts	51
Figure 7.19 Sending initial state to the server.....	51
Figure 7.20 Pressure sensor test code	52
Figure 7.21 Weight sensor test code	52
Figure 7.22 Ultrasonic sensor test code	53
Figure 7.23 Valve status sensor test code	53
Figure 7.24 GPS sensor test code	53
Figure 7.25 Wi-Fi test code	54
Figure 7.26 Firmware-Hardware integration test circuit wiring.....	55
Figure 7.27 Using Thunder Client to test API	56
Figure 7.28 Testing the AddJob form component	57
Figure 8.1 Pressure sensor readings.....	59
Figure 8.2 Loadcell weight readings.....	59



Figure 8.3 Ultrasonic sensor distance readings.....	59
Figure 8.4 Valve status sensor test results	60
Figure 8.5 Wi-Fi connection test results	60
Figure 8.6 Firmware successful compilation	60
Figure 8.7 Firmware-Hardware integration test results	61
Figure 8.8 HTTP GET request and response results on Thunder Client	61
Figure 8.9 NodeJS console results	62
Figure 8.10 MongoDB database after adding data	62
Figure 8.11 Sign-up page.....	63
Figure 8.12 Dashboard login page.....	64
Figure 8.13 System integration results: Dashboard	65
Figure 8.14 Registered trucks list	66
Figure 8.15 System integration test: Live data tracking	67
Figure 8.16 Registered Drivers list	68
Figure 8.17 FTATS Jobs list.....	69
Figure 8.18 Adding a new job.....	70
Figure 8.19 System integration test: Data anomaly notification	71
Figure 8.20 Prototype view 1.....	72
Figure 8.21 Prototype View 2.....	73
Figure 8.22 Prototype view 3.....	74
Figure 8.23 Image showing wires some sensors mounted on the underside of the prototype.....	75



LIST OF TABLES

Table 1 Hardware circuit pin connections	41
--	----



LIST OF ABBREVIATIONS

API	Application Programming Interface
IDE	Integrated Development Environment
IoT	Internet of Things
UI	User Interface
HTTP	Hypertext Transfer Protocol



5 INTRODUCTION

5.1 Introduction

This chapter introduces the research project, providing an overview of the context, motivation, and scope of the study.

5.2 Background

Fuel transportation is a vital component of various industries, including energy, logistics, and transportation. The demand for fuel is increasing globally, driven by population growth, urbanization, and economic development. As a result, the transportation of fuel has become a critical aspect of the global economy.

The fuel transportation industry faces significant safety risks and challenges, including fuel theft, spillage, and accidents. Fuel theft, in particular, is a major concern, with estimated annual losses ranging from \$5 billion to \$10 billion globally [1]. Fuel spillage and accidents also pose significant environmental and economic risks, resulting in soil and water contamination, property damage, and loss of life.

The lack of real-time monitoring and tracking capabilities exacerbates the safety risks and challenges associated with fuel transportation. Traditional methods of fuel transportation, such as manual monitoring and paper-based records, are inefficient and prone to errors. The need for innovative solutions to address these challenges has led to the development of real-time monitoring and tracking systems.

Real-time monitoring and tracking systems have the potential to revolutionize the fuel transportation industry by providing instant updates on fuel levels, location, and other vital parameters. These systems utilize cutting-edge technologies, including IoT sensors, cloud computing, and data analytics, to track fuel transportation in real-time. By enabling prompt response to deviations and anomalies, real-time monitoring and tracking systems can reduce fuel theft, spillage, and accidents, resulting in significant economic and environmental benefits.

5.3 Problem Statement

The transportation of fuel is prone to various safety risks and challenges, resulting in significant economic and environmental losses. The lack of real-time monitoring and tracking capabilities hinders prompt response to deviations and anomalies, exacerbating the issues.

5.4 Solution

This research project aims to design and develop a real-time fuel truck monitoring and alert system to address the safety risks and challenges associated with fuel transportation. The system will utilize cutting-edge technologies, including IoT sensors, cloud computing, and data analytics, to track vital parameters and enable prompt response to deviations.



5.5 Aim

The aim of this research project is to design and develop an innovative fuel truck anti-tempering system that enhances the safety and efficiency of fuel transportation.

5.6 Objectives

- To design and develop a real-time fuel truck monitoring and alert system
- To evaluate the system's performance and effectiveness
- To identify potential applications and limitations of the system

5.7 Justification

This research project is justified by the need for innovative solutions to address the safety risks and challenges associated with fuel transportation. The development of a real-time monitoring and alert system has the potential to reduce fuel theft, spillage, and accidents, resulting in significant economic and environmental benefits.

5.8 Conclusion

In this chapter, we have introduced the background, problem statement, solution, aim, objectives, and justification for the development of a fuel truck anti-tempering system. The next chapter will provide an overview of the current state of the art in IoT technologies and their applicability to truck monitoring systems.



6 LITERATURE REVIEW

6.1 Introduction

This literature review chapter provides an overview of the current state of research in the areas of fuel transportation safety, IoT technology, and real-time monitoring systems. It examines the key challenges and risks associated with fuel transportation, and reviews existing solutions and technologies that have been proposed or implemented to address these challenges. The chapter also explores the theoretical and conceptual frameworks that underpin the design and development of real-time monitoring systems, and discusses the key technologies and protocols used in IoT applications.

By reviewing the existing literature and research in this field, this chapter aims to provide a comprehensive understanding of the context and background of the project, and to identify the key research gaps and areas for further investigation.

6.2 Fuel Transportation Safety

Fuel transportation safety is a critical concern in the energy industry, as it involves the transportation of hazardous materials that can pose significant risks to people, the environment, and the economy. The transportation of fuel by road is particularly risky, as it involves the movement of large quantities of fuel over long distances, often through populated areas.

Several studies have highlighted the risks associated with fuel transportation, including the potential for accidents, spills, and theft. For example, a study by the National Tank Truck Carriers (NTTC) found that there were over 1,300 reported incidents involving tank trucks in the United States alone in 2020 [2]. These incidents resulted in millions of dollars in damages and several fatalities.

To address these risks, the fuel transportation industry has implemented various safety measures, including the use of double-hulled tanks, GPS tracking, and real-time monitoring systems. However, despite these measures, fuel transportation remains a high-risk activity, and there is a need for continued innovation and improvement in safety standards.

Several researchers have proposed the use of advanced technologies, such as IoT sensors and artificial intelligence, to improve fuel transportation safety. For example, a study by Zhang et al. proposed the use of IoT sensors to monitor fuel tank pressure and temperature in real-time, allowing for early detection of potential safety issues [3].

Overall, fuel transportation safety is a critical concern that requires continued attention and innovation. The use of advanced technologies, such as IoT sensors and real-time monitoring systems, has the potential to significantly improve safety standards in the fuel transportation industry.



6.3 IoT In Fuel Tank Monitoring

The Institute of Electrical and Electronics Engineers (IEEE) describes Internet of Things (IoT) as a network of items each embedded with sensors which are connected to the internet [4]. A block diagram of an IoT system is shown in Figure 6.1.

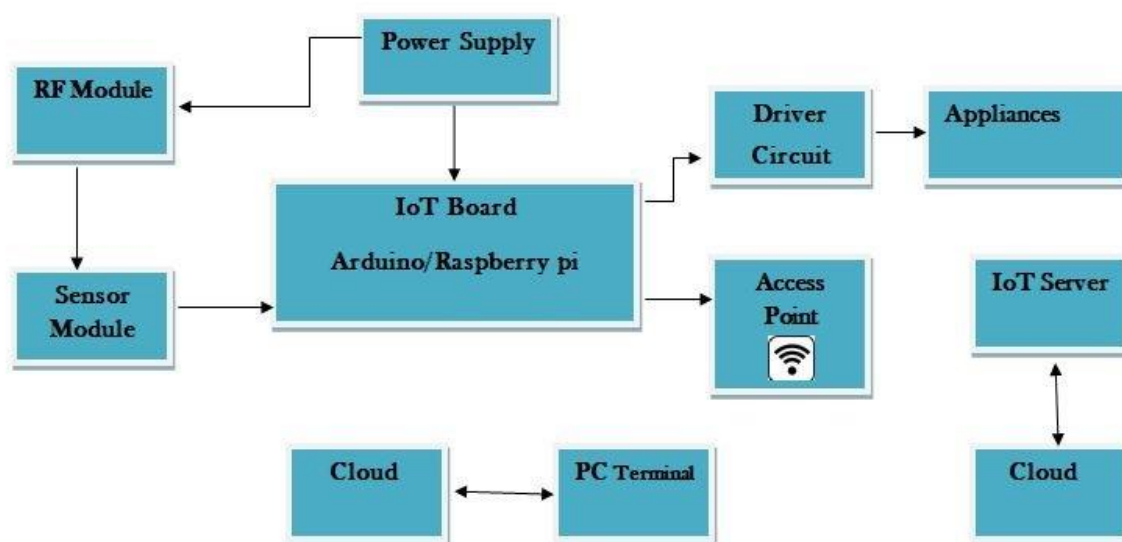


Figure 6.1 IoT block diagram

IoT technologies have been widely adopted for various applications in transportation, including real-time tracking of vehicles, cargo, and drivers. IoT solutions for truck monitoring rely on embedded sensors and location-based services to collect data and transmit it to a central database or cloud-based platform.

IoT-based truck monitoring systems can track vital data points such as speed, location, and fuel level, using GPS and other embedded sensors. This data is then processed, analyzed and transmitted via mobile or web interfaces to drivers, managers or support teams, keeping them informed about the status of the truck in real-time.

In a study by S. U. Khan et al [5], the authors conducted a literature review to summarize existing research methodologies for speed monitoring systems in vehicles. They proposed an IoT-enabled speed monitoring system designed to overcome the limitations of traditional systems. Similarly, M. S. Islam et al [6] proposed a fuel activity monitoring system that uses IoT to prevent fraud at petrol stations, benefiting both private car owners and truck owners.

Another area of interest in IoT in truck monitoring is refrigerated truck monitoring. The paper by S. Zheng et al [7] proposed a real-time monitoring and controlling system based on IoT for refrigerated trucks. The system aims to monitor temperature, humidity, and location of the truck to ensure the safety and quality of transported goods.



While there is significant interest in the application of IoT in truck monitoring, challenges still exist, such as security issues, interoperability, and integration with existing systems. S. Nieto et al [8] discussed the challenges of IoT technology and the opportunities it presents in the transportation sector. Significant research efforts are needed to carefully investigate the pros and cons of IoT technologies.

IoT-based monitoring solutions is an essential technology for logistics and transportation companies, especially those that operate a fleet of trucks. With IoT, fleet managers and drivers can operate with more efficiency and visibility, reducing fuel costs, improving safety and compliance, and optimizing cargo movements. As IoT technology continues to evolve, truck monitoring solutions will become even more sophisticated, opening up new possibilities to optimize logistics networks and supply chains.

6.4 Weight Monitoring

Weight monitoring is a critical aspect of fuel transportation, as it directly affects the safety and efficiency of the process. Accurate weight measurement helps to prevent overloading or underloading of fuel, which can lead to accidents or fuel waste. Several studies have highlighted the importance of weight monitoring in fuel transportation, and various technologies have been proposed to achieve this goal. For example, load cells and strain gauges have been used to measure the weight of fuel tanks, while IoT-based systems have been developed to monitor weight in real-time.

H. M. Adnan et al [9]. presents a portable vehicle weighing system using wireless sensor networks. The article discusses the limitations of traditional weighing systems, like weigh bridges and scales, and the advantages of using wireless sensor networks for weight monitoring in vehicles. The authors describe the design and implementation of a prototype system that uses load cells connected to a microcontroller for weight measurements, and wireless communication for data transmission to a cloud server. The article also discusses the use of machine learning algorithms for data analysis and decision-making based on the weight data collected by the system.

6.5 Location Tracking

Location tracking is a vital component of fuel transportation, as it enables real-time monitoring of fuel trucks and helps prevent fuel theft and diversion. Several studies have investigated the use of GPS technology for location tracking in fuel transportation [10, 11]. GPS-based systems provide accurate location data and have been shown to reduce fuel theft and improve delivery efficiency [12]. In addition, cellular networks and IoT technologies have been used to enhance location tracking and provide real-time updates [13].

6.6 Fuel Level Monitoring

Fuel level monitoring is critical to ensure that fuel is delivered in the correct quantity and to prevent fuel waste. Various sensors have been used for fuel level monitoring, including ultrasonic and float sensors [14, 15]. These sensors provide accurate fuel level data and can be integrated with IoT systems for real-time monitoring [16]. Fuel level monitoring has been shown



to reduce fuel waste and improve delivery efficiency [17]. This project uses an ultrasonic sensor to monitor fuel level in real-time.

6.7 Valve Status Monitoring

Valve status monitoring is essential to ensure that fuel is delivered safely and efficiently. Valve status monitoring systems use sensors to detect the state of fuel valves and prevent unauthorized access [18]. Several studies have investigated the use of sensor-based systems for valve status monitoring [19, 20]. These systems provide real-time data on valve status and can be integrated with IoT technologies for remote monitoring [21]. This project uses a sensor-based system to monitor valve status in real-time.

6.8 Pressure Monitoring

Pressure monitoring is critical to ensure that fuel is delivered safely and efficiently. Pressure sensors are used to monitor fuel tank pressure and detect any deviations from set values [22]. Several studies have investigated the use of pressure sensors for fuel tank pressure monitoring [23, 24]. Pressure monitoring has been shown to reduce the risk of fuel tank rupture and improve delivery safety [25]. This project uses a pressure sensor to monitor fuel tank pressure in real-time.

6.9 Inter-Device Communication

6.9.1 Message Query Telemetry Transport (MQTT)

MQTT (Message Queuing Telemetry Transport) is a machine-to-machine (M2M) connectivity protocol widely used for IoT applications. It is designed to transfer small data packets between devices in real-time, making it an ideal protocol for use with IoT devices that are connected to the internet but have limited resources such as low-bandwidth or intermittent connectivity. MQTT uses a publish-subscribe messaging paradigm, enabling a large number of devices to communicate with each other over a network quickly [26].

In relation to web technologies, MQTT can be integrated with web applications using various open-source libraries and MQTT brokers. MQTT-based web applications can use any web development language and can work with multiple devices.

MQTT can be used for multiple web technology applications, such as IoT sensors, real-time applications, and machine learning. Despite not being widely known in the web technologies domain, it has an important role in real-time messaging communication between devices and for handling large sets of data.

Overall, MQTT is a useful protocol in the web technologies field as it provides an efficient, scalable and secure means of communication between IoT devices and web applications. Developers can use MQTT to integrate IoT components into their web systems, and MQTT-based web applications can provide real-time information and advanced machine learning functions.



6.9.1.1 Pros of the MQTT Protocol

- Lightweight and efficient, requires low bandwidth and minimal processing power
- Supports both point-to-point [27] and publish-subscribe [28] messaging patterns
- Allows for reliable message delivery with quality of service (QoS) levels
- Supports authentication and access control, ensuring secure communication between devices
- Widely adopted and well-supported, with many open-source implementations and client libraries available

6.9.1.2 Cons of the MQTT Protocol

- Not suitable for large data payloads as it has a message size limitation
- May not be ideal for extremely low power devices with limited processing power
- Lack of built-in acknowledgment and timeout mechanisms can cause message loss in certain scenarios
- No standardized discovery mechanism can make it challenging to find MQTT servers on a network
- May require additional protocols such as Transport Layer Security (TLS) [29] for secure communication in some scenarios

6.9.2 Mosquitto

Mosquitto is one of the most popular MQTT brokers, which acts as a middle-layer between the publishers and the subscribers [30]. It is designed to be lightweight and efficient, making it ideal for use in small devices with limited resources. Mosquitto provides a publish/subscribe message exchange pattern, allowing devices to exchange information in a distributed environment. It also supports features like authentication and access control, making it a reliable solution for IoT (Internet of Things) applications.

Mosquitto is available on a variety of platforms including Windows, Linux, and macOS, and can be integrated with programming languages like JavaScript, Python, C, and Lua. It is widely used in applications like home automation, asset tracking, and telemetry.

6.9.2.1 Pros of Mosquitto

- Lightweight and efficient, which means it can run on low-powered devices with limited resources
- Open-source and free to use, making it a cost-effective choice for projects with limited budgets
- Works well with a wide range of programming languages and operating systems



- Includes a built-in websockets support for easy integration with web-based applications
- Offers support for SSL/TLS secure connections, making it a secure choice for IoT applications

6.9.2.2 Cons of Mosquitto

- Can be more difficult to set up and configure than some other MQTT brokers
- Limited scalability and performance compared to some commercial MQTT brokers
- Limited support options, since it is primarily community-driven and maintained
- May not be suitable for applications that require large scale data processing and analytics
- Some features and configurations may require knowledge of advanced networking concepts and protocols.

6.9.3 HTTP

HTTP (Hypertext Transfer Protocol) is a protocol used for transferring data over the internet. It is a request-response protocol, where a client sends a request to a server, and the server responds with the requested data. HTTP is a widely used protocol, and is the foundation of the web [31].

6.9.3.1 Pros of HTTP:

- Simplicity, flexibility, and scalability.
- platform-independence, meaning it can be used on any device or operating system.
- HTTP is a stateless protocol, which means that each request contains all the information necessary to complete the request.

6.9.3.2 Cons of HTTP:

- Lack of security, as data is sent in plain text. This makes it vulnerable to interception and eavesdropping.
- HTTP is a connectionless protocol, which means that each request requires a new connection to be established.

6.10 Web Technologies

One major advantage of employing a web-based UI (User Interface) is that the application can run on any device (Android TV, iPhone, Android phone, Desktop, etc.) and platform (Linux, Windows, macOS, IOS, etc.), provided the device in question has a web browser application installed.

6.10.1 HTML And CSS

HTML (Hypertext Markup Language) is the standard markup language used for creating web pages. It is responsible for describing the structure of a web page using a series of elements. These HTML elements, such as headings, paragraphs, and links, label pieces of content and tell the browser how to display this content. By using these elements, web developers can create web



pages that are both visually appealing and well-structured, making it easier for users to navigate and understand the content [32].

HTML elements are made up of tags and content. Tags consists of the element name surrounded by “<” and “>”. The elements have attribute options which gives them more specificity or adding functionality. HTML, therefore, plays a critical role in web development and is essential for creating engaging and functional websites [33].

Cascading Style Sheets (CSS) is used to style web pages, from the layout and typography to the color and visual effects [33]. CSS has been an essential component of web development since its introduction in 1996, and its use has only continued to grow. A study by the World Wide Web Consortium (W3C) found that 95% of all websites use CSS, making it a crucial skill for any web developer.

CSS has evolved significantly since its inception, with new features and properties being added regularly. The modular design of CSS makes it easy to customize and scale based on the specific needs of a project. Furthermore, the separation of presentation and content in CSS allows for greater re-usability and maintainability of code.

However, as with any technology, there are challenges associated with CSS. One major issue is browser compatibility, where certain CSS properties may not be supported across different browsers. Another issue is the complexity of CSS, which can be daunting for beginners and lead to poor coding practices.

Together, HTML and CSS can be used to develop visually appealing static web applications. To make the applications dynamic and more interactive, a third component has to be included:

6.10.2 JavaScript

JavaScript, an interpreted computer programming language [34], was introduced in 1995 as a way to add programs to web pages in the Netscape Navigator browser. The language has since been adopted by all other major graphical web browsers. It has made modern web applications possible— applications with which you can interact directly without doing a page reload for every action. JavaScript is also used in more traditional websites to provide various forms of interactivity and cleverness.

After its adoption outside of Netscape, a standard document was written to describe the way the JavaScript language should work so that the various pieces of software that claimed to support JavaScript were actually talking about the same language. This is called the ECMAScript standard, after the Ecma International organization that did the standardization. In practice, the terms ECMAScript and JavaScript can be used interchangeably—they are two names for the same language [35].

Although JavaScript is typically associated with web development and used primarily in web browsers, it can also be used outside the browser in a variety of contexts. For example, JavaScript can be used to create server-side applications, desktop and mobile applications, and even Internet of Things (IoT) devices. With the help of frameworks such as Node.js, JavaScript



has become a versatile language that can be used to build a wide range of applications beyond the web. Its popularity, flexibility, and ease of use make it a popular choice for developers looking to build modern applications across different platforms.

6.10.3 The Document Object Model

The Document Object Model (DOM) is a programming interface used by web browsers to represent web pages as a tree-like structure of nodes. Each node in the tree represents an element, attribute, or text in the HTML document, and these nodes can be manipulated using JavaScript.



Figure 6.2 HTML document and corresponding DOM representation

The DOM provides a powerful and flexible way for developers to create interactive web applications. With the DOM, you can dynamically modify the content and appearance of web pages, respond to user events like clicks and keyboard input, and create custom animations and effects.

A key benefit of the DOM is its platform independence. Because it is a standardized API, JavaScript code written for one web browser can be used on other browsers as well, making cross-browser development much easier.

