FINAL YEAR PROJECT

Motor Manager - Vehicle Management Application

Author:
Shakil ALI

Supervisor:
Dr. Alejandra BEGHELLI ZAPATA

A thesis submitted in fulfillment of the requirements for BSc Computer Science Degree

June 12, 2020

Declaration of Authorship

I, Shakil ALI, declare that this thesis titled, "Motor Manager - Vehcile Management Application" and the work presented in it are my own. I confirm that:

- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed: *Shakil Ali*

Date: 12th June 2020

UNIVERSITY OF LONDON

Abstract

Computing Department

BSc Computer Science Degree

Motor Manager - Vehicle Management Application

By Shakil ALI

There are over 37.5 million registered vehicles in the UK, as well as over 21,000 motor vehicle repair workshops. Due to the increasing number of vehicles and repair shops, an efficient method to manage them is required. *Motor Manager* is an Android application, which has been planned and developed to facilitate vehicle management. Its main function is storing vehicle data (e.g. registration number, transmission type, etc). However, it also encompasses other features such as a dark mode, file uploading and a voice input system. This application is aimed at individuals or organisations who want to manage a vehicle(s) e.g. garage owners, car rental providers. *Motor Manager* has a client side, programmed in the Java (Android-based) and XML languages. Google's Firebase development platform (BaaS) has been deployed as the server. This connects the client with Firebase's realtime (No-SQL) database and cloud storage services.

Acknowledgements

First and foremost, I would like to give praise to God (the most gracious, the most merciful). Next, I would like to give a special thanks to my project supervisor, Dr.Alejandra Beghelli Zapata. Throughout the project process, she provided continual and invaluable support. Dr.Zapata's persistence and selflessness were major factors in my success. For this, I will forever be indebted to her. Also, a big thank you to all the lecturers and staff I have had the pleasure of meeting, during my time at Goldsmiths. Last, but not least, I would like to thank my incredible family, especially my parents. Without their interminable encouragement and self-sacrificing nature, I would never have had the chance to study at University for a degree.

Contents

1 Introduction

1.1 Inspiration and Motivation

There are over 37.5 million registered vehicles in the UK [1] and 21,565 motor vehicle repair workshops [2] (as of 2020). These numbers are constantly increasing year on year. As a result, this gives rise to the need for vehicle management, on both an individualistic and societal scale. Computers have advanced globally, resulting in vehicle management and technology crossing paths. This can be seen in various modern software systems facilitating vehicle management.

This combination of this extensive rise in vehicle owners, technological advancement, and the need to manage personal vehicles, was the motivation behind developing *Motor Manager*.

1.2 Problem

As aforementioned, there is a rising number of vehicles on the road in Britain. The rise of vehicle use may be due to the growth in the car-rental market [3], which makes it simple for licensed individuals to gain access to a vehicle. Also, a rise in vehicles may lead to an increasing requirement of vehicle garages, as there may be more vehicle-related breakdowns and issues. As a result, the need for managing a greater number of vehicles becomes more prevalent.

1.3 Existing Solutions

In response to this demand, many solutions have emerged. For example, traditional means such as pen-and-paper, software tools (e.g. excel), and more recently mobile applications.

Based on the number of downloads from app stores, popular vehicle management mobile applications include *Drivvo, Simply Auto, Gas Manager*, and *Andi Car*. These applications, as well as many others with a lower number of downloads, provide users with an environment in which they could manage their vehicles. Common management features across these applications are account creation, a good user interface to enter and retrieve vehicle data, vehicle details storing and viewing capabilities, as well as data visuals (i.e. graphs derived from vehicle data).

The main weaknesses of these applications included a lack of accessibility features (voice input, audio description), ineffective security measures (data being lost between the user changing devices), and the high price.

1.4 Project Proposal

This project aims to provide users with a mobile application that provides efficient and simple vehicle management, overcoming some of the weaknesses of current popular applications. The proposed application, called *Motor Manager*, aims to include the basic features found on already existing applications (data storage and viewing, account creation, etc) but also targets to incorporate functionalities that are less common or inadequate in existing software systems, such as security and accessibility functionalities.

1.5 Report Structure

The rest of this report is structured as follows:

- Chapter 2 presents a review of the currently available applications for the problem of vehicle management
- Chapter 3 introduces the design process of *Motor Manager* and outlines how requirements were derived
- Chapter 4 details the technical implementation of *Motor Manager*
- Chapter 5 specifies the methods of testing applied to *Motor Manager*
- Chapter 6 presents a summary of the project, main findings, and future development plans

2 Background Research

This chapter presents the background research conducted to gain insight into the current applications in the market, stakeholder views, and possible technologies for use.

The first and second part of this chapter presents the results obtained after doing primary and secondary research on vehicle management systems. The last part focuses on researching technologies for development.

2.1 Vehicle Management Systems

2.1.1 Secondary Research

Secondary market research was carried out online to identify similar applications as well as their advantages and drawbacks. An application was deemed similar to *Motor Manager* if it has similar stakeholders and functionality. Stakeholders include individuals who own multiple vehicles and are required to keep track of their properties (e.g. car rental providers, mechanics). Basic functionality includes storing and retrieving data on multiple vehicles.

The search for different applications was carried out in the Google Play Store [4], App Store [5] and Amazon App Store [6]. The keywords used to find the applications were 'Vehicle Management System'. For each application store, the top five rated (according to the number of downloads) vehicle management applications were selected for further analysis. Table 1 lists the vehicle management applications selected from each app store. The number between parenthesis denotes the approximate number of downloads of each application (as of 25th March 2020).

Table 1: Application stores and their top 5 rated vehicle management applications

Google Play Store	App Store	Amazon App Store
Drivvo (1,000,000+)	Gas Manager (100,000+)	AndiCar (5000+)
Simply Auto (100,000+)	Reveal Manager (100,000+)	Cars Manager (1000+)
CarG (100,000+)	Quartix Vehicle Tracking (10,000+)	AutoWise Free (500+)
Car management (50,000+)	Kinesis (10,000+)	Heavy Vehicle Inspection & Fleet Management Software (500+)
Fleet Management App (5,000+)	MyCar (10,000+)	MyCars (500+)

To evaluate these applications, a search methodology [7] was used to identify aspects for comparison in mobile applications. This methodology consisted of performing a systematic search for the terms (application aspects) from different sources. For the search, the following sources were used: [8] [9] [10] [11]. Once the sources were found, a word cloud [12] was created to highlight the most common terms. Figure 1 shows the word cloud of the aspects detected by this search.

Figure 1: Word Cloud - Aspects

The main aspects obtained from the search (descending order of frequency):

- **Intuitiveness** - ability to understand or know something without any direct evidence or reasoning process
- **User Interface** - graphical display on a device that allows the user to interact with the device's apps, features, content, and functions
- **Accessibility** - accessible features for people with disabilities e.g. visually impaired
- **Target Audience** - a particular group at which an application is aimed at

- **Price** - the amount of money expected, required or given in payment for the application
- **Functions** - an activity that is natural to or the purpose of the application
- **Security** - measures in place within the application to ensure user data is secure
- **Size** - the storage space required by a device to download an application

To be able to analyse each application in terms of the above aspects, the rubric shown in Table 2 was created. An established book in the area of software engineering for competitive analysis was used to compile these criteria [13]. Also, these criteria have been produced specifically and solely to evaluate aspects of this project.

Table 2: Aspects - Rubric

	Rank		
Aspect	**× (0 points)**	**✓ (1 point)**	**✓ (2 points)**
Intuitiveness	None of these is present: Shadowed and labelled buttons Shadowed and labelled icons Shadowed and labelled sliders	At least one these is absent: Shadowed and labelled buttons Shadowed and labelled icons Shadowed and labelled sliders	These 3 features are present: Shadowed and labelled buttons Shadowed and labelled icons Shadowed and labelled sliders
User Interface	None of these is present: Different colours for background and elements Consistent design through application Familiar UI elements e.g. hamburger menu	At least one these is absent: Different colours for background and elements Consistent design through application Familiar UI elements e.g. hamburger menu	These 3 features are present: Different colours for background and elements Consistent design through application Familiar UI elements e.g. hamburger menu
Accessibility	None of these is present: Audio output Button shortcuts Text-to-speech	At least one these is absent: Audio output Button shortcuts Text-to-speech	At least one these is absent: Audio output Button shortcuts Text-to-speech
Functions	None of these is present: Store multiple vehicle details	At least one these is absent: Store multiple vehicle details	At least one these is absent: Store multiple vehicle details

	Edit and delete existing data Upload/Download files e.g. MOT certificate Visualise data using graphs	Edit and delete existing data Upload/Download files e.g. MOT certificate Visualise data using graphs	Edit and delete existing data Upload/Download files e.g. MOT certificate Visualise data using graphs
Security	None of these is present: Confirm password Email confirmation Biometric identification	At least one these is absent: Confirm password Email confirmation Biometric identification	At least one these is absent: Confirm password Email confirmation Biometric identification

Note: *Target Audience, Price and Size were not included in the rubric as they are not quantifiable in this project*

In the following, the aspects listed in Table 2 are discussed. Each application is analysed in terms of whether it meets the conditions expected for the aspect or not.

2.2 Aspect Analysis (Literature Review)

2.2.1 Intuitiveness

Software intuitiveness refers to how instinctively an individual can traverse and operate a system, without prior knowledge or experience with it [14]. In this work, intuitiveness will be evaluated exclusively according to the criteria established in Table 2. Thus, Table 3 has been compiled to highlight which applications from Table 1 adhere to the *intuitiveness* criteria from Table 2.

Table 3: Intuitiveness - Application Ranking

Application	Intuitiveness Rank	Comments
Google Play Store Applications		
Drivvo	✓	Meets criteria
Car management	✓	Meets criteria
CarG	✓	Meets criteria
Simply Auto	✓	Meets criteria
Fleet Management App	✓	Meets criteria
Apple App Store Applications		
Quartix Vehicle Tracking	✓	Meets criteria
Gas Manager	✓	Meets criteria

Reveal Manager	✗	Void of shadowed and labelled sliders
Kinesis	✓	Meets criteria
MyCar	✓	Meets criteria
Amazon App Store Applications		
AndiCar	✓	Meets criteria
Cars Manager	✓	Meets criteria
AutoWise Free	✗	Void of shadowed and labelled sliders
Heavy Vehicle Inspection & Fleet Management Software	✓	Meets criteria
MyCars	✓	Meets criteria

2.2.2 User Interface

User interfaces are software and/or hardware that bridge the world of human action and computer action. Applications and the interfaces we use to operate them are one of the things that humans adapt to [15]. Examples of hardware user interfaces are the screen, the keyboard, and the mouse. An example of software user interfaces is the way information is presented on a touch screen. Table 4 has been compiled to highlight which applications from Table 1 adhere to the *user interface* criteria from Table 2, considering the way information is diagrammed on screen as the user interface.

Table 4: User Interface - Application Ranking

Application	User Interface Rank	Comments
Google Play Store Applications		
Drivvo	✓	Meets criteria
Car management	✓	Meets criteria
CarG	✓	Meets criteria
Simply Auto	✓	Meets criteria
Fleet Management App	✓	Meets criteria
Apple App Store Applications		
Quartix Vehicle Tracking	✓	Meets criteria
Gas Manager	✓	Meets criteria
Reveal Manager	✓	Meets criteria
Kinesis	✓	Meets criteria
MyCar	✓	Meets criteria
Amazon App Store Applications		
AndiCar	✓	Meets criteria

Cars Manager	✓	Inconsistent design for every page
AutoWise Free	✓	Similar colours for UI elements and background
Heavy Vehicle Inspection & Fleet Management Software	✓	Meets criteria
MyCars	✓	Meets criteria

2.2.3 Accessibility

Accessibility, in a technological sense, is the design of products for people with disabilities (e.g. visual, hearing, physical) [16]. Table 5 has been compiled to highlight which applications from Table 1 adhere to the *accessibility* criteria from Table 2.

Table 5: Accessibility - Application Ranking

Application	Accessibility Rank	Comments
Google Play Store Applications		
Drivvo	×	Criteria not met
Car management	×	Criteria not met
CarG	×	Criteria not met
Simply Auto	✓	Meets criteria
Fleet Management App	×	Criteria not met
Apple App Store Applications		
Quartix Vehicle Tracking	×	Criteria not met
Gas Manager	×	Criteria not met
Reveal Manager	×	Criteria not met
Kinesis	×	Criteria not met
MyCar	×	Criteria not met
Amazon App Store Applications		
AndiCar	×	Criteria not met
Cars Manager	×	Criteria not met
AutoWise Free	×	Criteria not met
Heavy Vehicle Inspection & Fleet Management Software	×	Criteria not met
MyCars	×	Criteria not met

2.2.4 Target Audience

The target audience of an application consists of consumers or organisations that want to buy a company's products or services. This may be due to the need or want that company's product offers the consumers [17]. The general target audience for applications in this project is owners of multiple vehicles (e.g. mechanic, car rental providers). The applications *Fleet Management App* and *Heavy Vehicle Inspection & Fleet Management Software* have a more specific target audience: Heavy vehicle owners (possessors).

2.2.5 Price

Price refers to the cost of an application. The price can depend on many factors. For example, a company may have the aim of generating profit, therefore, they attach a price to their products. Another reason could be for a non-profit organization to be able to pay for operational costs (e.g. maintaining servers) [18]. Also, some applications have a purchasing price, whereas others charge users for the use of premium features.

2.2.6 Functions

Functions are the attributes and intended actions an application performs. The functions of an application vary between the needs and wants of the end-user [19]. For instance, in a vehicle management application, storing and viewing multiple vehicle details would be an expected function. Table 6 has been compiled to highlight which applications from Table 1 adhere to the *functions* criteria from Table 2.

Table 6: Functions - Application Ranking

Application	Functions Rank	Comments
Google Play Store Applications		
Drivvo	✓	Meets criteria
Car management	✓	Meets criteria
CarG	✓	Meets criteria
Simply Auto	✓	Meets criteria
Fleet Management App	✗	No visualisation of data via graphs
Apple App Store Applications		
Quartix Vehicle Tracking	✓	Meets criteria
Gas Manager	✓	Meets criteria
Reveal Manager	✗	No visualisation of data via graphs
Kinesis	✗	No visualisation of data via graphs

MyCar	✗	No visualisation of data via graphs
Amazon App Store Applications		
AndiCar	✗	Cannot upload/download files
Cars Manager	✗	Cannot upload/download files
AutoWise Free	✗	Cannot upload/download files
Heavy Vehicle Inspection & Fleet Management Software	✗	Cannot upload/download files
MyCars	✗	Cannot upload/download files

2.2.7 Security

The security aspect of an application refers to the comprehensive security components it entails [20]. Table 7 has been compiled to highlight which applications from Table 1 adhere to the *security* criteria from Table 2.

Table 7: Security - Application Ranking

Application	Security Rank	Comments
Google Play Store Applications		
Drivvo	✗	Lack of biometric identification
Car management	✗	Lack of biometric identification
CarG	✗	Lack of biometric identification
Simply Auto	✗	Lack of biometric identification
Fleet Management App	✗	Lack of biometric identification
Apple App Store Applications		
Quartix Vehicle Tracking	✗	Lack of biometric identification
Gas Manager	✗	Lack of biometric identification
Reveal Manager	✗	Lack of biometric identification
Kinesis	✗	Lack of biometric identification
MyCar	✗	Lack of biometric identification
Amazon App Store Applications		
AndiCar	✗	Lack of biometric identification and email confirmation
Cars Manager	✗	Lack of biometric identification
AutoWise Free	✗	Lack of biometric identification and email confirmation
Heavy Vehicle Inspection & Fleet Management Software	✗	Lack of biometric identification
MyCars	✗	Lack of biometric identification

2.2.8 Size (Storage Requirement)

The size (storage requirement) of an application refers to the space required on a device to download an application. The size of the application depends on the language it was programmed in and the complexity of the functions enclosed [21]. Therefore, the more functionalities an application encompasses, the greater the storage requirement for it is. Table 8 has been compiled to display applications from Table 1 and their storage requirements (descending order).

Table 8: Size (Storage Requirement) - Application Ranking

Application	Size
MyCar	112.6MB
Reveal Manager	109.8MB
Quartix Vehicle Tracking	51MB
Kinesis	42MB
Gas Manager	29.1MB
Heavy Vehicle Inspection & Fleet Management Software	28.1MB
Car management	15MB
Simply Auto	13MB
Drivvo	12MB
Fleet Management App	12MB
MyCars	5.4MB
AndiCar	4.7MB
AutoWise Free	1.1MB
CarG	773KB
Cars Manager	371.6KB

Lastly, Table 9 displays the full aspect list for the 15 applications and their respective total weighting. The application which scored the highest weighting (according to the criteria set in this project) was *Simply Auto* and the lowest weighted application was *AutoWise Free*. This matches the number of downloads found in Table 1, as *Simple Auto* was one of the most downloaded applications (100,000+) compared to *AutoWise Free* (500+), one of the least downloaded applications.

Furthermore, a vertical sum for the individual aspects allowed a gap in the market to be spotted amongst the selected applications. The aspects ‘Intuitiveness’ and ‘User Interface’ were generally found in the 15 applications consistently (both aspects scored 22). However, ‘Accessibility’ and ‘Security’, which scored a 5 and 15 respectively, were less common across the applications.

Consequently, the application and aspect analysis revealed knowledge of the strengths and weaknesses of major competitors. Thus, it outlined unique selling points (gaps in the market) such as accessibility and security functionality. Therefore, this information was incorporated when developing *Motor Manager*.

Table 9: Full Application Aspects Assessment

Application	Intuitiveness	User Interface	Accessibility	Functions	Security	Total Weighting	Comments
Simply Auto	2	2	2	2	1	**9**	Good application. Features include voice input. However, there are costs for the premium versions.
Drivvo	2	2	0	2	1	**7**	Good application. Not too many features regarding accessibility. Negative customer reviews consist of data being lost.
Car management	2	2	0	2	1	**7**	Good application. Not much mention of accessible features. Features e.g. data visualisation, work inconsistently
CarG	2	2	0	2	1	**7**	Good application. Accessibility features not mentioned. Customer reviews consist of loss data.
Fleet Management App	2	2	0	2	1	**7**	Good application. No accessibility features. Other features are not competent.
Quartix Vehicle Tracking	2	2	0	2	1	**7**	OK application. No accessibility features and must pay after three months.
Gas Manager	2	2	0	2	1	**7**	Good application. However, in-app purchases are required for a premium experience.
Kinesis	2	2	0	1	1	**6**	Good application. Although the application is very user-friendly, recent reviews suggest an update has caused the application to constantly crash. Therefore, this lowers user-experience.
My Car	2	2	0	1	1	**6**	Good application. However, features are basic and customer reviews contain complaints regarding data loss.
AndiCar	2	2	0	1	1	**6**	Good application. Customer reviews are mainly positive. No mention of accessibility features e.g. audio description for blind users.

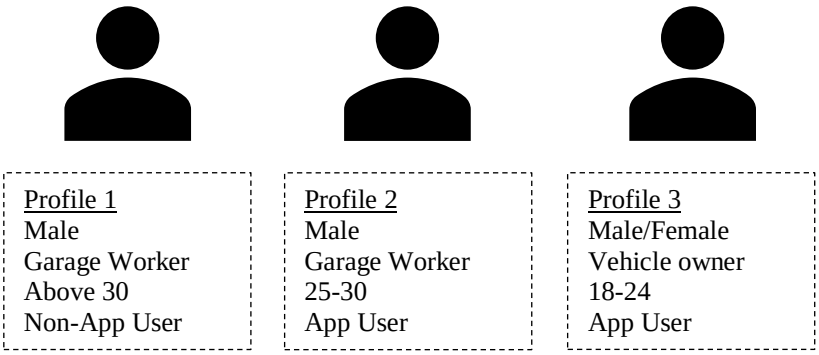
Heavy Vehicle Inspection & Fleet Management Software	2	2	0	1	1	**6**	Good application. Very good reviews. However, a con is that it is free for only a month.
MyCars	2	2	0	1	1	**6**	Good application. Simple, yet attractive interface. Recent reviews suggest new update causes crash after login.
Cars Manager	2	1	0	1	1	**5**	Satisfactory application. Basic functionality allows vehicle management to be carried out. No advanced features.
Reveal Manager	1	2	0	1	1	**5**	Satisfactory application. Basic features are included.
AutoWise Free	1	1	0	1	1	**4**	Very basic application. It allows the storage of vehicles. However, no further features. No reviewers also.
Apect Total Weighting	**28**	**28**	**2**	**22**	**15**		

2.3 Public Survey Result

To get a better understanding of the vehicle management application market, primary market research was conducted. This was in the form of a nine-question survey in Google Forms (Appendix A). It received fifteen responses in total. This is a small quantity, compared to commercial app store satisfaction surveys, and therefore may cause unrepresentative findings (this problem will be addressed below). The survey asked a range of questions, structured in a manner as to extract exploratory data, which would then be analysed. Consequently, the results found were used to aid in the system design and development of *Motor Manager*.

As aforementioned, the responses to the survey were limited. To ensure analysis of the primary research was unbiased and representative, profiles were created from the users who took part. Profiles are the typical responses produced by certain users, of a similar demographic [22]. The three main profiles which emerged from the survey are shown below in Figure 2.

Figure 2: Primary Market Research - Survey - Profiles



These profiles were derived from the first four exploratory questions regarding age, gender, work within a vehicle-related industry, and use of a vehicle management application. These profiles were then used to help group the answers to the remaining questions. The main findings are outlined below (Appendix B provides a more in-depth analysis of this research).

Profile 1:
Surveyors from profile 1 are over 30 years of age, male, work in a garage, and tend to be non-application users. They are non-app users, so they did not make use of any applications to assist them in their work, and therefore are not familiar with aspects of an application. Surveyors in this category prioritised 'User Interface' over other aspects if a new application was to be built. Profile 1 surveyors related that the main reason they didn't look to use applications to aid them previously was because they 'never thought of using one', 'inexperienced' with technology and concerns on data security. To interest them into use one the application would have to be simple, intuitive, and secure. Lastly, profile 1 users were equally split between logo 2 and 3.

Profile 2:
Surveyors from profile 2 are aged 25-30, male, work in a garage, and tend to be application users. This profile of surveyor tended to use applications that were not specific to vehicle management to assist them in their work e.g. Samsung notes, Microsoft excel. Individuals mainly thought the best aspect from the software they've previously utilised was 'Features'. Surveyors in this category believed if an application was to be built, 'User Interface' should be prioritised. Lastly, profile 2 users were also equally split between logo 2 and 3.

Profile 3:
Surveyors from profile 3 are aged 18-24, male/female, and tend to be app users. This profile of surveyor used vehicle management applications such as *aCar* and *Car Maintenance Reminder Lite*. Individuals mainly thought the best aspect from the software they've previously utilised was 'Features'. Surveyors in this category believed if an application was to be built, 'User Interface' should be prioritised. Lastly, this group preferred logo 2.

Overall, the survey and survey analysis provided the opportunity to develop acumen regarding vehicle management and its audience. An understanding which resulted was that the final software system had to have a very low barrier to entry (intuitive and simple), as well as be efficient for secure vehicle data storage and management. The findings of the primary market research complemented the aspect analysis (secondary market research) in that users want an application that securely stores user data. Differences found were that users from the survey wanted a user interface prioritised, whereas, a gap in the market identified from the aspect analysis was 'accessibility'. Therefore, these discoveries regarding the aspects (friendly user interface, intuitiveness, accessibility, security measures) were integrated during the design and development phase of *Motor Manager.*

2.4 Technology Review

Following on from the previous sections, findings have been discovered regarding general technical requirements. The application needs to be: friendly interface-wise, intuitive, accessible, and secure. The following subsections aim to identify and select appropriate technologies for this project and its technical requirements.

2.4.1 Development Environment

A development environment, commonly referred to as an IDE (Integrated Development Environment), is a software application that provides a broad range of facilities to users who want to develop software. The fundamental components of an IDE generally consist of a source code editor, debugger, and build automation tools [23]. Table 10 has been compiled to display the most commonly used IDE's for mobile applications, as well as their properties. The IDE's and their properties have been collected from multiple trusted sources [24] [25] [26].

Table 10: Integrate Development Environments - Properties

IDE	Supported Languages	Supported OS	Target OS	Target Audience	Price
Android Studio	Java, C, C++, Kotlin	Windows, MacOS, Linux	Android	Experienced	Free
Eclipse	Java, C, C++, C#, JavaScript, Python	All OS which supports Java	Android, iOS, Linux, MaxOS, Windows	Professional developers	Free

Visual Studio	C++, C, C#, Visual Basic PHP JavaScript	Windows, MaxOS, Linux	Windows, Android, iOS	Experienced	Free - $3000+
IntelliJ IDEA	Java, Scala, Groovy, Koitlin, JavaScript, TypeScript, SQL	Windows, MacOS, Linux	Any OS supporting Java	Professional developers (Java-specific)	Free - $500 (annually)
NetBeans	Java, C, C++, HTML, PHP, JavaScript	Windows, MacOS, Linux	Cross-platfrom	Professional developers	Free
Komodo	Java, JavaScript, PHP, Python, Ruby, HTML	Windows, MacOS, Linux	Cross-platform	Professional developers (web and mobile-specific)	Free - $400
Cordova	HTML, CSS, JavaScript	Windows, MacOS, Linux	Cross-platform	Web developers	Free
PhoneGap	HTML, CSS, JavaScript	Windows, MacOS, Linux, Android, Windows	Cross-platform	Web developers	Free
Appcelerator Titanium	JavaScript	Windows, MacOS, Linux	iOS, Android, Windows	JavaScript developers	Free - $100 (monthly)
App Inventor	Kawa	Windows, MacOS, Linux	Android	Students (Amateurs)	Free
AIDE	Java, C, C++, XML, HTML, CSS, JavaScript	Android	Android	Amateurs or mobile professionals	Free (in-app purchases available)

Table 10 outlined all the details regarding each IDE. This made it clearer as to which IDE was tailored to which target audience, programming languages, operating systems, and price. Consequently, this made the decision of selecting an IDE simpler.

Firstly, as this project is a student-based project, within a University, the cost would need to be kept to a minimum. As a result, the IDE's which were free

were more appealing to the circumstances. Next, the target audience of the IDE's was pivotal in the decision, as it had to accommodate a student with experience in programming. Also, ensuring the programming languages and supported operating systems was compliant with project resources.

The eventual decision for IDE was Android Studio [27]. Not only did it meet all the specified requirements above, but it also has many advantages (e.g. database server integration, in-built git facilities). Also, due to my previous experience with Android Studio and the Android programming language, it was ideal, as no knowledge had to be acquired fundamentally, as compared to the other IDE's (Appendix C contains more advantages of Android Studio).

2.4.2 Database Server

A database server is a type of server that facilitates the accessing and retrieval of data from a database, for other computers [28]. There is a variety of different database servers to choose from. However, some technical requirements were vital to being met:

- Ensure data integrity
- Integrate seamlessly with Android Studio (chosen IDE for this project)
- Serve multiple users concurrently

Table 11 has been compiled to display the most commonly used database servers for mobile applications, as well as their properties. The IDE's and their properties have been collected from multiple trusted sources [29] [30] [31].

Table 11: Database Servers - Properties

Firebase Server	Database Type	Querying Language	Storage Language	Database size	Price	Data Integrity Measures	Android Studio Integration	Simultaneous Users Served
SQLite	RDBMS	SQL	SQL	140TB	Free	Entity, Referential	✓	1024
Firebase	BAAS (NoSQL)	JavaScript	JSON	1GB	Free	Domain, User-Defined	✓	200K
Oracle	ORDBMS	SQL	SQL	128TB	Free	Entity, Referential	✓	2047
MySQL	RDBMS	SQL	SQL	256TB	Free	Entity, Referential	✓	4.3MN

Microsoft Access	DBMS	VBA	VBA	2GB	Free	Entity, Referential	×	255
MongoDB	NoSQL	JavaScript	JSON	16GB	Free	Domain, User-Defined	✓	128K
CouchDB	NoSQL	JavaScript	JSON	15GB	Free	Domain, User-Defined	✓	2048
CassandraDB	NoSQL	CQL	JSON	2GB	Free	Domain, User-Defined	✓	330K
***Note**: Base versions of databases are free, extra storage space will incur a charge*								

Table 11 outlined all the details regarding each database server. Therefore, this made the choice of selecting a database technology simpler. The technical requirements required a system that ensured data integrity, integrates easily with Android Studio, and served multiple users simultaneously. As a result, the chosen database server for this project was Firebase [32]. Additional reasons behind this decision consisted of the multiple benefits of Firebase to this project (Appendix D), as well as my experience and established knowledge regarding the platform.

2.5 Chapter 2 Summary

This chapter focussed on the background research required to begin planning for *Motor Manager*. Both primary and secondary market research conducted as described. Also, a technology review was carried out to decide which technology to use.

The results of the primary and secondary research were complimentary and concluded that users do require a simplistic application, user interface wise, as well as secure, so their data is safe.

The next chapter will aim to make use of this research and incorporate it into the system design of *Motor Manager.*

3 System Design

In the previous chapter, the main requirements were identified for the application: simple interface, intuitive, accessible, and secure. In this chapter, the designs of the system will be presented to complement the requirements found from research. This chapter will aim to build on the previous chapter findings and provide visual perspectives of *Motor Manager*.

The first part of this chapter defines the Systems Requirements Specification (SRS), which is a definitive list of requirements spanning from the general ones acquired from the previous chapter. The second part presents the Unified Modelling Language (UML) diagrams and technical diagrams.

3.1 System Requirements Specification (SRS)

The system requirements have been categorised into two groups: functional and non-functional. These requirements provide the basis for the operation of *Motor Manager*, as well as supplying a high-level understanding of possible limitations. These requirements were composed via the creation of a basic prototype and receiving user views. The methodological approach to gathering the requirements was based on the 'IEEE Guide for Developing System Requirements Specifications' [33].