The DOM is an essential tool for any web developer, and with its continuous evolution, it has become more powerful and efficient. With new versions of JavaScript and modern web development frameworks such as React and Angular, manipulating the DOM has become faster and more straightforward.

As web applications continue to grow in complexity and functionality, the DOM will remain a critical part of the web development process. Its versatility and broad adoption have made it a vital asset for creating engaging and interactive web applications.



6.10.4 Node.js

Created in 2009, Node.js is a JavaScript runtime environment that allows developers to build scalable, high-performance backend applications. Over the past decade, Node.js has grown in popularity due to its ease of use, flexibility, and ability to handle large amounts of data.

At its core, Node.js is designed to be fast and scalable. It uses a single-threaded, event-driven architecture that allows it to handle large numbers of requests simultaneously. The event loop, which is at the heart of Node.js, is responsible for handling I/O operations such as reading and writing to the file system or network sockets. This allows Node.js to perform non-blocking I/O operations that do not block the event loop, making it possible to handle many requests at once.

Node.js has a wide range of use cases. It is often used for building web applications, particularly those that require real-time collaboration or processing of large amounts of data. It can also be used for building APIs (Application Programming Interfaces) [36], command-line tools, and even desktop applications. Node.js is particularly well-suited for applications that require high concurrency and low latency, such as real-time applications.

In addition to its performance and scalability, Node.js has a large and active developer community. This community has created a wide range of libraries and frameworks that make it easier to build complex applications with Node.js. Popular frameworks include Express, Hapi, and Koa, while popular libraries include Socket.io, Passport, and Sequelize.

6.10.5 Libraries and Frameworks

6.10.5.1 React.js

React.js is an open-source JavaScript library designed by Facebook for building user interfaces or UI components. With React, developers can create reusable UI components that present data that changes over time. React uses a virtual DOM (Document Object Model), which is a lightweight representation of the actual DOM, to efficiently update the UI without having to make expensive calls to the browser [37].

React also makes it easy for developers to maintain large codebases by separating the application into smaller, reusable components. React encourages the use of a unidirectional data flow, where data flows downward from parent components to child components.

React has gained a lot of popularity because of its fast rendering speed and its ability to support server-side rendering [38], which enhances the performance of web applications. React is used by many web development teams, small and large, to build highly reusable and performant user interfaces.

6.10.5.2 Express.js

Express.js is a popular lightweight web framework for Node.js that provides a set of powerful features for building web applications. A literature review of research and blog posts related to



Express.js reveals that it is widely used in production environments and has a large and active community of developers.

Express.js simplifies the process of building robust and scalable web applications by providing various features such as routing, middleware, and model-view-controller (MVC) architecture [39] support. It also provides a range of plugins and modules that can be used to extend its functionality.

Together, Node.js and Express.js provide a powerful and flexible platform for building server-side web applications and APIs. Developers can take advantage of the features provided by Express.js, while leveraging the power of Node.js to build performant, scalable, and efficient web applications.

6.10.6 Database Technologies

6.10.6.1 Relational databases and SQL

A relational database is a type of database that stores and provides access to data points that are related to one another. Relational databases are based on the relational model, an intuitive, straightforward way of representing data in tables [40] (See Figure 6.3).

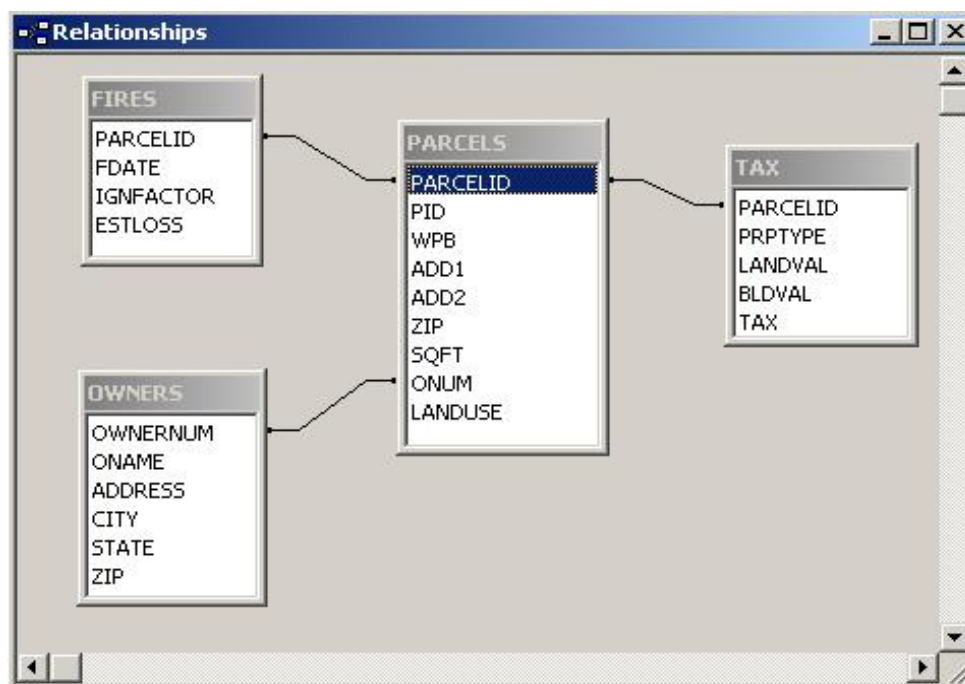


Figure 6.3: A relational database

SQL is a widely used domain-specific programming language for managing data held in a relational database management system. It is used for functions such as data manipulation, definition, and control. Research studies show that SQL databases are popular and widely used in



various programming applications. SQL databases are efficient at storing and querying structured data and have been used to handle large datasets in many industries, such as finance, e-commerce, and health care. With SQL, applications can create complex queries and easily retrieve data based on specific conditions, making it a powerful tool for data analysis and management. The most popular SQL database management systems are Oracle, MySQL, PostgreSQL, and Microsoft SQL Server

6.10.6.2 MySQL

MySQL is an open-source relational database management system that has been widely used for over a decade. It offers a wide range of features, including data storage and retrieval, scalability, security, and backup and recovery. Research shows that MySQL is one of the most popular databases in the world, especially in the web development industry. One reason for this popularity is its compatibility with a variety of programming languages and operating systems. Additionally, its documentation and community support make it easy for developers to get started with MySQL. Overall, MySQL continues to be a popular choice for companies and organizations looking for a reliable and flexible database management system.

6.10.6.3 Non-relational databases and NoSQL

A non-relational database is a database that does not use the tabular schema of rows and columns found in most traditional database systems. Instead, non-relational databases use a storage model that is optimized for the specific requirements of the type of data being stored [41].

NoSQL databases have become increasingly popular in recent years due to their flexibility and scalability in handling large amounts of unstructured or semi-structured data. In a systematic literature review of NoSQL databases, the data model used for a NoSQL database depends on the type of NoSQL database. NoSQL databases are categorized based on their storage type [42], which includes key-value stores, document stores, column-family stores, and graph databases. The review also notes that NoSQL databases have been adopted in various industries, including health care, where they have been used for large-scale data warehousing and analytics. While NoSQL databases have many advantages over traditional relational databases, they also come with some trade-offs, such as less strict consistency guarantees and a lack of standardization. NoSQL databases offer a promising alternative for handling big data applications, with new databases and technologies emerging to address the challenges faced by users in this rapidly evolving landscape. Popular NoSQL databases include MongoDB, Apache Cassandra and Redis.

6.10.6.4 MongoDB and Mongoose

MongoDB is a widely used NoSQL database that is designed for scalability, high performance, and flexibility in handling large amounts of unstructured data. Compared to relational databases, MongoDB is schema-free, meaning that it does not enforce a specific data structure or relationships between tables, allowing for more dynamic and agile development. Its flexible document-oriented data model is particularly well-suited for use cases that involve constantly evolving data, such as social media, e-commerce, and real-time analytics applications.

Mongoose is a JavaScript library that provides a schema-based modeling tool for MongoDB. Mongoose allows developers to define their data schema and model in a more structured and



scalable way, providing validation, type-casting, and other key features that are not available in the raw MongoDB driver.

The combination of MongoDB and Mongoose has become popular due to their ease of use, scalability, and flexibility. MongoDB is particularly suited for modern, agile software development methodologies that prioritize scalability, flexibility, and ease of use, while Mongoose provides a structured and object-oriented way of interacting with MongoDB. Together, they provide a powerful and efficient way to build modern web applications and services.

In addition, the MongoDB ecosystem includes a number of other powerful tools and technologies, such as MongoDB Atlas, a fully-managed cloud database service, and MongoDB Charts, a platform for data visualization and analysis. The MongoDB community is also active and growing, providing ample resources, tutorials, and documentation for developers to quickly get up to speed with the technology.

MongoDB and Mongoose are powerful tools for modern web development, offering a flexible and scalable approach to data management and access. With their ease of use, rich features, and active community, they are quickly becoming a go-to choice for many modern web applications.

```
const { Schema, model } = require('mongoose');

const UserSchema = new Schema(
  {
    fullName: { type: String, required: true },
    username: { type: String, required: true, unique: true },
    email: { type: String, required: true, unique: true },
    password: { type: String, required: true },
    bio: { type: String, required: true },
    level: { type: String, required: true, default: 'staff' },
    picture: { type: String },
    logs: { type: Array },
  },
  { timestamps: true }
);

const User = model('User', UserSchema);

module.exports = User;
```

Figure 6.4: Mongoose schema definition



6.10.7 The MERN Stack

The MERN stack allows developers to build modern web applications using JavaScript on both the front-end and back-end, thereby reducing the complexity of managing multiple programming languages and making web application development more streamlined and efficient. It provides a robust and efficient way to build real-time web applications, e-commerce platforms, and other data-intensive web applications that require scalability, flexibility, and agility.



The MERN stack is a popular choice for developing IoT applications as it offers a full-stack solution for building web applications using four technologies: MongoDB (a NoSQL database), Express.js (a Node.js framework for building APIs), React (a front-end JavaScript library), and



Node.js (a JavaScript runtime for server-side development). The MERN stack provides a seamless and efficient way to develop IoT applications that require the integration of multiple systems and data sources.

One of the key benefits of using the MERN stack in IoT is its ability to handle large volumes of data. MongoDB is known for its scalability, which allows for easier handling and management of large amounts of data. In addition, the React library enables the efficient rendering of data-intensive user interfaces by breaking them down into smaller components.

Another advantage of the MERN stack is its flexibility. Each component of the stack is modular, which means they can be easily replaced and customized to suit the specific needs of an IoT application. This makes it possible to integrate the MERN stack with other technologies and systems.

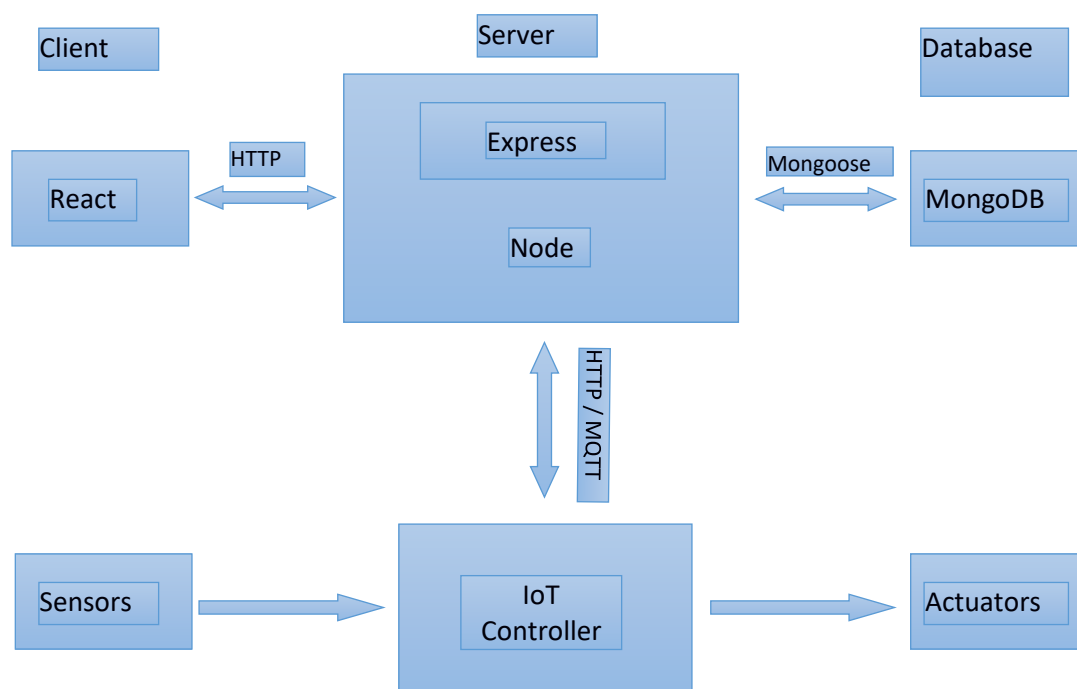


Figure 6.7: The MERN stack in IoT simple block diagram



6.11 Existing Systems

6.11.1 The Weighbridge

The working principle of a weighbridge involves the use of load cells that are installed under the platform of the weighbridge.



Figure 6.8: Weighbridge

As a vehicle drives onto the platform, the load cells detect the weight of the vehicle and transmit the data to a digital weight indicator that displays the weight of the vehicle. The load cells typically use strain gauge technology to measure the stress caused by the weight of the vehicle, which is converted into an electrical signal and transmitted to the weight indicator. Some weighbridges also use weigh-in-motion (WIM) technology, which measures the weight of moving vehicles using sensors embedded in the pavement or bridge structure. The data collected by the weighbridge can be used for a variety of purposes, such as verifying the weight of goods transported and ensuring compliance with weight restrictions on roads and bridges.

6.11.2 Fleet Management Systems (FMS)

FMS are used to track and manage fuel trucks, including their location, fuel level, and maintenance status [43]. However, these systems do not provide real-time monitoring and alerting capabilities.



6.11.3 IoT-based Fuel Monitoring Systems

IoT-based systems use sensors and wireless communication to monitor fuel levels and other parameters in real-time [44]. However, these systems do not provide a comprehensive solution for fuel transportation management.

6.11.4 GPS-based Tracking Systems

GPS-based systems are used to track fuel trucks and monitor their location in real-time [45]. However, these systems do not provide information on fuel levels, valve status, and other critical parameters.

6.11.5 Cloud-based Fuel Management Systems

Cloud-based systems use cloud computing and IoT technologies to monitor and manage fuel transportation [46]. However, these systems require high-speed internet connectivity and may not be suitable for remote areas.

In summary, existing systems provide partial solutions for fuel transportation management, but there is a need for a comprehensive system that provides real-time monitoring and alerting capabilities for all critical parameters.

6.12 Proposed Model

The proposed fuel truck anti-tampering system utilizes a combination of IoT technologies and cloud computing to provide real-time monitoring and alerting capabilities. The system consists of sensors installed on the fuel truck to monitor critical parameters such as location, fuel level, valve status, weight, and pressure. The sensors transmit the data to an ESP32 microcontroller, which sends the data to a NodeJS server via wireless communication.

The NodeJS server stores the data in a MongoDB database and triggers alerts to the frontend dashboard if any parameter deviates from the set values. The frontend dashboard, designed using React, displays the real-time data and alerts, enabling fleet managers to take prompt action in case of any anomalies. The system provides a comprehensive solution for fuel transportation management, enabling real-time monitoring, alerting, and decision-making.

The proposed system offers several advantages over existing systems, including real-time monitoring and alerting capabilities, and cloud-based storage and processing. The system is scalable, secure, and can be integrated with existing fleet management systems. The proposed system has the potential to significantly improve fuel transportation efficiency, reduce fuel theft and tampering, and enhance overall fleet management.



6.13 Literature Review of Components

6.13.1 ESP32

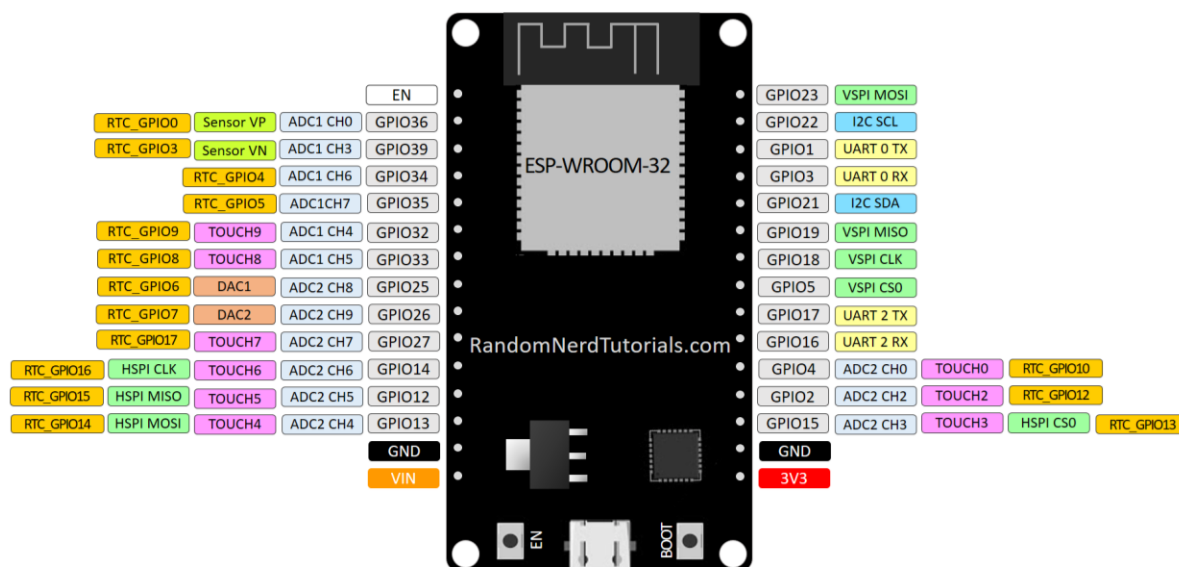


Figure 6.9: ESP32 pinout [image credits: RandomNerdTutorials.com]

ESP32 is a low-cost, low-power system-on-chip (SoC) microcontroller developed by Espressif Systems. It is a successor to the popular ESP8266 microcontroller and includes Wi-Fi and Bluetooth connectivity features. The ESP32 chip uses TSMC's low power 40 nm technology and is designed to achieve the best power and RF performance.

The ESP32 chip can be operated as a standalone system with minimal external circuitry, or as a slave device to a host MCU. It implements TCP/IP, full 802.11 b/g/n/e/i WLAN MAC protocol, Wi-Fi Direct specification, and Bluetooth 4.2 (and later) specifications. It has numerous features, including ADC, DAC, PWM, SPI, I2C, I2S, UART, SDIO, capacitive touch sensor, and more. One of the key features of the ESP32 is its dual-core processor, which enables it to handle multiple tasks simultaneously. This makes it ideal for applications that require complex computations or real-time data processing.

The ESP32 chip is quite popular in the IoT (Internet of Things) industry, and with its low cost and advanced features, it is a great alternative to more expensive options.



6.13.2 Ultrasonic Sensor

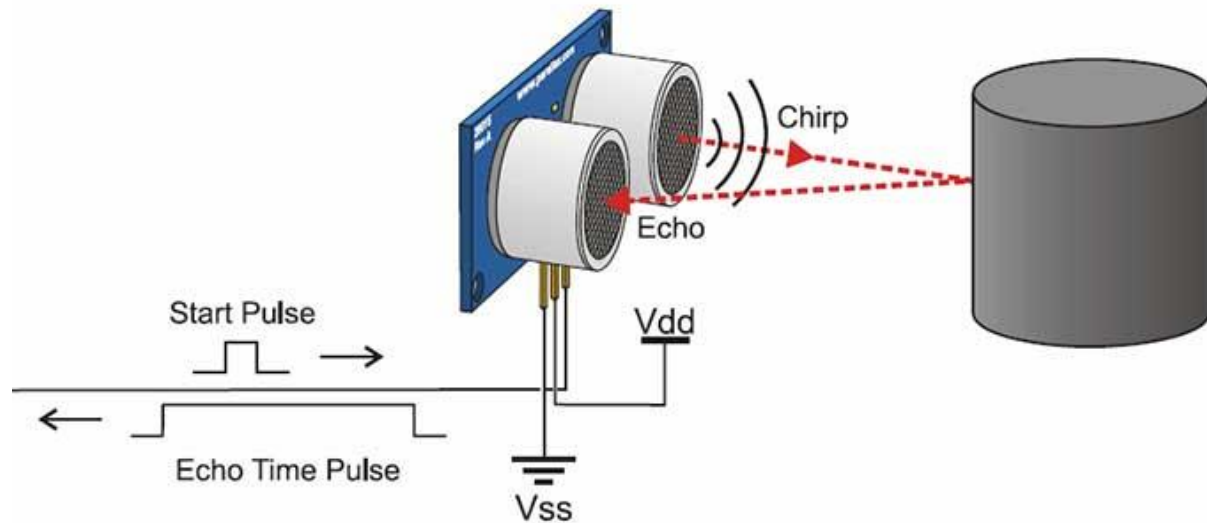


Figure 6.10: Ultrasonic sensor working principle

Ultrasonic sensors are electronic devices that use high frequency sound waves to detect the distance of an object from the sensor. They work on the principle of sending out ultrasonic waves and measuring the time taken for the waves to reflect back from an object. The sensor then calculates the distance based on the time taken for the wave to travel to and from the object. This makes ultrasonic sensors particularly useful for applications where precise distance measurement is required. Some common applications of ultrasonic sensors include distance measurement in robotics, parking assistance in cars, and liquid level measurement in tanks. Ultrasonic sensors can also be used for object detection and are often found in industrial and manufacturing environments. They can be interfaced with microcontrollers like the Arduino, and there are many types of ultrasonic sensors available in the market, such as HC-SR04, R305, JSN-SR04T, etc.