3.1.1 Storyboard

Storyboards provide a step-by-step presentation describing how people will perform a target activity using product concepts [34] and enables visualisation of a scenario in which an application may be utilised. A storyboard was initially created to clearly illustrate the need for *Motor Manager* (Appendix E). It depicts a frustrated mechanic in possession of multiple vehicles, with work required to be carried out. *Motor Manager* is then downloaded to help manage the workload. The storyboard produced the requirement for *Motor Manager* to be able to store multiple vehicles.

3.1.2 Wireframes

The next step was to create an initial model (wireframe). A wireframe is a layout of a web page that demonstrates what interface elements will exist on key pages. It is a critical part of the interaction design process and allows a visual representation of what the prototype could look like [35]. The wireframes depict the significant pages that will be found on *Motor Manager* (Appendix F). Similar to the storyboard, wireframes help envision what *Motor Manager* looks like, as well as the intended functionalities. The functional requirement obtained from the wireframes includes being able to traverse multiple pages of the application.

3.1.3 Prototypes

Further, after the creation of the storyboard and wireframes, where functional requirements were extracted, the prototyping for *Motor Manager* commenced. The prototype was based on the skeletal wireframes and the functionality adhered to the concept of the storyboard, as well as the previous research performed.

Adobe XD was selected as the prototyping software as it provided an efficient environment to produce mobile application prototypes [36]. It allows the creation of a basic prototype, which enables page transitioning and button interactivity. Adobe XD has a wide range of advantages including:

- A simple interface - can be used quite easily without prior knowledge
- Easy asset import (Android and IOS UI) i.e. use Android/IOS UI elements e.g. buttons, status bar, etc

As a result, an initial prototype was developed, consisting of the core vehicle management functionality (e.g. login/registration, saving vehicle details, displaying stored details), as well as some post MVP functions (graphs of stored data). Figure 3 shows the initial prototype with enumerated labels per screen. The screens account for the following:

1. **App Start Screen** - buttons to take the user to the *login* or *register* page
2. **Login Screen** - allows user to login
3. **Registration Screen** - allows user to register
4. **Main Menu Screen** - menu containing all pages of the application
5. **Vehicle Inventory Screen** - list of stored vehicles
6. **Vehicle 1 - Details Screen** - specific vehicle details
7. **Vehicle 1 - Data Visualisation Screen** - specific vehicle graphs

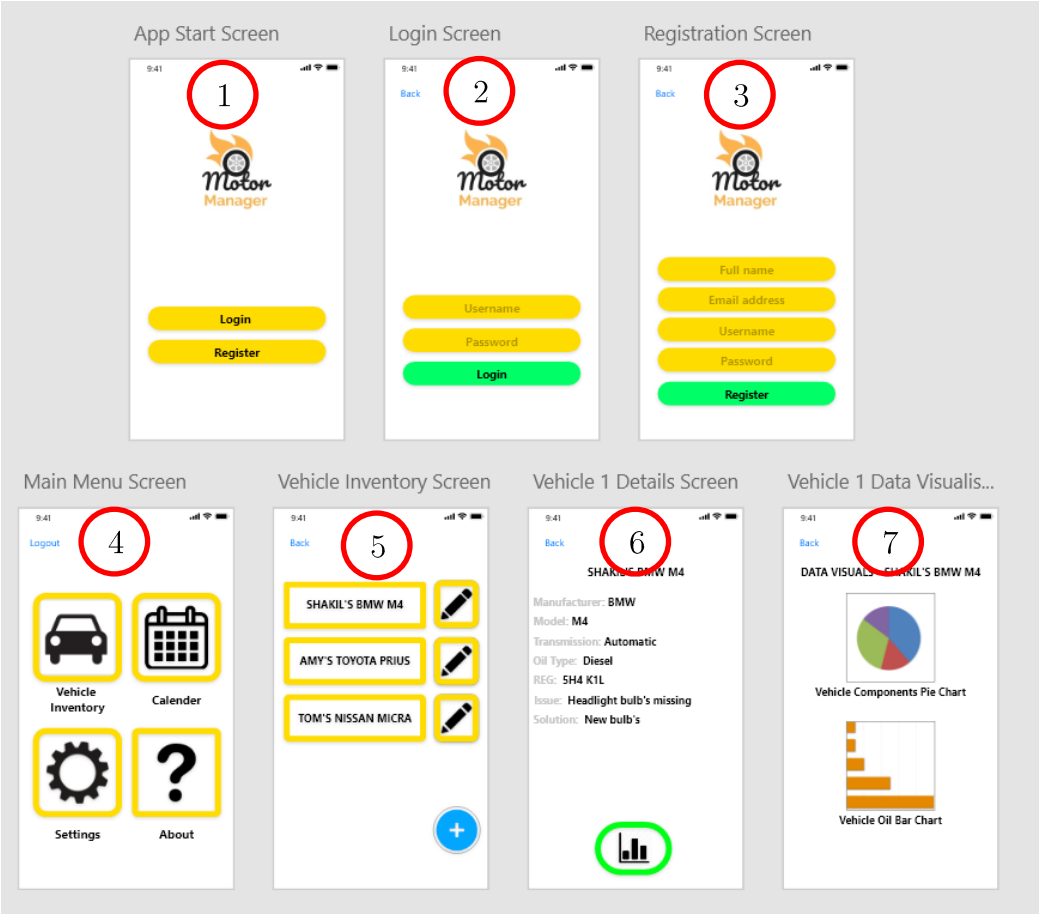


Figure 3: Prototype 1

After the conclusion of the first prototype, the IEEE guide recommends surveying to receive cursory feedback. As a result, a final prototype can be produced. The survey consisted of nine simple, exploratory questions (Appendix G). The initial questions were to find out the demographics of users partaking. The following questions were focusing on some of the aspects discussed in chapter 2, namely: simplicity, user-friendly, and functionality.

An analysis was carried out on the survey data to get a general understanding of user thoughts (Appendix H). There was a total of 7 responses (85.7% male, 14.3% female), with an average age of 25 and above. The overall consensus confirmed that:

- users liked the interface (average rating of 4.6/5)
- users found the interface intuitive (85.5%)
- an improvement that should be prioritised is more functionalities (e.g. export data, import data, download vehicle graphs)

These findings were considered when designing prototype two.

A second prototype, shown in Figure 4, was created based on the analysis from the first prototype. All of the original seven pages were present (opening, login, registration, menu, vehicle inventory, vehicle details, vehicle data visuals). The elements that the surveyors liked, such as design and intuitiveness, were maintained. Additional functional features as requested were added (download data, upload data, view data visuals, download graphs). These additions can be found on the last two screens: *Vehicle 1 Details Screen* and *Vehicle 1 Data Visualisation Screen*. The second prototype was more representative of the MVP for *Motor Manager,* and more compliant with user perspectives.

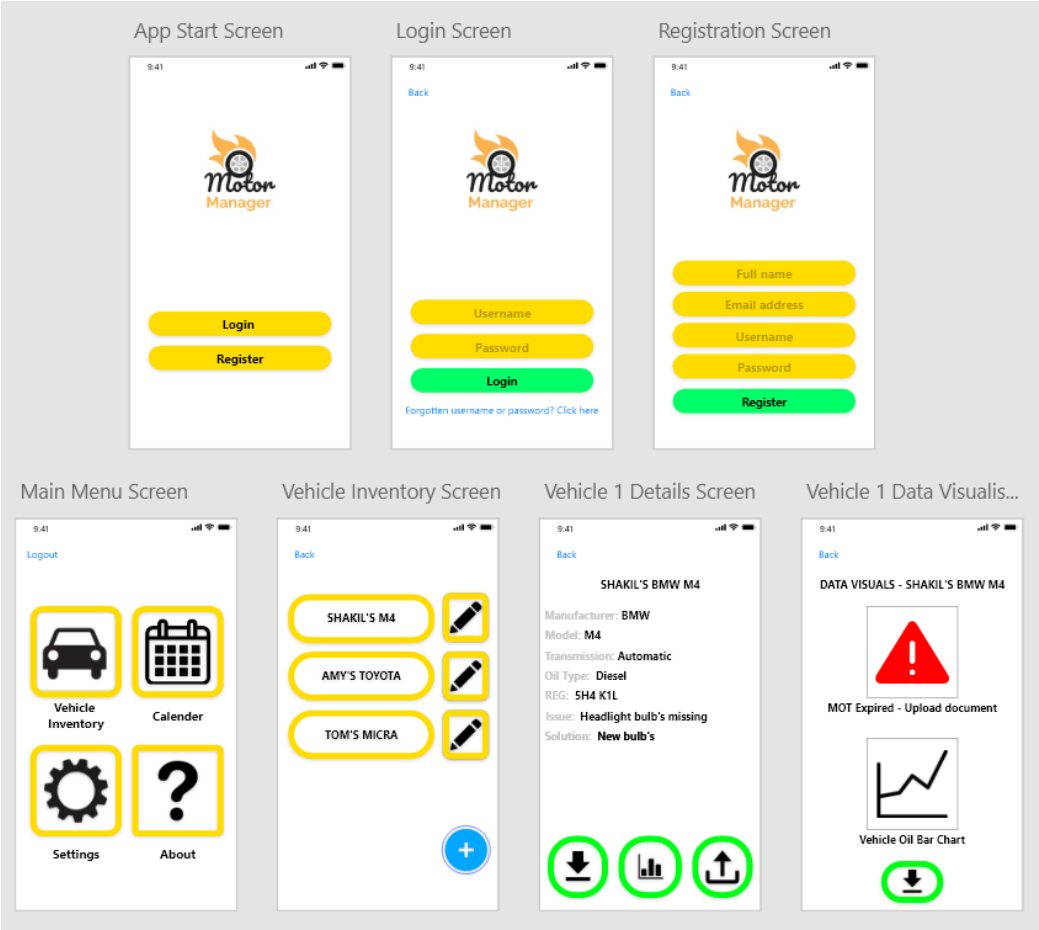


Figure 4: Prototype 2 (Final)

3.1.4 Functional Requirements

Functional requirements define and describe the functions a software must perform. Stating the functional requirements has allowed the capture of the intended behaviour of the system [37]. Once the second prototype was created and finalised, it was possible to derive functional requirements for the Minimum Viable Product (MVP), as seen in Table 12.

Table 12: Functional Requirements (MVP)

Functional Requirements		
ID	**Requirement**	**Source**
001	Application must be an app on a device (logo and name accurately applied)	Primary/Secondary research
002	Application must have a login option	Primary/Secondary research
003	Application must store and retrieve user login details in Firebase	Storyboard
004	Application must have a registration option	Primary/Secondary research
005	Application must store new user details e.g. full name, email, password, profile image.	Primary/Secondary research
006	Application must have the main menu when a user has logged in	Primary/Secondary research
007	Application must display the user's full name on login	Wireframes
008	Application must display the user's email address on login	Wireframes
009	Application must display the user's profile picture on login	Wireframes
010	Application must integrate with Firebase Realtime Database and store vehicle data	Prototype
011	Application must allow users to traverse different pages	Prototype
012	Application must allow the user to store multiple vehicles and their details e.g. vehicle image, registration number, description of work to be conducted.	Primary/Secondary research
013	The application must allow the user to click on a saved vehicle, and view its stored details	Prototype
014	The application must allow the user to logout successfully	Primary/Secondary research

3.1.5 Non-Functional Requirements

Table 13 contains non-functional requirements (MVP) for *Motor Manager*. Non-functional requirements specify criteria that can be used to assess how successful the operation of a system is, as opposed to its explicit role [38]. Prototypes are not required to produce non-functional requirements, however, deriving these requirements is pivotal for the user experience of an application. Non-functional requirements cover types of requirements (e.g. scalability capacity, interface, simplicity, reliability, recoverability, data integrity) that are not covered by functional requirements.

Table 13: Non-Functional Requirements (MVP)

Non-functional Requirements		
ID	**Requirement**	**Source**
001	Application must launch as soon as the user opens it	Primary/Secondary research
002	Application must inform the user of incomplete fields	Primary/Secondary research
003	Application must be able to handle 100 users at a time	Primary/Secondary research
004	Application must be portable (can download on any Android device)	Primary/Secondary research
005	The application must have the same user experience on all compatible devices	Primary/Secondary research
006	Application must allow storage of at least three vehicles	Primary/Secondary research
007	Application must ensure data is secure	Primary/Secondary research

See Appendix I for post-MVP functional and non-functional requirements.

3.2 System Diagrams

Throughout the process of designing and planning *Motor Manager*, many components had to be considered. This includes how the user would interact with the application (client-server relationship) as well as how the software will interact internally (front-end, server, and database). To visualise and outline these interactions diagrams were created. The following diagrams are built on the guidelines provided in the book: 'Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems' [39], as well as the requirements identified above.

3.2.1 UML Diagrams

The aim of UML (Unified Modelling Language) is to produce visual representations of systems in a coherent manner [40]. There are many types of UML diagrams. However, activity and sequence diagrams are most suitable for mobile applications [41]. Hence, they have been chosen to depict *Motor Manager* functionality.

Activity diagrams graphically represent workflows related to user activity [42]. The activity diagrams help visualise the requirement of traversing multiple pages. The activity diagram, shown and labelled in Figures 5, was created for *Motor Manager's* registration/login system. It illustrates a basic and main application-flow of *Motor Manager*, and the labels account for the following:

1. Opening of the application
2. Asks users whether they want to log in or register

3. Registering an account (if successful then step 4, else back to step 2)
4. The choice of exploring the *menu* pages or logging out
5. Displays the user's vehicle inventory
6. Allows users to *add vehicles*, *view properties* or *data visuals*

See Appendix J for more activity diagrams.

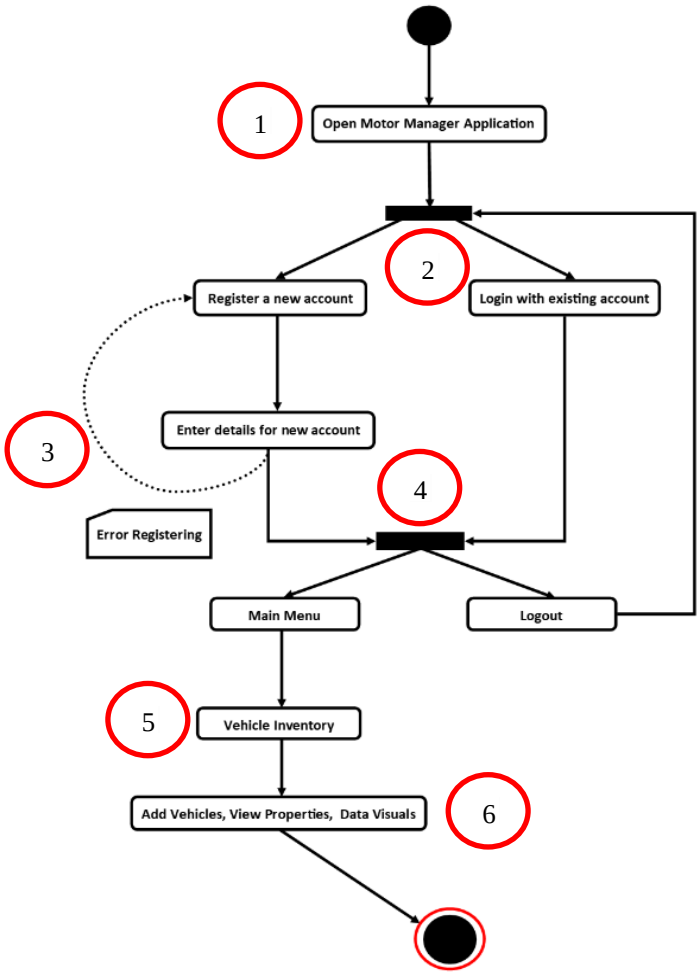


Figure 5: Activity Diagram - Registration/Login System

Sequence diagrams portray how objects (e.g. client, server, and database) interact with each other. The sequence diagram helps portray the requirement of storing vehicle data. Figure 6, displays a labelled sequence diagram, created for *Motor Manager* is in the form of an example (i.e. storing a BMW M4 details), as it emphasises the roles of different components of the software system. The arrows show the interactions between different components and the notes display alternative outcomes of those actions. The labels account for the following:

1. A garage employee downloads *Motor Manager* creates an account, which is then authenticated and stored
2. Displays users vehicle inventory, after retrieving data
3. BMW M4 details added and stored
4. Data visuals button is clicked and stored vehicle data is displayed graphically

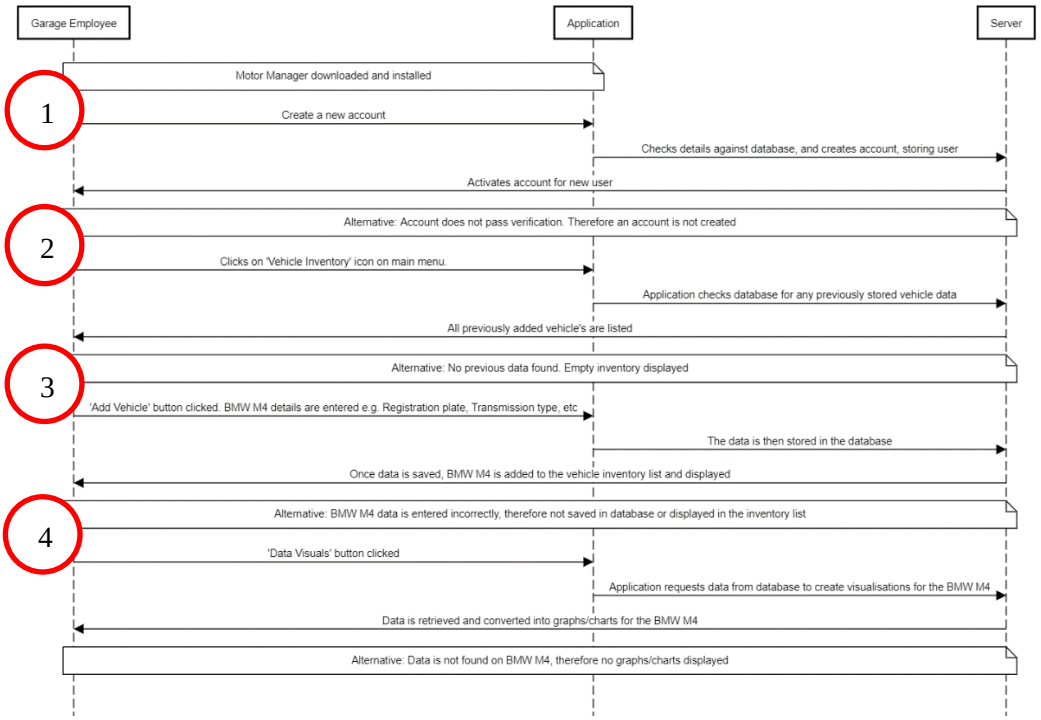


Figure 6: Sequence Diagram - Example adding vehicle details

3.2.2 Technical Architecture Diagram

The functional requirement derived previously in regards to storing vehicle data has led to the need to create a technical architecture diagram. This diagram outlines how the system will interact amongst its various units. A software technical architecture of a system is the structure of a system, which comprises of software elements, the externally visible properties of those elements, and the relationship among them [43]. Figure 7 displays the technical architecture diagram for *Motor Manager*. There are three main components:

- **Client** - presentation (visual) layer which the user interacts through
- **Backend** - acts as the ’middle-man’ between the database and client, stores and servers data
- **Database** - storage for user data

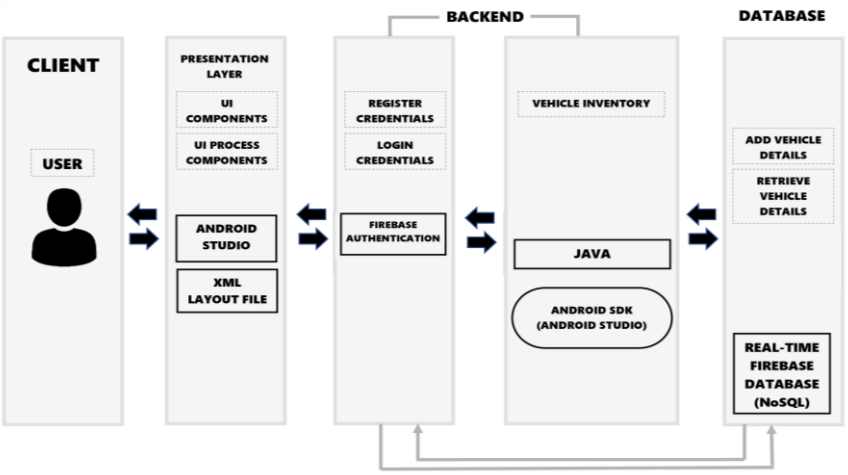


Figure 7: Technical Architecture Diagram

3.2.3 Class Diagrams

In this section, the class diagrams planned for the various pages of *Motor Manager* will be presented. Class diagrams are also UML diagrams. However, their main objective is to describe the structure of a system by displaying the system's classes, attributes, and methods, as opposed to the user experience. Relationships between class and objects are also exhibited. The class diagrams for *Motor Manager* MVP can be seen in Appendix K. The classes seen in these diagrams have been separated into groups via the principle of *Separations of Concerns* (SoC). This is so that each group is responsible for a separate function of the application:

- **Activities** - classes for screens of *Motor Manager*
- **Adapters -** classes which allow incompatible objects to collaborate
- **Fragments -** classes for screens of *Motor Manager* (specifically for the menu pages)
- **Helpers -** classes which contain code for additional (post-MVP) functionality
- **Models** - classes that define information about data.

3.2.4 Database Schema

The database schema, labelled, and shown in Figure 8, has been created for the MVP of *Motor Manager* and aims to portray the structure of the database. It stems from the majority of the functional requirements. Each table relates to another table via database relationships i.e. one-to-one, one-to-many, many-to-many. As aforementioned in chapter 1, Firebase's Realtime Database was the

selected backend server. Therefore, data will be stored in a *JSON* format. The labels account for the following *JSON* objects in the database:

- **Firebase Authentication** - stores user credentials
- **Vehicles** - stores user vehicle data
- **Vehicle Images** - stores vehicle images
- **User Profile Photos** - stores the user's profile image

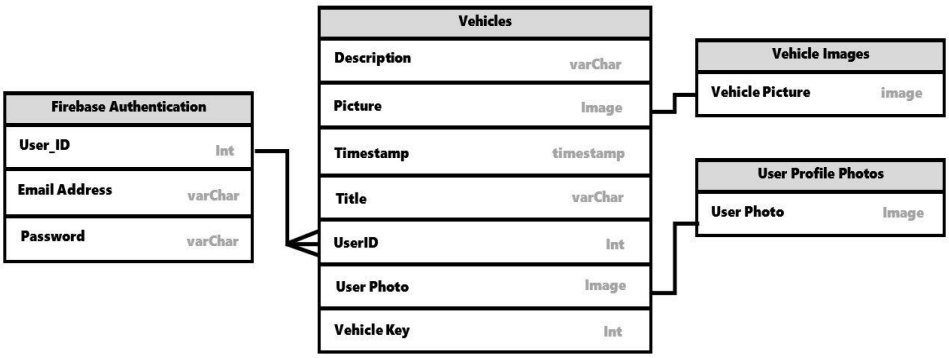


Figure 8: Database Schema (MVP)

3.3 Chapter 3 Summary

This chapter aimed to define the design process undertaken for the *Motor Manager* application. The 'IEEE Guide for Developing System Requirements Specifications' was used throughout, especially in the first section to explain the methodology of outlining functional and non-functional requirements. Through storyboards, wireframes, and prototypes, the system requirements were produced for the MVP. The second section exhibited the various diagrams associated with design procedure (e.g. class, activity, sequence, database schema).

Consequently, this provided a foundation for *Motor Manager* both visually and technically. The exterior of the software system was established, as well as the interior interactions between components.

The following chapter will provide a high-level, technical understanding of the implementation of the *Motor Manager* application. The implementation will build on the design advancement made from this chapter.

4 Implementation

This chapter details the technical implementation of *Motor Manager*, and all the various components required to produce the outlined system design.

The first section refers to the Java programming within Android Studio, and the second section describes the incorporation of the Firebase backend.

4.1 Android (Java) Programming

Android Studio was the selected IDE (Integrated Development Environment) for this project. All project (class) files can be found in the repository (Appendix W).

A kanban was created in *Trello* (Appendix W) to keep track of required and completed work. Also, an agile methodological approach was adopted when it came to the development phase [44]. Specifically, utilising the principles of *SoC* (Separations of Concerns) [45]. *Separations of Concerns* is a design concept that divides a computer program into different parts, such that each segment tackles a specific *concern*. A *concern* is a collection of knowledge affecting a computer program's code. As a result, the following five concerns were derived for the java classes (also seen in Figure 9):

- **Activities** - classes for screens of *Motor Manager*
- **Adapters** - classes which allow incompatible objects to collaborate
- **Fragments** - classes for screens of *Motor Manager* (specifically for the menu pages)
- **Helpers** - classes which contain code for additional (post-MVP) functionality
- **Models** - classes that define information about data

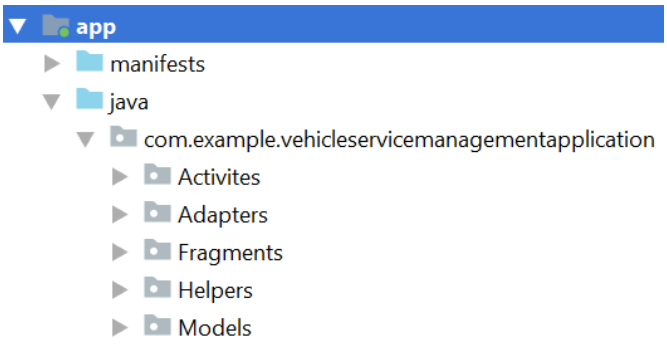


Figure 9: Principles of SoC - Java Class Folders

4.1.1 Activities

The *Activities* folder, seen in Figure 10, contains classes that encompass the code required for the functionality of various screens in *Motor Manager*. These involve the opening, register, login, home, and vehicle detail screens. The naming convention of each class (activity) was derived by the functionality they enclosed. All classes in *Activities* maintained a similar structure:

- Import statements
- Opening of class (*extends* and *implements* statements)
- Main method (*onCreate*)
- Specialised methods

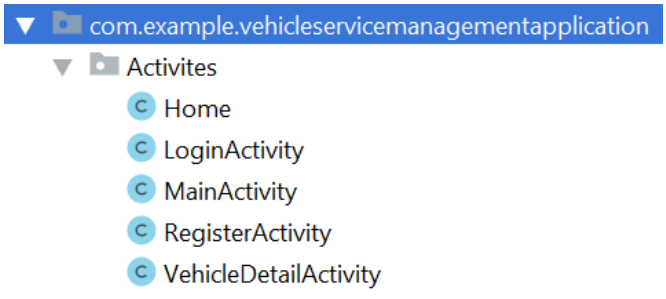


Figure 10: Activites folder and encompassed classes

Home.java

Home activity primarily serves the functionality for adding vehicles and their details. This is made possible via the combination of multiple methods. Firstly, the *Home* class extends the base class *AppCompatActivity*. This is due to the *Home* class utilising support library action bar features (all classes in *Activities* extend this parent class for the same reason). Also, this class implements interface *NavigationView.OnNavigationItemSelectedListener*. An interface is a collection of abstract methods. The methods are inherited by the class which implements their capsulising interface. In this case, the abstract method *OnNavigationItemSelectedListener* was inherited. This method is a listener for handling events on navigation items (e.g. Home, About, Settings).

Initialisation and the assignment operations of variables occurred next, this can be seen in Code Extract 1. The variables were of various types, as many each served a separate purpose. For instance, the variables *firebaseAuth* (authentication) and *currentUser* (currently logged in user) are both Firebase related variables, which ensure that the user who is logged in, has been authenticated correctly, and their data has been displayed accordingly. Additionally, the latter initialisations, i.e. *popAddVehicle, popUpPostImage* are for the fields on the add vehicle details pop-up. They consist of numerous data types such as *ImageView, TextView, Spinner,* and *String.*

Home.java

```
66  public class Home extends AppCompatActivity implements Navigati
67  {
68      // Variable Initialisations
69      private AppBarConfiguration mAppBarConfiguration;
70
71      // Request code variable for audio input
72      private static final int REQUEST_CODE_SPEECH_INPUT = 1000;
73
74      // Firebase database initialisations
75      FirebaseAuth firebaseAuth;
76      FirebaseUser currentUser;
77
78      // Dialog initialisation (pop-up to add vehicles)
79      Dialog popAddVehicle;
80
81      // Image variables for popup
82      ImageView popUpUserImage;
83      ImageView popUpPostImage;
84      ImageView popUpAddButton;
85
86      // Text variables to store field data
87      TextView popUpTitle;
88      TextView popUpDescription;
89      String popUpVehicleMake;
90      TextView popUpVehicleModel;
91      String popUpVehicleTransmission;
92      String popUpVehicleOil;
93      TextView popUpVehicleNote;
```

Code Extract 1: Home Class - Variable Initialisation

Furthermore, the methods were established next. Table 14 contains the method name, required parameters, return value, and intended functionality.

Table 14: Home Class - Methods

Method	Parameters	Returned Value	Functionality
onCreate	*Bundle* savedInstanceState	Void	Acts as the main method. Contains firebase database variables, on-click listener for floating action button (vehicle details addition popup), toggle for menu, 'Home' page is the default.
setupPopUpImageClick	None	Void	On-click listener to check if a user

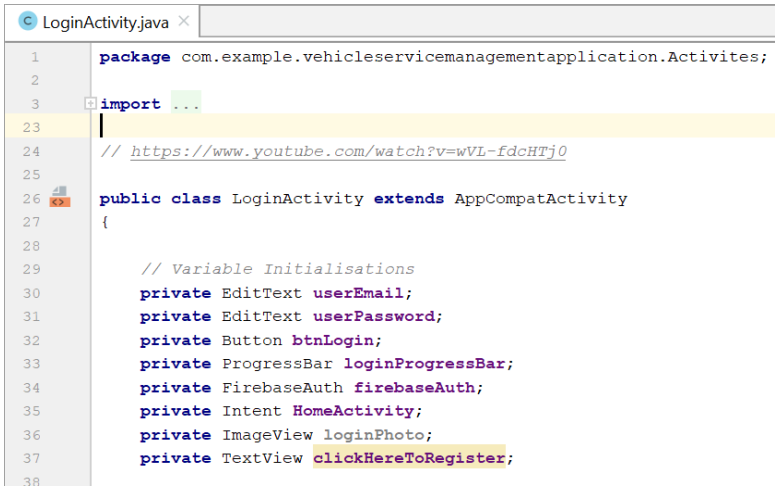
			wants to add a vehicle image. It also requests user permission before accessing the user's gallery.
checkAndRequestForPermission	None	Void	Checks if the user has granted permission for *Motor Manager* to access their external storage if not, it requests this permission via a *toast* (printed message).
openGallery	None	Void	If the user has granted *Motor Manager* permission to access their storage, this method will open their device gallery.
onActivityResult	*int* requestCode *int* resultCode *Intent* data	Void	Store the image the user selected for their vehicle, and then display the image in the pop-up vehicle details addition image area.
inPopup	None	Void	Checks if the fields are full. If they are, then the field values (except images) are converted to strings and stored in a database reference. The image is assigned a downloadable URL, string. If fields are not filled, a *toast* (message) appears on the screen informing the user.
addPost	*Post* post	Void	All the fields, which now have a Firebase reference, are sent to Firebases Realtime

			Database. Images are stored in Firebase's Cloud Storage.
showMessage	*String* message	Void	Displays *toast* (message) on the screen to the user.
onBackPressed()	None	Void	Autogenerated for the drawer layout (menu and pages).
onCreateOptionsMenu	*Menu* menu	Boolean	This displays the menu item to the action bar if it is present.
onNavigationItemSelected	*MenuItem* menuItem	Boolean	Takes the user to the page they selected from the menu.
updateNavHeader	None	Void	Collects the current user's username, email, and profile picture, and adds it to the navigation header.

LoginActivity.java

The *LoginActivity* is responsible for authenticating a user's login credentials, against those stored in Firebase, and signing them into the application. Imports for UI (User Interface) elements, on-complete listeners, and Firebase authentication objects were included.

Similarly, to the *Home* class, this class has the initialisation of numerous variables (e.g. *EditText, Button, Intent,* etc), seen in Code Extract 2. These variables align with the fields and items found on the *login* screen. For example, the *EditText* variable *userEmail* was created to store the user's email address.



Code Extract 2: LoginActivity Class - Variable Initialisation

The methods for this class were then created. Table 15 contains the method name, required parameters, return value, and intended functionality.

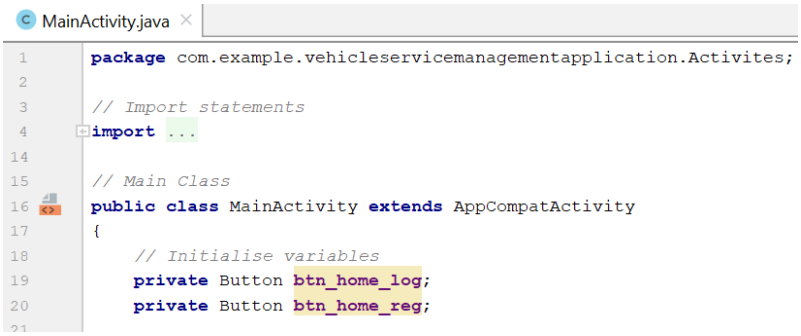
Table 15: LoginActivity Class - Methods

Method	**Parameters**	**Returned Value**	**Functionality**
onCreate	*Bundle* savedInstanceState	Void	Main method. Includes assignment operations of login screen fields. Code to allow the user to switch to the registration screen. On-click method to check when a user wants to log in. It also informs the user of missing fields or incorrect formatting.
signIn	*String* email *String* password	Void	Uses the Firebase Authentication object *firebaseAuth* and applies the built-in method *signInWithEmailAndPassword*. This method takes in an email address and password as arguments. If the credentials match those stored in Firebase's Authentication, then the *updateUI* method is called. If they fail, the user is informed via a message.
updateUI	None	Void	It starts the home activity.
showMessage	*String* userMessage	Void	Displays *toast* (message) on the screen to the user.
onStart	None	Void	Checks if the user that is currently authenticated is a null object. If not, the *updateUI* method is called.

MainActivity.java

The *MainActivity* is responsible for the opening screen of *Motor Manager*. The opening screen displays the *Login* and *Register* buttons, which take them to their respective screens. Imports for UI (User Interface) elements are included.

This class initialises two variables of type *Button*, as seen in Code Extract 3. These buttons correspond with the ones found on the opening screen.



Code Extract 3: MainActivity Class - Variable Initialisation

The methods for this class were then created. Table 16 contains the method name, required parameters, return value, and intended functionality.

Table 16: MainActivity Class - Methods

Method	**Parameters**	**Returned Value**	**Functionality**
onCreate	*Bundle* savedInstanceState	Void	Main method. Assignment operations for both *Login* and *Register* button. On-click listener methods for both *Login* and *Register* buttons, to check if the user clicks on them. If they do, the respective methods for opening the correct screen is called.
openLoginPage	None	Void	Opens Login*Activity.*
openRegisterPage	None	Void	Opens *RegisterActivity.*

RegisterActivity.java

The *RegisterActivity* is responsible for creating a new account, using the user's profile image, full name, email address, and password. Imports for UI (User Interface) elements, on-complete listeners, and Firebase authentication objects were included.

This class initialises multiple types of variables as seen in Code Extract 4. The UI elements correspond with the registration fields. A *FirebaseAuth* is also used to store user credentials in Firebases' Authentication.

RegisterActivity.java

```
1   package com.example.vehicleservicemanagementapplication.Activites;
2
3   import ...
34
35  // https://www.youtube.com/watch?v=yHAAg-RdKDY
36  // https://www.youtube.com/watch?v=o5x0HB-EYJY
37  // https://www.youtube.com/watch?v=6js4iUobzbc
38
39  public class RegisterActivity extends AppCompatActivity
40  {
41      // Variable Initialisations
42      // Specific to image upload
43      ImageView ImgUserPhoto;
44      static int PReqCode = 1;
45      static int REQUESTCODE = 1;
46      Uri chosenImgUri;
47
48      // Specific to fields on registration
49      private EditText userName;
50      private EditText userEmail;
51      private EditText userPassword;
52      private EditText userPassword2;
53      private ProgressBar loadingProgressBar;
54      private Button regButton;
55      private TextView clickHereToLogin;
56
57      // Firebase initialisation
58      private FirebaseAuth firebaseAuth;
```

Code Extract 4: RegisterActivity Class - Variable Initialisation

The methods for this class were then created. Table 17 contains the method name, required parameters, return value, and intended functionality.

Table 17: RegisterActivity Class - Methods

Method	Parameters	Returned Value	Functionality
onCreate	*Bundle* savedInstanceState	Void	Main method. Includes assignment operations of registration

			screen fields. Code to allow the user to switch to the login screen. On-click method to check when a user wants to register. It also informs the user of missing fields or incorrect formatting.
CreateUserAccount	*final String* name *String* email *String* password	Void	Takes email and password and checks if the email has been registered already. If not, a new account is created. There is an option for creating an account without a profile image. If the account registration is unsuccessful, an error message will be shown to the user.
updateUserAccount	*final String* name *Uri* chosenImgUri *final FirebaseUser* currentUser	Void	This updates the user's menu profile (name, email, and profile image). It also called the *updateUI* method.
updateUserAccountWithoutUserPhoto	*final String* name *final FirebaseUser* currentUser	Void	Similar to the method above, but for user accounts, without a profile image.
updateUI	None	Void	Once registration is complete, sends the user

			to the *Home* class (home page).
showMessage	*String* userMessage	Void	Displays *toast* (message) on the screen to the user.
checkAndRequestForPermission	None	Void	Checks if the user has granted permission for *Motor Manager* to access their external storage if not, it requests this permission via a *toast* (printed message).
openGallery	None	Void	If the user has granted *Motor Manager* permission to access their storage, this method will open their device gallery.
onActivityResult	*int* requestCode *int* resultCode *Intent* data	Void	Checks if *resultCode, requestCode,* and *data* variables are satisfied using a boolean *if* statement*. If they are*, stores the image the user selected and then displays the image in the menu profile.

VehicleDetailActivity.java

The *VehicleDetailActivity* contains code that is responsible for the display of vehicle details. Imports for UI elements and Firebase's Realtime Database objects were made. Also, the *Glide* library was added, for image loading and caching.

This class initialises one *ImageView*, for the vehicle image, and one *String* variable, for the vehicle key in the database. *TextView* variables were also

initialised as seen in Code Extract 5. They match the number of fields in the vehicle details addition form, as the data stored from these fields are displayed.

VehicleDetailActivity.java

```
28  import java.util.Locale;
29
30
31  // VehicleDetailActivity class
32  public class VehicleDetailActivity extends AppCompatActivity
33  {
34      // Variable Initialisation
35      // IMAGES
36      ImageView imageVehicle;
37      // TEXT
38      TextView textVehicleTitle;
39      TextView textVehicleMake;
40      TextView textVehicleModel;
41      TextView textVehicleTransmission;
42      TextView textVehicleOil;
43      TextView textVehicleDate;
44      TextView textVehicleDescription;
45      TextView textVehicleNote;
46      // STRING
47      String VehicleKey;
48
```

Code Extract 5: VehicleDetailActivity Class - Variable Initialisation

The methods for this class were then created. Table 18 contains the method name, required parameters, return value, and intended functionality.

Table 18: VehicleDetailActivity Class - Methods

Method	Parameters	Returned Value	Functionality
onCreate	*Bundle* savedInstanceState	None	Main method. Assignment operations for all initialised variables. Use the *Glide* library to send and display vehicle images. Uses '*.setText'* method to display vehicle data.
convertTimeStampToString	*long* time	String	Convert timestamp to date (DD-MM-YYYY).

4.1.2 Adapters

The *Adapters* folder, seen in Figure 11, contains a single class (*PostAdapter*). The intended purpose of *Adapters* classes is to allow incompatible objects to

cooperate. For instance, a class would take an input call for one object and transform it into output, which the second object recognises.

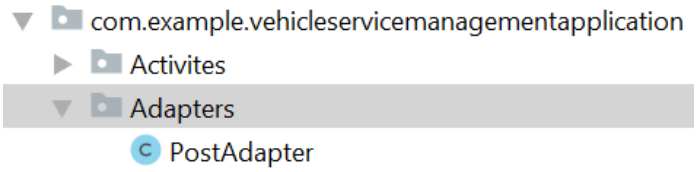


Figure 11: Adapters folder and encompassed classes

PostAdapter.java

This class takes the inputted data from the *Home* class (vehicle details addition pop-up form) and turns it into a form that the *VehicleDetailActivity* can handle. This is the reason *PostAdapter* extends the base class *RecyclerView.Adapter<PostAdapter.MyViewHolder>*. It facilitates binding between app-specific data to views that are displayed within a *RecyclerView*. It also creates a thumbnail (vehicle image and registration) for the vehicle on the home screen. Import statements include UI elements and the *Glide* library.

Initialisation and assignment operations were carried for variables m_(*Context*) and *m_Data* (*List<Post>*). The *context* variable stores the environment *Motor Manager* is currently in and the *list* variable stores *Post* (individual vehicle) objects. Code Extract 6 shows the *PostAdapter* class constructor.

```
// PostAdapter Class
public class PostAdapter extends RecyclerView.Adapter<PostAdapter.MyViewHolder>
{
    // Variable initialisation
    Context m_Context;
    List<Post> m_Data;

    // Constructor
    public PostAdapter(Context m_Context, List<Post> m_data)
    {
        // Assignment operations
        this.m_Context = m_Context;
        this.m_Data = m_data;

    // end of constructor
    }
```

Code Extract 6: PostAdapter Class - Constructor

The methods for this class were then created. Table 19 contains the method name, required parameters, return value, and intended functionality.

Table 19: PostAdapter Class - Methods

Method	Parameters	Returned Value	Functionality
PostAdapter	*Context* m_Context *List<Post>* m_Data	N/A	Create instances of the class who they are the constructor of. Assignment operations of variables occur.
onCreateViewHolder	*ViewGroup* parent *int* viewType	MyViewHolder	*Parent* is the *RecyclerView* (holds your cell). *ViewType* useful for different types of cells in a list e.g. vehicle registration, image. This ensures the layout file is inflated accurately by the types of cells.
onBindViewHolder	*MyViewHolder* holder *int* position	Void	Place vehicle image and registration in a specific format (thumbnail).
getItemCount	None	*int*	Returns the size of the *m_Data* (list variable).

An inner class, *MyViewHolder*, was also created. This ensured increased encapsulation, logical grouping of classes, and more readable code. The inner class extended *RecyclerView.ViewHoldler*, which describes the item view. Also, it contains metadata regarding its location within a *RecyclerView.*

A constructor, *MyViewHolder*, was created, as seen in Code Extract 7. This constructor assigned the vehicle registration and vehicle image to their respective elements id's (found in their *XML* layout files). Also, there is an on-click listener to check if a user has clicked on the thumbnail of the vehicle. If this has occurred, the vehicle details (e.g. registration number, image, transmission, etc) were sent to the *VehicleDetailActivity* class, to be displayed.

```
// Constructor
public MyViewHolder(@NonNull View itemView)
{
    super(itemView);

    // Assignment operation
    tv_Title = itemView.findViewById(R.id.row_post_title);
    imageVehicle = itemView.findViewById(R.id.row_post_image);

    // On click listener
    itemView.setOnClickListener(new View.OnClickListener()
    {
        @Override
        public void onClick(View v)
        {
            // Set an intent and store vehicle detail activity
            Intent vehicleDetailActivity = new Intent(m_Context, VehicleDetailActivity.class);
            // Store the position of the adapter in a variable
            int pos = getAdapterPosition();
            // Send the vehicle title
            vehicleDetailActivity.putExtra("Title", m_Data.get(pos).getTitle());
            // Send the vehicle image
            vehicleDetailActivity.putExtra("VehicleImage", m_Data.get(pos).getPicture());
            // Send the vehicle make
            vehicleDetailActivity.putExtra("Make", m_Data.get(pos).getMake());
            // Send the vehicle model
            vehicleDetailActivity.putExtra("Model", m_Data.get(pos).getModel());
            // Send the vehicle transmission
            vehicleDetailActivity.putExtra("Transmission", m_Data.get(pos).getTransmission());
            // Send the vehicle oil
            vehicleDetailActivity.putExtra("Oil", m_Data.get(pos).getOil());
            // Send the vehicle description
            vehicleDetailActivity.putExtra("Description", m_Data.get(pos).getDescription());
            // Send the vehicle detail post key
            vehicleDetailActivity.putExtra("VehicleKey", m_Data.get(pos).getVehicleKey());

            // Store timestamp of vehicle details addition
            long timeStamp = (long) m_Data.get(pos).getTimeStamp();
            // Send the the timestamp of vehicle detail
            vehicleDetailActivity.putExtra("VehicleDetailsAdditionDate", timeStamp);

            // From current context, start new activity to go to vehicle details activity
            m_Context.startActivity(vehicleDetailActivity);


        // end of on click method
        }
    });


// end of constructor
```

Code Extract 7: PostAdapter Class - Inner Class - Constructor

4.1.3 Fragments

As aforementioned, the *Fragments* folder contains the classes for screens of *Motor Manager*, specifically the menu pages, i.e. *home, settings, about.* A *Fragment* represents the behavior of a user interface in a *FragmentActivity*. A combination of multiple fragments can produce a multi-pane UI (a menu in this case). The *Motor Manager* application accommodates the *HomeFragment, AboutFragment, SettingsFragment* classes.

HomeFragment.java

The *HomeFragment* class extends the base class *Fragment*. Most imports, variables, and methods are auto-generated in Fragments as they have an already specified behavior of user interface. As seen in Code Extract 8, the variables created include a *PostAdapter* type variable to store an instance of that class, a *List<Post>* type variable to store the vehicles for an individual account, and a *RecyclerView* variable to display the user's vehicle as a list. Also, *FirebaseDatabase* and *DatabaseReference* variables have been created to retrieve stored vehicle data.

```
public class HomeFragment extends Fragment
{
    // TODO: Rename parameter arguments, choose names that match
    // the fragment initialization parameters, e.g. ARG_ITEM_NUMBER
    private static final String ARG_PARAM1 = "param1";
    private static final String ARG_PARAM2 = "param2";

    // TODO: Rename and change types of parameters
    private String mParam1;
    private String mParam2;

    private OnFragmentInteractionListener mListener;

    // Variable initialisation (not auto-generated)
    RecyclerView vehicleRecyclerView;
    PostAdapter postAdapter;
    List<Post> vehicleList;

    // Database related initialisations (not auto-generated)
    FirebaseDatabase firebaseDatabase;
    DatabaseReference databaseReference;
```

Code Extract 8: HomeFragment Class - Variable Intialisation

The methods for this class were then created. Table 20 contains the method name, required parameters, return value, and intended functionality.

Table 20: HomeFragment Class - Methods

Method	Parameters	Returned Value	Functionality
HomeFragment	None	N/A	Autogenerated, default constructor.
newInstance	*String* param1 *String* param2	HomeFragment	Autogenerated. Renames and changes types and numbers of parameters
onCreate	*Bundle* savedInstanceState	Void	Autogenerated main method. Stores parameters.

onCreateView	*LayoutInflater* inflater *ViewGroup* container *Bundle* savedInstanceState	View	Autogenerated. Links *HomeFragment* with its XML file. This is to display the *RecyclerView* which lists the stored vehicles for that user. Database reference "Vehicles" has also been stored.
onStart	None	Void	Makes fragment visible to the user.
onButtonPressed	*Uri* uri	Void	Checks if the user has clicked on a vehicle thumbnail.
onAttach	*Context* context	Void	Called when the fragment gets related to its activity.
onDetach	None	Void	Called just before a fragment is no longer related to its activity.

Lastly, this class contains an autogenerated interface *onFragmentInteractionListener*, as seen in Code Extract 9. This interface must be implemented by activities which contain the *HomeFragment* to allow interaction between this fragment and the activity.

```
public interface OnFragmentInteractionListener
{
    // TODO: Update argument type and name
    void onFragmentInteraction(Uri uri);
}
```

Code Extract 9: HomeFragment Class - Interface

ProfileFragment.java

The *ProfileFragment* class extends the base class *Fragment*. It has been created for the *About* page of *MotorManager*. Imports, variables, and methods are auto-generated in Fragments as they have an already specified behavior for their user interface. This fragment has no specialised methods or altered code, as it is only being used as an about page. Consequently, the *XML* file for this fragment is solely edited.

SettingsFragment.java

The *SettingsFragment* class extends the base class *Fragment*. It has been created for the *Settings* page of *MotorManager*. Imports, variables, and methods are auto-generated in Fragments as they have an already specified behavior for their user interface. This fragment has no specialised methods or altered code, as it is only being used as a settings page. Consequently, the *XML* file for this fragment is solely edited.

4.1.4 Helpers

Helpers encompassed classes for post-MVP functionalities e.g. dark mode, file upload. See Appendix L for these classes.

4.1.5. Models

Models encompass classes which have fields and actions of data which is being stored. Each *model* class maps to a single database table. In this project, only one model class *Post.java* has been created.

Post.java
The *Post* class has fields for vehicle detail storage (e.g. vehicle make, model, transmission), and it stores in a single database named "Vehicles*"*. This class contains a single import for Firebases' *ServerValue* class, which contains placeholder values to utilise when writing to the database. Code Extract 10 displays the variable initialisations for each vehicle detail field. Also, *object* and *string* variables were created to store the *timeStamp* (date and time when the details were stored) and *vehicleKey* (unique vehicle identifier) respectively.

```
// Post class
public class Post
{
    // Initialising variables which will be stored in database
    private String title;
    private String make;
    private String model;
    private String transmission;
    private String oil;
    private String description;
    private String note;
    private String picture;
    private String userId;
    private String userPhoto;
    private Object timeStamp;
    private String vehicleKey;
```

Code Extract 10: Post Class - Variable Intialisation

Code Extract 11 presents the parametrised constructor which was created to assign the declared variables their respective parameter values. Once a *Post* object is created (when a vehicle details are being stored), arguments would need to be supplied, which then get assigned to the corresponding variables.

```
// Constructor for all initialised variables
public Post(String title, String make, String model, String transmission,
            String oil, String description, String note,
            String picture, String userId, String userPhoto)
{
    this.title = title;
    this.make = make;
    this.model = model;
    this.transmission = transmission;
    this.oil = oil;
    this.description = description;
    this.note = note;
    this.picture = picture;
    this.userId = userId;
    this.userPhoto = userPhoto;
    this.timeStamp = ServerValue.TIMESTAMP;

// end of constructor
}
```

Code Extract 11: Post Class - Constructor

4.2 XML

XML (Extensible Markup Language) files are primarily used in android studio to design screens of an application. They are generally found in the *layout* folder. Each time an *activity* (class) file is created, an *XML* file is automatically generated. Figure 12 presents all the layout files used in the production of *Motor Manger*.

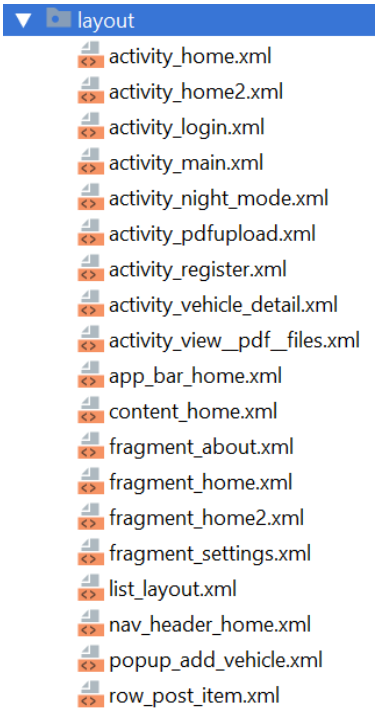


Figure 12: Layout Folder - XML Files

Specific layout files were also created to provide separate design and interactive functionality to certain pages. For example, *app_bar_home.xml*, *content_home.xml*, and *nav_header_home.xml* were all generated for the menu UI. Figure 13 displays the additional components contained in the *res* (resources) folder, which supply the data for features such as colour, dimensions, and style.

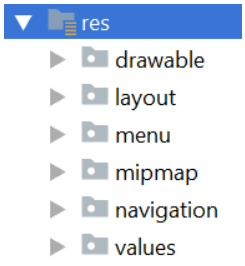


Figure 13: Resources Folder

As seen in Figure 14, *XML* files contained in the *layout* folder consist of the sections *Design* and *Text*. *Design* provides a preview to the application screen and *Text* is where the elements (e.g. images, buttons, text) seen on the preview are coded. All screens of *Motor Manager* can be seen in Appendix M.

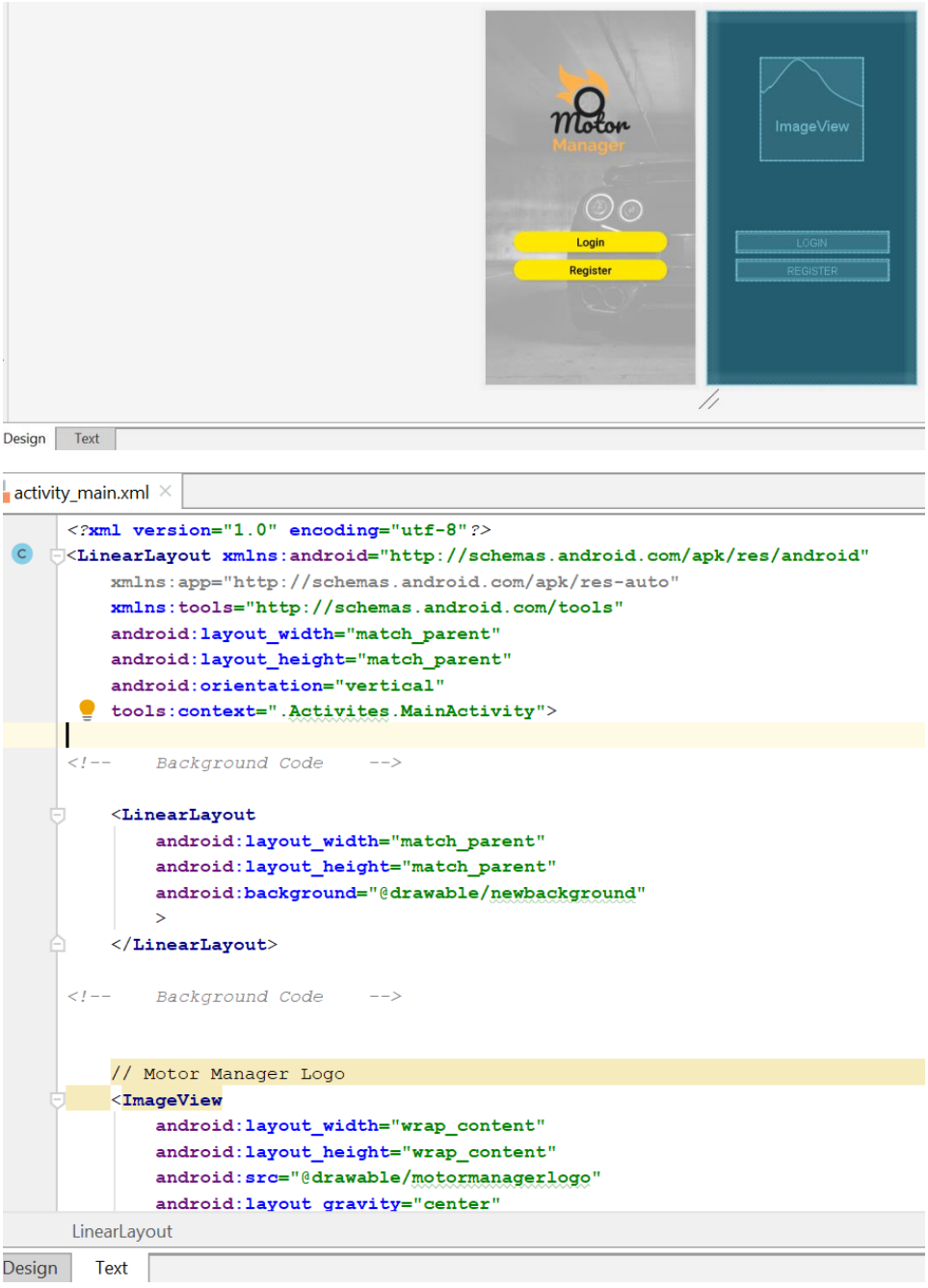


Figure 14: Main Activity XML - Design and Text

Figure 15 shows a code snippet from *activity_register.xml*. The code contained within the *ImageView* tags is responsible for the *ImageView* on the register page. It is assigned an *id* so that it can be referred to in class files. The rest of the code outlines the dimensions and positioning of the image.

```
// Profile Image Avatar
<ImageView
    android:id="@+id/regUserPhoto"
    android:layout_width="100dp"
    android:layout_height="100dp"
    android:layout_gravity="center"
    android:src="@drawable/userphoto"
    android:layout_marginTop="50dp"
    tools:ignore="ContentDescription">
</ImageView>
```

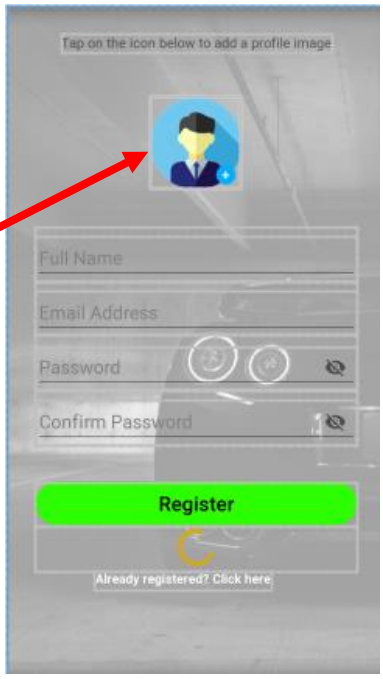


Figure 15: Register Page XML - Profile ImageView Code

4.3 Google's Firebase Development Platform

4.3.1 Firebase Authentication

Firebase Authentication is a backend service provided by Firebase to authenticate users on an application. The Firebase console displays registered users. In this project, Firebase authentication is used in the login and register activities (classes). Code Extract 12 displays the in-built, Firebase *signIn* method, which requires two parameters (email and password).

```
// Method to sign users in
private void signIn(String email, String password)
{
    // Sign users in with email and password
    firebaseAuth.signInWithEmailAndPassword(email, password).addOnCompleteListener((task) → {
            // Conditional to check if task is successful
            if(task.isSuccessful())
            {
                // Make progress bar invisible
                loginProgressBar.setVisibility(View.INVISIBLE);
                // Make login button visible
                btnLogin.setVisibility(View.VISIBLE);
                // After successful login, take user to correct screen
                updateUI();
            }
            // Else conditional if unsuccessful
            else
            {
                // Display error message
                showMessage(task.getException().getMessage());
                // Make login button visible
                btnLogin.setVisibility(View.VISIBLE);
                // Make progress bar invisible
                loginProgressBar.setVisibility(View.INVISIBLE);
            }
    });
}
```