6.13.3 GPS Module

A GPS module is a device that can receive signals from GPS satellites and use those signals to determine its own location on the Earth's surface. These modules typically consist of an antenna and a receiver, and they can be used in a variety of applications, including navigation, tracking, and monitoring. GPS technology has become ubiquitous in modern life, with GPS modules being used in everything from smartphones to cars to drones. Additionally, GPS modules are often used in conjunction with other sensors and devices to provide location-based data that can be used to support a variety of applications.



Figure 6.11: GPS module

6.13.4 Load Cell

Load cells are a type of weight sensor used to measure force or weight in various applications, ranging from industrial manufacturing to healthcare. They work by converting the mechanical force applied to them into an electrical signal that can be read by a data acquisition system.

Numerous types of load cells have been developed over the years, including strain gauge load cells, hydraulic load cells, and piezoelectric load cells. Each type has its own advantages and disadvantages, and the choice of load cell depends on the specific application requirements.

6.13.5 Pressure Sensor

Piezoresistive pressure sensors are widely used in industrial applications due to their high accuracy, reliability, and durability. The Honeywell ABP Series is a popular choice for fuel tank pressure monitoring due to its high sensitivity, low hysteresis, and compatibility with a wide range of fluids. The sensor's piezoresistive material changes its electrical resistance in response to changes in pressure, allowing for accurate pressure measurement.

6.13.6 Valve

Solenoid valves are widely used in fuel management systems due to their fast response time, high reliability, and low power consumption. The Parker Hannifin 2-Way Normally Closed Solenoid Valve is a popular choice for fuel tank valve control due to its high flow rate, low pressure drop, and compatibility with a wide range of fuels. The valve's solenoid coil actuates the valve opening and closing, allowing for precise control of fuel flow.



6.13.6.1 Valve Status Sensor

Binary switch sensors are widely used to monitor valve status, providing a simple and reliable solution for detecting the OPEN or CLOSED state of a valve. These sensors typically consist of a switch or relay that changes state in response to the valve's position, providing a digital output indicating the valve's status.

Binary switch sensors offer several advantages, including high reliability, low power consumption, and compatibility with a wide range of valves. They are also resistant to mechanical stress and vibration, making them suitable for use in harsh environments. Additionally, binary switch sensors provide a clear and unambiguous indication of the valve's status, eliminating the need for complex signal processing or interpretation.

6.14 Conclusion

This chapter dived into the various truck monitoring systems currently in the market, analyzing their use cases and where there is room for improvement. The main metrics of focus were weight, location and fuel level. Web technologies used in IoT applications were introduced: HTML, CSS, JavaScript, Node.js, Express.js, React.js, and database technologies. The proposed system was briefly outlined and a literature review of the components to be used was presented.



7 METHODOLOGY

7.1 Introduction

This chapter outlines the methodology employed in the development and implementation of the Fuel Truck Anti-Tampering System. The methodology adopted in this project is a systematic and structured approach, which ensures the effective and efficient development of the system. This chapter provides an overview of the research design, hardware and software components, testing and validation procedures, and data analysis techniques used in the project.

The methodology chapter is divided into several sections, which provide a detailed description of the following:

- Research design and approach
- Hardware and software components used in the system
- System development and integration
- Testing and validation procedures
- Data analysis techniques

This chapter aims to provide a comprehensive understanding of the methodology employed in the project, which will help readers to appreciate the efforts and resources invested in the development of the Fuel Truck Anti-Tampering System.

7.2 Research Design and Approach

This section of the project outlines the research methodology used to develop and implement the fuel truck anti-tampering system. The approach consisted of two main phases:

7.2.1 System Requirements Analysis:

This initial phase involved identifying and understanding the needs and functionalities required for the anti-tampering system. This was achieved through:

- A comprehensive review of existing literature and industry reports to understand fuel theft methods, existing technologies, and potential sensor types.
- Consultations with fuel company personnel to understand their concerns, preferred features, and operational constraints.

7.2.2 System Design and Development

Based on the findings from the requirements analysis, the following steps were taken:

- Selection of appropriate sensors and a microcontroller for data acquisition and communication.
- Design and development of software components, including microcontroller firmware, a server-side application, and a front-end application.
- Prototyping and testing of the system to ensure accuracy, reliability, and user-friendliness.

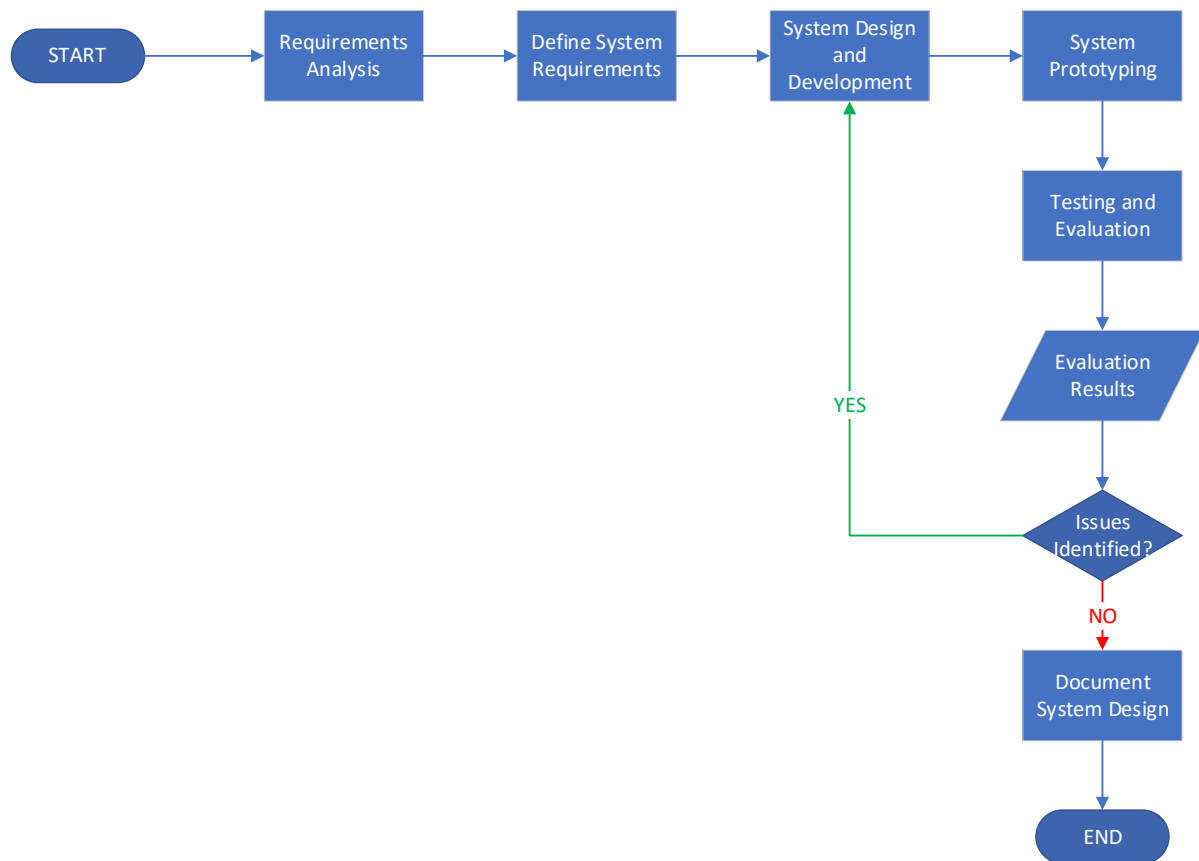


Figure 7.1 Research Design and Approach flow chart

This systematic approach ensured a thorough understanding of the problem and user needs, followed by effective system design, development, and testing. The final system addresses fuel theft concerns while being user-friendly and reliable.

7.3 Hardware Components Selection

The selection of appropriate hardware components was crucial for ensuring the functionality, reliability, and cost-effectiveness of the fuel truck anti-tampering system. Here's a breakdown of the key components chosen:

7.3.1 Sensors

7.3.1.1 Fuel Level Sensor:

Options considered:

- Ultrasonic
- capacitive,
- resistive.

Selection criteria:



- Accuracy
- compatibility with chosen fuel types
- installation feasibility within the fuel tank.

The HC-SR04 ultrasonic sensor was chosen for non-intrusive measurement and wide compatibility

7.3.1.2 Weight Sensor

Options considered:

- Load cells
- Axle weight sensors.

Selection criteria:

- Measurement range to accommodate full truck capacity
- Durability for harsh road conditions
- Ease of integration with the truck chassis.

Load cells mounted between the truck frame and suspension were chosen due to their direct weight measuring capability, ease of installation, durability and scalability

7.3.1.3 Pressure Sensor

Options considered:

- Differential pressure sensor
- Gauge pressure sensor.

Selection criteria:

- Pressure range to monitor both positive and negative pressure changes within the tank (indicating potential leaks or siphoning)
- Operating temperature range suitable for fuel types

Differential pressure sensor was chosen for versatile monitoring

7.3.1.4 Valve State Sensor

Options considered:

- Magnetic reed switch (Binary switch sensor)
- Mechanical limit switch.

Selection criteria:

- Compatibility with existing fuel outlet valve mechanism
- Reliable detection of open/closed positions
- Resistance to tampering attempts

Magnetic reed switch was chosen for temper resistance, and simple and reliable detection



7.3.1.5 Location Sensor

Options considered:

- GPS (Global Positioning System)
- Cellular Network Tracking
- GNSS (Global Navigation Satellite System)
- Radio Frequency Identification (RFID)

Selection criteria

- Accuracy Requirements
- Operational Area
- Cost
- Power Consumption

GPS technology (Neo GPS module) was chosen for cost-effectiveness, standalone functionality, high accuracy (within 2.5m), integration and ease of use.

7.3.2 Microcontroller

The ESP32 microcontroller was chosen due to its:

- Built-in Wi-Fi and Bluetooth connectivity for data transmission.
- Sufficient processing power for sensor data acquisition and communication.
- Low power consumption for extended battery life in the field.
- Availability of development tools and libraries for ease of programming.

7.4 Software Components Selection

The software components for the fuel truck anti-tampering system were chosen based on their functionality, compatibility, and ease of development. Here's a breakdown of the key software selections:

7.4.1 Microcontroller Firmware

Programming Language – C/C++ is a popular choice for embedded systems due to its efficiency, low-level control, and extensive library support for sensor interfacing and communication protocols.

7.4.2 Communication Protocol

HTTP over a Wi-Fi network was chosen as it is easy to implement and has a large community for support.

7.4.3 Server-Side Application

Development Framework – Node.js was selected for its:

Event-driven architecture – Efficiently handling real-time data streams from multiple trucks.

JavaScript – Familiarity with JavaScript by developers can expedite development time.



Extensive ecosystem of libraries – Provides readily available tools for data storage, communication, and web server functionality.

7.4.4 Database

MongoDB – A NoSQL document database chosen for its:

Scalability – Easily accommodates the growing volume of sensor data from multiple trucks.

Flexibility – Stores sensor data in a flexible JSON format, allowing for easy integration of various sensor types and potential future expansion.

7.4.5 Front-End Application

Development Framework – React was selected for its:

Component-based architecture – Enables modular development and user interface construction.

Virtual DOM – Improves rendering performance and user experience.

Large developer community – Provides extensive resources and support for troubleshooting and advanced features.

7.4.6 Additional Considerations

7.4.6.1 Security

Implementing secure coding practices and user authentication mechanisms to protect sensitive data transmission and storage.

7.4.6.2 Data Visualization Libraries

Utilizing libraries like D3.js, GuageChart.js or Chart.js to create informative and interactive dashboards for displaying real-time and historical sensor data.

7.5 System Design

This section details the design of the hardware circuit and software algorithm for the fuel truck anti-tampering system.

7.5.1 Hardware Circuit Design

The hardware circuit for the fuel truck anti-tampering system serves as the interface between the sensors and the microcontroller unit (MCU). See Figure 7.2.

Microcontroller Unit (MCU): The ESP32 serves as the central processing unit for the system, collecting sensor data and transmitting it to the server application.

Voltage Regulator: The LM7805 regulator IC regulates the incoming power supply from batteries connected in series to a voltage level, 5V, suitable for the MCU and sensor operation.

Power Supply: The system is typically powered by a rechargeable battery with sufficient capacity to ensure continuous operation.



Signal Amplifier: As the signal from the loadcell is not large enough to be detected by the MCU, an HX711 amplifier is used to amplify this signal

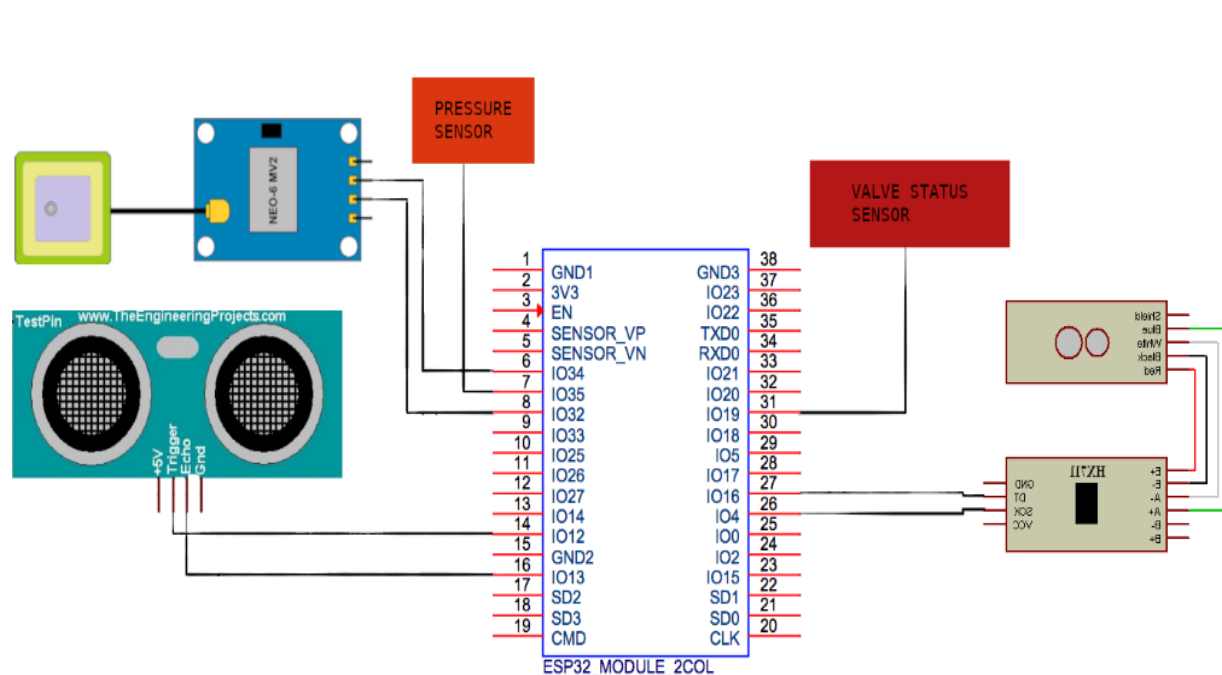


Figure 7.2 Hardware circuit schematic

7.5.2 Firmware Algorithm Design

The firmware algorithm for the fuel truck anti-tampering system defines the logical steps followed by the microcontroller firmware to process sensor data and trigger actions. Microsoft Visio was used for visually representing this algorithm.

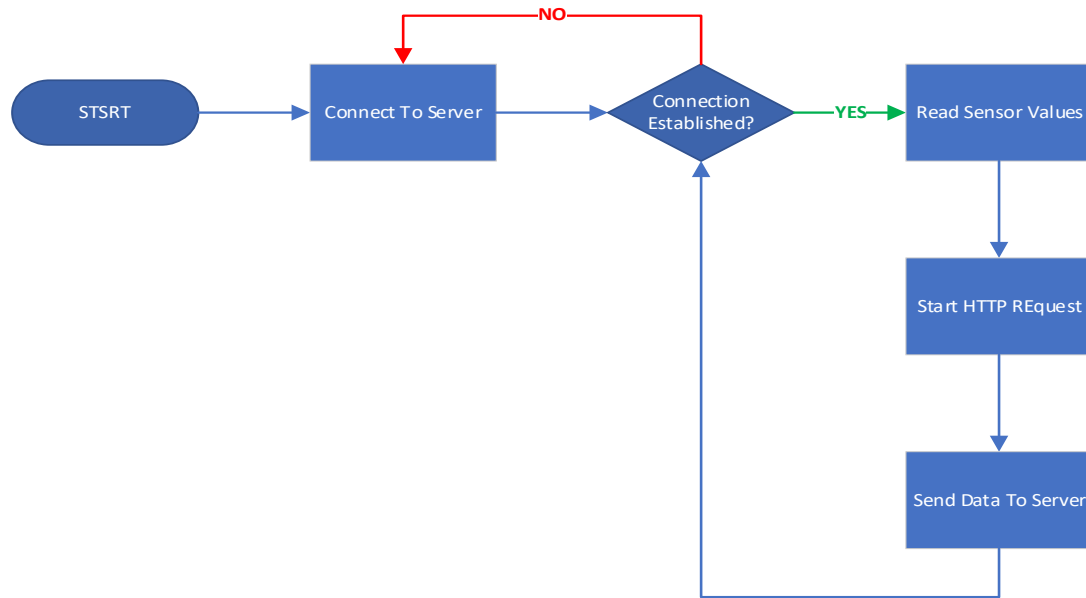


Figure 7.3 ESP32 Firmware Algorithm

7.5.3 Database Design

The database schema was designed to include a collection for each truck, with documents within each collection storing timestamps and sensor readings. This design allows for easy scalability as more trucks are added to the system and simplifies data retrieval for analysis and visualization on the front-end dashboard. The design of the database schema is shown in Figure 7.4 below.

```
plateNo: String,  
make: String,  
driver: String,  
level: Number,  
valve: Boolean,  
pressure: Number,  
weight: Number,  
compromised: Boolean,  
jobComplete: Boolean,  
gps: Object,
```

Figure 7.4 Truck Database Design



7.5.4 Frontend Application Design

To avoid re-inventing the wheel, the Material Dashboard 2 React – v2.1.0 by Creative Tim was used as a dashboard to display data from the server. The Dashboard has an open-source MIT license (Appendix B). Edits were then made to the design to meet the Fuel Truck Anti-Tempering System design objectives.

7.6 System Development and Integration

After the system design was validated for each unit of the system the system development and integration phase was carried out.

7.6.1 Hardware Integration

Wiring of the hardware circuit (MCU and sensors) was done according to the design in Figure 7.2.

Table 1 Hardware circuit pin connections

ESP32	HC-SR04	GPS	HX711	LOADCELL	VALVE STATUS	PRESSURE SENSOR
4			SCK			
12	TRIG					
13	ECHO					
16			DT			
19					✓	
32		RX				
34		TX				
35						✓
			E-	BLACK		
			E+	RED		
			A_	WHITE		
			A+	GREEN		

Table 1 shows the pin connections followed for the hardware circuit integration. To power up the circuit, a 12V series combination of batteries was used. An LM7805CV regulator IC was then used regulates the incoming power supply to 5V, suitable for the ESP32 and sensors used. A smoothing capacitor was connected between the 7805 output and common ground of the circuit to ensure a stable voltage supply for the microcontroller and sensors as shown in Figure 7.5.