Code Extract 12: LoginActivity Method - signIn

4.3.2 Firebase Realtime Database

Firebases' *Realtime Database* was used to store the vehicles and their details. Code Extract 13 shows the *addPost* method found in the *Home* class. The database reference used for the location of vehicle details was "Vehicles". Figure 16 presents how the vehicle data is stored in a *JSON* format.

```java
// Method to add vehicle details
private void addPost(Post post)
{
    // Initialised and assigned database variable to current instance
    FirebaseDatabase database = FirebaseDatabase.getInstance();
    // Initialise and assign database reference
    DatabaseReference myRef = database.getReference( path: "Vehicles").push();

    // Store database reference key in a variable
    String key = myRef.getKey();
    // Add a vehicle key to post
    post.setVehicleKey(key);
    // Send vehicle data to Firebase Realtime Database
    myRef.setValue(post).addOnSuccessListener((OnSuccessListener) (aVoid) → {
            // Inform user data stored successfully
            showMessage("Vehicle Successfully Added");
            // Make progress bar invisible
            popUpClickProgressBar.setVisibility(View.INVISIBLE);
            // Make vehicle add button visible
            popUpAddButton.setVisibility(View.VISIBLE);
            // Remove pop up dialog box
            popAddVehicle.dismiss();

        // end of on success method
    });
// end of addPost method
```

Code Extract 13: Home Class Method - addPost (Add Vehicle Details)

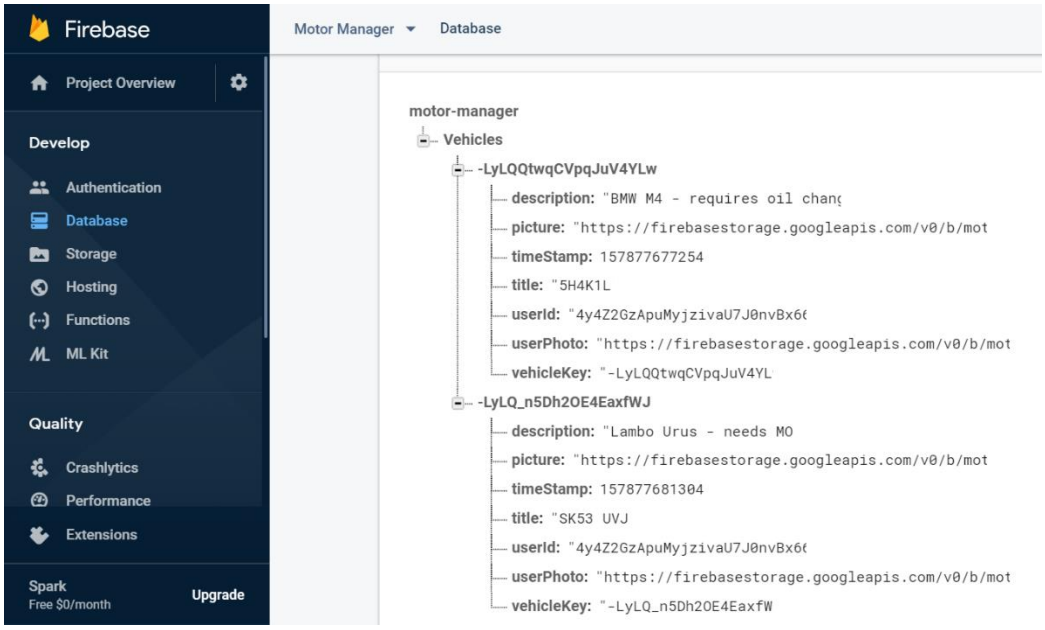


Figure 16: Firebase Console - Realtime Database

4.3.3 Firebase Storage

Lastly, *Firebase Storage* is a cloud service, primarily used for storage of files. *Motor Manager* incorporates this service to store user-profiles and vehicle images. Code Extract 14 and 15 present the storage references for vehicle images and user profile images, respectively. Figure 17 displays the storage folders (containing images) in the Firebase console.

```
// Create variables for storage reference and file path
StorageReference storageReference = FirebaseStorage.getInstance().getReference().child("Vehicle_Images");
```

Code Extract 14: Home Class - Storage Reference - Vehicle Images

```
// Variable to store user profile image storage reference
StorageReference mStorage = FirebaseStorage.getInstance().getReference().child("users_profilePhotos");
final StorageReference imgFilePath = mStorage.child(chosenImgUri.getLastPathSegment());
```

Code Extract 15: Home Class - Storage Reference - User Profile Images

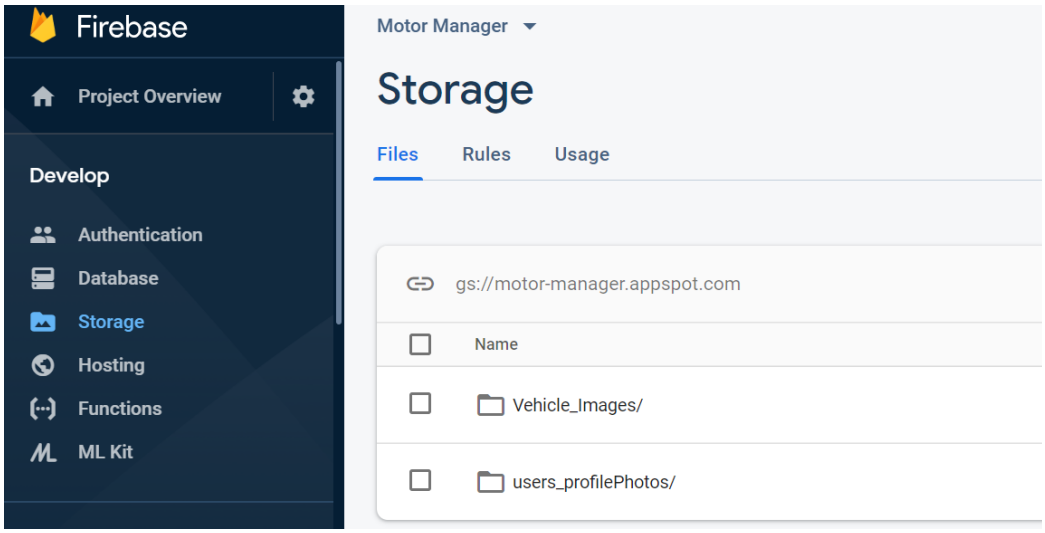


Figure 17: Firebase Console - Storage

See Appendix N for more *Firebase Development Platform* images.

4.4 Chapter 4 Summary

This chapter aimed to describe in detail the implementation process of *Motor Manager*. Development in *Android Studio* and *Firebase* integration was outlined.

An agile methodological approach was adopted for developing *Motor Manager*. The principle of *Separation of Concerns* was also implemented. Activities (classes) that had specific functionalities were separated. This increased the readability and maintainability of the code. The *XML* files for the design element of *Motor Manager* also adhered to these principles. Code extracts and figures were included to provide an insight into the main areas of the program.

The following chapter will describe the testing phase for the *Motor Manager* application. The testing stage will build on the implementation of this chapter.

5 Testing

This chapter describes the testing carried out for *Motor Manager*, following the implementation from the previous chapter. Various testing methodologies were incorporated throughout development, as well as on completion of the application, to assure quality consistently.

The following sections outline *white box testing* methods, analyses *black box testing* results, and describe *Firebase's Robo testing* output.

5.1 White Box Testing

During development, an agile workflow was adopted. This meant continuously coding certain functions and then running the application to check if they work. This comes under *white box testing.* White box testing is testing associated with the internal structure of an application [46]. Class files were created specifically for white box testing test cases (Appendix W). *Espresso* [47] and *JUnit4* [48] libraries were incorporated to write the programs for the tests. White box testing was fulfilled in three stages and documented via test cases:

- **Unit Testing** - individual units being testing
- **Integration Testing** - units combined and tested as a group
- **Systematic Testing** - test functions against a variety of inputs

5.1.1 Unit Testing

Unit testing consisted of testing individual components and ensuring they are semantically correct. Table 21 displays the unit test cases for *Motor Manager* (MVP) and their outcomes. Functions and classes are grouped by their responsibility in the program. Code Extract 16 displays an example of a unit test to launch the opening screen. Firstly, a *view* variable is created and assigned to the *login* button on the opening screen (*findViewById* takes in the UI element id i.e. in this case *btn_home_login*). This is so that if this button is detected, it confirms the opening screen has launched successfully.

Table 21: Unit Tests (MVP)

Test Case ID	Test	Class File (Java)	Test Class File (Java)	Status (Pass/Fail)
Activities				
1	onCreate(Bundle savedInstanceState)	MainActivity	MainActivityUnitTests	PASS
2	openLoginPage()	MainActivity	MainActivityUnitTests	PASS
3	openRegisterPage()	MainActivity	MainActivityUnitTests	PASS
4	onCreate(Bundle savedInstanceState)	Home	HomeUnitTests	PASS
5	setupPopUpImageClick()	Home	HomeUnitTests	PASS

6	checkAndRequestForPermission()	Home	HomeUnitTests	PASS
7	openGallery()	Home	HomeUnitTests	PASS
8	onActivityResult(int requestCode, int resultCode, @Nullable Intent data)	Home	HomeUnitTests	PASS
9	inPopup()	Home	HomeUnitTests	PASS
10	addPost(Post post)	Home	HomeUnitTests	PASS
11	showMessage(String userMessage)	Home	HomeUnitTests	PASS
12	onBackPressed()	Home	HomeUnitTests	PASS
13	onCreateOptionsMenu(Menu menu)	Home	HomeUnitTests	PASS
14	onNavigationItemSelected(@NonNull MenuItem menuItem)	Home	HomeUnitTests	PASS
15	updateNavHeader()	Home	HomeUnitTests	PASS
16	onCreate(Bundle savedInstanceState)	LoginActivity	LoginActivityUnitTests	PASS
17	signIn(String email, String password)	LoginActivity	LoginActivityUnitTests	PASS
18	updateUI()	LoginActivity	LoginActivityUnitTests	PASS
19	showMessage(String userMessage)	LoginActivity	LoginActivityUnitTests	PASS
20	onStart()	LoginActivity	LoginActivityUnitTests	PASS
21	onCreate(Bundle savedInstanceState)	RegisterActivity	RegisterActivityUnitTests	PASS
22	CreateUserAccount(final String name, String email, String password)	RegisterActivity	RegisterActivityUnitTests	PASS
23	updateUserAccount(final String name, Uri chosenImgUri, final FirebaseUser currentUser)	RegisterActivity	RegisterActivityUnitTests	PASS
24	updateUserAccountWithoutUserPhoto(final String name, final FirebaseUser currentUser)	RegisterActivity	RegisterActivityUnitTests	PASS
25	updateUI()	RegisterActivity	RegisterActivityUnitTests	PASS
26	showMessage(String userMessage)	RegisterActivity	RegisterActivityUnitTests	PASS
27	checkAndRequestForPermission()	RegisterActivity	RegisterActivityUnitTests	PASS
28	openGallery()	RegisterActivity	RegisterActivityUnitTests	PASS
29	onActivityResult(int requestCode, int resultCode, @Nullable Intent data)	RegisterActivity	RegisterActivityUnitTests	PASS

30	onCreate(Bundle savedInstanceState)	VehicleDetailActivity	VehicleDetailActivityUnitTests	PASS
31	convertTimeStampToString(long time)	VehicleDetailActivity	VehicleDetailActivityUnitTests	PASS
Adapters				
32	onCreateViewHolder(@NonNull ViewGroup parent, int viewType)	PostAdapter	PostAdapterUnitTests	PASS
33	onBindViewHolder(@NonNull MyViewHolder holder, int position)	PostAdapter	PostAdapterUnitTests	PASS
34	getItemCount()	PostAdapter	PostAdapterUnitTests	PASS
Fragments				
35	newInstance(String param1, String param2)	HomeFragment	HomeFragmentUnitTests	PASS
36	onCreate(Bundle savedInstanceState)	HomeFragment	HomeFragmentUnitTests	PASS
37	onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)	HomeFragment	HomeFragmentUnitTests	PASS
38	onStart()	HomeFragment	HomeFragmentUnitTests	PASS
39	onButtonPressed(Uri uri)	HomeFragment	HomeFragmentUnitTests	PASS
40	onAttach(Context context)	HomeFragment	HomeFragmentUnitTests	PASS
41	onDetach()	HomeFragment	HomeFragmentUnitTests	PASS
42	newInstance(String param1, String param2)	ProfileFragment	ProfileFragmentUnitTests	PASS
43	onCreate(Bundle savedInstanceState)	ProfileFragment	ProfileFragmentUnitTests	PASS
44	onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)	ProfileFragment	ProfileFragmentUnitTests	PASS
45	onButtonPressed(Uri uri)	ProfileFragment	ProfileFragmentUnitTests	PASS
46	onAttach(Context context)	ProfileFragment	ProfileFragmentUnitTests	PASS
47	onDetach()	ProfileFragment	ProfileFragmentUnitTests	PASS
48	newInstance(String param1, String param2)	SettingsFragment	SettingsFragmentUnitTests	PASS
49	onCreate(Bundle savedInstanceState)	SettingsFragment	SettingsFragmentUnitTests	PASS
50	onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)	SettingsFragment	SettingsFragmentUnitTests	PASS
51	onButtonPressed(Uri uri)	SettingsFragment	SettingsFragmentUnitTests	PASS
52	onAttach(Context context)	SettingsFragment	SettingsFragmentUnitTests	PASS
53	onDetach()	SettingsFragment	SettingsFragmentUnitTests	PASS

Helpers				
No classes in this folder (created for SoC purposes) are required for the Motor Manager MVP				
Models				
54	getTitle()	Post	PostUnitTests	PASS
55	getMake()	Post	PostUnitTests	PASS
56	getModel()	Post	PostUnitTests	PASS
57	getTransmission()	Post	PostUnitTests	PASS
58	getOil()	Post	PostUnitTests	PASS
59	getDescription()	Post	PostUnitTests	PASS
60	getNote()	Post	PostUnitTests	PASS
61	getPicture()	Post	PostUnitTests	PASS
62	getUserId()	Post	PostUnitTests	PASS
63	getUserPhoto()	Post	PostUnitTests	PASS
64	getTimeStamp()	Post	PostUnitTests	PASS
65	getVehicleKey()	Post	PostUnitTests	PASS
66	setTitle(String title)	Post	PostUnitTests	PASS
67	setMake(String make)	Post	PostUnitTests	PASS
68	setModel(String model)	Post	PostUnitTests	PASS
69	setTransmission(String transmission)	Post	PostUnitTests	PASS
70	setOil(String oil)	Post	PostUnitTests	PASS
71	setDescription(String description)	Post	PostUnitTests	PASS
72	setNote(String note)	Post	PostUnitTests	PASS
73	setPicture(String picture)	Post	PostUnitTests	PASS
74	setUserId(String userId)	Post	PostUnitTests	PASS
75	setUserPhoto(String userPhoto)	Post	PostUnitTests	PASS
76	setTimeStamp(Object timeStamp)	Post	PostUnitTests	PASS
77	setVehicleKey(String vehicle Key)	Post	PostUnitTests	PASS

```
//  Unit test - launching Main Activity (onCreate)
@Test
public void mainActivityTestLaunch()
{
    // Look for view once launched - to confirm successful launch
    View view = nActivity.findViewById(R.id.btn_home_login);
    // If view is not null - then has launched successfully
    assertNotNull(view);

// end of mainActivityTestLaunch method
}
```

Code Extract 16: Unit Test - mainActivityTestLaunch method

5.1.2 Integration Testing

Integration testing consisted of testing components as a group and ensuring they are cooperative. Table 22 displays the integration test cases for *Motor Manager* (MVP) and their outcomes. The column 'Test Case Objective' describes the type of test performed. Code Extract 17 displays an example of an integration test to launch the login page from the opening screen. It initially checks if the *login* button is present on the opening screen (*findViewById* locates its id) using the *assertNotNull* method. The *login* button is then clicked using the *perform* method. An *activity* variable is then created to store a *monitor* for the *LoginActivity* and provides a time limit (5000ms) in which it must open within. A final *assertNotNull* checks whether the *login* page has opened.

Table 22: Integration Tests (MVP)

Test Case ID	**Test Case Objective**	**Test Case Description**	**Expected Result**	**Test File (Java)**	**Status (Pass/Fail)**
1	Check the link between the *opening* screen and *login* screen	Click the 'Login' button	To be directed to the *login* page	MainToLoginIntegrationTest	PASS
2	Check the link between the *opening* screen and *register* screen	Click the 'Register' button	To be directed to the *register* page	MainToRegisterIntegrationTest	PASS
3	Check if an account be registered via the *register* screen	Register a new account by filling in the required fields e.g. full name, email, profile image, etc	To be directed to the *home* screen of *Motor Manager*	RegisterToHomeIntegrationTest	PASS
4	Check if an already registered account accessed via the *login* screen	Login using credentials which have already been registered e.g. USER: shakil@gmail.con, PASSWORD: hello123	To be directed to the *home* screen of *Motor Manager*	LoginToHomeIntegrationTest	PASS

5	Check the link between the *home* page and *about* page	Click the menu icon, to open the side-bar navigation, and click 'About'	To be directed to the *about* page of *Motor Manager*	HomeToAboutIntegrationTest	PASS
6	Check the link between the *home* page and *settings page*	Click the menu icon, to open the side-bar navigation, and click 'Settings'	To be directed to the *settings* page of *Motor Manager*	HomeToSettingsIntegrationTest	PASS
7	Check the link between the *home page* and *opening* screen (signing out)	Click the menu icon, to open the side-bar navigation, and click 'Sign out'	To be directed to the *opening* screen of *Motor Manager*	HomeToMainIntegrationTest	PASS
8	Check the link between *home* and *vehicle details* screen	Click on an uploaded vehicle from the home screen	To be directed to the vehicle details page (for a specific, selected vehicle)	HomeToVehicleDetails IntegrationTest	PASS

```
// Integration test - launching login activity from main activity
@Test
public void mainToLogin()
{
    // If view not null, then find element (login button)
    assertNotNull(nActivity.findViewById(R.id.btn_home_login));
    // Click on 'login' button
    onView(withId(R.id.btn_home_login)).perform(click());
    // Wait until the monitor has been hit (stored in activity variable)
    Activity loginActivity = getInstrumentation().waitForMonitorWithTimeout(monitorLogin, timeOut: 5000);
    // If view is not null - then has launched successfully
    assertNotNull(loginActivity);
    // End activity
    loginActivity.finish();


// end of main to login method
}
```

Code Extract 17: Integration Test - mainToLogin method

5.1.3 Systematic Testing

Systematic testing consisted of testing functions against multiple inputs, focusing on system validation, rather than functional verification (unit testing) or function interoperability (integration testing). Table 23 displays the systematic test cases for *Motor Manager* (MVP) and their outcomes. Test cases have been organised according to their functions. Code Extract 18 displays an example of a systematic test on the login screen. Multiple *string* variables were created for the *email* and *password* fields. The method *testLogin1* was one of four tests applied to the login form (checked whether a non-email address and correct password would log a user in).

Table 23: Systematic Tests (MVP)

Test Case ID	Test Case Objective	Test Case Description	Expected Result	Test File (Java)	Status (Pass/Fail)
Registration Functionality					
1	Check if the registration text input fields can take string values	Enter a string in each text input field. For example, for the 'Full Name' field, enter 'Shakil Ali'	The entered string will be recognised as an accepted value	RegisterActivity SystematicTest	PASS
2	Check if the registration text input fields can take integer values	Enter an integer in each text input field. For example, for the 'Full Name' field, enter '0123456'	The entered string will not be recognised as an accepted value	RegisterActivity SystematicTest	PASS
3	Check if the registration text input fields can take special characters e.g.*/?	Enter special characters in each text input field. For example, for the 'Full Name' field, enter '*?>@}+£'	Entered characters will not be recognised as an accepted value	RegisterActivity SystematicTest	PASS
4	Check if registration can be completed (without adding a profile image)	Complete registration form without adding a profile image	Account created successfully without profile image	RegisterActivity SystematicTest	PASS
5	Check if registration can be	Complete registration form	Account created successfully,	RegisterActivity SystematicTest	PASS

	completed (by adding a profile image)	by adding a profile image	displaying profile image		
Login Functionality					
6	Check if a non-email address can be used to login	Complete login field: *username*, with a non-email address e.g. 'Shakil'	The error message should be displayed regarding the entered value	LoginActivitySystematicTest	PASS
7	Check if an email address can be used to login	Complete the login field: *username*, with an email address e.g. 'shakil@gmail.com'	Account should be logged in to successfully	LoginActivitySystematicTest	PASS
8	Check if email address case sensitivity is considered	Complete the login field: *username*, with an email address e.g. 'SHAKIL@GMAIL.COM'	Account should be logged in to successfully	LoginActivitySystematicTest	PASS
9	Check if password case sensitivity is considered	Complete the login field: *password*, with a different capitalisation of the original password. For example, original password: hello123, test password: HELLO123	Error message should appear informing that details entered are incorrect	LoginActivitySystematicTest	PASS
Vehicle Detail Addition Functionality					
10	Check if the vehicle detail addition pop-up text input fields can take string values	Enter a string in each text input field. For example, for the 'Registration' field, enter 'SHAKIL' (except vehicle image upload and drop-down selection)	The entered string will be recognised as an accepted value	VehicleDetailActivitySystematicTest	PASS
11	Check if the vehicle detail addition pop-up text input fields can take	Enter an integer in each text input field. For example, for the 'Registration' field, enter '574413' (except	Entered integer will be recognised as an accepted value	VehicleDetailActivitySystematicTest	PASS

	integer values	vehicle image upload and drop-down selection)			
12	Check if the vehicle detail addition pop-up text input fields can take special characters	Enter a string in each text input field. For example, for the 'Registration' field, enter '?*)(}£$' (except vehicle image upload and drop-down selection)	Entered special characters will be recognised as an accepted value	VehicleDetailActivitySystematicTest	PASS

```
// Original credentials
private String email_original = "shakil@gold.ac.uk";
private String password_original = "hello123";


// Test credentials #1
private String email1 = "shakil";
private String password1 = "hello123";

// Test credentials #2
private String email2 = "shakil@gold.ac.uk";
// use password 1

// Test credentials #3
private String email3 = "SHAKIL@GOLD.AC.UK";
// user password 1

// Test credentials #4
// user email 2
private String password2 = "HELLO123";


@Test
//  Systematic test - launching Login Activity and signing in
public void testLogin1()
{
    // Input email into email field
    Espresso.onView(withId(R.id.loginEmail)).perform(typeText(email1));
    // Input password into password field
    Espresso.onView(withId(R.id.loginPassword)).perform(typeText(password_original));
    // Close soft keyboard
    Espresso.closeSoftKeyboard();
    // Perform button click (on login button)
    Espresso.onView(withId(R.id.loginBtn)).perform(click());

// end of testLogin1 method
}
```

Code Extract 18: Systematic Test - Test credentials and mainToLogin method

See Appendix O for post-MVP white box testing.

5.2 Black Box Testing

Once the development of *Motor Manager* was finalised, testing had to be carried out with the end-users, to check if requirements had been fulfilled. This meant *black-box testing* methods had to be employed. Black box testing explores the functionality of the software without prior knowledge of its interior framework [49]. This primarily occurs in the form of *user testing*.

5.2.1 User Testing

User testing is a black-box testing method which occurs post-application development. It consisted of potential end-users using *Motor Manager* and answering surveys (designed to assess the functionality of the software). The survey can be seen in Appendix P (MVP functionality) and Appendix Q (post-MVP functionalities). To help testers traverse all areas of the application as well as test all capabilities, an instruction sheet was provided (Appendix R).

The first survey consisted of seventeen questions, with an additional (optional) five questions. The extra questions were to acquire user feedback. A total of 30 responses were received. The second survey consisted of five questions and 30 responses were also received (the same individuals took both surveys).

The first four exploratory questions, from the first survey, were to get an overview of the profile of users testing *Motor Manager*. They asked for age, gender, profession, and whether they required a vehicle for work or owned a vehicle of their own. The majority of testers (43.3%) were 18-24 years of age and males made up 66.7% of the sample. Also, 90% of users (27 out of 30) owned a vehicle or had a vehicle(s). Figure 18 displays a pie chart, depicting the various professions of the testers in the sample. The largest profession was *students* (46.7%), however, 53.3% of the testers worked in a vehicle-oriented environment.

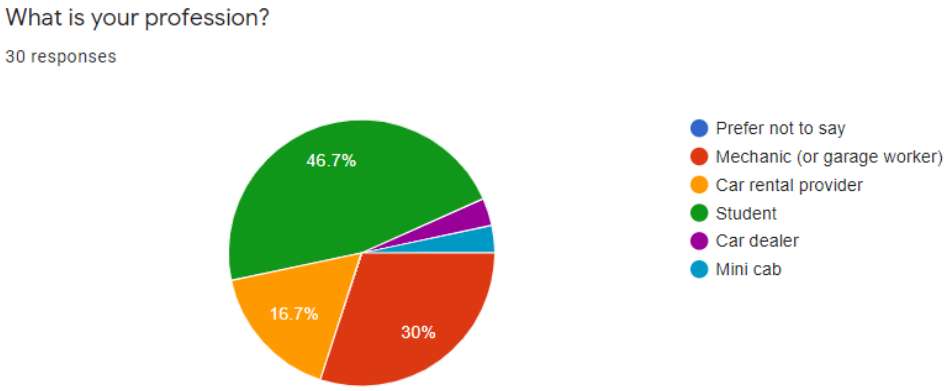


Figure 18: Tester Professions - Pie Chart

The next thirteen questions, of the first survey, were regarding the functionalities of the application. They were simultaneously being answered by the surveyors as they used the instruction sheet to traverse *Motor Manager*. All 30 surveyors found that the MVP for the application worked properly, as all thirteen questions were answered with 'Yes', 100% of the time (Appendix S). As a result, this indicated that no significant issues arose in the functionality of the *Motor Manager* MVP. The results of the second survey were the same, as testers answered 'Yes', 100% of the time (Appendix T). As a result, this indicated that no significant issues arose in the post-MVP functionality of *Motor Manager*. Overall, no bugs had to be rectified and the first version of *Motor Manager* was rendered successful.

Furthermore, as aforementioned, there were an additional five questions, from the first survey, seeking user feedback (Appendix S). Table 24 below summarises the responses.

Table 24: Additional Questions Responses - Summary

Question	Response	Analysis
Did the application meet the Minimum Viable Product (MVP) requirements?	All the thirty surveyors answered 'Yes' (definition of MVP was provided in the instructions sheet)	MVP requirements fulfilled successfully, therefore no coding changes required
How intuitive did you find the application? (1 = Not intuitive 5 = Intuitive)	• 1 (0%) • 2 (0%) • 3 (20%) • 4 (60%) • 5 (20%)	Testers generally found the application intuitive, therefore this aspect has been fulfilled to an extent
How good was the user interface? (1 = Not good 5 = Excellent)	• 1 (0%) • 2 (3.3%) • 3 (23.3%)	Testers generally found the interface to be of a good standard, therefore this aspects

	• 4 (53.3%) • 5 (20%)	has been fulfilled to an extent
What was the best aspect of *Motor Manager*?	• Intuitiveness (20%) • User Interface (53.3%) • Accessibility (0%) • Price (0%) • Functions (23.3%) • Security (3.3%)	Testers found that the best aspect was the *user interface*, which matches the applications from the secondary research (literature review) outcomes
What can be added or improved in future versions of *Motor Manager*?	• Make the user interface more specific to vehicle management e.g. vehicle wallpaper • Edit vehicle details • Customisable displayable • Audio output functions • Image recognition e.g. camera opens up and detects registration plate number	Additions already implemented in the post-MVP *Motor Manager* application: • Voice input • Dark mode • File upload • File download • Delete vehicle detail The following will be implemented in future versions: • Graphical displays, visualising vehicle data e.g. line graph estimating oil refill date • Edit vehicle details • Customisable user display • Image recognition • Audio output

5.3 Firebase Robo Testing

Firebase Development Platform contains a testing environment: *Test Lab*. *Test Lab* facilitates *robo testing* of an application across multiple devices. Various configurations can be applied and results (e.g. videos, logs, screenshots) are outputted. *Robo testing* is an automated test, whereby an applications internal structure is traversed. It was conducted to confirm whether non-functional requirements were met. The results (crawl graphs and performance statistics) displayed that the *Motor Manager* application was successful in meeting these requirements. For further details, please see Appendix U for the *Firebase Robo Test* results.

5.4 Chapter 5 Summary

This chapter aimed to describe the various testing measures implemented during and after development to ensure the functionalities of *Motor Manager* were at a satisfactory standard.

White box testing methods (unit, integration, systematic) were employed during development. This allowed bugs to be spotted relatively quickly, as well as give an insight into whether the application is performing as required. Black box user testing was also carried out with potential end-users in the form of a survey. Results portrayed that both the MVP and post-MVP functionalities for *Motor Manager* were successful following the requirements outlined previously. *Firebase Robo Testing* was also carried out, as a form of acceptance and performance testing, to confirm *Motor Manager* met non-functional requirements.

The next chapter will present the conclusions of this work as well as possible lines of future work.

6 Conclusion

This chapter will aim to evaluate the project and its overall success. It will provide a summary of the project processes and actions taken, an overview of the main findings and describe a development plan, for changes which could be made in the future, to improve *Motor Manager* further.

6.1 Project Summary

This project was carried out in 5 phases to produce the final application.

The first phase included researching current solutions to the problem of vehicle management and gaining stakeholder views on the same topic. Primary research was conducted in the form of a survey, whereby stakeholders provided their views on any existing systems or methods they utilised. Also, what they would like to see in future applications. The secondary market research conducted a literature review of the top 15 existing applications from three popular application stores (Google Play Store, App Store, Amazon App Store). The application analysis compared 5 aspects: Intuitiveness, User Interface, Accessibility, Functions, Security. A further three aspects (Target Audience, Application Size, Price) were also noted for comparison.

In the second phase, technical research went into selecting the development environment and database server. This resulted in *Android Studio* and *Firebase* technologies being employed, respectively.

In the third phase, system design began with initial storyboards, wireframes, and prototypes. The prototypes were designed in *AdobeXD* and a survey was carried out to get user insights. Using user feedback, a final prototype was conceived. Next, a *Systems Requirements Specification* (SRS) was created. It encompassed functional and non-functional requirements, which were derived from the storyboards, wireframes, and prototypes. Other system design diagrams were also created (sequence, activity, class, technical architecture, database schema), intending to give a visualisation into the technical framework of *Motor Manager*.

Following the design of the application, the fourth phase - implementation - commenced. *Motor Manager* was created in *Android Studio* and the *Separation of Concerns* (Soc) principle was adopted. This was so that the class files could be organised according to their function. The *Firebase Realtime Database* and *Storage* were integrated with the application to store user data.

Upon the conclusion of development, the last phase of testing began. Testing consisted of white box and black box testing. Black box testing was carried out in the form of a survey, in which an individual tested the application (following an instruction sheet), then submitted their answers. White box testing was

carried out via unit tests, integration tests, and systematic tests. These tests had their own *Java* classes created, with test cases, and the test results were documented. After the completion of testing, *Motor Manager* was deemed successful.

6.2 Evaluation

This section presents the evaluation of the project. It encompasses the main findings from the different phases of producing *Motor Manager*, as well as how an improvement could have been made.

6.2.1 Main Findings and Further Work

Background Research:
As aforementioned, primary and secondary market research was conducted. Primary market research was conducted in the form of a survey. There was a total of 15 responses received. Three distinct profiles were drawn from the responses: Profile 1 (male, garage worker, above 30, non-app user), Profile 2 (male, garage worker, 25-30, app user), Profile 3 (male/female, vehicle owner, 18-24, app user). The consensus portrayed that the application, which was to be developed, had to have a low barrier to entry (intuitive and simple interface) and facilitate efficient vehicle management. An improvement that may have been more beneficial for this project, is collecting more survey results. This will make data more representative and provide a more specific idea as to what a future vehicle management application should consist of.

Secondary market research was conducted on existing applications, development environments, and database servers. The literature review cross-examined 15 applications (5 each from Google's Play Store, App Store, Amazon App Store). Main results displayed that according to the criteria outlined in this project, intuitiveness, user interface, and functions were found to be prominently adequate in the applications. However, the analysis showed that accessibility and security functionalities were lacking. Consequently, these aspects were prioritised when developing *Motor Manager.* An improvement could be to review a wider number of applications, including other application stores. Additionally, further research and comparative-analysis could also be carried out to evaluate vehicle management systems from development environments and database servers.

System Design:
The system design phase aimed to establish the visual appearance of *Motor Manager*. Wireframes, storyboard, and prototypes were created. The prototype had a survey that testers had to complete. Utilising the feedback, a final prototype was produced. The Systems Requirements Specification (SRS) was derived from these models. It presented the functional and non-functional requirements. There were also additional diagrams (sequence, activity, class,

database schema) to display the technical functionality of *Motor Manager.* An enhancement could be to research more diagrams, to portray the application's functional capabilities more concisely.

Implementation:
The implementation consisted of coding classes according to their functionality. The principle of *Separations of Concerns (Soc)* was adopted and five class categories were created: Activities, Adapters, Fragments, Helpers, Models. Each category concerns a different functionality of *Motor Manager.* Also, the application was integrated with the *Firebase Authentication*, *Realtime Database*, and *Storage*. Once the Minimum Viable Product (MVP) was established (a registered user, storing and viewing vehicle data), post-MVP functionalities were programmed. These included deleting vehicle data, uploading and download files (related to a vehicle), dark mode, voice input for vehicle detail fields. Further work regarding the implementation could be including more complex functionalities e.g. edit existing data, visualising data graphically.

Testing:
Lastly, testing was carried out to ensure the quality and integrity of *Motor Manager* were maintained. Black box user testing was carried out in the form of a survey. Testers were provided with instructions and asked to submit their findings via the survey. Results reflected that all functions of *Motor Manager* worked as intended. White box testing was carried out in three stages: unit tests, integration tests, and systematic tests. These tests ensured individual components as well as units of programs, worked cooperatively. The conclusion of these tests also displayed that all intended functionalities worked as required. An improvement could be to carry out more vigorous types of testing e.g. acceptance testing. Also, having more people (stakeholders) test the application would produce a clearer picture of *Motor Manager*'s success.

6.2.2 Self Evaluation

This project consisted of me having inspiration for a vehicle management project, carrying out various forms of market research to get a better understand of the current and potential market, producing system diagrams to visualise the functionalities and appearance of *Motor Manager*, implementing the application in *Android Studio* and integrating it with *Firebase*, and interrogating the application via multiple testing methods. Undergoing this project process on my own has been very significant in my progress as a software developer. It is the first time I have planned and carried out a project of this scale, on my own. The experience gained will be pivotal for me going forward in future work.

Throughout this project I have developed skills, which will aid me in my career:

- **Project and time management:** Developing *Motor Manager* required multiple phases and planning. Each phase had to be completed under the deadline, to ensure the was project completed on time.
- **Technical skills:** Code was written in the *Java (Android)* programming language and *XML*. *JSON* was also written for the database rules in the *Firebase* console. This developed my programming abilities as well as improved my problem-solving capabilities. Additionally, organising the code into segments, relating to their functionality, enhanced my logical thought process.
- **Written communication:** Throughout this project, I have to create an interim report as well as this final project report. As a result, this has allowed me to practice and improve my ability when presenting technical information in a written document.

Overall, this project provided me with an insight into how Computer Science can be employed to solve problems in the real world. I am extremely content with the level of progress I was able to achieve in the time provided and believe this project has been a fundamental building block for my future endeavors.

6.3 Future Development Plan

Although the *Motor Manager* MVP was achieved, along with a few post-MVP functionalities, there are still many improvements that can be made, to benefit users and enable *Motor Manager* to compete alongside existing software systems.

The following subsections outline a future development plan for *Motor Manager,* in terms of which existing aspects can be improved and any new functionalities that can be incorporated.

6.3.1 Email/Password Change

The current application does not allow changing registered credentials e.g. email, password. This will be incorporated in future versions of *Motor Manager*. If the user requests an email or password change, they will receive an email informing them of the steps they have to take. The new credentials will then be updated in the database.

6.3.2 Account Deactivation

This will provide the user with an option to deactivate their account. This option is currently available in the *Firebase* console; however, it is currently not accessible via the application interface.

6.3.3 Customisable Display

There is a dark mode in the *Settings* page of *Motor Manager.* It changes the theme of the application to a ‘dark’ (black) colour. In future versions, other colour schemes will be available. This may also aid users with colour impairments.

6.3.4 Edit Vehicle Data

The current application allows users to create, delete, and view vehicle data. These operations are three out of four of the CRUD (create, read, update, delete) operations, and editing vehicle details will be incorporated in later versions.

6.3.5 Data Visuals (Artificial Intelligence)

Using artificial intelligence and stored vehicle data, the application will display graphs showing the current status of the vehicle (e.g. oil levels) and ‘smartly’ informing users when servicing is required (e.g. refueling required in two weeks).

6.3.6 OCR (Optical Character Recognition)

OCR (Optical Character Recognition) will allow users to upload images and have the text on the images, converted into text for fields. For example, if the user uploads an image of a vehicle’s number plate, this will be converted into text for the vehicle registration field.

6.3.7 Platforms

Motor Manager is currently only available on Android. However, future versions will look to expand onto other platforms e.g. IOS.

7 Bibliography

[1] D. f. T. a. D. a. V. L. Agency, "Vehicle licensing statistics: January to March 2017," UK Government, 15 06 2017. [Online]. Available: https://www.gov.uk/government/statistics/vehicle-licensing-statistics-january-to-march-2017. [Accessed 30 04 2020].

[2] S. R. Department, "Number of motor vehicle repair workshops in the United Kingdom 2004-2020," Statista, 28 10 2015. [Online]. Available: https://www.statista.com/statistics/547017/uk-number-of-vehicle-repair-workshops/. [Accessed 30 04 2020].

[3] Mordorintelligence, "CAR RENTAL MARKET - GROWTH, TRENDS, AND FORECAST (2020 - 2025)," Mordorintelligence, 01 01 2020. [Online]. Available: https://www.mordorintelligence.com/industry-reports/car-rental-market. [Accessed 01 05 2020].

[4] Google, "Google Play Store," Google, 07 03 2020. [Online]. Available: https://play.google.com/store?hl=en. [Accessed 06 11 2019].

[5] Apple, "App Store," App, 07 03 2020. [Online]. Available: https://www.apple.com/uk/ios/app-store/. [Accessed 06 11 2019].

[6] Amazon, "Amazon App Store," Amazon, 07 03 2020. [Online]. Available: https://www.amazon.co.uk/gp/feature.html?ie=UTF8&docId=1000644603. [Accessed 06 11 2019].

[7] G. Paré and S. Kitsiou, Handbook of eHealth Evaluation: An Evidence-based Approach, Victoria: University of Victoria, 2017.

[8] C. S. Fleisher, Business and Competitive Analysis: Effective Application of New and Classic Methods, Sydney: Macquarie University, 2007.

[9] A. Sharifulin, How to Do Competitive Analysis on Mobile Apps with Ease, Helsinki: Appfollow, 2019.

[10] T. C. Lacerda and C. G. v. Wangenheim, Systematic literature review of usability capability/maturity models, vol. 1, Florianópolis: Federal University of Santa Catarina, 2013, p. 16.

[11] C. Fleisher and B. Bensiussan, Standing Out From The Crowd: Improving Your Mobile App With Competitive Analysis, vol. 1, New Jersey: FT Press (Financial Times Prentice Hall), 2007, p. 1.

[12] J. Davies, "Word Cloud Generator," JasonDavies, 07 03 2020. [Online]. Available: https://www.jasondavies.com/wordcloud/. [Accessed 01 12 2017].

[13] B. E. Bensoussan and C. S. Fleisher, Business and Competitive Analysis: Effective Application of New and Classic Methods, New Jersey: Pearson FT Press, 2015.

[14] A. Blackler, Intuitive Use of Products, Brisbane: Queensland University of Technology, 2003.

[15] A. J.Ko and E. Whitmire, User Interface Software and Technology, Washington: University of Washington, 2018.

[16] S. Horton, A Web for Everyone: Designing Accessible User Experiences, USA: Rosenfeld Media, 2014.

[17] P. Berney, Mobile Marketing, London: Kogan Page , 2019.

[18] A. Genadinik, Mobile App Marketing And Monetization: How To Promote Mobile Apps Like A Pro: Learn to promote and monetize your Android or iPhone app. Get hundreds of thousands of downloads & grow your app business, New York: Semantic Valley LLC, 2014.

[19] D. Hermes, Xamarin Mobile Application Development: Cross-Platform C# and Xamarin.Forms Fundamentals, New York: Apress, 2015.

[20] J. N.Helfrich, Security for Software Engineers, London: Chapman and Hall/CRC, 2018.

[21] J. McCallister, Mobile Apps Made Simple: The Ultimate Guide to Quickly Creating, Designing and Utilizing Mobile Apps for Your Business - 2nd Edition (mobile application, ... programming, android apps, ios apps), Birmingham: Packt Publishing, 2014.

[22] Z. Abedjan, L. Golab and F. Naumann, Profiling relational data: a survey, Switzerland: Springer Link, 2015.

[23] T. Li, C. Chan, Z. Tang and X. Lin, Improve Developers' Coding Experience, Birmingham: University of Birmingham, 2016.

[24] R. Khandozhenko, "CROSS-PLATFORM MOBILE APPLICATION DEVELOPMENT," Oulu University of Applied Sciences, Oulu, 2014.

[25] B. Feigin, "MOBILE APPLICATION DEVELOPMENT “The search for common ground in a divided"," University of Birmingham, Birmingham, 2008.

[26] J. Iversen and M. Eierman, Learning Mobile App Development: A Hands-on Guide to Building Apps with iOS and Android 1st Edition, Boston: Addison-Wesley Professional, 2013.

[27] Android, "Android Studio," Android, 20 08 2019. [Online]. Available: https://developer.android.com/studio. [Accessed 15 02 2020].

[28] L. Anderson, Advantage Database Server: A Developer's Guide, Indiana: AuthorHouse, 2007.

[29] A. Thakur, "Top 11 Open Source Database for Your Next Project," *Geekflare*, vol. 1, no. 1, p. 1, 2019.

[30] K. Anderson, "2019 Open Source Database Report," DZone, Wisconsin, 2019.

[31] H. DuPreez, "Top 10 Databases for 2019," Database Journal, California, 2019.

[32] Google, "Firebase Development Platform," Google, 17 02 2020. [Online]. Available: https://firebase.google.com/. [Accessed 17 02 2020].

[33] IEEE, IEEE Guide for Developing System Requirements Specifications, New York: IEEE-SA Standards Board, 1998.

[34] K. Holtzblatt and H. Beyer, Contextual Design (Second Edition), England: Elsevier Inc, 2017.

[35] N. Young, "What is wireframing?," ExperienceUX, 26 02 2019. [Online]. Available: https://www.experienceux.co.uk/faqs/what-is-wireframing/. [Accessed 26 02 2020].

[36] Adobe, "AdobeXD," Adobe, 10 02 2020. [Online]. Available: https://www.adobe.com/uk/products/xd/prototyping-tool.html. [Accessed 26 02 2020].

[37] K. Wiegers, Software Requirements 2, UK: Microsoft Press, 2003.

[38] S. Paradkar, Mastering Non-Functional Requirements, India: Packt Publishing Ltd, 2017.

[39] M. Kleppman, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems, UK: O'Reilly Media, 2017.

[40] M. Zaleski, "UML Model Report," University of Toronto, Toronto, 2003.

[41] L. Brisolara and A. Parada, Automating mobile application development: UML-based code generation for Android and Windows Phone, Rio Grande do Sul: Universidade Federal de Pelotas, 2015.

[42] Nishadha, "UML Diagram Types | Learn About All 14 Types of UML Diagrams," Creately, 30 04 2019. [Online]. Available: https://creately.com/blog/diagrams/uml-diagram-types-examples/. [Accessed 04 02 2020].

[43] L. Bass, P. Clements and R. Kazman, "Chapter 1 - The Architecture Business Cycle," in *Software Architecture in Practice*, US, Pearson Education (US), 2003, p. 560.

[44] J. Highsmith, Agile Project Management, London: Pearson Education, 2004.

[45] I. Alsmadi, Software Engineering Ontology Separation of Concerns: Introduction,Software Processes,People, Tools, The Software Product, The Project, Software Metrics., Riga: VDM Verlag, 2009.

[46] L. Crispin, Agile Testing: A Practical Guide for Testers and Agile Teams, Boston: Addison-Wesley Professional, 2008.

[47] Google, "Developers - Espresso," 27 12 2019. [Online]. Available: https://developer.android.com/training/testing/espresso. [Accessed 17 04 2020].

[48] JUnit, "JUnit 4," 01 01 2020. [Online]. Available: https://junit.org/junit4/. [Accessed 17 04 2020].

[49] B. Beizer, Black-Box Testing: Techniques for Functional Testing of Software and Systems, New Jersey: Wiley, 2008.

8 Appendices

8.1 Appendix A: Primary Research - Survey Questions

Vehicle Management Application

I am a student in the Department of Computing, and as part of my project for Computing Projects, I am collecting data regarding vehicle management applications. The survey is designed to help me gather relevant information about individual views on vehicle management. I can use the data to answer my research questions. This survey will be open until 22/11/2019. Your participation is entirely optional and voluntary. Your identity will remain anonymous. All data will be destroyed after the project has completed. The whole survey is likely to take around 5 minutes to complete. Please take your time to read the questions carefully, and answer as truthfully as possible. Should you have any questions, please feel free to contact me: sali011@gold.ac.uk. Thank you very much for your time.

* Required

1. **Which age category do you fall under?**
   * *Mark only one oval.*
   - ◯ Under 18
   - ◯ 18-24
   - ◯ 25-30
   - ◯ Above 30
   - ◯ Prefer Not To Say

2. **What gender would you classify yourself?**
   * *Mark only one oval.*
   - ◯ Prefer Not To Say
   - ◯ Female
   - ◯ Male
   - ◯ Other: _____

3. **Do you work in a garage or any vehicle-servicing related industry?** * *Mark only one oval.*
   - ◯ Yes
   - ◯ No
   - ◯ Maybe

4. **Have you ever used/required a vehicle management application before?** * *Mark only one oval.*
   - ◯ Yes
   - ◯ No
   - ◯ Maybe
   - ◯ Prefer Not To Say

5. **If you answered 'Yes'/'Maybe' to the last question, which applications have you used (or currently use)? Else, select 'Have not used any'** *
Check all that apply.

- [ ] Have not used any
- [ ] aCar
- [ ] Car Maintenance Reminder Lite
- [ ] Other: _____

6. **If you have used a vehicle management application before, which aspect was the best in your opinion? (Select 'None' if you have never used a vehicle management application)** * *Mark only one oval.*

- ◯ None
- ◯ Interface (the actual app layout e.g. main menu, help page)
- ◯ Features (e.g. visualisation of background, multiple vehicle storage)
- ◯ Ability to be used on multiple devices (e.g. smartphone, tablet)
- ◯ Other: _____

7. **If a new vehicle management application was to be created, which feature would you like prioritised?** *
Mark only one oval.

- ◯ Prefer Not To Say
- ◯ Interface (the actual app layout e.g. main menu, help page)
- ◯ Features (e.g. visualisation of background, multiple vehicle storage)
- ◯ Ability to be used on multiple devices (e.g. smartphone, tablet)
- ◯ Other

8. **If you have never used a vehicle management application, please explain why not, in a few words (If you have just type 'I have'). Also, what would interest you in using one?** *

9. **Which logo is the most attractive to you?**
* *Mark only one oval.*

◯ Logo 1

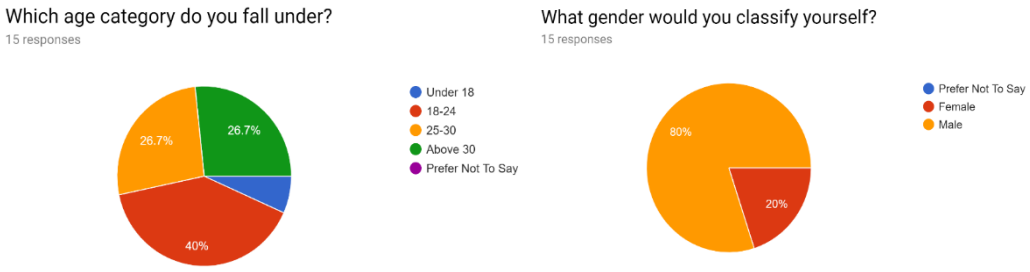
◯ Logo 2

◯ Logo 3

8.2 Appendix B: Primary Research - Survey Analysis

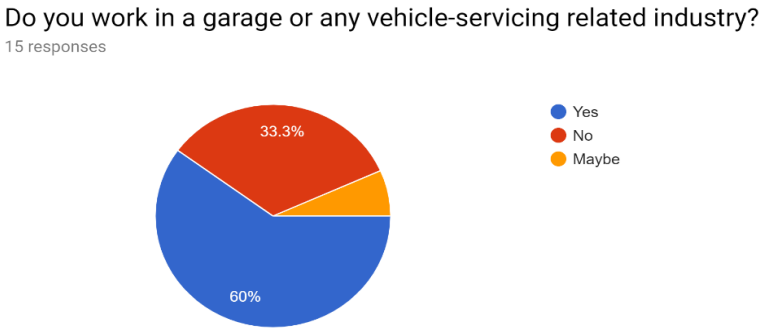
To get a better understanding of the vehicle management applications' market, primary market research was conducted. This was in the form of a nine-question survey in Google Forms. It received 15 responses in total. This document aims to outline the key findings from the survey.

The initial questions were quite generic; based on age and gender. They were required to gain insight into the demographic. The first pie chart below, clearly displays that the largest age category, which took part in this survey, were aged 18-24 (6 people out of 15). The age '25-30' and 'Above 30' will also be considered, as they make up a combined 53.4% of the total sample. The 'Under 18' category will be disregarded, due to the lack of data, which may lead to unrepresentative outcomes. The second pie chart informs us that 80% of the surveyors were male (12 out of 15 people). Therefore, we can conclude that the target audience is predominantly males, over the age of 18.



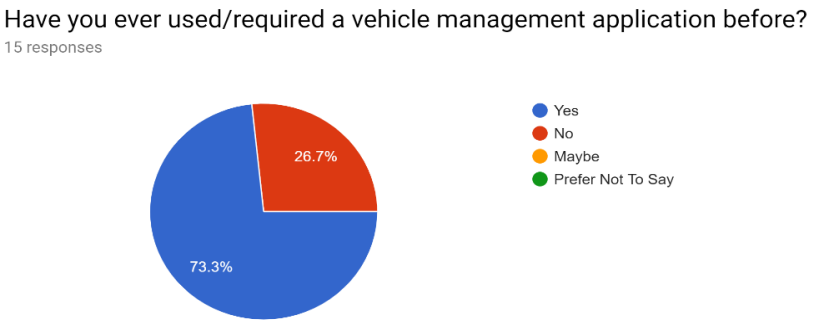
Age Category and Gender - Pie Charts

The next question was asking whether the surveyor works in a garage or any other vehicle-servicing-related industry. The pie chart below displays that 60% of the sample did work in a vehicle-servicing industry. Thus, the answers provided by these individuals will be significant concerning the vehicle management application.



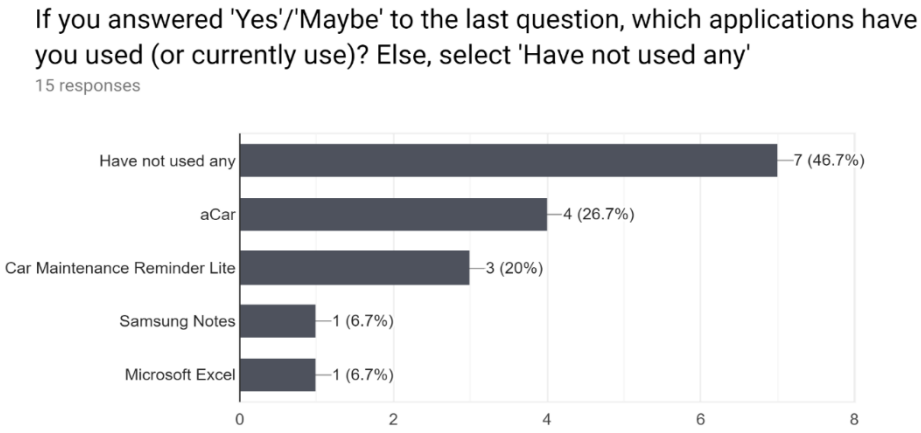
Work in garage or vehicle-servicing industry - Pie Charts

The next set of questions was about whether the surveyor has ever needed or used a vehicle management application, which apps they have used, and what their favourite features on those apps were. The graph below depicts that 73.3% (11 people) have used an application to aid in their vehicle management. As a result, we can infer that they are familiar with what these types of apps encompass, and what is required of them.



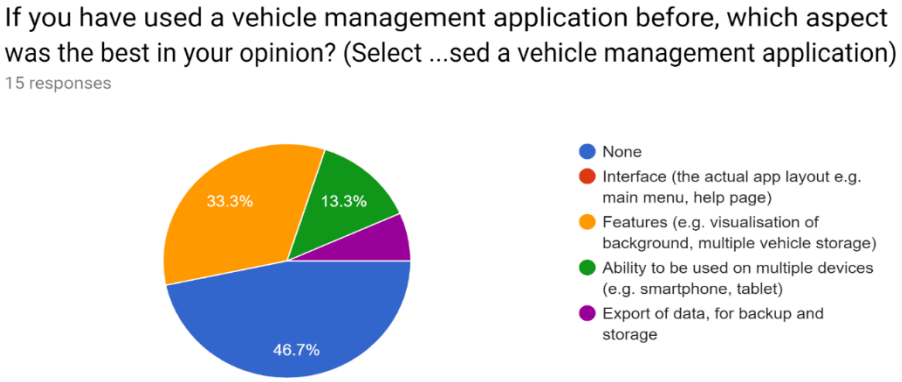
Used vehicle management applications previously - Pie Chart

The following bar chart displays the various applications used. The most used was ‘aCar’, an Android application. Surprisingly, there was software, which is not vehicle management specific. This included ‘Samsung Notes’ and ‘Microsoft Excel’. Consequently, the use of these programs may have been used due to their data inputting features.



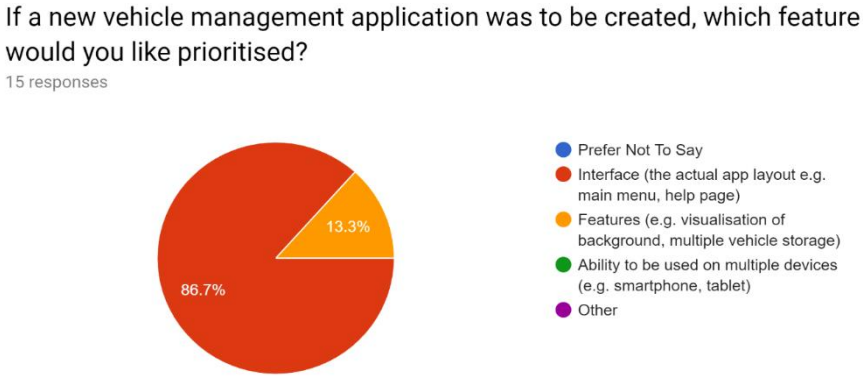
Applications/Software used for vehicle management - Bar Chart

Next, the diagram below portrays which components of the software, individuals who used them, found the best. A third of the total survey (33.3%) found the features of the applications e.g. data visualisation, to be the best quality. Therefore, utilising this information, when building the application, it must incorporate features which users will find simple to use, yet effective.



Best feature on used applications/software - Pie Chart

After, the following question was regarding what users would like prioritised if a new application were to be made. The pie chart demonstrates that the majority of users (13 out of 15) preferred the interface to be the priority.



Feature to be prioritised - Pie Chart

My penultimate question asks why an individual has not used a vehicle management application before, and what would interest them in using one. The results primarily contained answers such as: “I did not require one” or “I never thought of using one”. As a result, I believe my applications’ target audience becomes clearer (owners of multiple vehicles) following these answers. As well as this, I know that I need to create a platform that makes it easy for everyone to utilise.

Finally, my last question was about which logo was most attractive. There were three options. The majority chose ‘Logo 2’ (60%).

Logo 1 Logo 2 Logo 3

Logo options

Overall, most ultimately, the survey and survey analysis have provided me the opportunity to develop my acumen, regarding vehicle management and its audience. I understand that I must create software that has a very low barrier to entry (intuitive and simple) as well as efficient for high-level, vehicle data storage, and management.

8.3 Appendix C: Advantages of Android Studio

Android Studio has been the selected developing environment for programming the 'Motor Manager' application. It encompasses many tools that will make the process of development much more efficient. This document aims to outline the advantages of Android Studio, and why it was chosen specifically for this project.

First and foremost, Android Studio is free to download and use. The availability of the Android SDK (Software Development Kit) is widespread and free online. As a result, there are no costs incurred in utilising the IDE. Also, there are no in-software costs. Therefore, projects can be developed whenever the need arises, without any start-up costs.

Also, Android Studio makes uses of the Java programming language. Java is an OOP (Object Oriented Programming) language, which is a common language used in modern times. This means that the language used within Android Studio is well-known (especially to me). As a result, online support resources are abundant. For instance, there are many online video tutorials, as well as blog posts, regarding common Android Studio-related issues.

Furthermore, the recent updates to Android Studio have rendered the 'Layout Editor' more intuitive. Changes have made management and interaction of the design of an application easier. Also, there is an option in the 'Layout Editor' to manually position elements; code is dynamically altered in the XML file. Android Studio allows the preview of layout XML's side-by-side with the code. Consequently, this improves workflow.

A huge benefit of Android Studio for my Motor Manager application is the helpful VCS (Version Control System) integration. The main feature of VCS which will aid in the development process of my project is Github. The built-in VCS interface makes using git features simple and quick. The code can be committed, pushed, and pulled. There is also a helpful static code check when runs at the time of committing files. Warning and error messages appear if bugs are present, and a review option is made available. This is effective when it comes to error detection and resolution.

Additionally, similar to many IDE's and code editors, Android Studio has a built-in typo-check in real-time. For instance, if a user misspells a common method's name e.g. Toast();. Android Studio will highlight the area and provide suggestions. Furthermore, once a variable or method is declared, Android Studio will auto-fill the user's code if characters match. This improves the efficiency of the development process.

Lastly, alongside Android Studio's auto-completing features, it also can generate commonly implemented activities. For example, you can generate 'Login' activity, with the base code provided. Consequently, this boosts productivity and aids time efficiency.

8.4 Appendix D: Advantages of Firebase

Motor Manager will make use of Google's Firebase development platform and its various products. Firebase (created by Firebase Inc, now acquired by Google) is a mobile and web application development platform. It has 18 products and is used by 1.5million applications worldwide. The Firebase console provides the user with a friendly interface to access and edit firebase properties.
The Firebase development platform products I aim to use for Motor Manager include:

- **Authentication** (to track users and their details e.g. email address)
- **Realtime Database** (to store vehicle details in a JSON format)
- **Storage** (to store images, such as user profile images or vehicle images)
- **Performance** (monitor user experience regarding their networks and waiting times e.g. application start-up time)
- **Test Lab** (automatic testing environment for applications e.g. Robo test and XCTest)

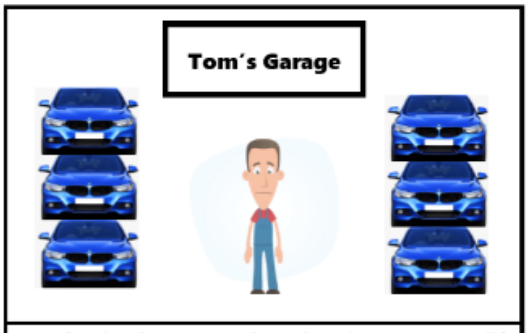
There are many advantages to the selected Firebase products. They will be integral for the functionality of Motor Manager.