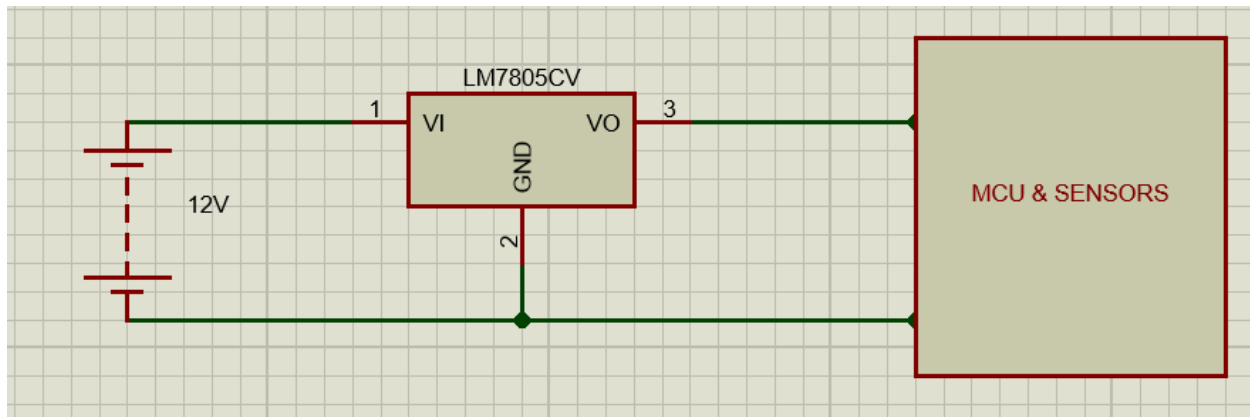


Figure 7.5 Power supply voltage regulation

7.6.2 Microcontroller Firmware Development

To simplify the development of the ESP32 firmware program, the Arduino IDE was used as it was designed to ease the development such programs. Pre-built libraries for the various sensors publicly available from the Arduino community were also included in the development (Figure 7.6),

```
1  #include <Arduino.h>
2  #include <ArduinoHttpClient.h>
3  #include <WiFi.h>
4  #include <NewPing.h>
5  #include <TinyGPSPlus.h>
6  #include <SoftwareSerial.h>
7  #include <ArduinoJson.h>
8  #include "HX711.h"
```

Figure 7.6 Imported libraries

Global variables were declared in the program for the server address and port where the server was to be running. Other global variables were also used to hold pin numbers where the sensors were connected:

```
24
25  char serverAddress[] = "192.168.43.190"; // server address # check on list of connected device in
26  int port = 1999;
27
28
```



```
29  WiFiClient wifi;
30  HttpClient client = HttpClient(wifi, serverAddress, port);
31
32  #define TRIGGER_PIN 12 // Arduino pin tied to trigger pin on the ultrasonic sensor.
33  #define ECHO_PIN    13 // Arduino pin tied to echo pin on the ultrasonic sensor.
34  #define MAX_DISTANCE 200 // Maximum distance we want to ping for (in centimeters). Maximum sensor range is about 4m (400cm)
35
36  NewPing sonar(TRIGGER_PIN, ECHO_PIN, MAX_DISTANCE); // NewPing setup of pins and maximum distance
37
38
39  // RX and TX pin from context of controller!
40  //controller RX <--> GPS TX
41  // controller TX <--> GPS RX
42  static const int RXPin = 34, TXPin = 32;
43  static const uint32_t GPSPin = 9600;
44
45  // The TinyGPSPlus object
46  TinyGPSPlus gps;
47
48  // hx711 object
49  HX711 scale;
50  uint8_t dataPin = 16;
51  uint8_t clockPin = 4;
52
53  // The serial connection to the GPS device
54  SoftwareSerial ss(RXPin, TXPin);
55
```

Figure 7.7 Global variable declaration

The setup function was used to establish a Wi-Fi connection and initialize serial communication ports and modules as shown in Figure 7.9. HX711 offset and direction of pins were also specified in the setup.

```
94  void loop(){
95
96      //GPS
97      while (ss.available() > 0)
98          if(gps.encode(ss.read()))
99              sendInfo();
100
101      if((millis() > 5000) && (gps.charsProcessed() < 10)) {
102          Serial.println(F("No GPS detected: check wiring."));
103          sendInfoWithoutGps();
104          // while(true);
105      }
106  }
```

Figure 7.8 Loop function



```
56 void setup() {
57     pinMode(valvePin, INPUT);
58     pinMode(15, OUTPUT);
59     //set the resolution to 12 bits (0-4096)
60     analogReadResolution(12);
61
62     Serial.begin(9600);
63     while(!Serial){delay(100);}
64
65     scale.begin(dataPin, clockPin);
66     // Obtained from HX711 calibration
67     scale.set_offset(35644);
68     scale.set_scale(9.298984);
69
70     Serial.println();
71     Serial.println("*****");
72     Serial.print("Connecting to ");
73     Serial.println(ssid);
74
75     WiFi.begin(ssid, password);
76
77     while (WiFi.status() != WL_CONNECTED) {
78         digitalWrite(15, HIGH);
79         Serial.print(".");
80         delay(500);
81         digitalWrite(15, LOW);
82         Serial.print(".");
83         delay(500);
84     }
```

Figure 7.9 Setup function

Figure 7.8 shows the loop function whose sole purpose is to determine if data is available from the GPS module. If data is available, the function to send sensor data along with GPS coordinates is called or else a function to send sensor readings without GPS coordinates is sent.

To enable the ESP32 MCU to send data to the NodeJS server the ArduinoJson library was used to convert the sensor readings data into a JSON object compatible with the server API expected input data.

The data in JSON format is then sent in the body of an HTTP POST request made to the system server API endpoint as shown in Figure 7.10 below.



```
153 // send JSON data to server
154 String endpoint = "/truck/update/tank/" + truckId;
155 client.beginRequest();
156 client.post(endpoint);
157 client.setHeader("Content-Type", "application/json");
158 client.setHeader("Content-Length", sensorDataObjectString.length());
159 client.setHeader("Connection", "close");
160 client.beginBody();
161 client.print(sensorDataObjectString);
162 int statusCodePost = client.responseStatusCode();
163 String responsePost = client.responseBody();
164 client.endRequest();
165
```

Figure 7.10 HTTP request to send sensor data

7.6.3 Firmware – Hardware Integration

With the hardware in place and the firmware program developed, the next step taken was the compilation of the code from C++ language into machine code compatible with the MCU. The resulting machine code was then uploaded from the programming computer into the ESP32 microcontroller via a USB data cable

7.6.4 Database Schema Development

The database schema of project was developed using Mongoose. The design shown in Figure 7.4 was implemented to produce the database schema model below:

```
const mongoose = require('mongoose');

const TruckSchema = mongoose.Schema({
  plateNo: { type: String, required: true, unique: true },
  make: { type: String, required: true },
  driver: { type: String },
  level: { type: Number, default: 0.0 },
  valve: { type: Boolean, default: 0 },
  pressure: { type: Number, default: 0.0 },
  weight: { type: Number, default: 0.0 },
  compromised: { type: Boolean, default: 0 },
  jobComplete: { type: Boolean, default: 1 },
  gps: {
    type: Object,
    default: {
      longitude: 0.0,
      latitude: 0.0,
    },
  },
  setWeight: { type: Number, default: 0.0 },
  setLevel: { type: Number, default: 0.0 },
  setPressure: { type: Number, default: 0.0 },
});

module.exports = mongoose.model('Truck', TruckSchema);
```

Figure 7.11 Mongoose schema



Other models for system users, notifications and drivers were also developed so that the system could be used as a complete fleet management center. See Appendix C – G.

7.6.5 Server API Development

The following dependencies were installed:

- bcrypt
- cors
- express
- jsonwebtoken
- mongoose

bcrypt is a popular npm package for password hashing. It provides a secure and efficient way to hash passwords, making them difficult to crack in case of a security breach. The package uses a salted hash function to protect against attacks like rainbow tables and brute force attacks. It allows for customization of the hashing algorithm, making it easy to adjust the level of security based on specific needs.

```
{
  "name": "fuel_truck_anti-tempering_system_back",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index",
    "dev": "nodemon index"
  },
  "license": "ISC",
  "dependencies": {
    "bcrypt": "^5.1.0",
    "cors": "^2.8.5",
    "express": "^4.18.2",
    "jsonwebtoken": "^9.0.0",
    "mongoose": "^6.9.0"
  }
}
```

Figure 7.12 Installed dependencies

The npm package cors is a middleware that allows servers to control access to resources from other domains. It provides a simple way to enable cross-origin resource sharing (CORS) in a Node.js application. This package helps to prevent security issues that arise when client-side web applications request data from unauthorized origins. The cors package can be customized to allow or restrict access based on specific origins, headers, and HTTP methods. Thus, cors is a useful package to ensure security and access control in web applications.



The npm package `jsonwebtoken` provides a simple and secure way to implement JSON Web Tokens (JWT). It allows developers to generate, sign, and verify JWTs for authentication and authorization in Node.js applications. The `jsonwebtoken` package also offers various configuration options for creating and verifying JWTs, including secret keys, expiration times, and custom claims. JSON (JavaScript Object Notation) Web Tokens are a popular mechanism for securely transmitting information between parties as a JSON object. They are often used for authentication and authorization in web applications, providing a means to verify the legitimacy of requests and protect sensitive data.

The back-end project structure applies “separation of concerns” by bundling the Node.js files for routes and data models in different directories (Figure 7.13).

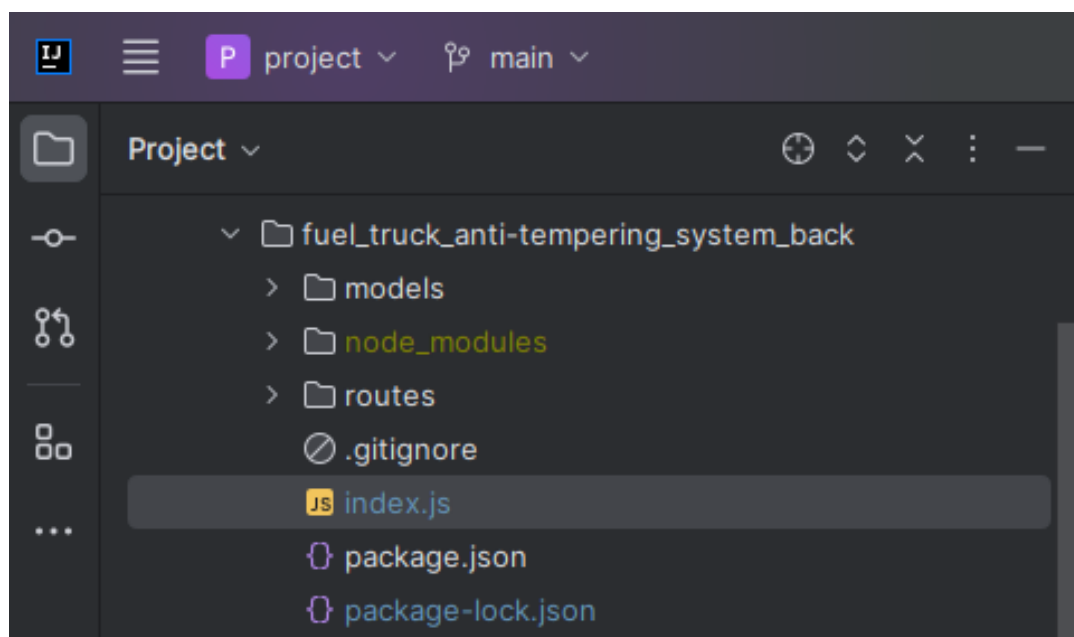


Figure 7.13 Back-end folder structure

The file `index.js` is the entry point of the application. The file imports the required node packages into the project. All route files are also imported into `index.js`. The MongoDB database uri is specified, and mongoose uses it to connect to the database.



```
1 const express = require('express');
2 const app = express();
3 const mongoose = require('mongoose');
4 const cors = require('cors');
5
6 const user = require('./routes/users');
7 const truck = require('./routes/trucks');
8 const job = require('./routes/jobs');
9 const driver = require('./routes/drivers');
10 const notification = require('./routes/notifications');
11
12 app.use(cors());
13
14 app.use(express.json());
15
16 const db_url = 'mongodb://0.0.0.0:27017/ftats';
17
18 mongoose.set('strictQuery', true);
19 mongoose.connect(db_url);
20 const db = mongoose.connection;
21 db.on('error', (error) => console.log(error));
22 db.once('open', () => console.log('DB Connected'));
23
24 app.get('/', (req, res) => {
25   res.end("hello");
26 });
27
28 app.listen(1999, (req, res) => {
29   console.log('Server running on port 1999');
30 });
```

Figure 7.14 Server API entry point

7.6.6 Front End Application Development

Like in the server, the directory structure of the front-end was made so that it is easier to understand the project thus reducing debugging time in case of errors.

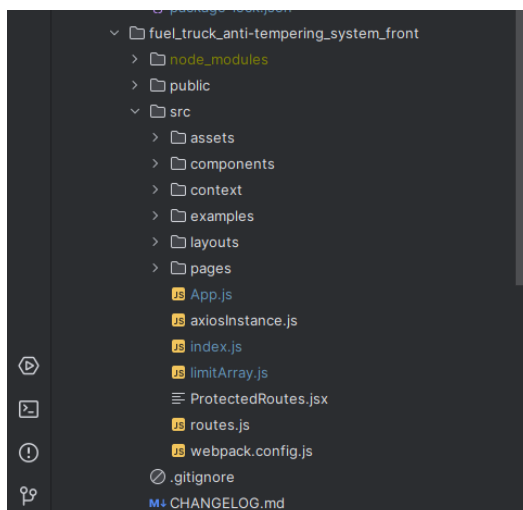


Figure 7.15 Front-end application project structure



The assets folder was used to hold the images and theme data used in the UI (User Interface). The pages directory contains the React files for rendering the different pages of the web application front-end in the web browser. The project entry point is the file index.js which specifies the ID of the HTML element in the public HTML document in which the React code will be inserted. The file imports and uses the BrowserRouter and MaterialUIControllerProvider components into the project.

```
207 <Routes>
208   <Route element={}<ProtectedRoutes />}>
209     {getRoutes(routes)}
210     <Route path='/addtruck' element={}<AddTruck /> } />
211     <Route path='/addjob' element={}<AddJobs /> } />
212     <Route path='/managetruck' element={}<ManageTruck /> } />
213     <Route path='/adddriver' element={}<AddDriver /> } />
214     <Route path='/assigndriver' element={}<AssignDriver /> } />
215     <Route path='/managejob' element={}<ManageJob /> } />
216   </Route>
217   <Route path='/authentication/sign-in' element={}<SignIn /> } />
218   <Route path='/authentication/sign-up' element={}<SignUp /> } />
219   <Route path='/addnot' element={}<Addnotification /> } />
220   <Route path='*' element={}<Navigate to='/dashboard' /> } />
221 </Routes>
```

Figure 7.16 Application routes

The entry point imports and calls the App React component to be rendered. The App component imports and uses the theme components from the assets directory to define application styling. The components to be rendered for each route are also defined in the App.js component (Figure 7.16).

The component that displays truck data in real-time uses ReactJS's useEffect hook to periodically fetch data from the server and re-render the UI whenever there is a change in the state of application data (Figure 7.17).

The AddJob component was developed for setting initial state of the tank whenever a new delivery is to be made.

The component renders a form with the fields to select the driver, tank fuel level, pressure, valve status, and weight.

A form submit handler function was then written to send this data collected from the form to the application server. See Figure 7.18

This initial state is then used to check for any deviations along the route before the job is completed.



```
59
60   useEffect(() => {
61     let isMounted = true;
62     const controller = new AbortController();
63
64     const fetchTruckData = async () => {
65       try {
66         const res = await axios.get(url);
67         const fetchedTruckData = await res.data.truck;
68         isMounted && setTruckData(fetchedTruckData);
69       } catch (err) {
70         console.log(
71           `Component \'ManageTruck.js\' failed to fetch truck data with the following error:\n${err}
72         );
73       }
74     };
75     fetchTruckData();
76     const fetchTruckDataPeriodically = setInterval(
77       () => fetchTruckData(),
78       7000
79     );
80     return () => {
81       clearInterval(fetchTruckDataPeriodically);
82       isMounted = false;
83       controller.abort(); //cancel any pending requests when the component unmounts
84     };
85   }, []);
86
87   let { level, valve, pressure, weight, gps, setWeight } = truckData;
```

Figure 7.17 Fetching truck data from the server

The application continually checks the real-time tank state against the initial set conditions. Whenever a significant change is detected before a delivery is completed an alert is raised indicating the exact location and time when this change was detected (7.18).



```
fetchTrucks().then((data) => {
  data.forEach((truck) => {
    const {level, weight, pressure, setLevel, setWeight, setPressure, jobComplete, valve, driver, _id} = truck;
    let message = '';
    if (jobComplete === false) {
      if (setLevel - level > 10) {
        message += 'level, ';
      }

      if (setPressure - pressure > 10) {
        message += 'pressure, ';
      }

      if (setWeight - weight > 5) {
        message += 'weight, ';
      }
      if (message.length > 3) message += 'reduced. ';

      if (valve) message += 'Valve opened!';
    }

    if (message.length > 3) {
      axios
        .post('/truck/addAlert', { driver, _id, message })
        .then((res) => console.log(res.data));
    }
  });
});
```

Figure 7.18 Detecting metric changes and raising alerts

The alert also provides information about which of the metrics being tracked deviated. The status of the truck in question is then set to “compromised”.

```
addJobs.js
8
9 const handleSubmit = (evt) => {
10   evt.preventDefault();
11   axios
12     .post('/job', {
13       company,
14       status,
15       jobNo,
16       driverId,
17       goods,
18       weight,
19       level,
20       pressure,
21     })
22     .then(function (response) {
23       console.log(response);
24       if (response.data) {
25         alert(response?.data?.message);
26       }
27     })
28     .catch(function (err) {
29       console.log(err);
30       if (!err?.response) {
31         alert('No Server Response');
32       } else if (err.response?.status) {
33         alert(err.response?.data?.message);
34       }
35     });
36 };
37
```

Figure 7.19 Sending initial state to the server



7.7 Testing and Validation Procedures

The testing and validation phase was conducted to ensure that the system met the required specifications and performed as expected. The system was tested in a simulated environment, where sensors were connected to the ESP32 MCU and data displayed in various forms depending on the type of test being carried out.

7.7.1 Hardware Unit Tests

The hardware unit tests were conducted to verify the functionality and performance of each hardware component of the fuel truck anti-tempering system. Individual sensors were tested to ensure accurate readings. The ESP32 programs used to test the components are shown in Figure 7.20 – 7.24

```
1  const int pressPin = 35;
2
3  void setup() {
4      //set resolution to 12 bits
5      analogReadResolution(12);
6      Serial.begin(9600);
7  }
8
9  void loop() {
10     //read value every 2 seconds
11     int pressure = analogReadMilliVolts(pressPin);
12     Serial.print("Value: ");
13     Serial.println(pressure);
14     delay(2000);
15 }
```

Figure 7.20 Pressure sensor test code

```
7
8  #include "HX711.h"
9
10 HX711 scale;
11
12 uint8_t dataPin = 16;
13 uint8_t clockPin = 4;
14
15
16 void setup()
17 {
18     Serial.begin(115200);
19     Serial.println(__FILE__);
20     Serial.print("LIBRARY VERSION: ");
21     Serial.println(HX711_LIB_VERSION);
22     Serial.println();
23
24     scale.begin(dataPin, clockPin);
25
26     scale.set_offset(35644);
27     scale.set_scale(9.298984);
28 }
29
30
31 void loop()
32 {
33     Serial.print("UNITS: ");
34     Serial.print(scale.get_units(20)/10);
35     Serial.println();
36     delay(1000);
37 }
38
```

Figure 7.21 Weight sensor test code



```
1 #include <NewPing.h>
2
3 #define TRIGGER_PIN 12 // Arduino pin tied to trigger pin on the ultrasonic sensor.
4 #define ECHO_PIN 11 // Arduino pin tied to echo pin on the ultrasonic sensor.
5 #define MAX_DISTANCE 200 // Maximum distance we want to ping for (in centimeters). Maximum sensor distance is rated at 400-500cm.
6
7 NewPing sonar(TRIGGER_PIN, ECHO_PIN, MAX_DISTANCE); // NewPing setup of pins and maximum distance.
8
9 void setup() {
10   Serial.begin(115200); // Open serial monitor at 115200 baud to see ping results.
11 }
12
13 void loop() {
14   delay(50); // Wait 50ms between pings (about 20 pings/sec). 29ms should be the shortest delay between pings.
15   Serial.print("Ping: ");
16   Serial.print(sonar.ping_cm()); // Send ping, get distance in cm and print result (0 = outside set distance range)
17   Serial.println("cm");
18 }
```

Figure 7.22 Ultrasonic sensor test code

```
2
3 #define VALVE 12 // Arduino pin tied to signal pin of valve status sensor.
4 bool status;
5
6 void setup() {
7   Serial.begin(115200); // Open serial monitor at 115200 baud to see ping results.
8   pinMode(VALVE, INPUT);
9 }
10
11 void loop() {
12   Serial.print("Valve status: ");
13   status = digitalRead(VALVE) ? "OPEN" : "CLOSED";
14   Serial.println(status);
15 }
```

Figure 7.23 Valve status sensor test code

```
1 #include <TinyGPSPlus.h>
2 #include <SoftwareSerial.h>
3
4 static const int RXPin = 4, TXPin = 3;
5 static const uint32_t GPSPBaud = 4800;
6 TinyGPSPlus gps;
7 SoftwareSerial ss(RXPin, TXPin);
8
9 void setup() {
10   Serial.begin(115200);
11   ss.begin(GPSPBaud);
12   Serial.println(F("DeviceExample.ino"));
13   Serial.println(F("A simple demonstration of TinyGPSPlus with an attached GPS module"));
14   Serial.print(F("Testing TinyGPSPlus library v. ")); Serial.println(TinyGPSPlus::libraryVersion());
15   Serial.println(F("by Mikal Hart"));
16   Serial.println();
17 }
18
19 void loop() {
20   while (ss.available() > 0)
21     if (gps.encode(ss.read()))
22       displayInfo();
23
24   if (millis() > 5000 && gps.charsProcessed() < 10)
25   {
26     Serial.println(F("No GPS detected: check wiring."));
27     while(true);
28   }
29 }
```

Figure 7.24 GPS sensor test code

Additionally, the Wi-Fi communication interface was tested to ensure stable and reliable operation.



```
1 #include <WiFi.h>
2
3 const char* ssid    = "test";
4 const char* password = "12345677";
5
6 void setup() {
7     Serial.begin(115200);
8     while(!Serial){delay(100);}
9
10    Serial.println();
11    Serial.println("*****");
12    Serial.print("Connecting to ");
13    Serial.println(ssid);
14
15    WiFi.begin(ssid, password);
16
17    while (WiFi.status() != WL_CONNECTED) {
18        delay(500);
19        Serial.print(".");
20    }
21
22    Serial.println("");
23    Serial.println("WiFi connected");
24    Serial.println("IP address: ");
25    Serial.println(WiFi.localIP());
26 }
27
28 void loop(){
29
30 }
```

Figure 7.25 Wi-Fi test code

7.7.2 Firmware Test

The “verify” functionality of the Arduino IDE was used to validate the ESP32 code for any errors and datatype inconsistencies. This helped to ensure that successful compilation of the C++ code into machine code.

7.7.3 Firmware – Hardware Integration Test

After the sketch was uploaded into the ESP32 microcontroller, the serial monitor was used to display the values being read by the sensors and test the overall functionality of the system.

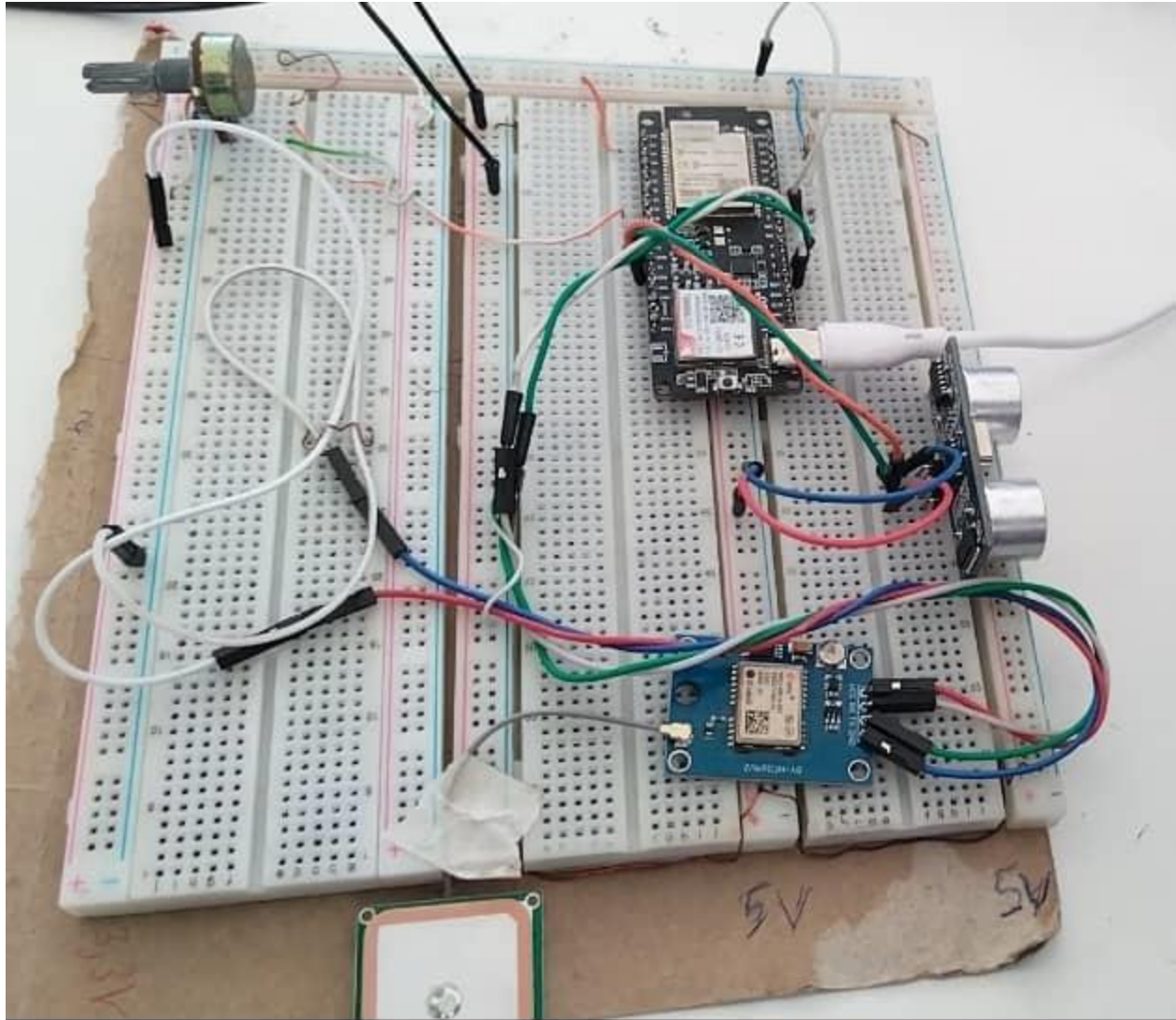


Figure 7.26 Firmware-Hardware integration test circuit wiring

7.7.4 Server API Tests

Thunder Client, a Visual Studio Code extension, was used to test the server API. HTTP requests were made to the API endpoints of the server and the response received from the server was displayed to ensure that the server was behaving as expected.

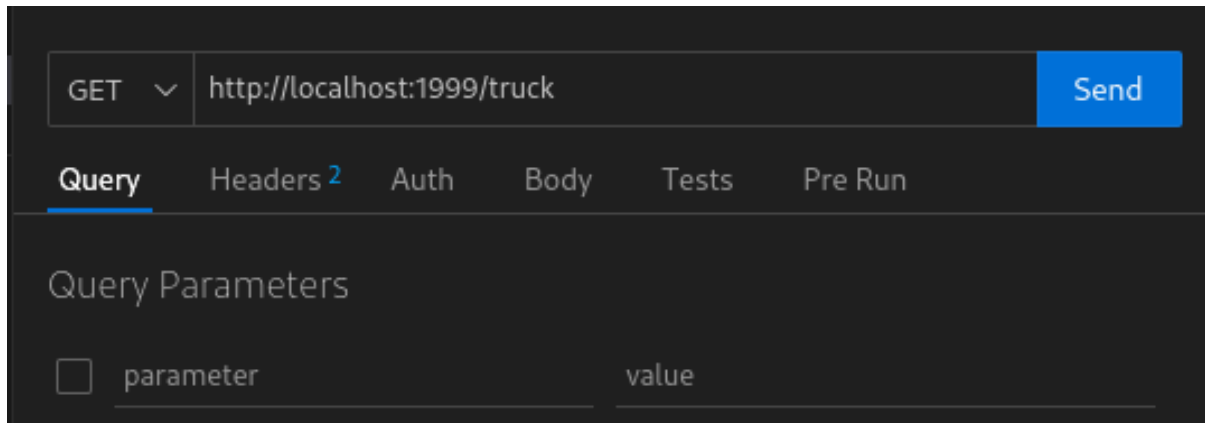


Figure 7.27 Using Thunder Client to test API

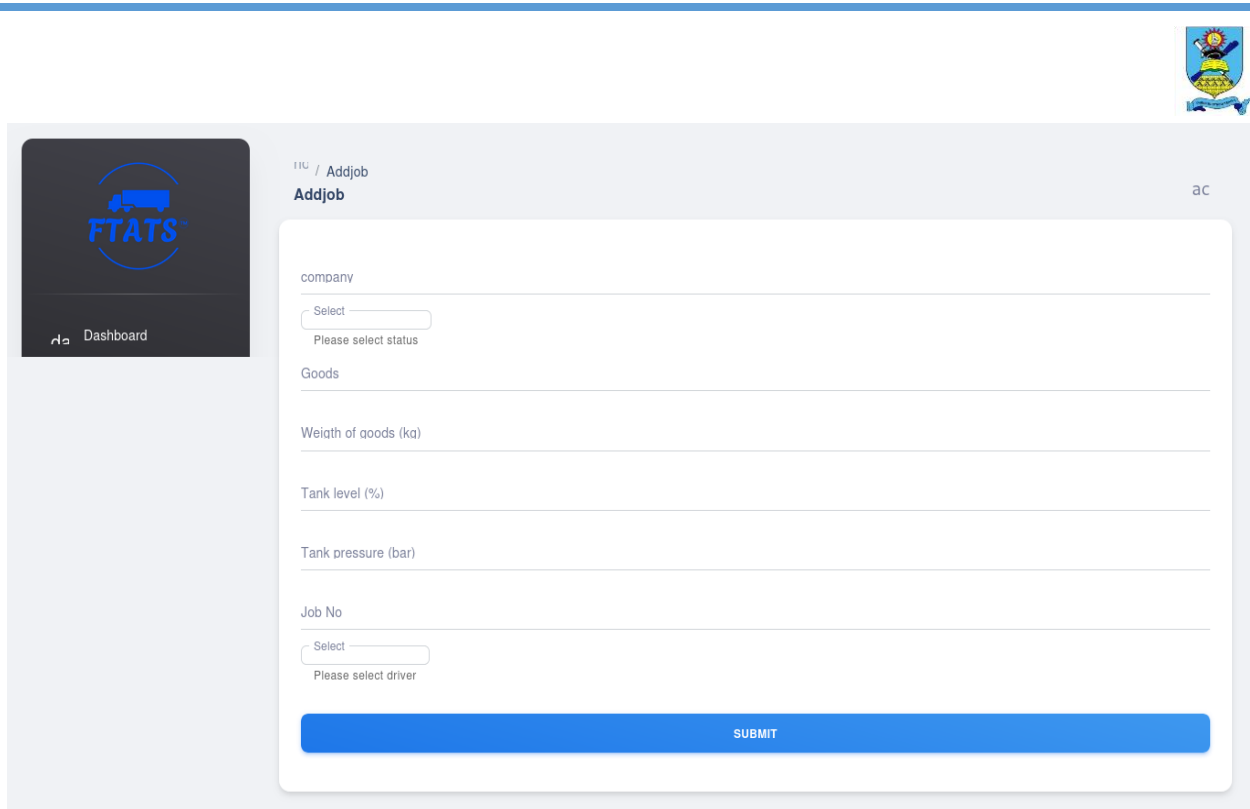
Strategic calls to JavaScript's inbuilt `console.log` function were made to display the state of the server at various stages of request handling.

MongoDB Compass desktop application was used to inspect the database after an HTTP request was made to the server to modify the database

7.7.5 Front End Component Tests

Frontend component tests were conducted to ensure that the React dashboard components functioned as expected and displayed accurate data. These tests focused on verifying the correctness of the user interface, component interactions, and data rendering. We wrote unit tests for each component, using Jest and React Testing Library, to isolate and test individual components in isolation. For example, we tested that the Fuel Level component correctly displayed the fuel level data received from the backend, and that the Alert component triggered the correct alert message when the fuel level fell below a certain threshold.

UI integration tests were also performed to verify that the components worked together seamlessly. We tested scenarios such as navigating between pages, triggering alerts, and updating data in real-time. These tests ensured that the frontend components correctly communicated with the backend API and displayed the correct data to the user. By writing comprehensive frontend component tests, we were able to catch and fix bugs early in the development process, ensuring a high-quality and reliable user interface for the fuel truck monitoring system.



The screenshot displays a web application interface for 'FTATS'. On the left is a dark sidebar with the 'FTATS' logo and a 'Dashboard' link. The main content area is titled 'Addjob' and contains a form with the following fields: 'company' (with a 'Select' dropdown and 'Please select status' text), 'Goods', 'Weight of goods (kg)', 'Tank level (%)', 'Tank pressure (bar)', 'Job No' (with a 'Select' dropdown and 'Please select driver' text), and a prominent blue 'SUBMIT' button at the bottom. A university crest is located in the top right corner of the page.

Figure 7.28 Testing the AddJob form component

7.7.6 System Integration Tests

The system integration phase involved combining the hardware and software components of the fuel truck anti-tampering system to ensure seamless communication and data exchange. This phase was critical in ensuring that the entire system functioned as a cohesive unit, providing real-time tracking and monitoring of the fuel truck's location, fuel level, valve state, weight, and pressure. The ESP32 microcontroller was integrated with the sensors to collect data, which was then transmitted to the NodeJS server via HTTP over Wi-Fi.

The NodeJS server was responsible for receiving the sensor data and storing it in the MongoDB database. The server was also integrated with the React frontend, which displayed the collected data in a user-friendly dashboard. The integration of the server and frontend enabled real-time updates and alerts to be sent to the dashboard whenever a change in the set values of any of the metrics being tracked occurred. This ensured that any potential tampering or anomalies were quickly identified and addressed.

To ensure robust system integration, various tests were conducted to simulate real-world scenarios and identify any potential issues. These tests included simulating sensor data transmission, testing the accuracy of the sensor readings, and verifying the functionality of the alert system. Additionally, the system was tested for security vulnerabilities to ensure that the data transmitted and stored was secure and protected from unauthorized access.



7.8 Data Analysis Techniques

The collected sensor data from the fuel truck anti-tampering system will be analyzed using various techniques to identify potential fuel theft attempts and optimize fleet management. Here are some key data analysis techniques that will be employed:

7.8.1 Descriptive Statistics

Mean, Median, Standard Deviation – These will be used to understand the average, central tendency, and variability of sensor readings (fuel level, weight, pressure) for each truck across different timeframes (daily, weekly, monthly).

Minimum and Maximum Values – Identifying minimum and maximum readings can help detect abnormal sensor behavior or potential tampering attempts (e.g., sudden drops in fuel level).

7.8.2 Statistical Anomaly Detection:

Thresholding – Pre-defined thresholds were established for each sensor reading based on historical data and operational knowledge. Deviations exceeding these thresholds might indicate potential theft attempts.

7.8.3 Visualization Techniques

Interactive Dashboards – The collected data was visualized in interactive dashboards using libraries like ReactGuagehCart.js or Chart.js. These dashboards allow for real-time monitoring of sensor readings, historical data exploration, and anomaly visualization to facilitate informed decision-making.

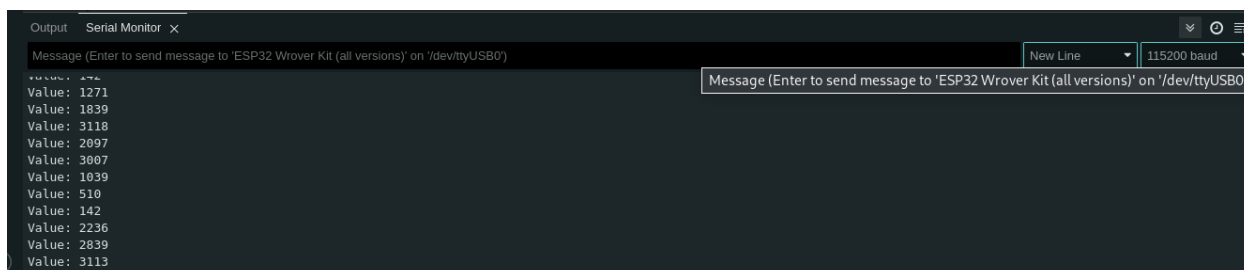


8 RESULTS AND DISCUSSION

8.1 Hardware Unit Tests Results

8.1.1 Pressure Sensor

The results of connecting the pressure sensor to the specified ESP32 pin and executing the pressure test program are shown in Figure 8.1 below. The Arduino IDE in-built Serial Monitor was used to print and inspect the pressure readings



```
Output Serial Monitor x
Message (Enter to send message to 'ESP32 Wrover Kit (all versions)' on '/dev/ttyUSB0') New Line 115200 baud
Value: 1271
Value: 1839
Value: 3118
Value: 2097
Value: 3007
Value: 1039
Value: 510
Value: 142
Value: 2236
Value: 2839
Value: 3113
```


Figure 8.1 Pressure sensor readings

The results of unit tests of other hardware components are shown in Figure 8.2 – 8.5



```
Output Serial Monitor x
Message (Enter to send message to 'ESP32 Wrover Kit (all versions)' on '/dev/ttyUSB0') New Line 115200 baud
UNITS: 127
UNITS: 139
UNITS: 318
UNITS: 209
UNITS: 67
UNITS: 109
UNITS: 510
UNITS: 142
UNITS: 223
UNITS: 839
UNITS: 113
```

Figure 8.2 Loadcell weight readings



```
Output Serial Monitor x
Message (Enter to send message to 'ESP32 Wrover Kit (all versions)' on '/dev/ttyUSB0') New Line 115200 baud
config: 0, SPIWP: 0xee
clk_drv: 0x00, q_drv: 0x00, d_drv: 0x00, cs0_drv: 0x00, hd_drv: 0x00, wp_drv: 0x00
mode: DIO, clock div: 1
load: 0x3fff0030, len: 1344
load: 0x40078000, len: 13964
load: 0x40080400, len: 3600
entry 0x400805f0
Ping: 20cm
Ping: 5cm
Ping: 7cm
Ping: 11cm
```

Figure 8.3 Ultrasonic sensor distance readings



```
Output Serial Monitor x
Message (Enter to send message to 'ESP32 Wrover Kit (all versions)' on '/dev/ttyUSB0') New Line 115200 baud
configip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0030,len:1344
load:0x40078000,len:13964
load:0x40080400,len:3600
entry 0x400805f0
Valve status: OPEN
Valve status: OPEN
Valve status: CLOSED
Valve status: OPEN
```

Figure 8.4 Valve status sensor test results

```
Output Serial Monitor x
Message (Enter to send message to 'ESP32 Wrover Kit (all versions)' on '/dev/ttyUSB0') New Line 115200 baud
mode:DIO, clock div:1
load:0x3fff0030,len:1344
load:0x40078000,len:13964
load:0x40080400,len:3600
entry 0x400805f0

*****
Connecting to test
...
WiFi connected
IP address:
192.168.43.168
```

Figure 8.5 Wi-Fi connection test results

The results of the hardware unit tests confirmed the integrity of the system hardware components.

8.2 Firmware Test Results

The completed firmware code compiled successfully to indicate that there were no syntax, type and undeclared variable errors in the code.

```
Sketch uses 753217 bytes (57%) of program storage space. Maximum is 1310720 bytes.
Global variables use 44776 bytes (13%) of dynamic memory, leaving 282904 bytes for local variables. Maximum is 327680 bytes.
```

Figure 8.6 Firmware successful compilation

8.3 Firmware – Hardware Integration Test Results

The results of integrating all the system hardware components and the ESP32 program are shown in Figure 8.7 below:



```
...
WiFi connected
IP address:
192.168.43.168

Sending with gps data...

Wait 10 seconds

Post Response Status Code: 200

Post Response: {"gps":{"latitude":-20.231424,"longitude":28.4385},"level":84,"weight":57,"pressure":142,"valve":false}

Pretty JSON Object:
{
  "gps": {
    "latitude": -20.231424,
    "longitude": 28.4385
  },
  "level": 84,
  "weight": 57,
  "pressure": 142,
  "valve": false
}
```

Figure 8.7 Firmware-Hardware integration test results

The response status code of 200 indicate that the HTTP request made by the ESP32 to the placeholder server – used for testing purposes – was handled successfully without any errors.