Advantages of Firebase:

- Access and use of the Firebase Development Platform are currently free of charge (a Google account is required).
- No download is required. Firebase is available online, via web browsers.
- Firebase has a user-friendly console. The console displays various sections such as 'Authentication', 'Database' (Realtime or Firestore), 'Storage', 'Test Lab', and many more. This allows the user to monitor all sections of their application efficiently.
- Firebase backend has SDK's (Software Development Kit) that are available to use for authentication within applications.
- Firebase has two databases: Realtime and Firestore. Firebase Realtime Database is being used for Motor Manager. The Firebase Realtime Database is cloud-hosted. Data is stored in a JSON format and synchronised in real-time to all connected clients. The advantage this brings is that data being stored does not require a strict schema, as any data, of any type, can be stored. Also, the database is updated in real-time, so we can check whether the application is sending and retrieving data accurately. Rules can be applied to the database also. The maximum of 1GB can be stored (on the free service).
- The storage section makes use of Google's Cloud Storage. As a result, users can download files they have stored. The maximum of 5GB can be stored (on the free service).
- Firebase also has automatic testing services. This occurs in the 'Test Lab'. Robo Tests and test XCTest. As a result, once an applications' APK is uploaded, a user can test if the various pages of the application's work accordingly or not. Also, single methods can be tested, check whether appropriate outputs are occurring.

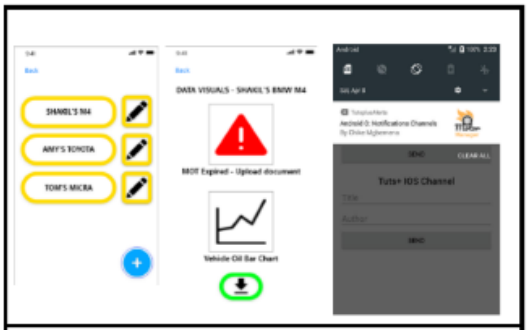
Lastly, Firebase is very scalable. Firebase can handle 100,000 users at a time, and over many regions.

8.5 Appendix E: Storyboard



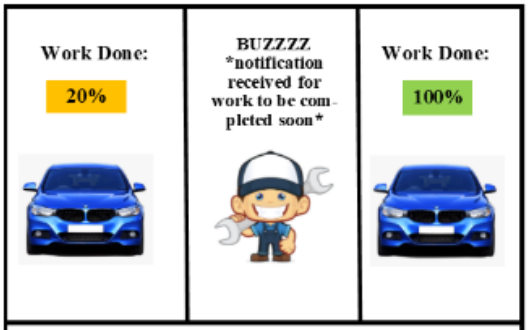
"(*Sigh*) Ah, I have so much work to do on so many different vehicles. I wish there was an efficient way of managing all their details and client requirements!"

"Eureka! 'Motor Manager' can solve my problems. It will let me store and monitor multiple vehicle data. Let me look at the features and download it right away!"



"Wow, look at the features Motor Manager has. It allows me to view stored vehicle details, visualise data ,such as oil levels, and receive reminders about work which is due!"

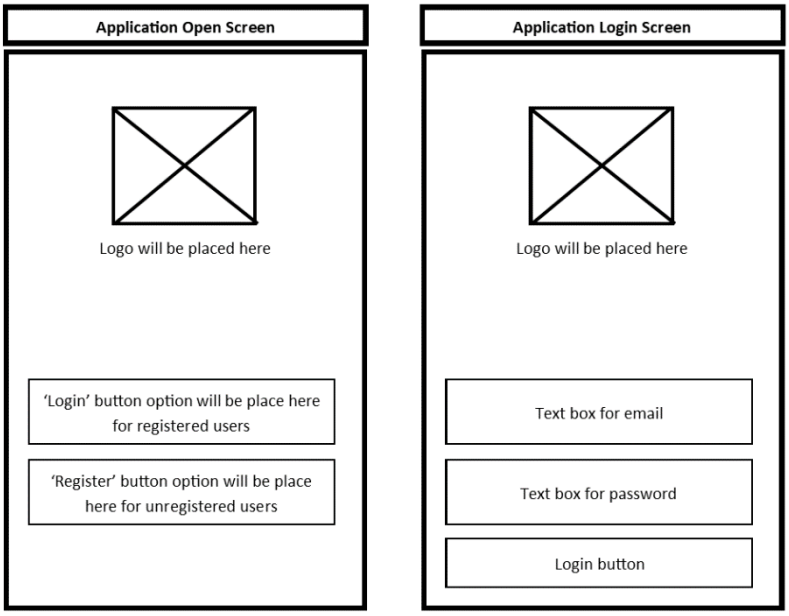
~ MOTOR MANAGER BEING DOWNLOADED AND INSTALLED FROM THE GOOGLE PLAYSTORE~

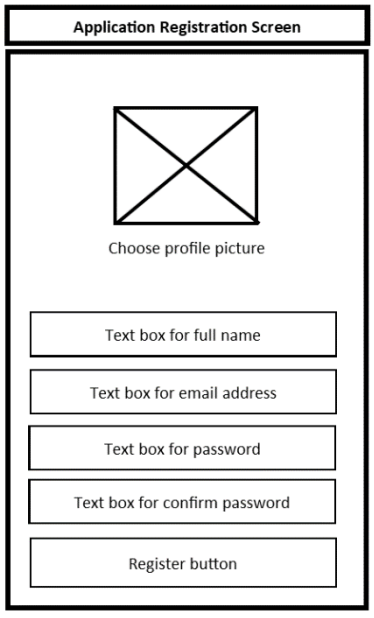


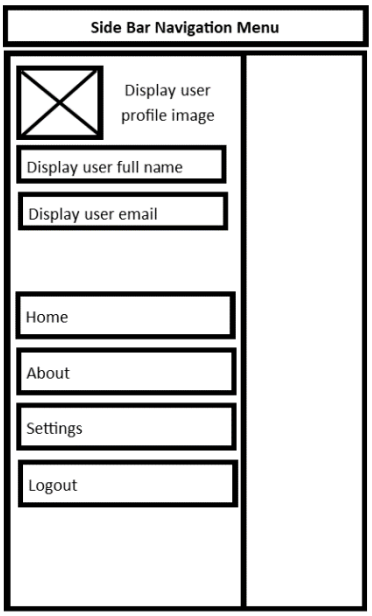
"Wow. I am so happy now. I am able to keep track of all vehicles I am working on. I also received these automatic notifications, reminding me when deadlines are due!"

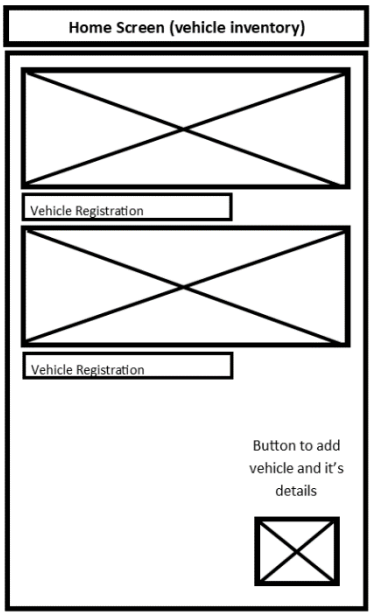
"YES! This application has saved my life. All the vehicles in my garage will now be monitored effectively. I will never fall behind in my work again!"

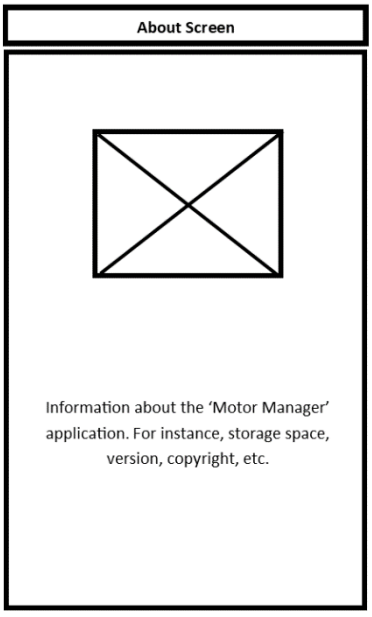
8.6 Appendix F: Wireframes

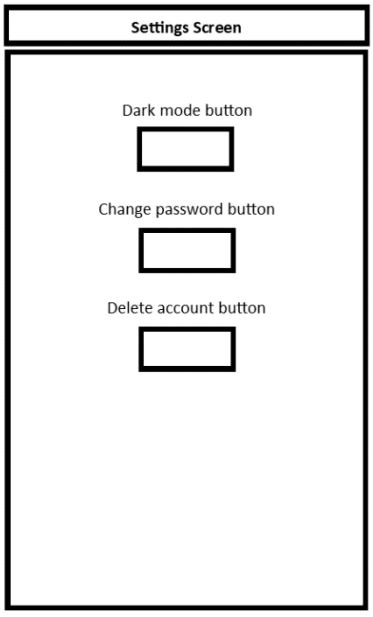


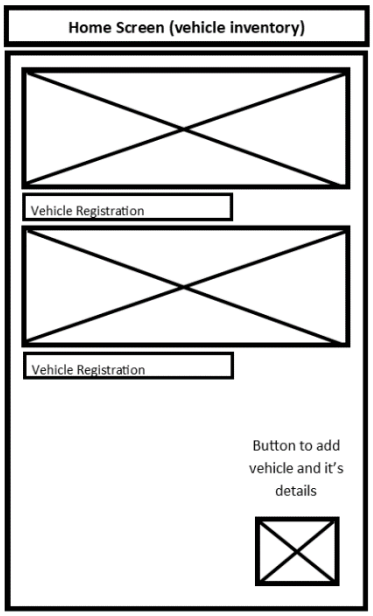


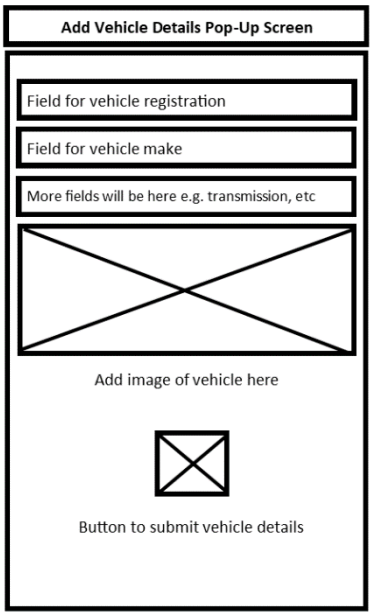

Side Bar Navigation Menu
Display user profile image
Display user full name
Display user email
Home
About
Settings
Logout

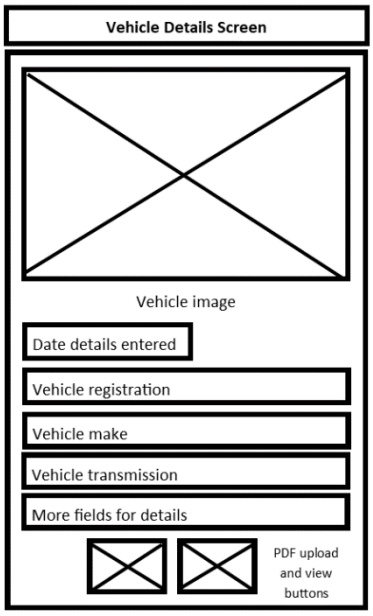

Home Screen (vehicle inventory)
Vehicle Registration
Vehicle Registration
Button to add vehicle and it's details


About Screen
Information about the 'Motor Manager' application. For instance, storage space, version, copyright, etc.


Settings Screen
Dark mode button
Change password button
Delete account button


Home Screen (vehicle inventory)
Vehicle Registration
Vehicle Registration
Button to add vehicle and it's details


Add Vehicle Details Pop-Up Screen
Field for vehicle registration
Field for vehicle make
More fields will be here e.g. transmission, etc
Add image of vehicle here
Button to submit vehicle details


Vehicle Details Screen
Vehicle image
Date details entered
Vehicle registration
Vehicle make
Vehicle transmission
More fields for details
PDF upload and view buttons

8.7 Appendix G: Prototype Survey

Motor Manager - Prototype Survey

I am a student in the Department of Computing, and as part of my project for Computing Projects, I am collecting data regarding my vehicle management application prototype. The survey is designed to help me gather relevant information about individual views on my prototype. I can use the data to answer my research questions. This survey will be open until 3/12/2019. Your participation is entirely optional and voluntary. Your identity will remain anonymous. All data will be destroyed after the project has completed. The whole survey is likely to take around 5 minutes to complete. Please take your time to read the questions carefully, and answer as truthfully as possible. Should you have any questions, please feel free to contact me: sali011@gold.ac.uk. Thank you very much for your time.

* Required

1. **Which age category do you fall under?** *
   Mark only one oval.
   - ◯ Under 18
   - ◯ 18-24
   - ◯ 25-30
   - ◯ Above 30
   - ◯ Prefer Not To Say

2. **What gender would you classify yourself?** *
   Mark only one oval.
   - ◯ Prefer not to say
   - ◯ Female
   - ◯ Male
   - ◯ Other: _____

3. **How attractive is the application?** *
   Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Very

4. **How easy-to-use (user-friendly) is the application?** *
   Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Very

5. **How intuitive was the application?** *
   Mark only one oval.

	1	2	3	4	5	
Not at all	◯	◯	◯	◯	◯	Very

6. **Did buttons function as required (e.g. 'Login' take you to the main menu)?** *
Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ Maybe

7. **What was the best feature of the prototype?** *
Mark only one oval.

- ◯ Design
- ◯ Features
- ◯ Structure (e.g. Login or registration screen leads to main menu, then leads to vehicle inventory etc)
- ◯ Other

8. **What could be improved?** *
Check all that apply.

- ☐ Design
- ☐ Features
- ☐ Structure
- ☐ Other

9. **Any additional comments? (Type 'No' if you have no further comments to add)** *

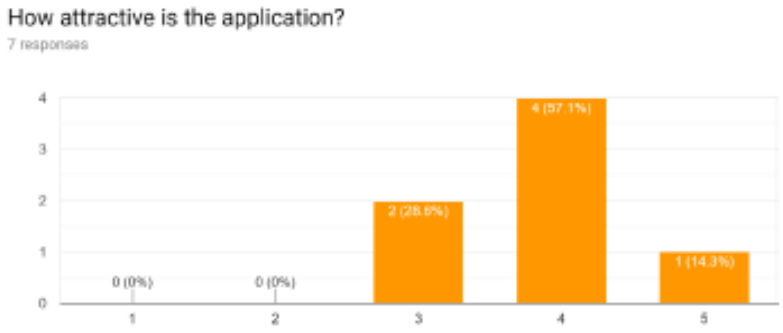
8.8 Appendix H: Prototype Survey - Analysis

In order to get an initial conceptualisation of my application, I designed a prototype. The prototype was designed in Adobe XD. Next, I designed a survey to gain target audience views. The results and analysis are below, and they will be incorporated when creating the final prototype. Total responses: 7. The surveyors consisted of the same individuals from the initial survey (mainly garage and multiple-vehicle owners). In addition, extreme users e.g. individuals who had never used any data storage systems previously, also partook. The views of these individuals will aid in determining factors which can persuade them to become users.

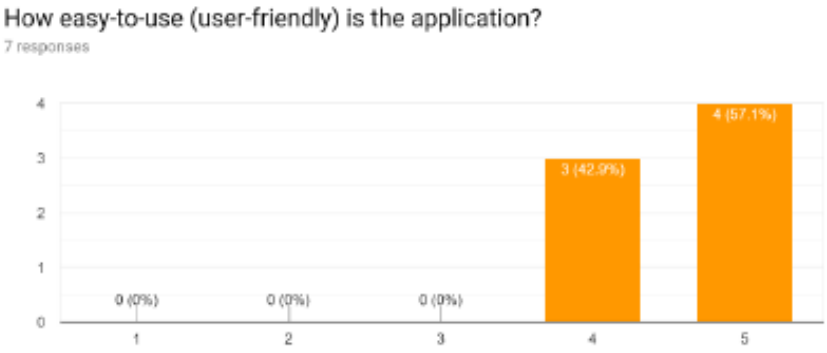
The survey begins with very basic, general questions, regarding age and gender. This was so as to collect background data. As we can see, the highest age group in this survey is 'Above 30' (3), and the most common gender is 'Male' (6). Consequently, we can verify that the demographic consists of mainly males, which are 18 and above, for this application.

Age Category and Gender (Pie Charts)

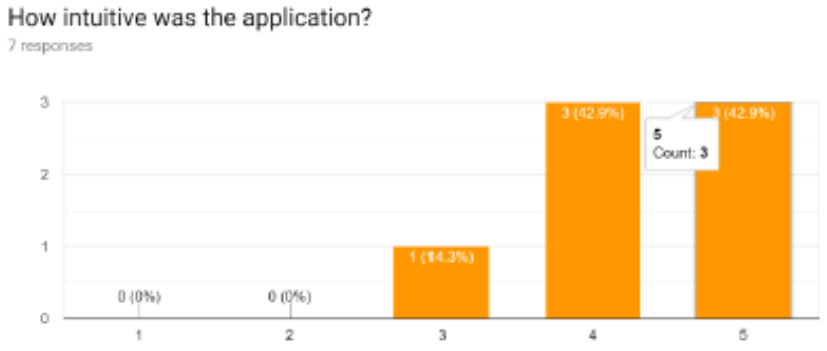
Following these questions, there were three scale-questions. Surveyors were asked to rate on a scale of 1-5 ('Not at all' to 'Very') the attractiveness, user-friendliness, and intuitiveness. As shown by the bar charts below, the majority thought positively about the prototype. There were two responses of '3' (neutral) on attractiveness and one response of '3' (neutral) on intuitiveness. As a result, this means I need to improve how the application appears, as well as how easy-to-understand it is at initial use.



Prototype Attractiveness (Bar Chart)

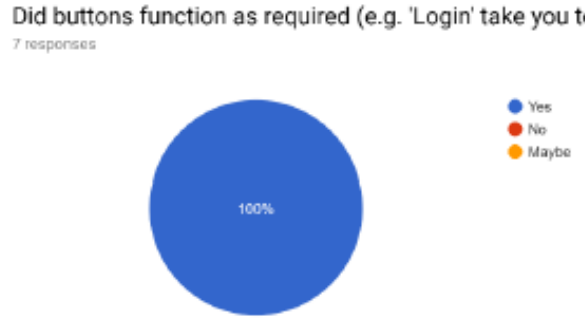


User Friendliness (Bar Chart)



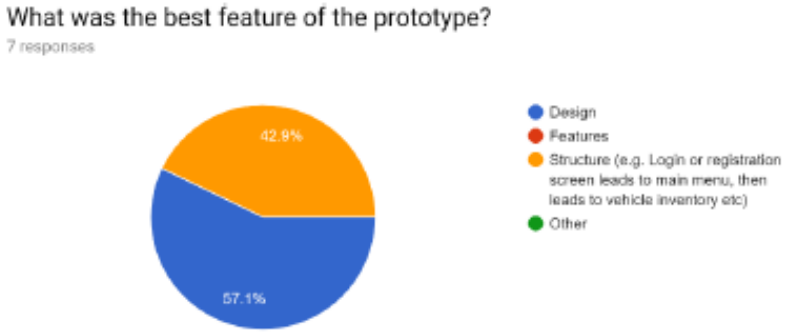
Intuitiveness (Bar Chart)

Next, this question asked users about the button functionality of the prototype. It is clearly depicted in the pie chart below, that all surveyors found the buttons to be operative. Consequently, I need to ensure my final prototype maintains this.

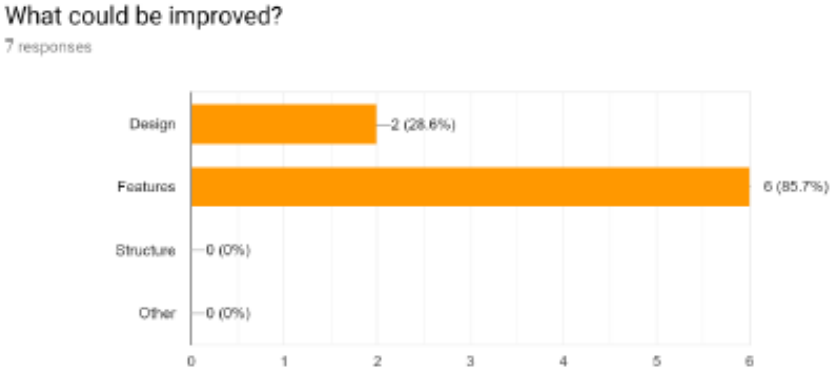


Button Functionality (Pie Chart)

Furthermore, the next two question's were in regard to the best feature of the prototype and what could be improved. The majority (4 people) confirmed 'Design' to be the best feature, with 'Structure' chosen by the remaining 3 individuals. 'Features' was voted as the component which needs to be improved (6 people). This leads me to believe, I need to ensure I keep design consistent, whilst improving the features available. However, a note to be taken is that this prototype is resembling the MVP (Minimum Viable Product) for Motor Manager, thus, it's simplistic nature.



Best feature of the prototype (Pie Chart)



Component to be improved (Bar Chart)

Lastly, the final question was deployed to retrieve qualitative data. Individuals had the option of leaving any additional comments they had. The bulk of responses mentioned including more features or improve intuitiveness on buttons. Therefore, from this, I understand that additional features and improved buttons need to be integrated.

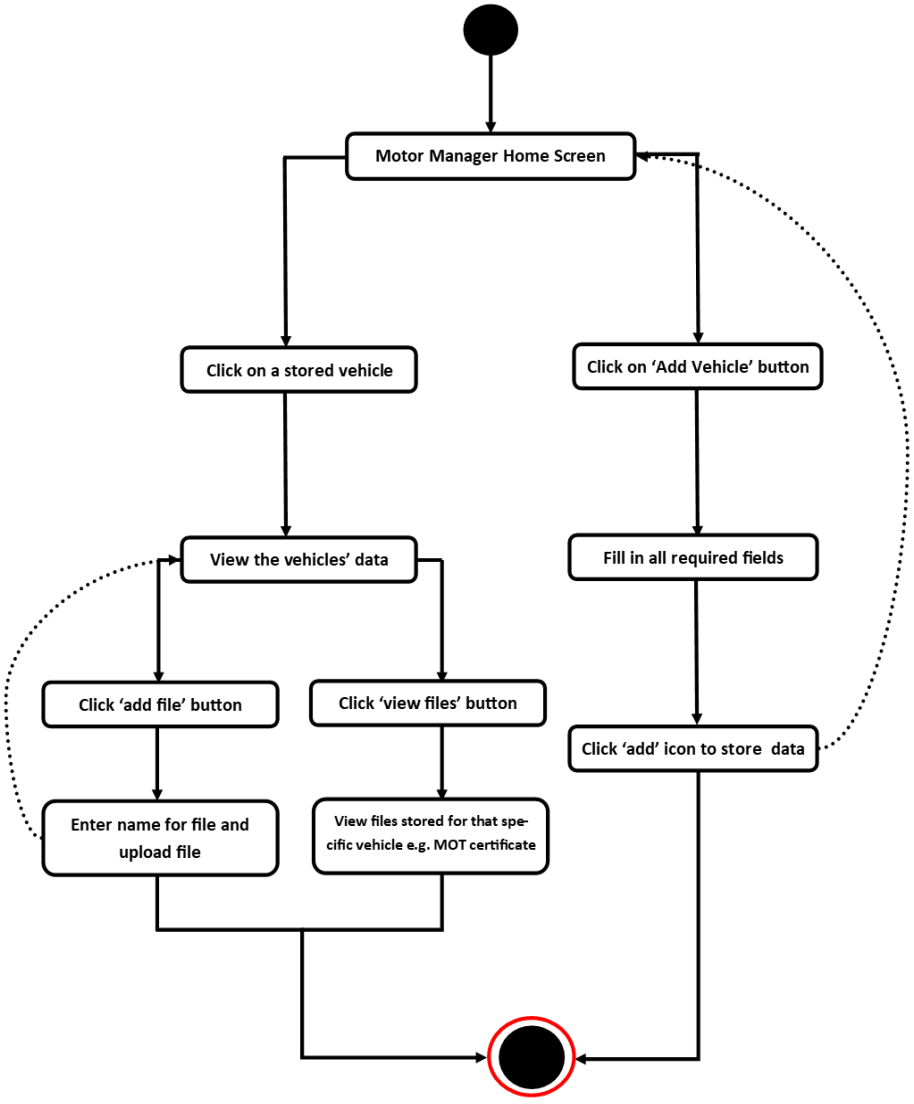
Overall, most ultimately, I believe that this survey has allowed me to get my target audience's perspective. In my final prototype, I will work on accommodating views such as additional features and increased intuitiveness.

8.9 Appendix I: Functional and Non-Functional Requirements (Post-MVP)

MVP Functional Requirements	
ID	**Requirement**
001	Application must be an app on a device (logo and name accurately applied)
002	Application must have a login option
003	Application must store and retrieve user login details in Firebase
004	Application must have a registration option
005	Application must store new user details e.g. full name, email, password, profile image.
006	Application must have the main menu when a user has logged in
007	Application must display the user's full name on login
008	Application must display the user's email address on login
009	Application must display the user's profile picture on login
010	Application must integrate with Firebase Realtime Database and store vehicle data
011	Application must allow users to traverse different pages
012	Application must allow the user to store multiple vehicles and their details e.g. vehicle image, registration number, description of work to be conducted.
013	The application must allow the user to click on a saved vehicle, and view its stored details
014	The application must allow the user to logout successfully
Post-MVP Functional Requirements	
015	Application must allow users to input text to fields using their voice
016	Application must allow users to attach a file to a vehicle
017	Application must allow users to view any files attached to a vehicle
018	Application must allow users to download any files attached to vehicles
019	Application must have a dark mode
020	Application must allow the user to delete stored vehicle data

MVP Non-functional Requirements	
ID	**Requirement**
001	Application must launch as soon as the user opens it
002	Application must inform the user of incomplete fields
003	Application must be able to handle 100 users at a time
004	Application must be portable (can download on any Android device)
005	The application must have the same user experience on all compatible devices
006	Application must allow storage of at least three vehicles
007	Application must ensure data is secure
Post-MVP Non-functional Requirements	
008	Application must display voice-inputted text on a preview screen
009	Application must allow the user to name files attached to vehicles
010	Application must allow the attachment of at least three files for a vehicle