8.4 Server API Tests Results

The response from making a GET request to the API endpoint to fetch trucks data is shown in Figure 8.8 below. Figure 8.9 shows the logs made by the server as it processed the GET request whilst Figure 8.10 shows the results in the database after making a POST request to the API endpoint to add truck data to the database.

```
GET http://localhost:1999/truck Send
Status: 200 OK Size: 578 Bytes Time: 86 ms

Query Parameters
parameter value

Response
1 {
2   "trucks": [
3     {
4       "_id": "662bf28c920ec610546933a4",
5       "plateNo": "abc001",
6       "make": "Truck 1",
7       "level": 83.03249359,
8       "valve": false,
9       "pressure": 2740,
10      "weight": 15.34000015,
11      "compromised": true,
12      "jobComplete": false,
13      "gps": {
14        "latitude": 0,
15        "longitude": 0
16      },
17      "setWeight": 1200,
18      "setLevel": 98,
19      "setPressure": 9.05,
20      "v": 0,
21      "driver": "driver001"
22    },
23    {
24      "_id": "662bf296920ec610546933a9",
25      "plateNo": "abc002",
26      "make": "Truck 2"
```

Figure 8.8 HTTP GET request and response results on Thunder Client



```
OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  GITLENS  COMMENTS  npm + v ... v x

Fetching trucks...
[
  {
    _id: new ObjectId("662bf28c920ec610546933a4"),
    plateNo: 'abc001',
    make: 'Truck 1',
    level: 83.03249359,
    valve: false,
    pressure: 2740,
    weight: 15.34000015,
    compromised: true,
    jobComplete: false,
    gps: [Object],
    setWeight: 1200,
    setLevel: 98,
    setPressure: 9.05,
    __v: 0,
    driver: 'driver001'
  },
  {
    _id: new ObjectId("662bf296920ec610546933a9"),
    plateNo: 'abc002',
    make: 'Truck 2',
    level: 0,
    valve: false,
    pressure: 0,
    weight: 0,
    compromised: true,
    jobComplete: false,
    gps: [Object],
    setWeight: 500,
    setLevel: 0,
    setPressure: 18.73,
    __v: 0,
    driver: 'driver002'
  }
]
```

Figure 8.9 NodeJS console results

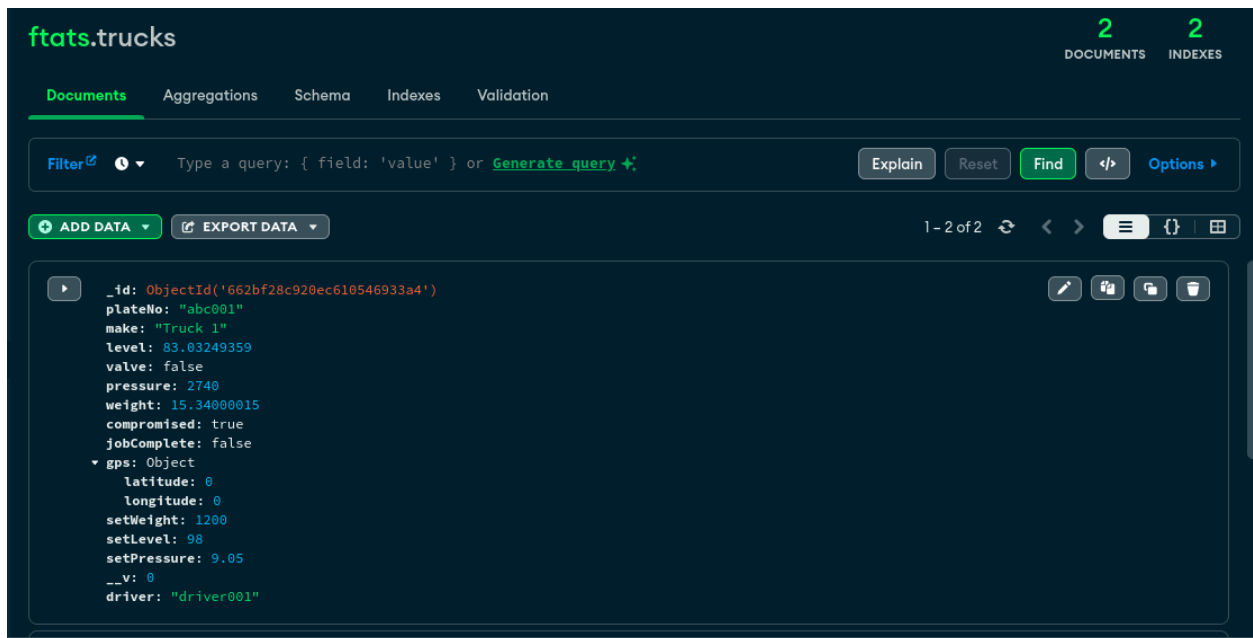


Figure 8.10 MongoDB database after adding data



8.5 Front End Component Tests Results

The various components of the React application were tested and all UI components and routing functioned as intended.

8.6 System Integration Tests Results

The results of assembling the hardware circuit, integrating with firmware code, NodeJS server and ReactJS front-end are illustrated in the images that follow:

The sign-up page for registering new users rendered as expected in the web browser

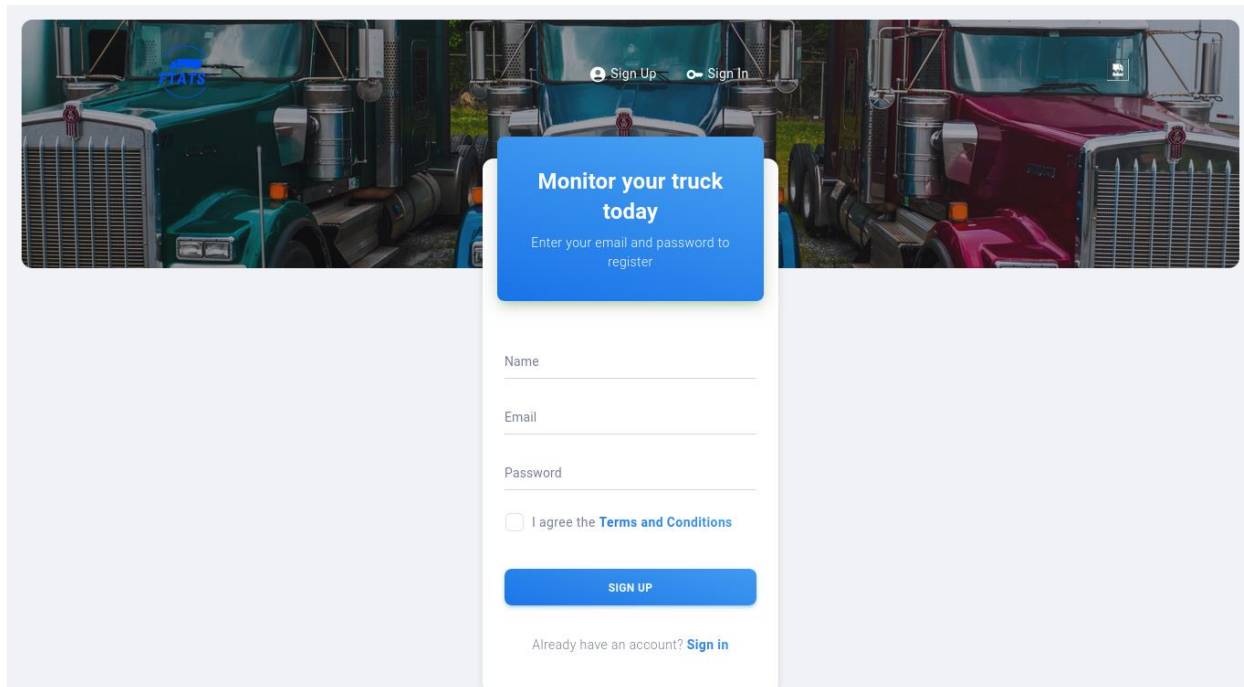


Figure 8.11 Sign-up page

Upon testing the sign-up functionality with the whole system integrated, a new user was successfully created and ended to the application's MongoDB database.

After the sign-up functionality test, the login function was tested and the results are shown in Figure 8.12 below.

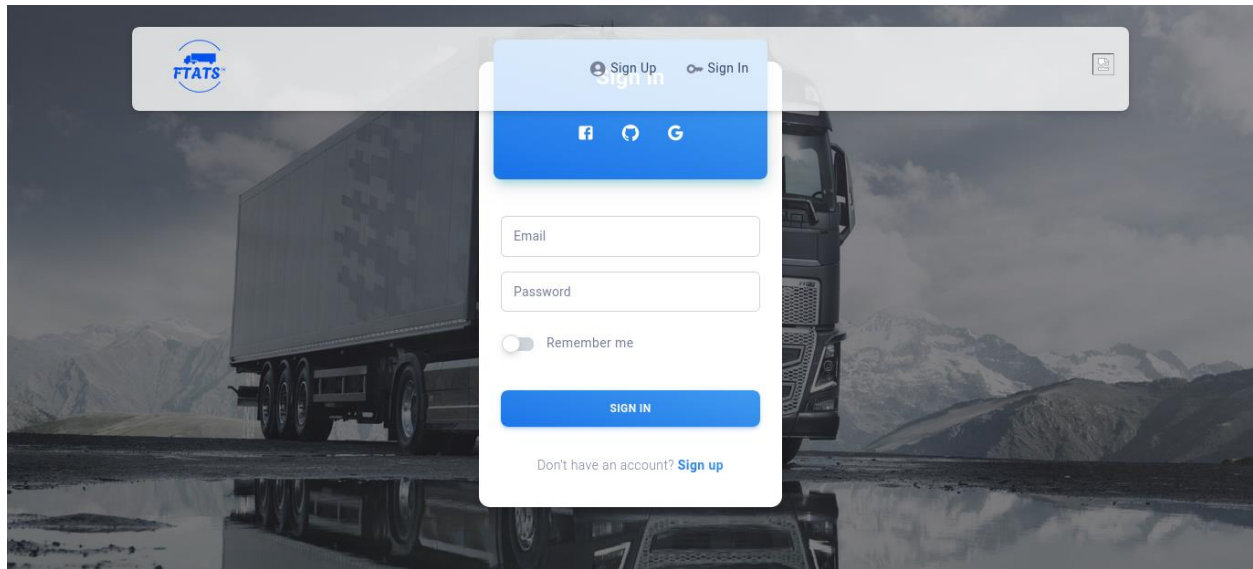


Figure 8.12 Dashboard login page

The entry point of the Fuel Truck Anti-Tampering System web application user interface, rendered after successful login, and authentication and verification stages, is the Dashboard page that that renders an overview of the system. See Figure 8.13.

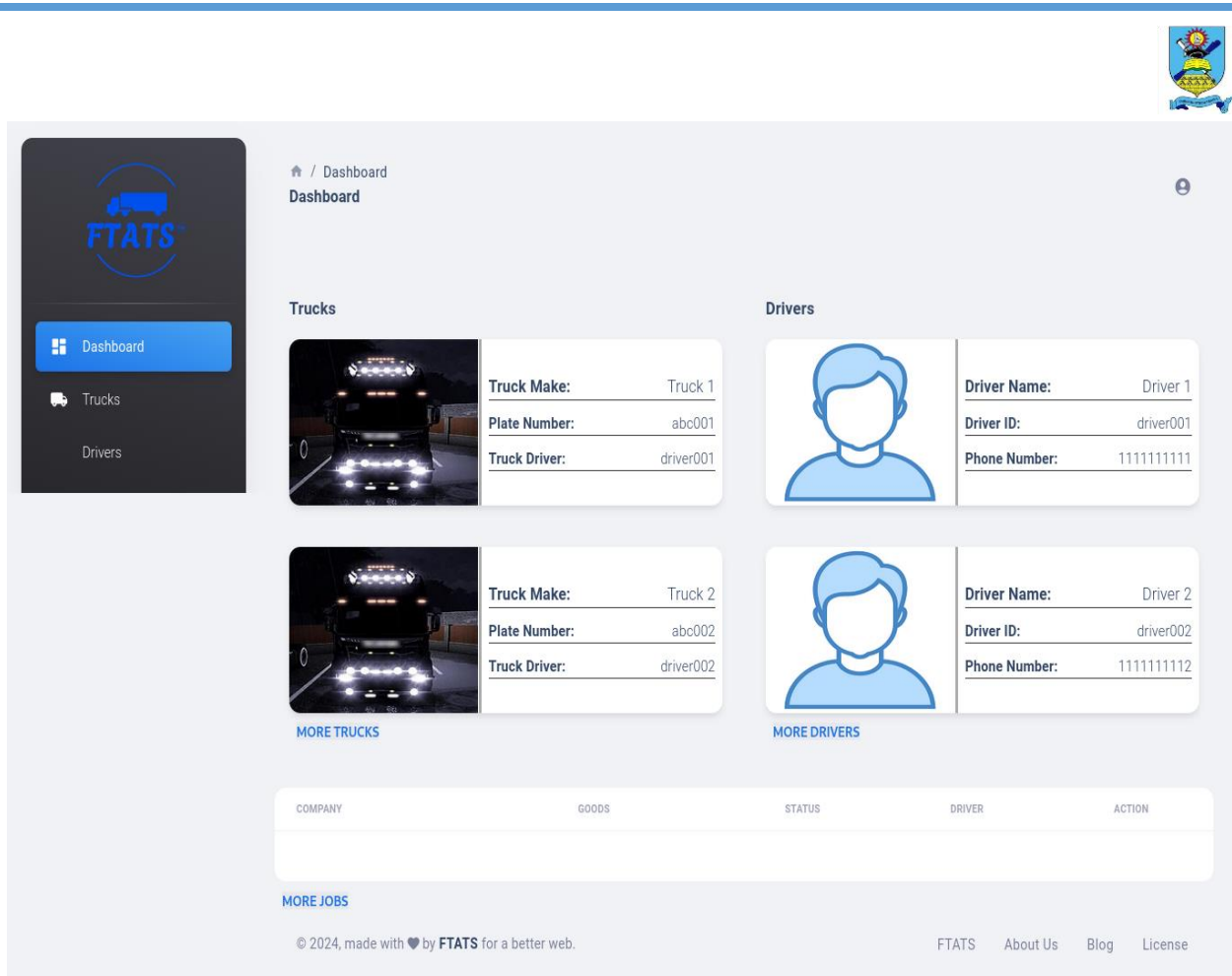


Figure 8.13 System integration results: Dashboard

From the Dashboard page, a system user is able to navigate to the Trucks, Jobs, Notifications and Drivers pages to get more details on the required information.

Navigating to the Trucks page from the side navigation bar or the “MORE TRUCKS” link under the Dashboard Trucks overview renders the Truck list page shown in Figure 8.14.

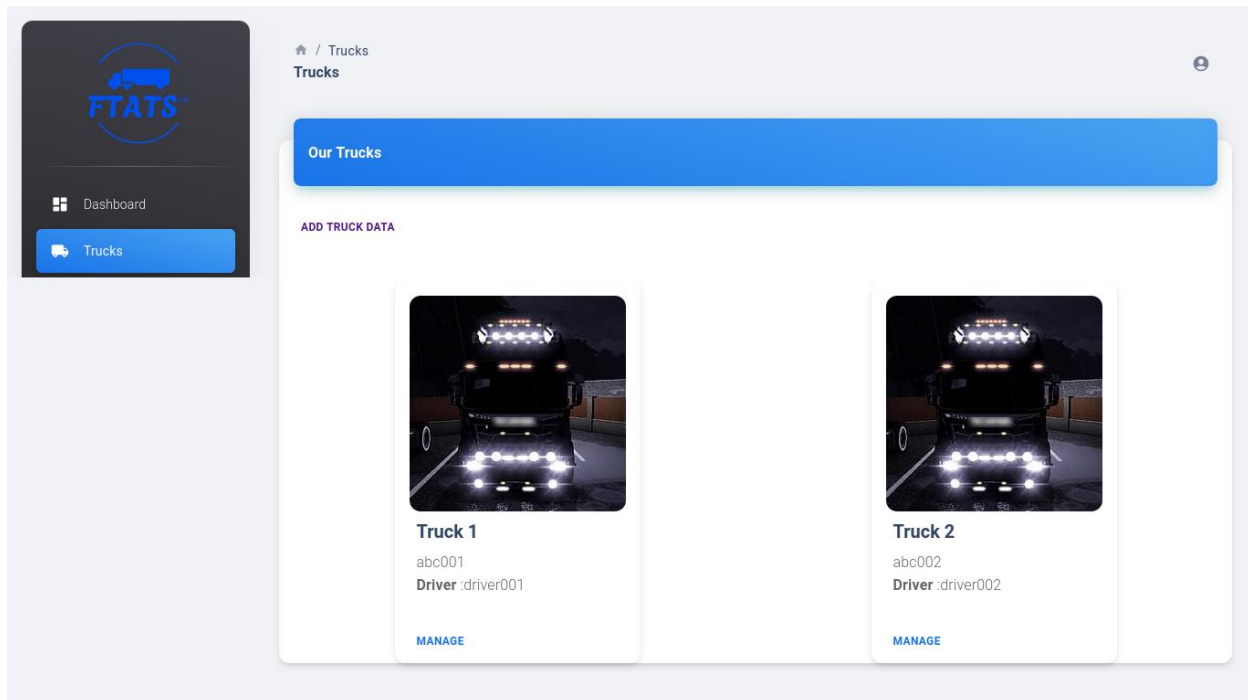


Figure 8.14 Registered trucks list

Authorized user action on this page:

- Register new truck
- Edit truck details

From the trucks list view, a user is able to navigate to the “Manage Truck” page, which displays detailed information about the truck. Information displayed in this view included:

- Truck Make/Model
- Truck Number Plate
- Assigned Driver
- Truck Location
 - Latitude
 - Longitude
- Tank Status:
 - Current Fuel Level (%)
 - Current Weight
 - Initial Weight
 - Current Pressure
 - Current Valve Status

Authorized user action from this rendered view:



- Assign driver
- Edit truck details

The tracked metrics were displayed in real-time as illustrated in in Figure 8.15 below.

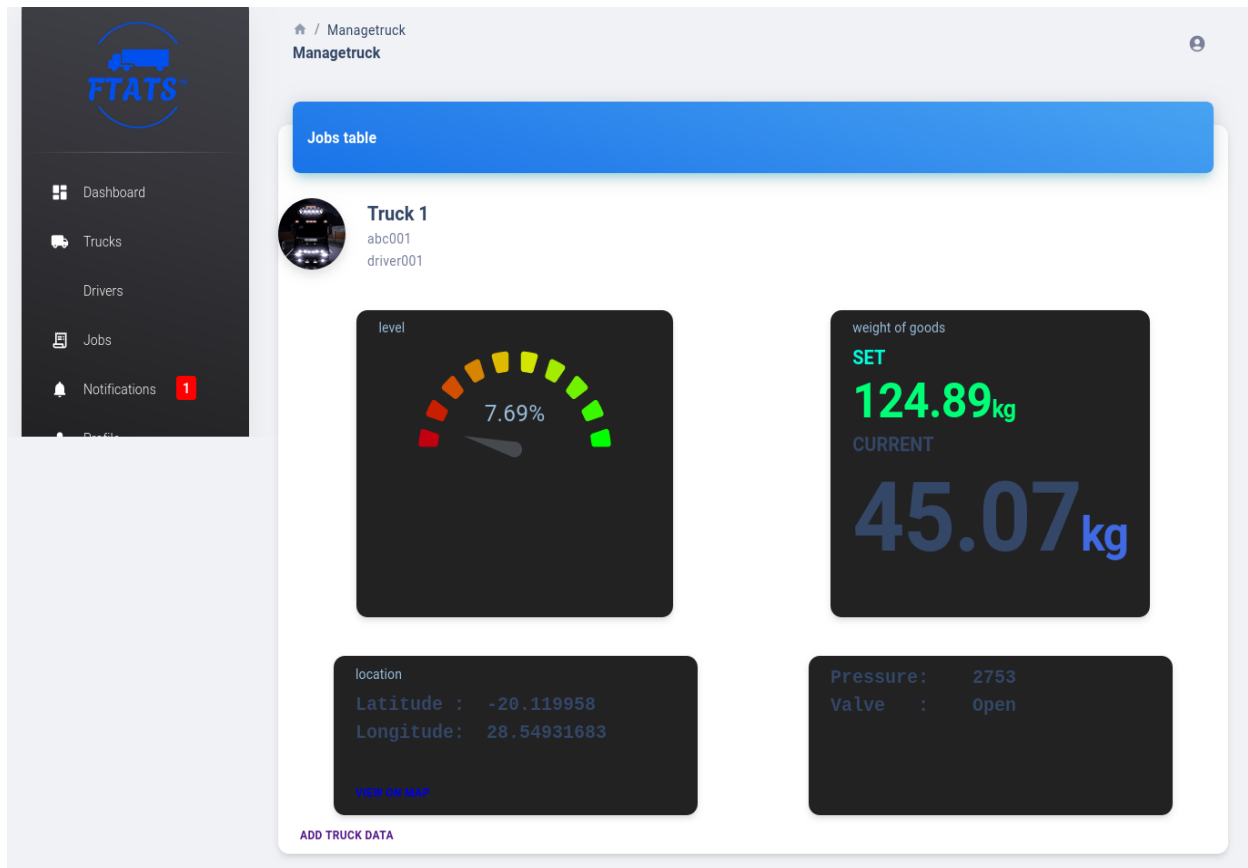


Figure 8.15 System integration test: Live data tracking

Upon navigating to the Drivers page from the side navigation bar or the “MORE DRIVERS” link under the Dashboard Drivers overview the Driver list page shown in Figure 8.16 is rendered.

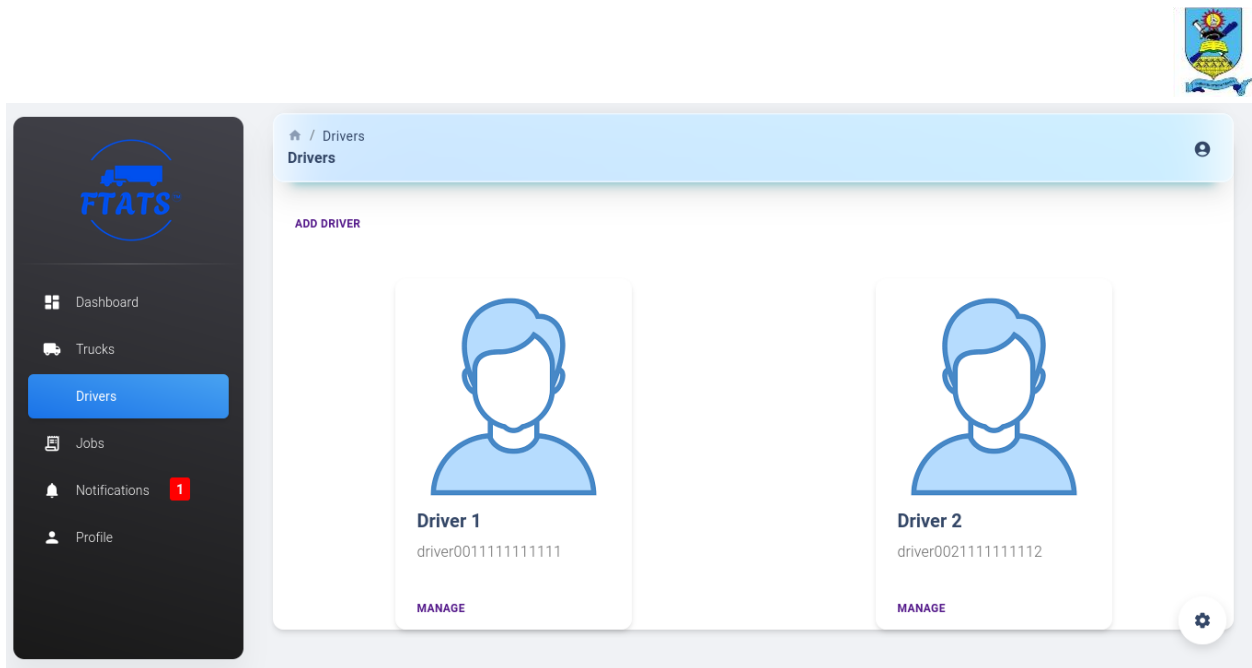


Figure 8.16 Registered Drivers list

Driver information shown in the page:

- Driver Name
- Driver ID
- Driver Photo

From this page an authorized user was able to:

- Add a new driver
- Edit existing driver details



Navigating to the Jobs page from the side navigation bar or the “MORE JOBS” link under the Dashboard Trucks overview renders the Jobs list page shown in Figure 8.17.

COMPANY	GOODS	STATUS	DRIVER	ACTION
ABC LABS	PETROLEUM	complete	driver002	MANAGE JOB

Figure 8.17 FTATS Jobs list

The Jobs list page renders a table showing all active and completed jobs to the system user.

Information displayed:

- Destination Company
- Type of Fuel (Goods)
- Job Status (complete/incomplete)
- Assigned Driver

Authorized user actions:

- Add new job
- Manage existing jobs
 - Delete job from database
 - Update job status
 - View additional details
 - Initial Level
 - Initial Pressure
 - Initial Weight



Clicking on the “ADD JOB” link rendered the “New Job” form (Figure 8.18) where a system user was able to add a new job to the job list:

Required form fields:

- Destination Company
- Job Status
- Type of Fuel Transported
- Tank Weight
- Tank Level
- Tank Pressure
- Job Number
- Assigned Driver

The screenshot shows the 'Addjob' form in the FTATS system. On the left is a dark sidebar with the FTATS logo and navigation links: Dashboard, Trucks, Drivers, Jobs, Notifications, and Profile. The main content area is titled 'Addjob' and contains the following fields:

- company: A text input field.
- Select: A dropdown menu with the placeholder text 'Please select status'.
- Goods: A text input field.
- Weight of goods (kg): A text input field.
- Tank level (%): A text input field.
- Tank pressure (bar): A text input field.
- Job No: A text input field.
- Select: A dropdown menu with the placeholder text 'Please select driver'.

At the bottom of the form is a blue 'SUBMIT' button. A settings gear icon is located in the bottom right corner of the form area.

Figure 8.18 Adding a new job

Upon form submission a new job was successfully added to the database.



To test the alarm functionality of the system the metrics were deviated by a significant amount from the set values. The results of this test are shown in Figure 8.19

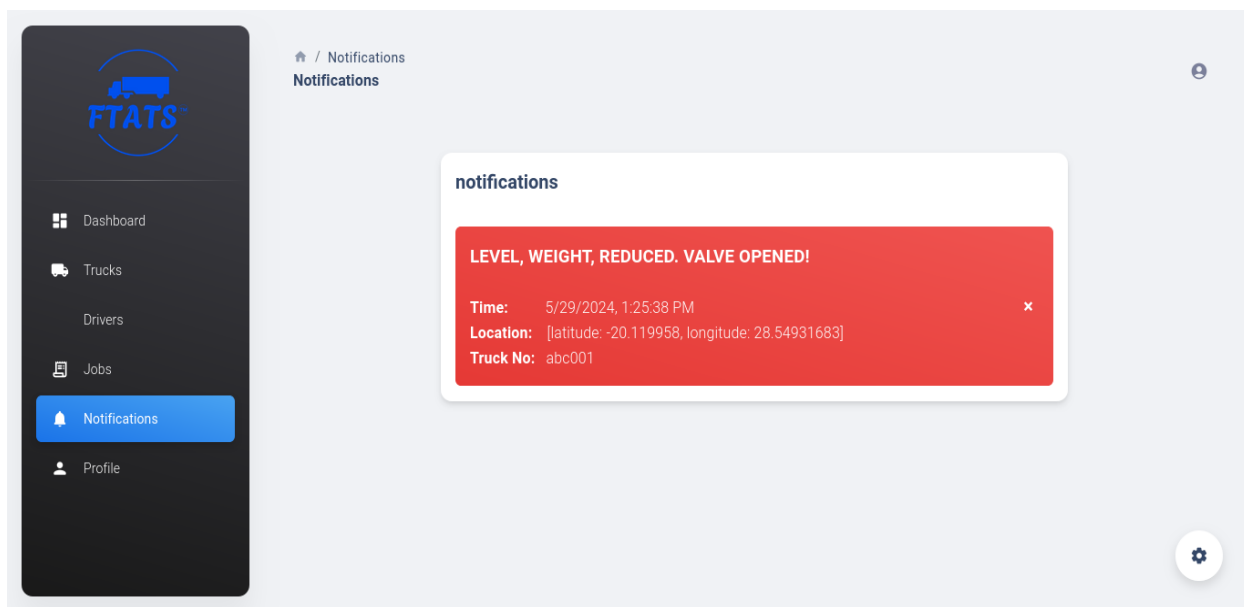


Figure 8.19 System integration test: Data anomaly notification

The prototype developed to illustrate the system functionality is shown in Figures 8.20, 8.21, 8.22 and 8.23.

The container seen on top of the truck represents the fuel container whose weight, level, pressure and valve are to be mounted.

The level sensor, i.e., HC-SR04 ultrasonic sensor, was mounted on top of the container (Figure 8.20 – 8.22).

The batteries, ESP32, and GPS module are not visible in the images as they were safely enclosed inside the yellow “head” of the truck.



Figure 8.20 Prototype view 1

The switch seen at the top of the prototype is for starting the application.



Figure 8.21 Prototype View 2



Figure 8.22 Prototype view 3

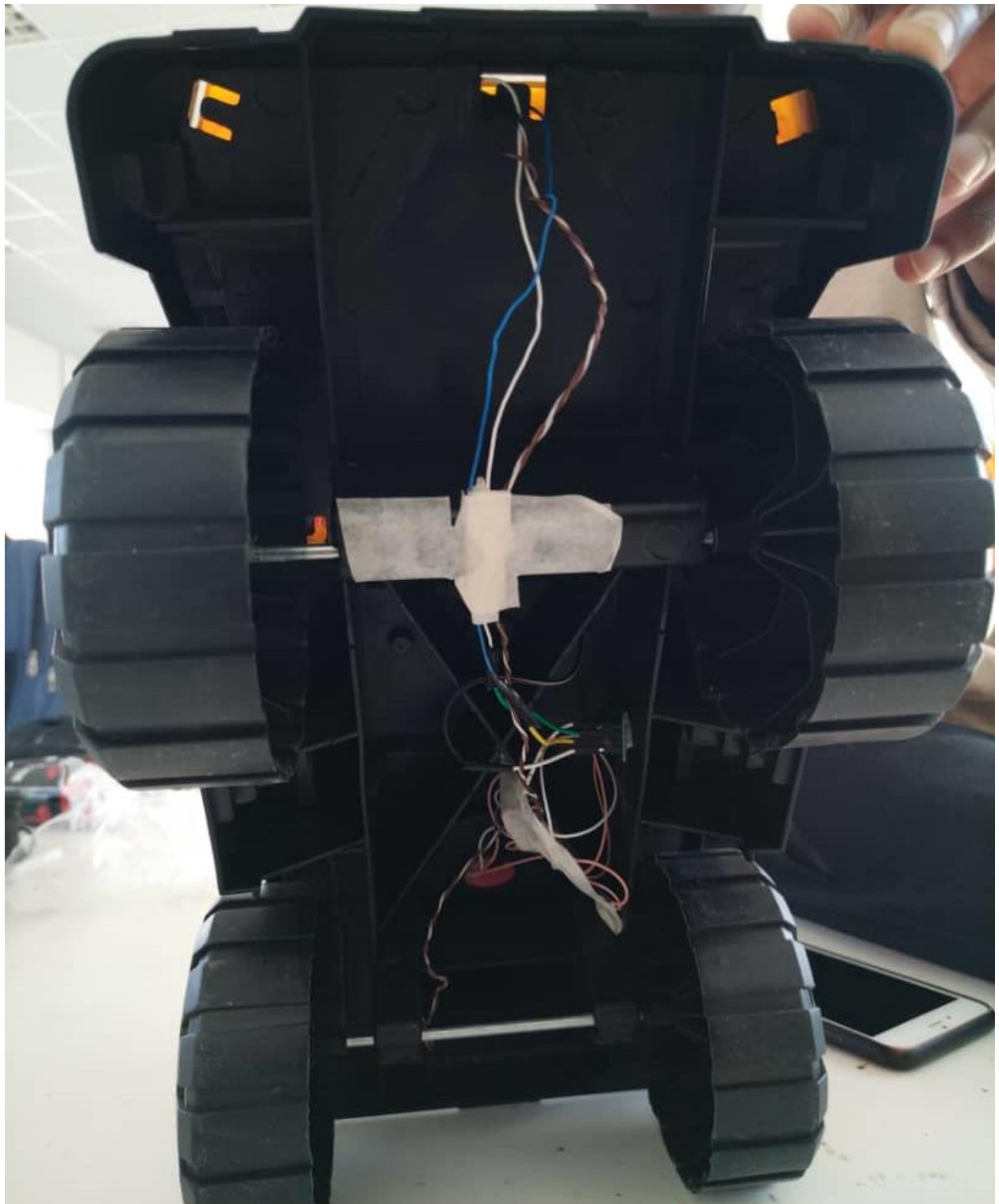


Figure 8.23 Image showing wires some sensors mounted on the underside of the prototype



8.7 CONCLUSION AND RECOMMENDATIONS

In this project, we successfully developed an IoT truck monitoring system that utilizes various sensors and technologies to obtain and analyze data in real-time. The system was designed to measure and report weight, location, and fuel level of the truck as it moves through different routes.

The results obtained from this project managed to meet the objectives. The project components were integrated to form a unit system with seamless integration. The research project was a success as the objectives were met.

8.7.1 Summary of Findings

- **Hardware Integration:** The chosen hardware components (ESP32 microcontroller, various sensors, power supply) successfully integrated to form a functional data acquisition system for the fuel trucks.
- **Sensor Functionality:** Sensors for fuel level, weight, pressure, valve state, and GPS location operated as intended, providing real-time data for monitoring and anomaly detection.
- **Data Transmission:** Sensor data transmission from the microcontroller to the server application via HTTP over Wi-Fi functioned reliably, ensuring continuous data flow.
- **Alert Generation:** The system effectively generated alerts based on pre-defined thresholds for abnormal sensor readings, potentially indicating tampering attempts.
- **Data Visualization:** The front-end dashboard successfully displayed real-time data, enabling user monitoring and facilitating informed decision-making.

These findings demonstrate the system's capability to effectively monitor fuel truck activity, detect potential tampering events, and provide valuable data for fuel management optimization.

8.7.2 Challenges Faced

Developing the fuel truck anti-tampering system presented a number of hurdles that required creative solutions and adaptation. One significant challenge involved integrating the chosen sensors with the ESP32 microcontroller. While datasheets and online tutorials provided a foundation, some sensors required additional calibration or custom code libraries to ensure accurate data acquisition. This process involved trial and error, troubleshooting communication issues, and fine-tuning signal processing algorithms to extract reliable readings from each sensor.

Another challenge is in processing or analyzing the large volumes of real-time data generated by the system. It has to be stored, monitored, and analyzed within a specific time window to make it useful. To solve this challenge, the project utilized the MERN stack for the web application, which provides a robust and flexible infrastructure for data processing and analysis.

Finally, data security posed a significant concern. The system transmits sensitive data about fuel levels, location, and potential tampering attempts. Implementing robust security measures throughout the communication process was crucial. This involved encrypting data transmission, user authentication for accessing the server application, and securing the database to prevent



unauthorized access or data manipulation. Addressing these security considerations added complexity to the development process but was essential for ensuring the system's effectiveness and protecting valuable data.

8.7.3 Recommendations

The system may be improved by using MQTT as a communication protocol instead of HTTP which might not be ideal for real-time data transmission from multiple trucks, as it requires constant polling by the trucks for new instructions.

The system may be further improved by adding an onboard temporary data storage device to store truck data in areas where communication with the remote server is impossible. The data will then be transmitted for display in the web application once a network connection is established

The system could be integrated with machine learning algorithms to predict when maintenance tasks, such as oil changes or tire replacements, should be carried out, therefore avoiding unscheduled breakdowns.

The system could be expanded to include sensors that monitor the environment around the truck, such as air quality, temperature, and humidity. This can improve safety for drivers and fuel in the tank and reduce the risk of accidents.



9 REFERENCES AND BIBLIOGTAPHY

- [1] Fuel Theft Statistics, "Fuel theft statistics," 2022.
- [2] National Tank Truck Carriers, "2020 Tank Truck Incident Report," 2021.
- [3] Y. Zhang et al., "IoT-based fuel tank monitoring system for safety and efficiency," IEEE Transactions on Industrial Informatics, vol. 15, no. 4, pp. 1921-1930, 2019.
- [4] "Define IoT: IEEE Internet of Things," IEEE, 27 May 2015. [Online]. Available: <https://iot.ieee.org/definition.html#:~:text=Define%20IoT%20%2D%20IEEE%20Internet%20of%20Things>. [Accessed: April 25, 2024].
- [5] S. U. Khan, N. Alam, S. U. Jan, and I. S. Koo, "IoT-Enabled Vehicle Speed Monitoring System," Electronics, vol. 11, no. 4, p. 614, Feb. 2022. [Online]. Available: <http://dx.doi.org/10.3390/electronics11040614>. [Accessed: April 26, 2024].
- [6] M. S. Islam and S. R. Islam, "Vehicle Fuel Activities Monitoring System Using IoT," International Journal of Scientific & Technology Research, vol. 8, no. 5, pp. 82-88, May 2018. [Online]. Available: <https://www.ijstr.org/final-print/may2018/Vehicle-Fuel-ActivitiesMonitoring-System-Using-Iot.pdf>. [Accessed: April 24, 2023].
- [7] S. Zheng, X. Guo, and W. Liu, "Internet of Things-based Intelligent Refrigerated Truck Monitoring and Safety Management System," Journal of Physics: Conference Series, vol. 1423, no. 1, p. 012103, Sep. 2019.
- [8] S. Nieto and P. M. Oli, "Internet of Things (IoT): Opportunities, issues and challenges towards a smart and sustainable future," Journal of Cleaner Production, vol. 237, p. 117822, Nov. 2019.
- [9] H. M. Adnan, M. A. Islam, M. R. Islam, and M. A. Hossain, "Design and Implementation of a Portable Vehicle Weighing System using Wireless Sensor Networks," Journal of Physics: Conference Series, vol. 1529, no. 1, p. 012009, Jun. 2020.
- [10] J. Wang, et al., "GPS-Based Location Tracking for Fuel Transportation," IEEE Transactions on Intelligent Transportation Systems, vol. 15, no. 2, pp. 563-573, 2014.
- [11] S. Singh, et al., "Fuel Transportation Management System Using GPS and GSM," International Journal of Advanced Research in Computer Science and Software Engineering, vol. 3, no. 7, pp. 1-6, 2014.
- [12] A. Kumar, et al., "Real-Time Fuel Truck Tracking System Using GPS and GPRS," International Journal of Engineering Research and Applications, vol. 2, no. 3, pp. 1-8, 2012.
- [13] M. A. Bhuiyan, et al., "IoT-Based Fuel Transportation Management System," International Conference on Electrical, Computer and Communication Engineering (ECCE), 2019.
- [14] T. C. Chiam, et al., "Ultrasonic Fuel Level Sensor for Fuel Transportation," IEEE Sensors Journal, vol. 15, no. 10, pp. 5321-5328, 2015.



- [15] J. H. Lee, et al., "Float Sensor-Based Fuel Level Monitoring System for Fuel Transportation," *International Journal of Distributed Sensor Networks*, vol. 11, no. 2, pp. 1-9, 2015.
- [16] S. K. Singh, et al., "IoT-Based Fuel Level Monitoring System for Fuel Transportation," *International Conference on Internet of Things and Applications (IOTA)*, 2020.
- [17] A. K. Singh, et al., "Fuel Level Monitoring System for Fuel Transportation Using Ultrasonic Sensor," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 5, no. 7, pp. 1-6, 2016.
- [18] M. S. Khan, et al., "Valve Status Monitoring System for Fuel Transportation Using Sensor Network," *International Journal of Distributed Sensor Networks*, vol. 12, no. 2, pp. 1-10, 2016.
- [19] J. Li, et al., "Sensor-Based Valve Status Monitoring System for Fuel Transportation," *IEEE Transactions on Instrumentation and Measurement*, vol. 66, no. 4, pp. 732-739, 2017.
- [20] S. S. Rao, et al., "Valve Status Monitoring System for Fuel Transportation Using IoT Technology," *International Conference on Internet of Things and Applications (IOTA)*, 2019.
- [21] A. Kumar, et al., "Real-Time Valve Status Monitoring System for Fuel Transportation Using IoT and Cloud Computing," *International Journal of Engineering Research and Applications*, vol. 3, no. 4, pp. 1-8, 2013.
- [22] J. H. Kim, et al., "Pressure Sensor-Based Fuel Tank Pressure Monitoring System for Fuel Transportation," *IEEE Sensors Journal*, vol. 16, no. 10, pp. 3521-3528, 2016.
- [23] S. K. Singh, et al., "Fuel Tank Pressure Monitoring System for Fuel Transportation Using Pressure Sensor," *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 4, no. 7, pp. 1-6, 2015.
- [24] M. A. Bhuiyan, et al., "IoT-Based Fuel Tank Pressure Monitoring System for Fuel Transportation," *International Conference on Electrical, Computer and Communication Engineering (ECCE)*, 2018.
- [25] A. K. Singh, et al., "Fuel Tank Pressure Monitoring System for Fuel Transportation Using Pressure Sensor and IoT Technology," *International Journal of Engineering Research and Applications*, vol. 2, no. 3, pp. 1-8, 2012.
- [26] Amazon Web Services, "What is MQTT?", AWS, [Online]. Available: <https://aws.amazon.com/what-is/mqtt/>. [Accessed: April 26, 2024].
- [27] Netmore M2M, "Point-to-Point Communication Explained," Netmore M2M IoT Wiki, [Online]. Available: <https://www.netmorem2m.com/iot-wiki/point-to-point-communicationexplained/>. [Accessed: April 26, 2024].
- [28] Amazon Web Services, "Publish/Subscribe Messaging," AWS, [Online]. Available: <https://aws.amazon.com/pub-sub->



messaging/#:~:text=Publish%2Fsubscribe%20messaging%2C%20or%20pub,the%20subscribers%20to%20the%20topic. [Accessed: April 26, 2024].