8.10 Appendix J: Activity Diagram - Vehicle Details Addition & Viewing



8.11 Appendix K: Class Diagrams

Figure K.1: Class Diagram - Home

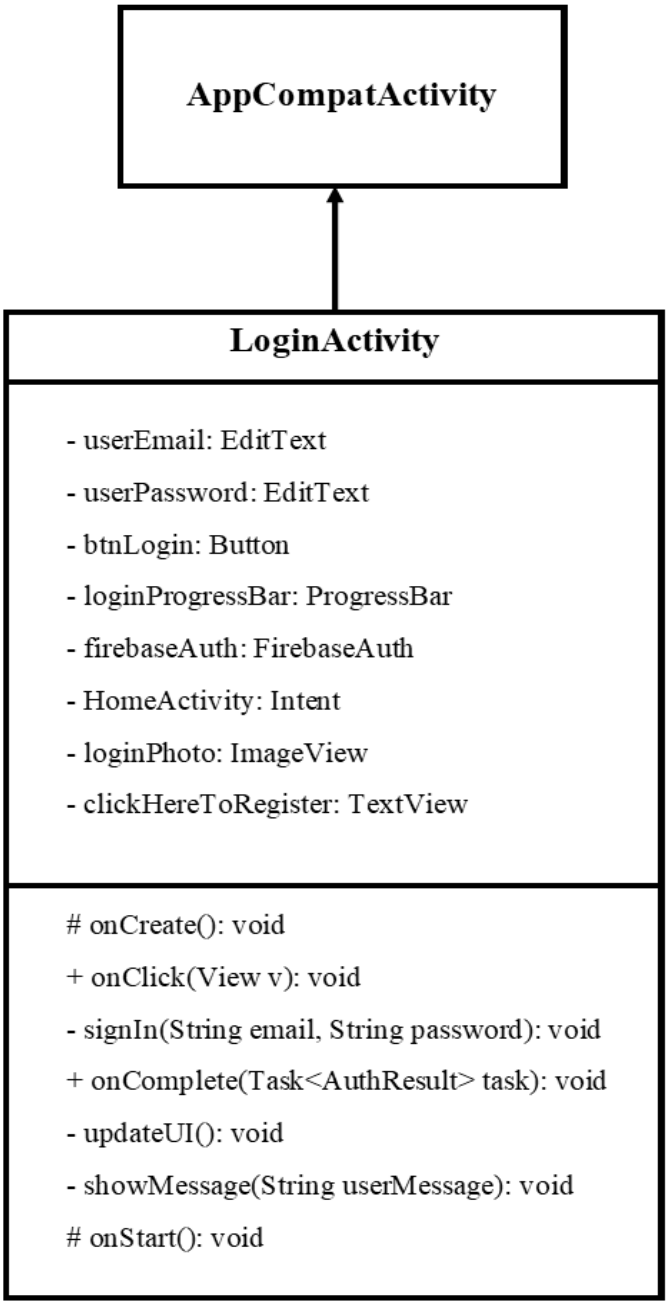


Figure K.2: Class Diagram - Login

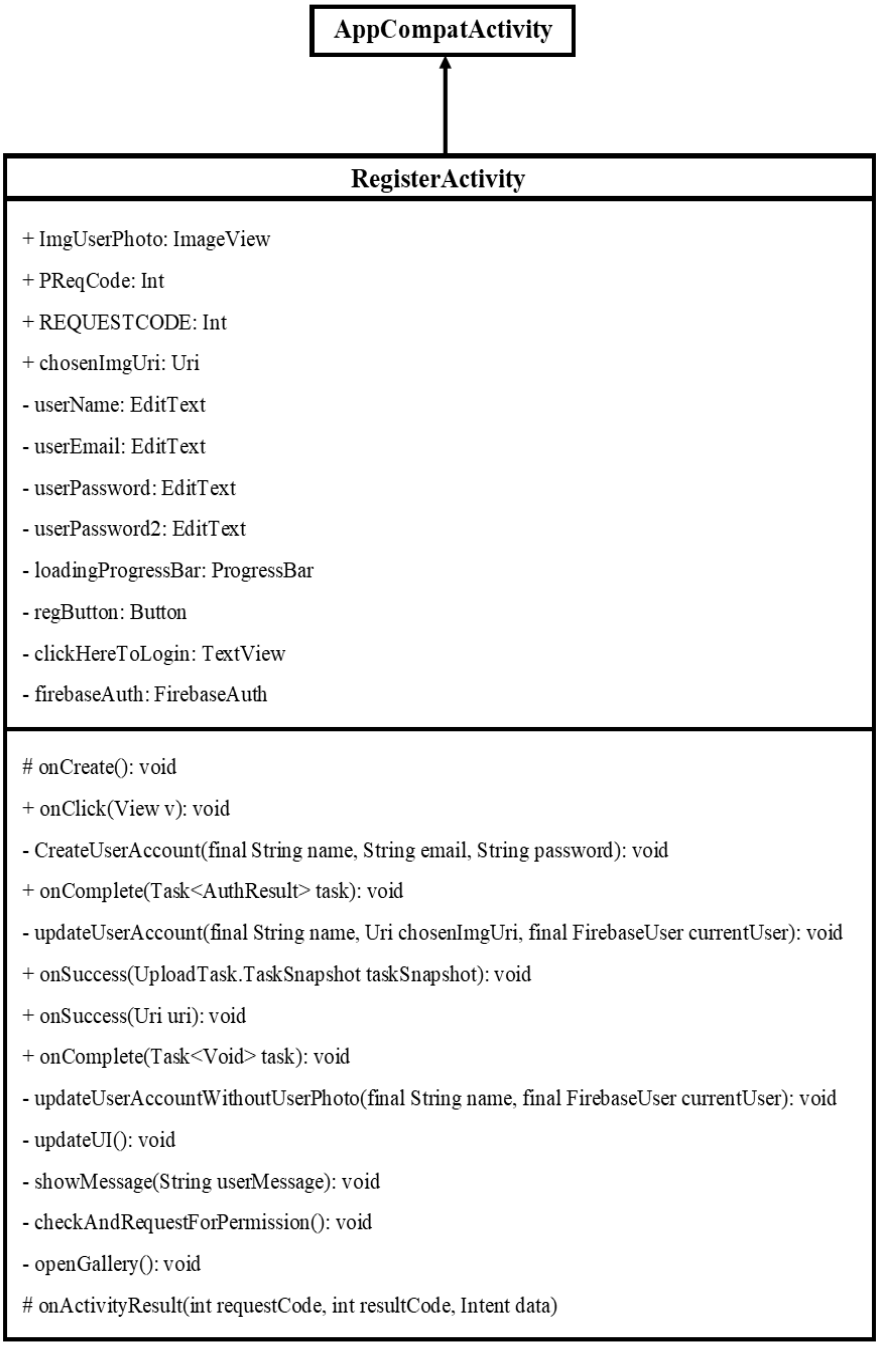


Figure K.3: Class Diagram - Register

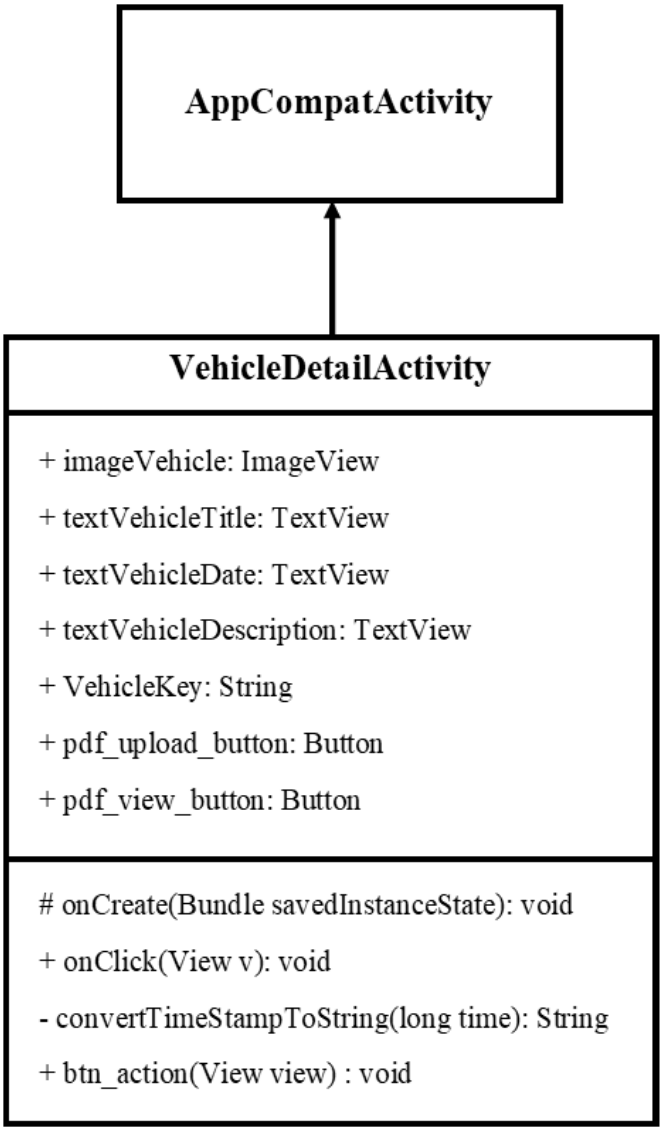


Figure K.4: Class Diagram - Vehicle Details

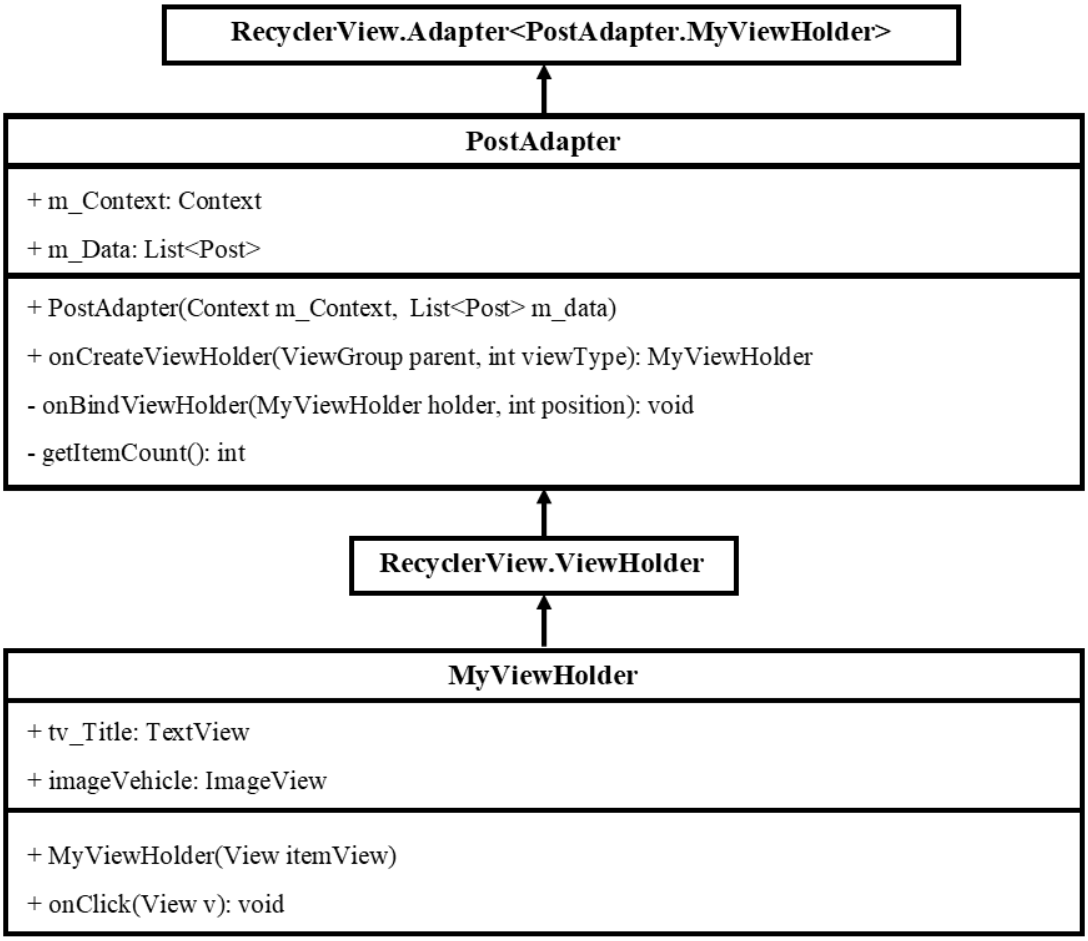


Figure K.5: Class Diagram - PostAdapter

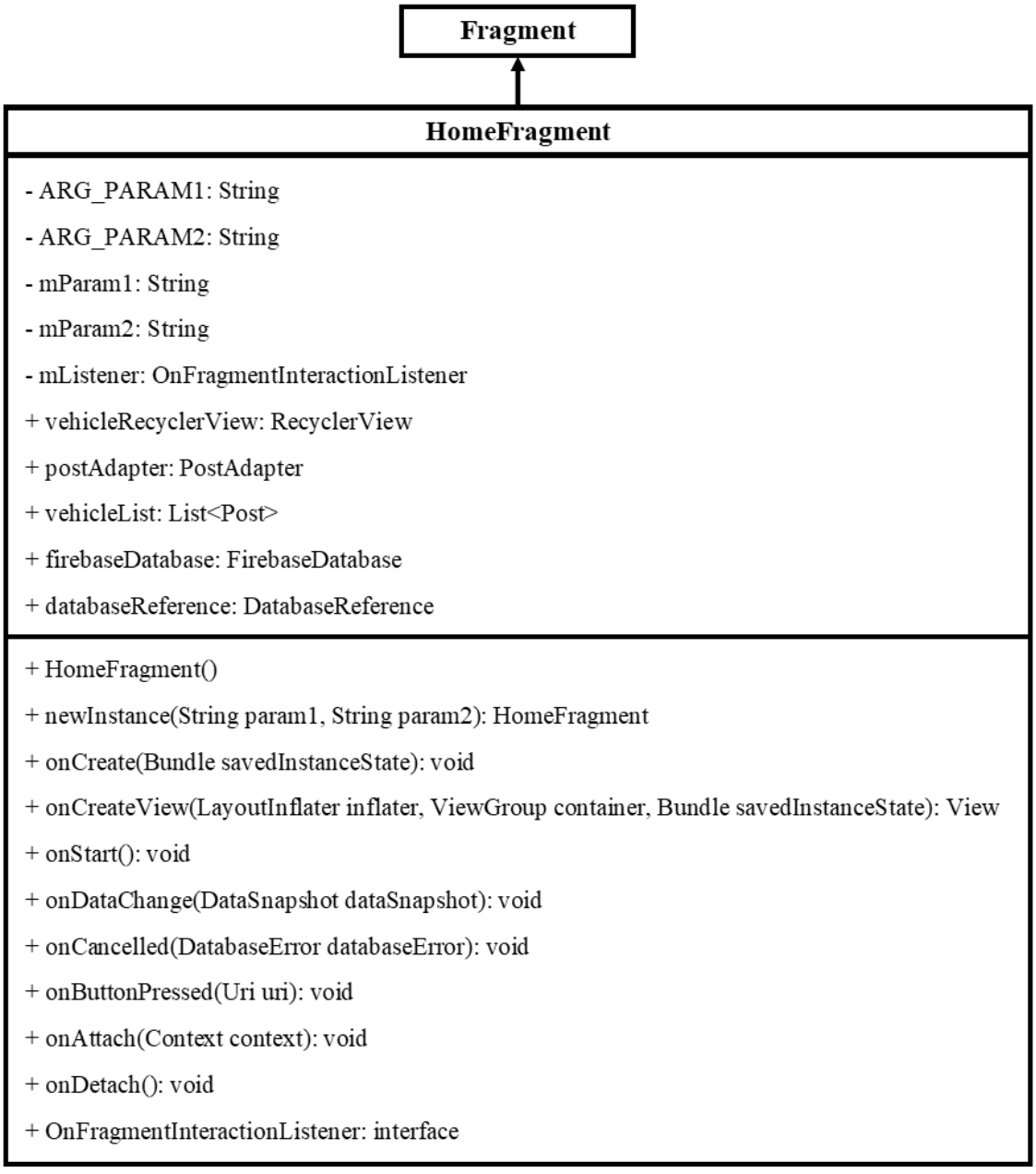


Figure K.6: Class Diagram - HomeFragment

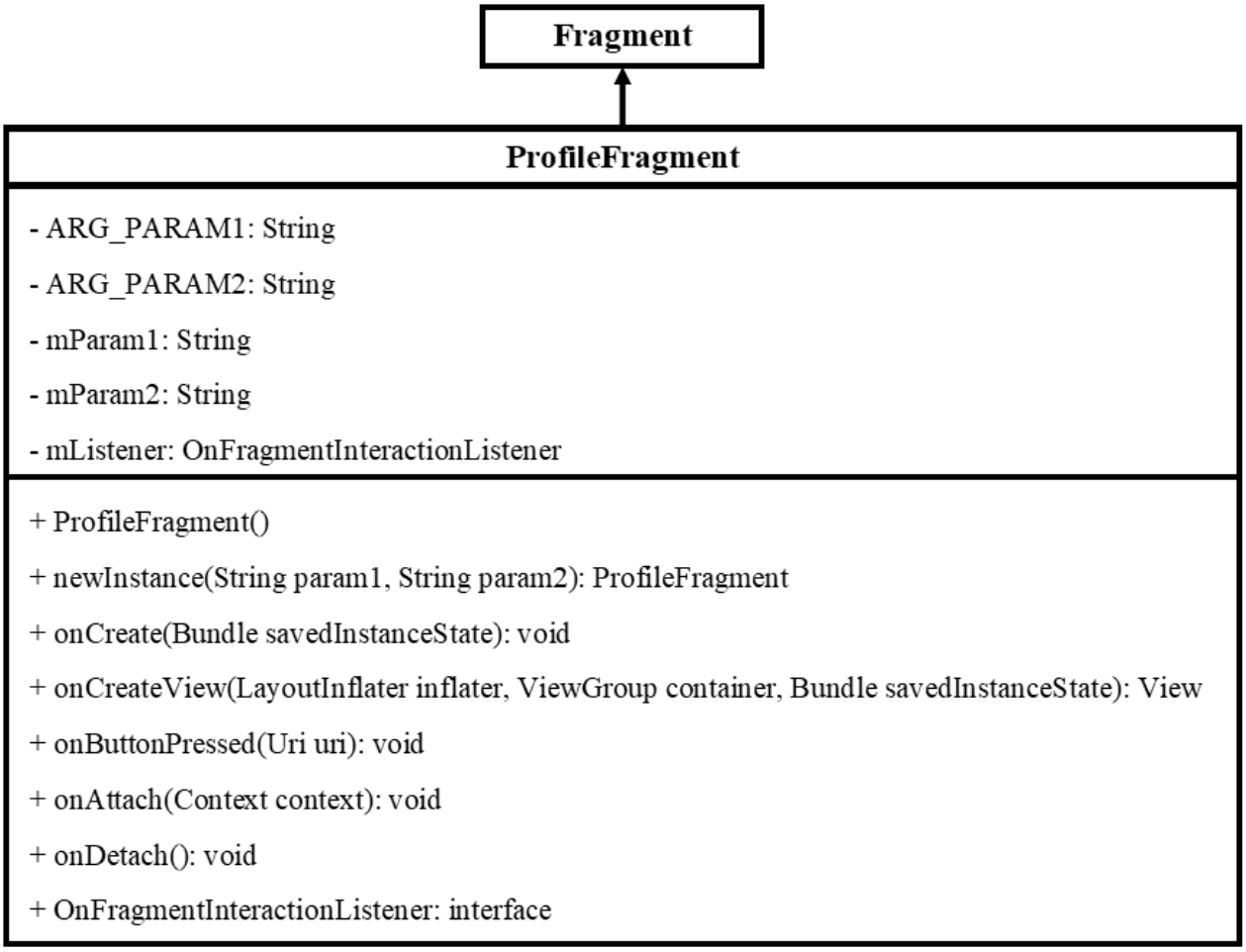


Figure K.7: Class Diagram - ProfileFragment

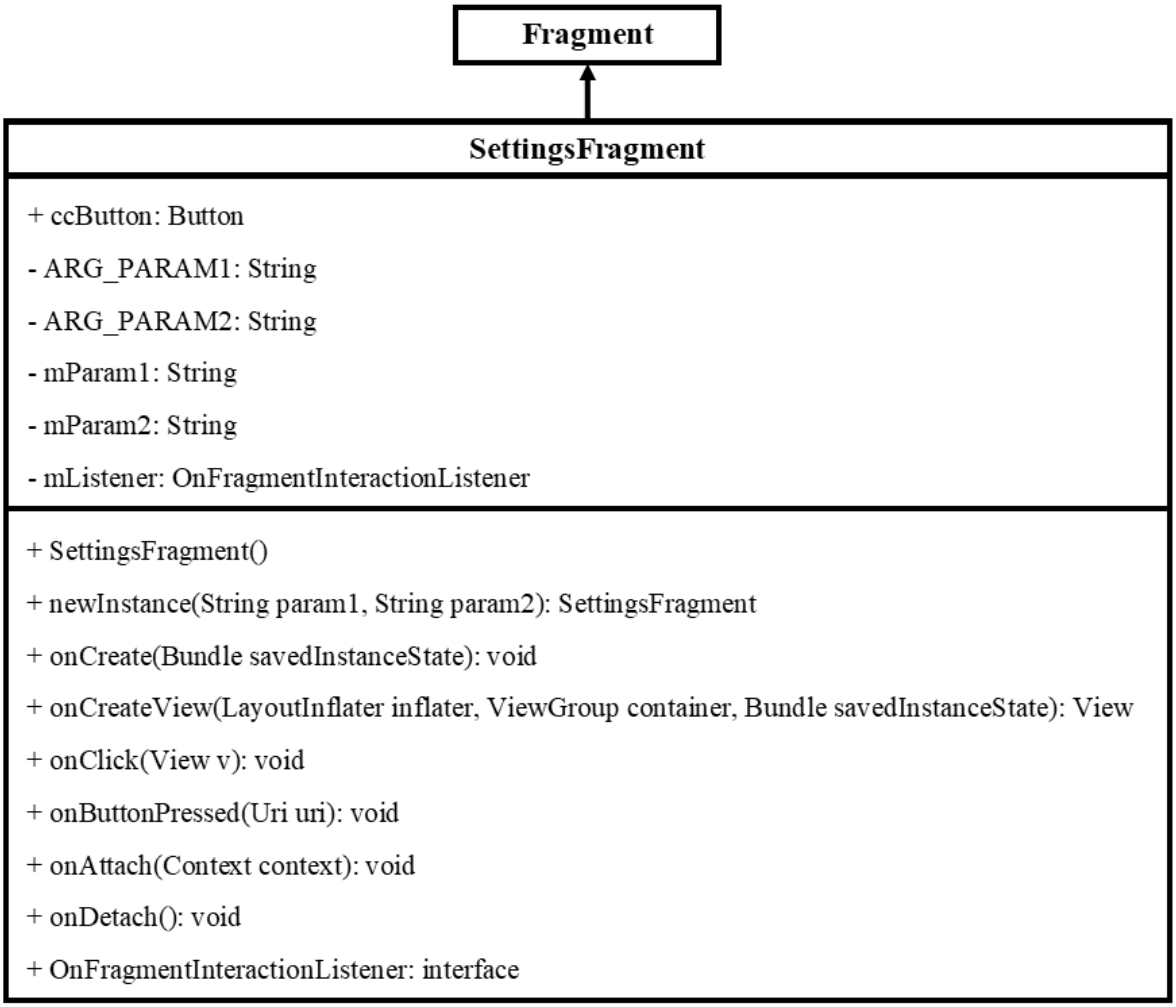


Figure K.8: Class Diagram - SettingsFragment

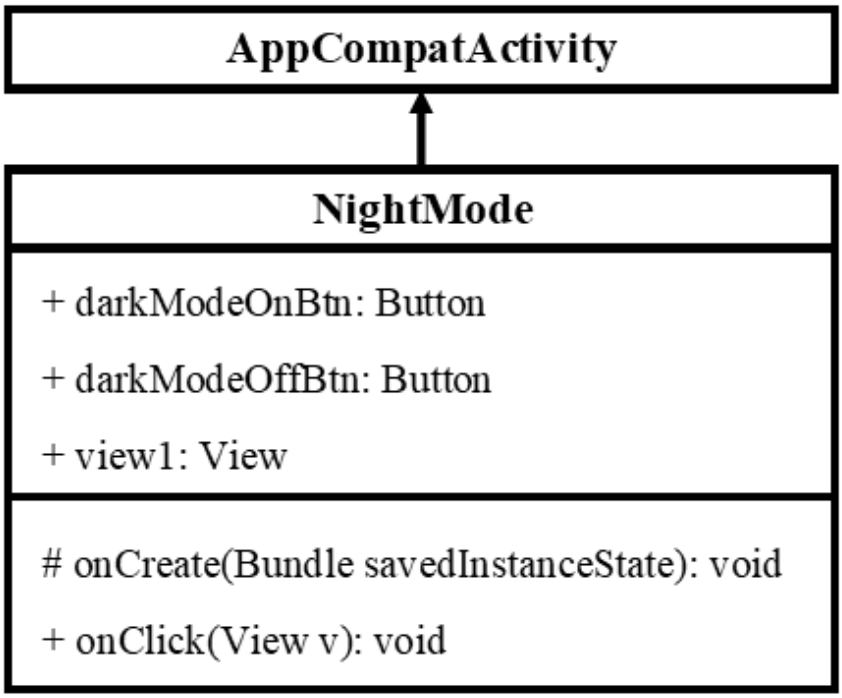
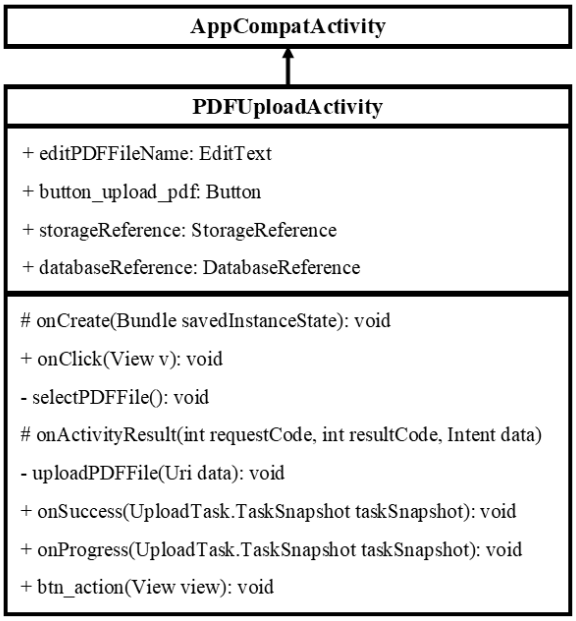


Figure K.9: Class Diagram - NightMode



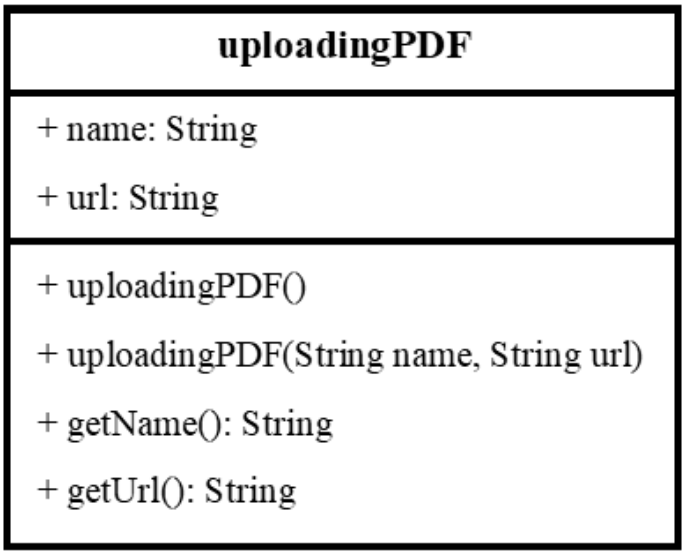


Figure K.10: Class Diagram - PDFUploadActivity and uploadingPDF

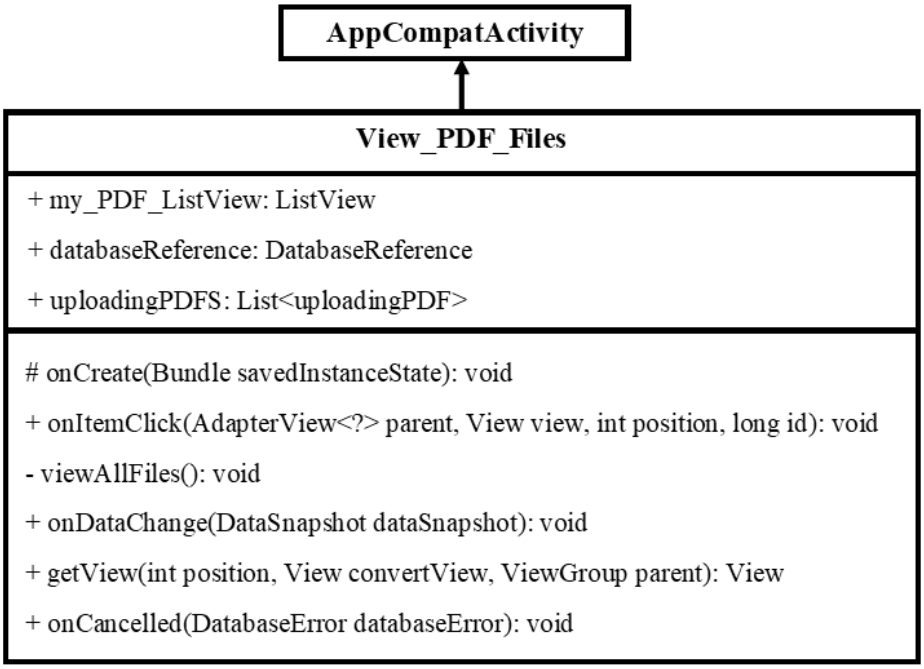


Figure K.11: Class Diagram - View PDF Files

Figure K.12: Class Diagram - Post

8.12 Appendix L: Implementation (Post-MVP Classes)

Helpers

Helpers encompassed classes for the post-MVP-functionalities e.g. dark mode, file upload. Each class contained different methods for the functionalities.

NightMode.java
This class has the program code for the dark (night) mode in *Motor Manager*. This mode can be assessed on the *settings* page of the application. It changes the white and orange theme to black and orange. The imports include two buttons and a view element. The buttons are respectively for 'On' and 'Off', and the *view* element stores the current window display. The main method initialises these variables as seen in the code extract below. There are on-click listeners on each button. If the 'On' button is clicked, dark mode is switched on, and vice versa for the 'Off' button.

```
// Variable Assignments
darkModeOnBtn = findViewById(R.id.darkModeOnButton);
darkModeOffBtn = findViewById(R.id.darkModeOffButton);

view1 = this.getWindow().getDecorView();


// On click listener for dark mode on
darkModeOnBtn.setOnClickListener((v) → {
        // Set background colour to black
          view1.setBackgroundResource(R.color.black);
        AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_YES);

    // end of on click method
});

// On click listener for dark mode off
darkModeOffBtn.setOnClickListener((v) → {
        // Set background colour to white
          view1.setBackgroundResource(R.color.white);
        AppCompatDelegate.setDefaultNightMode(AppCompatDelegate.MODE_NIGHT_NO);

        // end of on click method
});
```

PDFUploadActivity.java
The *PDFUploadActivity* class is responsible for the file uploading functionality in *Motor Manager*. This function can be utilised when on the individual vehicle detail page to attach files (e.g. MOT certificate) to the vehicle. Imports include UI elements (e.g. buttons, progress bar, edit text, toast, view). There are also *Firebase* imports (storage references and databases).

The code extract below displays the variable initialisations and assignment operations that are carried out for the edit text, button, and *Firebase* (storage and database references). There is also an on-click listener on the PDF upload button, to check when a user has clicked it.

PDFUploadActivity.java

```
38      // Main Method
39      @Override
40      protected void onCreate(Bundle savedInstanceState)
41      {
42          super.onCreate(savedInstanceState);
43          setContentView(R.layout.activity_pdfupload);
44
45          // Assignment Operations
46          editPDFFileName = findViewById(R.id.text_pdfName);
47          button_upload_pdf = findViewById(R.id.btn_pdfUpload);
48
49          storageReference = FirebaseStorage.getInstance().getReference();
50          databaseReference = FirebaseDatabase.getInstance().getReference( path: "PDFs");
51
52
53          // On click listener for PDF upload button
54          button_upload_pdf.setOnClickListener((v) -> {
60                  // Method for selecting PDF from device
61                  selectPDFFile();
62
63              // end of on click method
64          });
66
67      // end of main method
68      }
```

The methods for this class were then created. The table below contains the method names, required parameters, return value, and intended functionality of each method in this class.