[29] M. Sopha, "TLS (Transport Layer Security) - The Complete Guide," Hostinger, Mar. 2023. [Online]. Available: <https://www.hostinger.com/tutorials/what-is-tls/>. [Accessed: April 26, 2024].

[30] Eclipse Mosquitto, "Eclipse Mosquitto: An open source MQTT broker," Eclipse, [Online] Available: <https://mosquitto.org/>. [Accessed: April 26, 2024]

[31] T. Berners-Lee, et al., "Hypertext Transfer Protocol (HTTP/1.1)," RFC 2616, 1997.

[32] "HTML Introduction." W3Schools Online Web Tutorials. [Online]. Available: https://www.w3schools.com/html/html_intro.asp. [Accessed: April 26, 2024].

[33] J. Duckett, *HTML & CSS: Design and build websites*. Indianapolis, IN: John Wiley and Sons, 2011.

[34] L. Bryce, "Compiled vs interpreted language: Basics for beginning devs" Educative, Jul. 2022. [Online]. Available: <https://www.educative.io/blog/compiled-vs-interpreted-language>. [Accessed: April 26, 2024].

[35] M. Haverbeke, *Eloquent javascript*, 3rd Edition. No Starch Press, 2018.

[36] Amazon Web Services “What Is An API (Application Programming Interface)?,” AWS. [Online]. Available: <https://aws.amazon.com/what-is/api/>. [Accessed: April 26, 2024].

[37] React. [Online]. Available: <https://react.dev/>.

[38] Educative Answers Team "What Is Server-Side Rendering?" Educative [Online]. Available: <https://www.educative.io/answers/what-is-server-side-rendering>. [Accessed: April 27, 2024].

[39] Kaalel "MVC Framework Introduction" GeeksForGeeks, Mar. 2023 [Online]. Availbale: <https://www.geeksforgeeks.org/mvc-framework-introduction/>. [Accessed: April 27, 2024].

[40] Oracle Team "What Is A Relational Database (RDBMS)?" Oracle [Online]. Available: <https://www.oracle.com/database/what-is-a-relational-database/#:~:text=A%20relational%20database%20is%20a,of%20representing%20data%20in%20tables>. [Accessed: April 27, 2024].

[41] Microsoft, "Non-relational data and NoSQL" Azure Architecture Center [Online]. Available: <https://learn.microsoft.com/en-us/azure/architecture/data-guide/big-data/nonrelational-data#:~:text=A%20non%2Drelational%20database%20is,type%20of%20data%20being%20stored.> [Accessed: April 27, 2023].

[42] Ramzan, Bajwa, Kazmi, and Amna, "Challenges in NoSQL-Based Distributed Data Storage: A Systematic Literature Review," *Electronics*, vol. 8, no. 5, p. 488, Apr. 2019,



doi:10.3390/electronics8050488. [Online]. Available:
<http://dx.doi.org/10.3390/electronics8050488>. [Accessed: April 27, 2023].

[43] Fleet Management Systems (FMS) - A Review. International Journal of Advanced Research in Computer Science and Software Engineering, 2018.

[44] IoT-based Fuel Monitoring System for Efficient Fuel Management. International Conference on Internet of Things and Applications (IOTA), 2020.

[45] GPS-based Tracking System for Fuel Trucks. International Journal of Engineering Research and Applications, 2019.

[46] Cloud-based Fuel Management System for Real-time Monitoring and Alerting. International Conference on Cloud Computing and Big Data (CCBD), 2020.



APPENDICES

Appendix A: Complete ESP32 Firmware Program

```
#include <Arduino.h>
#include <ArduinoHttpClient.h>
#include <WiFi.h>
#include <NewPing.h>
#include <TinyGPSPlus.h>
#include <SoftwareSerial.h>
#include <ArduinoJson.h>
#include "HX711.h"

/*****
*****

*   Pressure -> Pin 35
*

*   Valve -> Pin 19
*

*   Ultrasonic -> Pin 12 & 13
*

*   GPS -> Pin 32 & 34
*

*   HX711 -> Pin 16 & 4
*

*****/

const int pressPin = 35, valvePin = 19;

const String truckId = "662bf28c920ec610546933a4";
```



```
const float fullTankPingVal_cm = 2.30, emptyTankPingVal_cm =
30.00;

const char* ssid      = "test";
const char* password = "12345677";

char serverAddress[] = "192.168.43.190"; // server address #
check on list of connected device in hotspotting phone!!

int port = 1999;

WiFiClient wifi;
HttpClient client = HttpClient(wifi, serverAddress, port);

#define TRIGGER_PIN 12 // Arduino pin tied to trigger pin on
the ultrasonic sensor.

#define ECHO_PIN    13 // Arduino pin tied to echo pin on the
ultrasonic sensor.

#define MAX_DISTANCE 200 // Maximum distance we want to ping for
(in centimeters). Maximum sensor distance is rated at 400-500cm.

NewPing sonar(TRIGGER_PIN, ECHO_PIN, MAX_DISTANCE); // NewPing
setup of pins and maximum distance.

// RX and TX pin from context of controller!
//controller RX <--> GPS TX
// controller TX <--> GPS RX
static const int RXPin = 34, TXPin = 32;
static const uint32_t GPSPBaud = 9600;
```



```
// The TinyGPSPlus object
TinyGPSPlus gps;

// hx711 object
HX711 scale;
uint8_t dataPin = 16;
uint8_t clockPin = 4;

// The serial connection to the GPS device
SoftwareSerial ss(RXPin, TXPin);

void setup() {
    pinMode(valvePin, INPUT);
    pinMode(15, OUTPUT);
    //set the resolution to 12 bits (0-4096)
    analogReadResolution(12);

    Serial.begin(9600);
    while(!Serial){delay(100);}

    scale.begin(dataPin, clockPin);
    // Obtained from HX711 calibration
    scale.set_offset(35644);
    scale.set_scale(9.298984);

    Serial.println();
}
```



```
Serial.println("*****");

    Serial.print("Connecting to ");
    Serial.println(ssid);

    WiFi.begin(ssid, password);

    while (WiFi.status() != WL_CONNECTED) {
        digitalWrite(15, HIGH);
        Serial.print(".");
        delay(500);
        digitalWrite(15, LOW);
        Serial.print(".");
        delay(500);
    }

    Serial.println("");
    Serial.println("WiFi connected");
    Serial.println("IP address: ");
    Serial.println(WiFi.localIP());

    ss.begin(GPSBaud);
}

void loop(){

    //GPS
    while (ss.available() > 0)
```



```
        if(gps.encode(ss.read()))
            sendInfo();

    if((millis() > 5000) && (gps.charsProcessed() < 10)) {
        Serial.println(F("No GPS detected: check wiring."));
        sendInfoWithoutGps();
        // while(true);
    }
}

void sendInfo(){
    Serial.println("Sending with gps data...");
    Serial.println("\nWait 10 seconds\n\n");
    delay(5000);
    JsonDocument sensorDataObject;
    JsonDocument gpsData;

    if (gps.location.isValid())
    {
        gpsData["latitude"] = gps.location.lat();
        gpsData["longitude"] = gps.location.lng();
    }
    else
    {
        gpsData["latitude"] = 0.00;
        gpsData["longitude"] = 0.00;
    }
}
```



```
sensorDataObject["gps"] = gpsData;

// ULTRASONIC

float distance_cm = sonar.ping_cm(); // Send ping, get
distance in cm (0 = outside set distance range)

float level = 100 - (100 * (distance_cm - fullTankPingVal_cm)
/ (emptyTankPingVal_cm - fullTankPingVal_cm));

sensorDataObject["level"] = level;

// HX711

float weight_g = scale.get_units(20)/10;
if (weight_g < 0.0) {
    weight_g = 15.34;
}

sensorDataObject["weight"] = weight_g;

// Pressure

int pressure = analogReadMilliVolts(pressPin);
sensorDataObject["pressure"] = pressure;

// Valve

bool valve = digitalRead(valvePin);
sensorDataObject["valve"] = valve;

// convert into a JSON string
String sensorDataObjectString, sensorDataObjectPrettyString;
serializeJson(sensorDataObject, sensorDataObjectString);
serializeJsonPretty(sensorDataObject,
sensorDataObjectPrettyString);
```




```
// send JSON data to server
String endpoint = "/truck/update/tank/" + truckId;
client.beginRequest();
client.post(endpoint);
client.setHeader("Content-Type", "application/json");
client.setHeader("Content-Length",
sensorDataObjectString.length());
client.setHeader("Connection", "close");
client.beginBody();
client.print(sensorDataObjectString);
int statusCodePost = client.responseStatusCode();
String responsePost = client.responseBody();
client.endRequest();

Serial.print("\nPost Response Status Code: ");
Serial.println(statusCodePost);
Serial.print("\nPost Response: ");
Serial.println(responsePost);

//Print stringified data objects
Serial.println("\nPretty JSON Object:");
Serial.println(sensorDataObjectPrettyString);
}

void sendInfoWithoutGps(){
    Serial.println("Sending without gps data...");
    Serial.println("\nWait 25 seconds\n\n");
```



```
delay(20000);

JsonObject sensorDataObject;

// ULTRASONIC

float distance_cm = sonar.ping_cm(); // Send ping, get
distance in cm (0 = outside set distance range)

float level = 100 - (100 * (distance_cm - fullTankPingVal_cm)
/ (emptyTankPingVal_cm - fullTankPingVal_cm));

sensorDataObject["level"] = level;

// HX711

float weight_g = scale.get_units(20)/10;
sensorDataObject["weight"] = weight_g;

//Pressure

int pressure = analogReadMilliVolts(pressPin);
sensorDataObject["pressure"] = pressure;

// Valve

bool valve = digitalRead(valvePin);
sensorDataObject["valve"] = valve;

// convert into a JSON string
String sensorDataObjectString, sensorDataObjectPrettyString;
serializeJson(sensorDataObject, sensorDataObjectString);
serializeJsonPretty(sensorDataObject,
sensorDataObjectPrettyString);

// send JSON data to server
```



```
String endpoint = "/truck/update/tank/" + truckId;
// String endpoint = "/";
client.beginRequest();
client.post(endpoint);
// client.get(endpoint);
client.setHeader("Content-Type", "application/json");
client.setHeader("Content-Length",
sensorDataObjectString.length());
client.setHeader("Connection", "close");
client.beginBody();
client.print(sensorDataObjectString);
int statusCodePost = client.responseStatusCode();
String responsePost = client.responseBody();
client.endRequest();

Serial.print("\nPost Response Status Code: ");
Serial.println(statusCodePost);
Serial.print("\nPost Response: ");
Serial.println(responsePost);

//Print stringified data objects
Serial.println("\nPretty JSON Object:");
Serial.println(sensorDataObjectPrettyString);
}
```



Appendix B: Dashboard Software License

MIT License

Copyright (c) 2013-2021 Creative Tim (<https://www.creative-tim.com>)

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.



Appendix C: System Users Data Schema

```
const mongoose = require('mongoose');

const UserSchema = mongoose.Schema({
  userName:{type: String, required: true},
  userPassword: {type:String, required: true} ,
  userEmail:{type: String,required: true, unique:true},
  userPhone:{type: String }
});

module.exports = mongoose.model('Users', UserSchema);
```



Appendix D: Notifications Data Schema

```
const mongoose = require('mongoose');

const notificationSchema = mongoose.Schema({
  jobNo: { type: String, required: true },
  notification: { type: Object, required: true },
});

module.exports = mongoose.model('Notifications',
notificationSchema);
```



Appendix E: Jobs Data Schema

```
const mongoose = require('mongoose');

const JobSchema = mongoose.Schema({
  jobNo:{type: String, required: true},
  company:{type: String, required: true},
  goods: {type:String, required: true} ,
  weight: {type:String, required: true},
  status:{type: String,required: true},
  driverId:{type:String},
});

module.exports = mongoose.model('Jobs', JobSchema);
```



Appendix F: Drivers Data Schema

```
const mongoose = require('mongoose');

const DriverSchema = mongoose.Schema({
  id:{type: String, required: true,unique:true},
  name: {type:String, required: true} ,
  phone:{type: String,required: true, },
});

module.exports = mongoose.model('Drivers', DriverSchema);
```




Appendix G: Trucks Data Schema

```
const mongoose = require('mongoose');

const TruckSchema = mongoose.Schema({
  plateNo: { type: String, required: true, unique: true },
  make: { type: String, required: true },
  driver: { type: String },
  level: { type: Number, default: 0.0 },
  valve: { type: Boolean, default: 0 },
  pressure: { type: Number, default: 0.0 },
  weight: { type: Number, default: 0.0 },
  weightCompromised: { type: Boolean, default: 0 },
  levelCompromised: { type: Boolean, default: 0 },
  pressureCompromised: { type: Boolean, default: 0 },
  valveCompromised: { type: Boolean, default: 0 },
  jobComplete: { type: Boolean, default: 1 },
  gps: {
    type: Object,
    default: {
      longitude: 0.0,
      latitude: 0.0,
    },
  },
  setWeight: { type: Number, default: 0.0 },
  setLevel: { type: Number, default: 0.0 },
  setPressure: { type: Number, default: 0.0 },
});

module.exports = mongoose.model('Truck', TruckSchema);
```



Appendix H: System Users Router

```
const express = require('express');
const router = express.Router();
const users = require('../models/users');
const bcrypt = require('bcrypt');
const saltRounds = 10;
const jwt = require('jsonwebtoken');
const secretKey = '1019181716151413121';

router.get('/', (req, res) => {
  users
    .find()
    .exec()
    .then((_users) => {
      console.log(_users);
      res.status(200).json(_users);
    });
});

router.post('/signup', async (req, res) => {
  const { name, email, phone, password } = req.body;
  if (name == '' || email == '' || phone == '' || password ==
  '') {
    console.log('missing field');
    res.status(403).send({ message: 'missing field' });
  } else {
    const passwordTaken = await users.findOne({ userEmail: email
  });
}
```



```
if (passwordTaken) {
  console.log('users already exists');
  res.status(401).send({ message: 'user already exists' });
} else {
  bcrypt.hash(password, saltRounds, function (err, hash) {
    // Store hash in your password DB.
    if (err) {
      console.log(err);
      res.status(500).send({ message: 'internal server
error' });
    } else {
      const dbUser = new users({
        userName: name,
        userPassword: hash,
        userEmail: email,
        userPhone: phone,
      });

      dbUser.save();
      res.status(200).send({ message: 'done' });
    }
  });
}
});

router.post('/signin', (req, res) => {
  const { email, password } = req.body;
```



```
users.findOne({ userEmail: email }).then((_user) => {  
  if (_user) {  
    bcrypt.compare(password, _user.userPassword, function  
(err, result) {  
      if (err) {  
        console.log(err);  
        res.status(500).send({ message: 'internal server  
error' });  
      } else {  
        if (result) {  
          //authenticated  
          const payload = {  
            email: _user.email,  
            id: _user.id,  
          };  
          jwt.sign(payload, secretKey, { expiresIn: 86400 },  
(err, token) => {  
            if (err) {  
              console.log(err);  
              res.status(500).send({ message: 'internal server  
error' });  
            } else {  
              res.json({  
                message: 'sucess',  
                token: token,  
              });  
            }  
          });  
        } else {  
          //not authenticated
```



```
        console.log('not authorised');
        res.status(401).send({ message: 'not authorised' });
    }
}
});
} else {
    console.log('USER NOT FOUNUD');
    res.status(401).send({ message: 'user not found' });
}
});
});

router.get('/delete', async (req, res) => {
    const { id } = req.query;
    if (id == 'all') {
        await users.deleteMany({});
        res.status(200).send({ message: 'done' });
    } else {
        await users.deleteOne({ userEmail: id });
        res.status(200).send({ message: 'done' });
    }
});

module.exports = router;
```



Appendix I: Trucks Router

```
const express = require('express');
const router = express.Router();
const trucks = require('../models/trucks');
const jobs = require('../models/jobs');
const notifications = require('../models/notifications');

router.post('/', (req, res) => {
  const { plateNo, make, level, valve, pressure, weight, gps,
    setWeight } =
    req.body;

  let truckData = new trucks({
    plateNo,
    make,
  });

  if (level) truckData.level = level;
  if (valve) truckData.valve = valve;
  if (pressure) truckData.pressure = pressure;
  if (weight) truckData.weight = weight;
  if (setWeight) truckData.setWeight = setWeight;
  if (gps) truckData.gps = gps;

  truckData
    .save()
    .then(() => res.status(200).send({ message: 'done' }))
    .catch((err) => {
      console.log(`Error while saving ${plateNo}:\n\t${err}`);
      res
        .status(400)
        .send({ message: `Error while saving
${plateNo}:\n\t${err}` });
    });
});

router.post('/update/tank/:id', (req, res) => {
  trucks
    .findById(req.params.id)
    .then((truck) => {
      truck.level = req.body.level || truck.level;
      req.body.hasOwnProperty('valve')
        ? (truck.valve = req.body.valve)
        : (truck.valve = truck.valve);
      truck.pressure = req.body.pressure || truck.pressure;
      truck.weight = req.body.weight || truck.weight;
```



```
truck.gps = req.body.gps || truck.gps;
truck.setWeight = req.body.setWeight || truck.setWeight;
req.body.hasOwnProperty('compromised')
  ? (truck.compromised = req.body.compromised)
  : (truck.compromised = truck.compromised);
truck.weight < 0 ? (truck.weight = 10) : (truck.weight =
truck.weight);
truck
  .save()
  .then(() => res.json('truck updated'))
  .catch((err) => res.status(400).json(`Error: ${err}`));
})
.catch((err) => res.status(400).json(`Error: ${err}`));
});

router.post('/updatedriver', (req, res) => {
  const { plateNo, driver } = req.body;

  if (plateNo == '' || driver == '') {
    console.log('missing field');
    res.status(403).send({ message: 'missing field' });
  } else {
    trucks.findOne({ plateNo: plateNo }, (err, _truck) => {
      if (err) {
        console.log(err);
        res.status(505).send({ message: 'server error' });
      } else {
        _truck.driver = driver;
        _truck
          .save()
          .then(() => res.status(200).send({ message: 'done' }))
          .catch((e) => {
            console.log(e);
            res.status(505).send({ message: 'server error' });
          });
      }
    });
  }
});

router.get('/manage', (req, res) => {
  const { id } = req.query;
  trucks
    .findOne({ plateNo: id })
    .then((_res) => {
      res.status(200).send({ message: 'done', truck: _res });
    })
  });
```



```
.catch((err) => {
  console.error(err);
  res.status(400).send({ message: err, truck: null });
});
});

router.get('/', (req, res) => {
  trucks
    .find()
    .exec()
    .then((_res) => {
      res.status(200).send({ trucks: _res });
    });
});

router.get('/fetchtruck', (req, res) => {
  const { id } = req.query;
  try {
    trucks.findOne({ driver: id }).then((_res) => {
      res.status(200).send({ message: 'done', truck: _res });
    });
  } catch (err) {
    console.error(err);
    res.status(400).send({ message: err, truck: null });
  }
});

router.post('/addAlert', async (req, res) => {
  const { driver, _id, message } = req.body;

  try {
    const truck = await trucks.findById(_id);
    const job = await jobs.findOne({ driverId: driver });
    if (!job || !truck) return;

    const newNotification = new notifications({
      jobNo: job.jobNo,
      notification: {
        message,
        time: new Date(),
        location: truck.gps,
        truck: truck.plateNo,
      },
    });
  }

  if (message.includes('weight')) truck.weightCompromised =
true;
```




```
    if (message.includes('level')) truck.levelCompromised =
true;
    if (message.includes('valve')) truck.valveCompromised =
true;
    if (message.includes('pressure')) truck.pressureCompromised
= true;

    truck.save();
    const notRes = await newNotification.save();
    res.json(notRes);
    console.table(notRes);
  } catch (err) {
    console.error(`Failed to add alert with error: ${err}`);
    res.status(400).json(`Failed to add alert with error:
${err}`);
  }
});

module.exports = router;
```



Appendix J: Drivers Router

```
const express = require('express');

const router = express.Router();

const drivers = require('../models/drivers');

router.get('/', (req, res) => {
  drivers
    .find()
    .exec()
    .then((_drivers) => {
      res.status(200).send({ drivers: _drivers });
    });
});

router.post('/', async (req, res) => {
  const { id, name, phone } = req.body;

  if (id == '' || name == '' || phone == '') {
    res.status(403).send({ message: 'missing field' });
  } else {
    const not_unique = await drivers.findOne({ id: id });
    if (not_unique) {
      console.log('driver already exists');
      res.status(401).send({ message: 'user already exists' });
    } else {
      const driverData = new drivers({
        id,
        name,
```



```
        phone,  
    });  
  
    driverData.save();  
    res.status(200).send({ message: 'done' });  
  }  
}  
});  
  
module.exports = router;
```