Method	Parameters	Returned Value	Functionality
onCreate	*Bundle* savedInstanceState	Void	Main method. Assignment operations for the initialised variables. Also contains an *onclicklistener* on the upload button.
selectPDFFile	None	Void	This method allows the user to select a file from their device, which they want to upload, and then store it in an intent.
onActivityResult	*int* requestCode *int* resultCode, *Intent* data	Void	This method checks if the *requestCode* and *resultCode* are 1, and *data* is not null, meaning

			a file has been chosen. If the check if successful, the *data* variable has the 'getData' method applied to it and passed to the uploadPDFFile method.
uploadPDFFile	*Uri* data	Void	This method initializes the *ProgressDialog* and displays it with a custom title. A storage reference has also been assigned to the *Firebase* storage section. If the file is successfully uploaded, an *onSuccess* method will run, adding the name and URL of the file to *Firebase*. The progress bar is then hidden.
Btn_action	*View* view	Void	This method is for the button which takes users to the *view PDF class* (contains all uploaded files).

View_PDF_Files.java

The *View_PDF_Files* class is responsible for displaying and viewing uploaded files. This class is utilised when the 'file list' button is clicked, and the file list is displayed. This list contains all the files associated with a specific vehicle. Import statements include a *ListView* (to display the files in a list fashion), a database reference (to retrieve file data from), and a *List* (containing the files).

The code extract below displays how the variables are initialised in the *onCreate* (main method). The *ListView* variable is assigned to its element on the *XML* file, the *List* variable is assigned to a new, empty *ArrayList*. An *onItemClickListener* is applied to the *ListView* to check if a user clicks on any files. If a click is detected, the corresponding file is retrieved and displayed.

```
// Assignment Operations
my_PDF_ListView = (ListView)findViewById(R.id.myPDFListView);
// Empty Array List
uploadingPDFS = new ArrayList<>();

// Call method to view stored PDF files
viewAllFiles();

// Item click listener on list view
my_PDF_ListView.setOnItemClickListener((parent, view, position, id) -> {
        // Store PDF file position in list
        uploadingPDF uploadingPDF = uploadingPDFS.get(position);

        // Create new intent
        Intent intent = new Intent();
        // Set intent type
        intent.setType(Intent.ACTION_VIEW);
        // Set intent data
        intent.setData(Uri.parse(uploadingPDF.getUrl()));
        // Start the activity
        startActivity(intent);

    // end of on item click method
});
```

The methods for this class were then created. The table below contains the method names, required parameters, return value, and intended functionality of each method in this class.

Method	Parameters	Returned Value	Functionality
onCreate	*Bundle* savedInstanceState	Void	Main method. Includes assignment operations of the initialised variables. There is also an *onItemClickListener* on the *ListView*.
viewAllFiles	None	Void	This method retrieves all the stored files and places them in a chronological list.

uploadingPDF.java

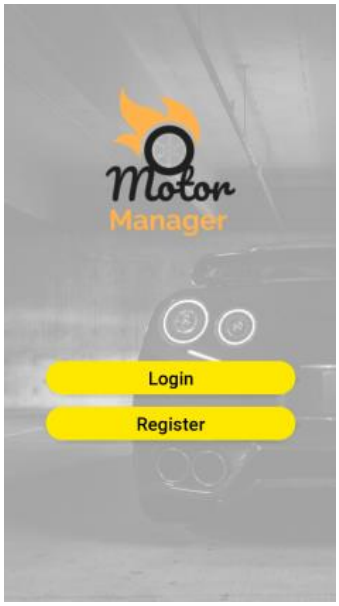
The *uploadingPDF* class is responsible for file details in the database. It creates an object per file, encompassing the file *name* and *URL*. Initialisations of two *String* variables are made (*name* and *URL*). The code extract below presents the base constructor (default, no parameters) and two variable constructor, which is used to store each file as an object in the database.

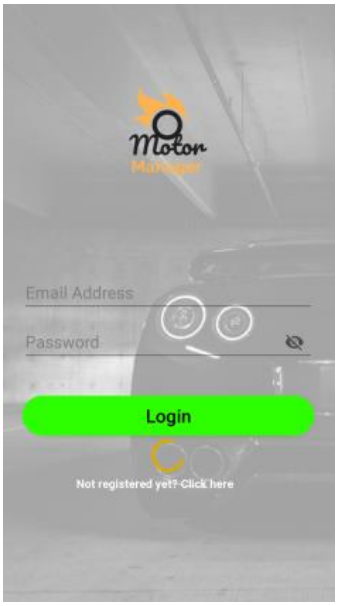
```
// Empty constructor (base)
public uploadingPDF()
{

// end of constructor
}

private uploadingPDF(String name, String url)
{
    this.name = name;
    this.url = url;

// end of two parameter constructor
}
```

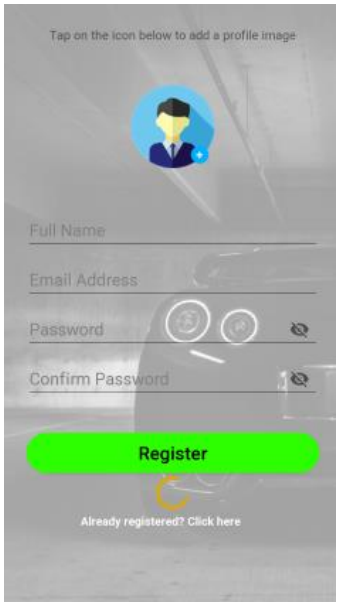
The methods for this class were then created. The table below contains the method names, required parameters, return value, and intended functionality of each method in this class.

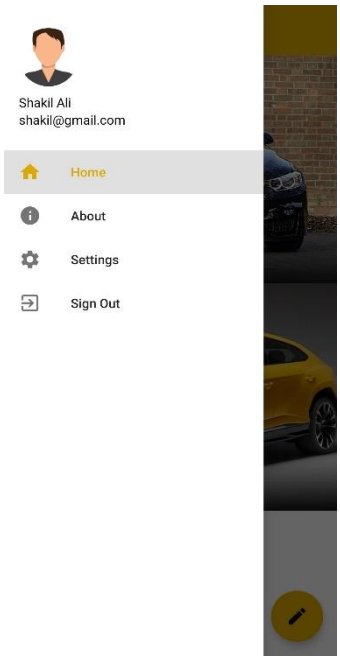
Method	**Parameters**	**Returned Value**	**Functionality**
uploadingPDF	None	N/A	This constructor is the default, non-parameterised, constructor. The purpose of this constructor is for general initialisation.
uploadingPDF	*String* name *String* url	Void	This constructor is a two-string parameter. These arguments are then assigned to the global *name* and *url* variables.
getName	None	String	This is a getter for the name of the file.
getUrl	None	String	This is a getter for the url of the file.

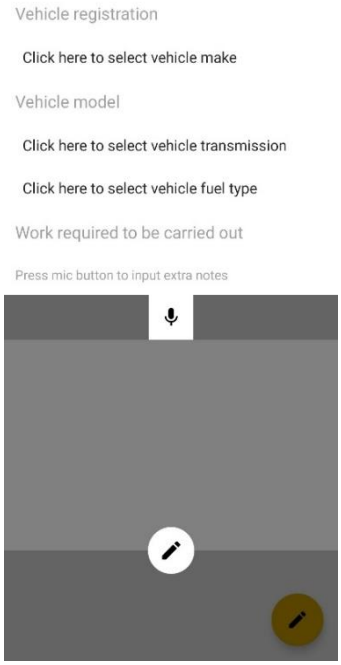
8.13 Appendix M: Motor Manager Screens

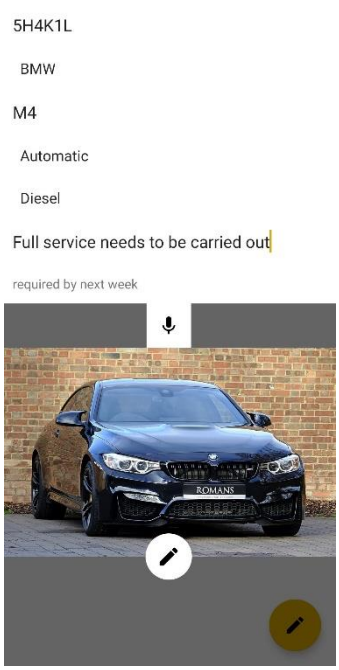


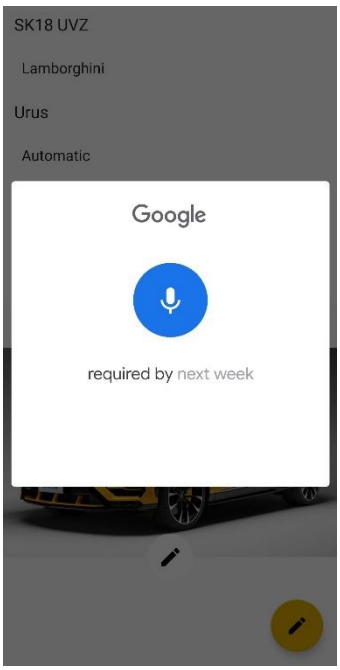


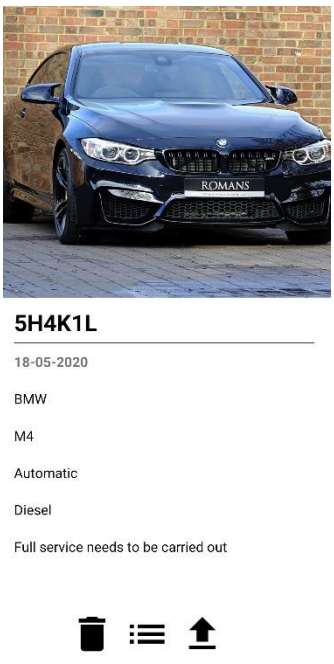


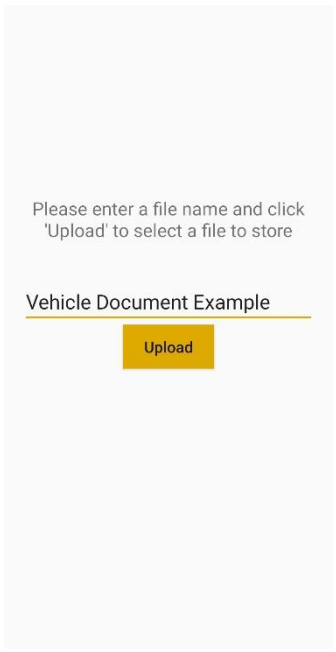


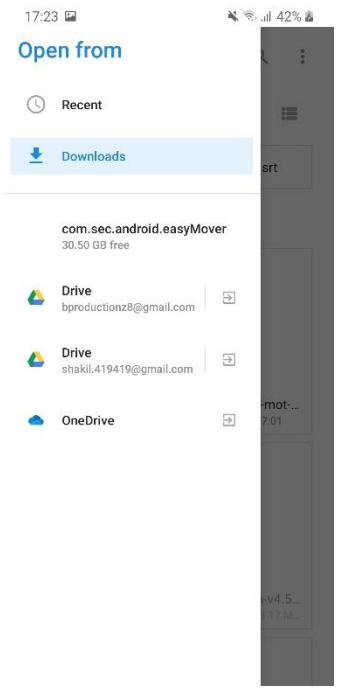



5H4K1L
BMW
M4
Automatic
Diesel
Full service needs to be carried out
required by next week

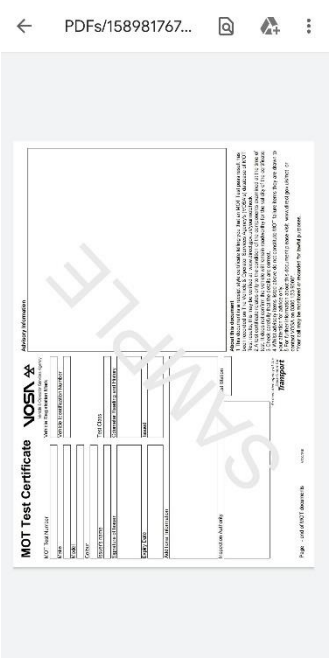

SK18 UVZ
Lamborghini
Urus
Automatic
Google
required by next week

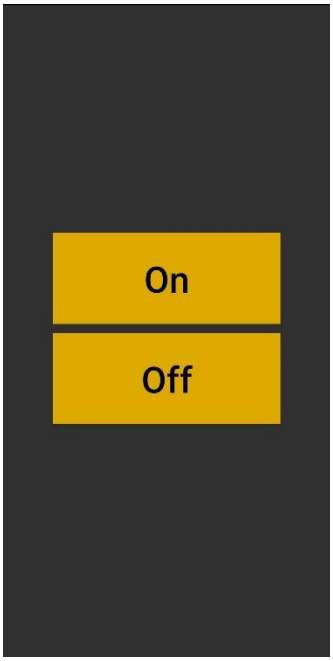

5H4K1L
18-05-2020
BMW
M4
Automatic
Diesel
Full service needs to be carried out

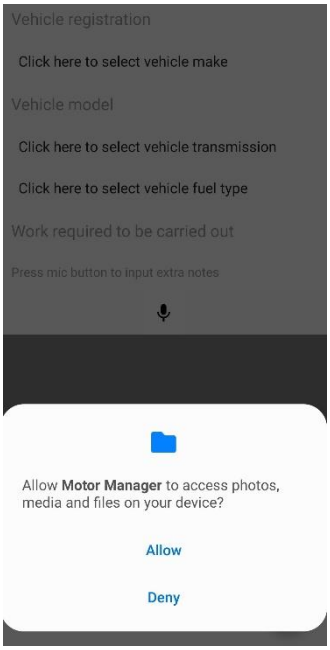

Please enter a file name and click 'Upload' to select a file to store
Vehicle Document Example
Upload


17:23
42%
Open from
Recent
Downloads
com.sec.android.easyMover
30.50 GB free
Drive
bproductionz8@gmail.com
Drive
shakil.419419@gmail.com
OneDrive


Vehicle Document Example



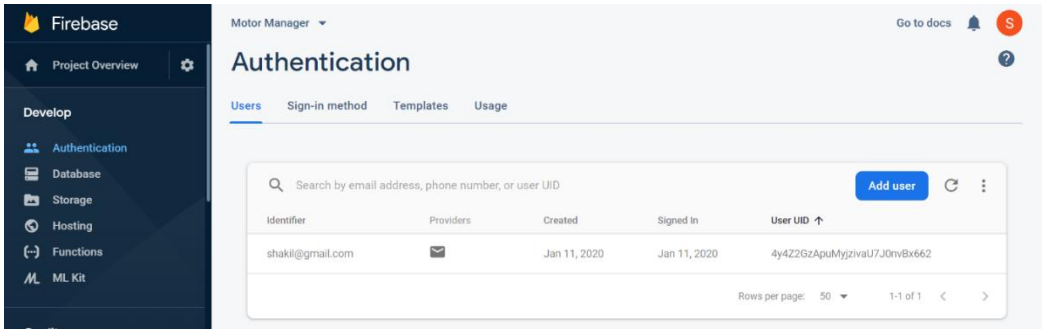




See Appendix W for the User Guide (explains the *Motor Manager* screens further).

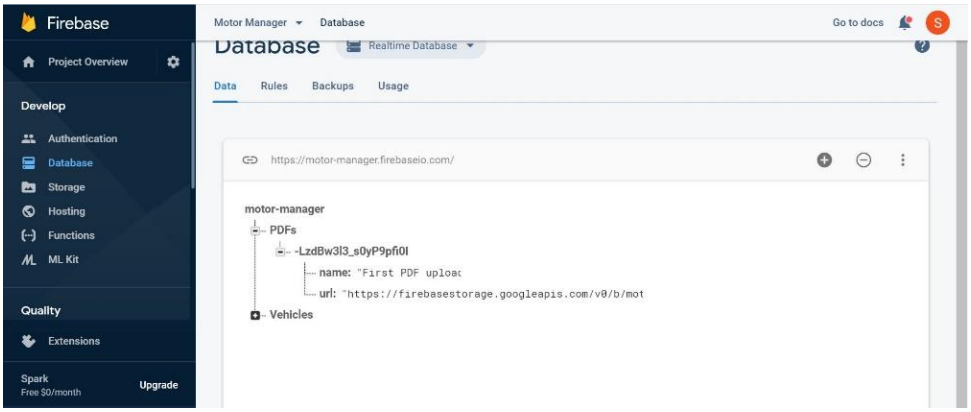
8.14 Appendix N: Firebase Development Platform - Extra Images

Firebase Authentication:

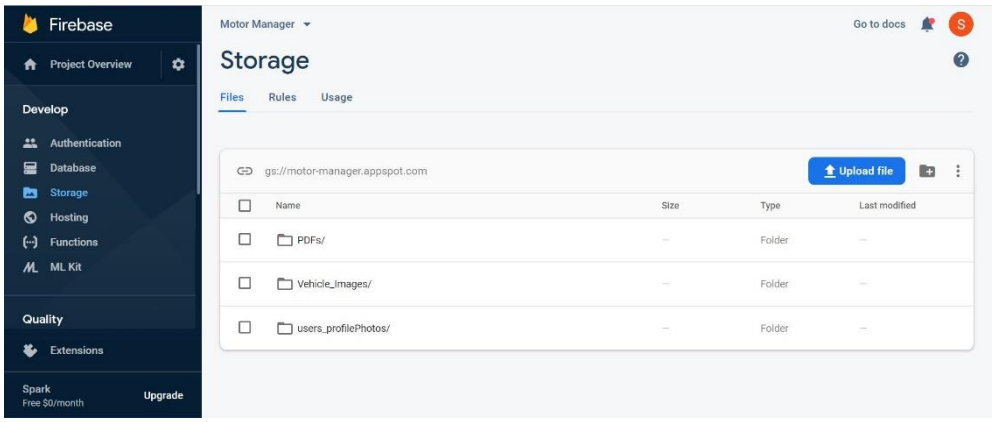


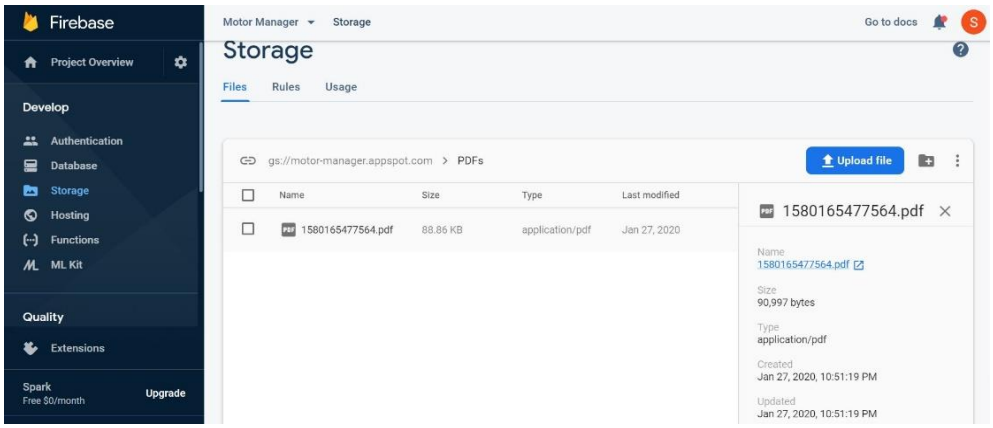
Firebase Realtime Database - PDF functionality:

```
// Initialisations
EditText editPDFFileName;
Button button_upload_pdf;

StorageReference storageReference;
DatabaseReference databaseReference;

// Main Method
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_pdfupload);

    // Assignment Operations
    editPDFFileName = findViewById(R.id.text_pdfName);
    button_upload_pdf = findViewById(R.id.btn_pdfUpload);

    storageReference = FirebaseStorage.getInstance().getReference();
    databaseReference = FirebaseDatabase.getInstance().getReference( path: "PDFs");
```

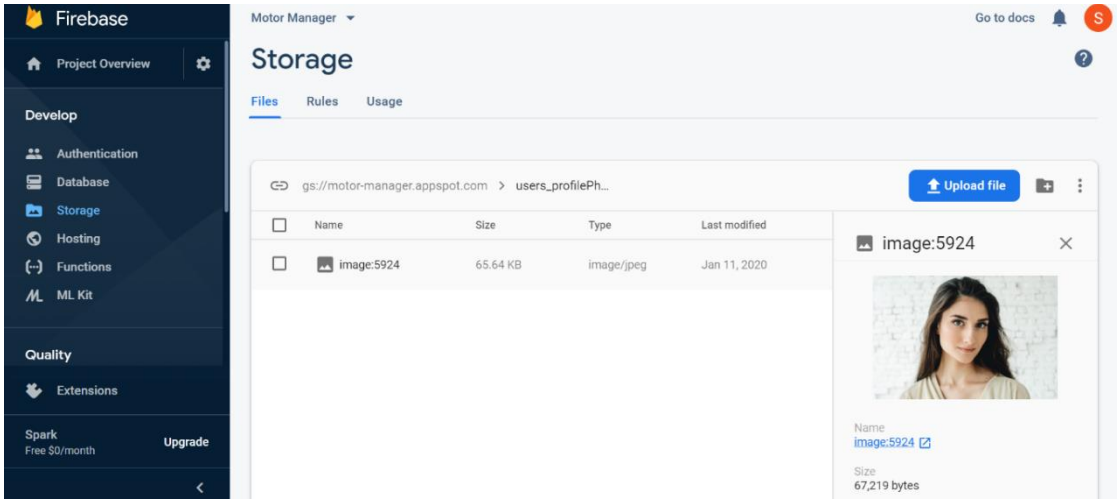


Firebase Storage - PDF functionality:

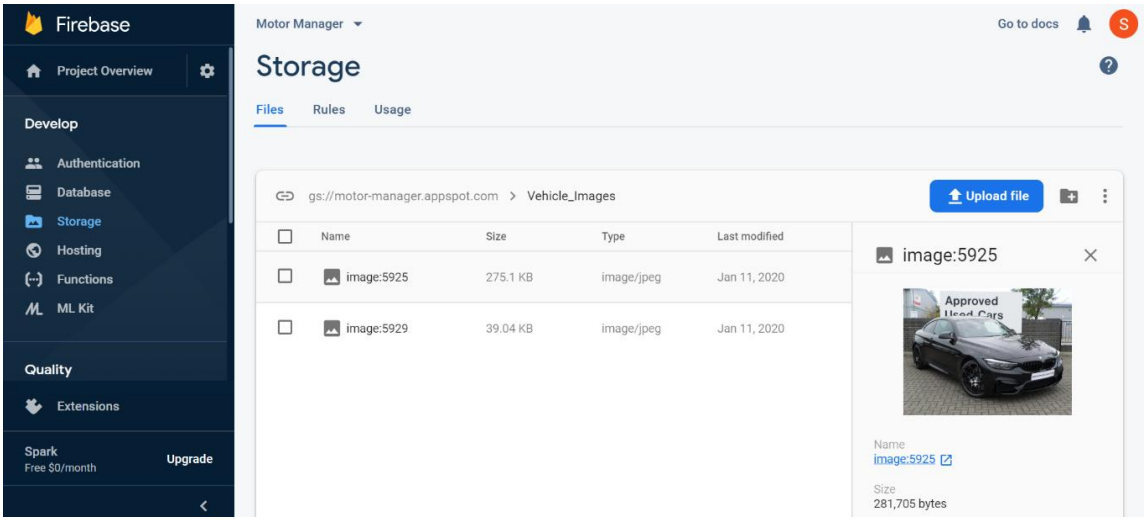




Firebase Storage - User Profile Image:



Firebase Storage - Vehicle Image:



8.15 Appendix O: White Box Testing (Post-MVP)

Post-MVP Unit Tests

Test Case ID	Test	Class File (Java)	Test Class File (Java)	Status (Pass/Fail)
Activities				
1	speak()	Home	HomeUnitTests	PASS
2	deleteVehicle(String vehicle)	VehicleDetailActivity	VehicleDetailActivityUnitTests	PASS
3	btn_action(View view)	VehicleDetailActivity	VehicleDetailActivityUnitTests	PASS
Adapters				
No Post-MVP Adapters classes				
Fragment				
No Post-MVP Fragment classes				
Helpers				
4	onCreate(Bundle savedInstanceState)	NightMode	NightModeUnitTests	PASS
5	onCreate(Bundle savedInstanceState)	PDFUploadActivity	PDFUploadActivityUnitTests	PASS
6	selectPDFFile()	PDFUploadActivity	PDFUploadActivityUnitTests	PASS
7	onActivityResult(int requestCode, int resultCode, Intent data)	PDFUploadActivity	PDFUploadActivityUnitTests	PASS
8	uploadPDFFile(Uri data)	PDFUploadActivity	PDFUploadActivityUnitTests	PASS
9	btn_action(View view)	PDFUplaodActivity	PDFUploadActivityUnitTests	PASS
10	onCreate(Bundle savedInstanceState)	View_PDF_Files	View_PDF_Files_UnitTests	PASS
11	viewAllFiles()	View_PDF_Files	View_PDF_Files_UnitTests	PASS
12	uploadingPDF()	uploadingPDF	uploadingPDFUnitTests	PASS
13	uploadingPDF(String name, String url)	uploadingPDF	uploadingPDFUnitTests	PASS
14	getName()	uploadingPDF	uploadingPDFUnitTests	PASS
15	getUrl()	uploadingPDF	uploadingPDFUnitTests	PASS
Models				
No Post-MVP Models classes				

Post-MVP Integration Tests

Test Case ID	Test Case Objective	Test Case Description	Expected Result	Test File (Java)	Status (Pass/ Fail)
1	Check the link between the vehicle details screen and file upload screen	Click the 'File Upload' button	To be directed to the upload file page	VehicleDetailToFileUploadIntegrationTest	PASS
2	Check the link between the vehicle details screen and the files list screen	Click the 'File List' button	To be directed to the upload file page	VehicleDetailToFileListIntegrationTest	PASS
3	Check the link between the file list and file	Click on any file from the file list	To display the file	FileListToFileIntegrationTest	PASS

Post-MVP Systematic Tests

Test Case ID	Test Case Objective	Test Case Description	Expected Result	Test File (Java)	Status (Pass/Fail)
No systematic tests were carried out for Post-MVP functions					

8.16 Appendix P: Black Box User Testing - Survey (MVP)

Black Box User Testing

I am a student in the Department of Computing, and as part of my project for Computing Projects, I am using this survey to collect data from users after testing my vehicle management application: Motor Manager. The survey is designed to help me gather relevant information about how well my applications' functions perform. This survey will come with an instruction sheet to help traverse the user to test all the functions of the software. This survey will be open until 19/03/2020. Your participation is entirely optional and voluntary. Your identity will remain anonymous. All data will be destroyed after the project has completed. The whole survey is likely to take around 5 minutes to complete. Please take your time to read the questions carefully, and answer as truthfully as possible. Should you have any questions, please feel free to contact me: sali011@gold.ac.uk. Thank you very much for your time.

* Required

1. Which age category do you fall under? *

Mark only one oval.

- ◯ Prefer not to say
- ◯ Under 18
- ◯ 18-24
- ◯ 25-30
- ◯ Above 30

2. What is your gender? *

Mark only one oval.

- ◯ Prefer not to say
- ◯ Female
- ◯ Male
- ◯ Other: _____

3. What is your profession? *

Mark only one oval.

- ◯ Prefer not to say
- ◯ Mechanic (or garage worker)
- ◯ Car rental provider
- ◯ Student
- ◯ Other: _____

4. Do you own a vehicle or does your profession require you to be in possession of a vehicle(s)? *

Mark only one oval.

- ◯ Prefer not to say
- ◯ Yes
- ◯ No
- ◯ Not Sure

5. Does the application open successfully when the 'Motor Manager' application icon is clicked? *

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ Not sure

6. Are you able to view the opening screen (displaying 'Login' and 'Register' buttons)? *

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ Not sure

7. Are you able to open the 'Login' screen successfully? *

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ Not sure

8. Are you able to open the 'Register' screen successfully? *

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ Not sure

9. Does the application inform you of missing fields (e.g. full name)? *

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ Not sure

10. Are you successfully able to register an account? *

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ Not sure

11. Are you able to click the side-bar navigation button (menu icon) and log out via the 'Sign Out' option? *

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ Not sure

12. Once logged out, are you able to sign-in again, via the 'Login' page? *

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ Not sure

13. Click the menu icon and go through all the pages of the application. Do they work accordingly (e.g. 'About' takes you to the 'About' page)? *

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ Not sure

14. Does the vehicle detail addition pop-up form appear when you click the pencil icon on 'Home'? *

Mark only one oval.

- Yes
- No
- Not sure

15. Are you able to upload vehicle details (and if not, does the application inform you that there are empty fields)? *

Mark only one oval.

- Yes
- No
- Not sure

16. Does the 'Home' page update with the vehicle you just submitted (i.e. the vehicle image, registration number)? *

Mark only one oval.

- Yes
- No
- Not sure

17. When you click on the vehicle thumbnail, are you able to view the details you uploaded? *

Mark only one oval.

- Yes
- No
- Not sure

Optional Additional Questions

These optional questions are for user insight and future development purposes.

18. Did the application meet MVP requirements (see the instructions sheet for MVP requirements)?

Mark only one oval.

- ◯ Yes
- ◯ No
- ◯ Other: _____

19. How intuitive did you find the application?

Mark only one oval.

	1	2	3	4	5	
Not intuitive	◯	◯	◯	◯	◯	Very intuitive

20. How good was the user interface?

Mark only one oval.

	1	2	3	4	5	
Poor	◯	◯	◯	◯	◯	Exceptional

21. What was the best aspect of Motor Manager? (see the instructions sheet for definitions)

Mark only one oval.

- ◯ Intuitiveness
- ◯ User Interface
- ◯ Accessibility
- ◯ Price
- ◯ Functions
- ◯ Security

22. What aspects can be improved (or added) in future versions of the application?

_____

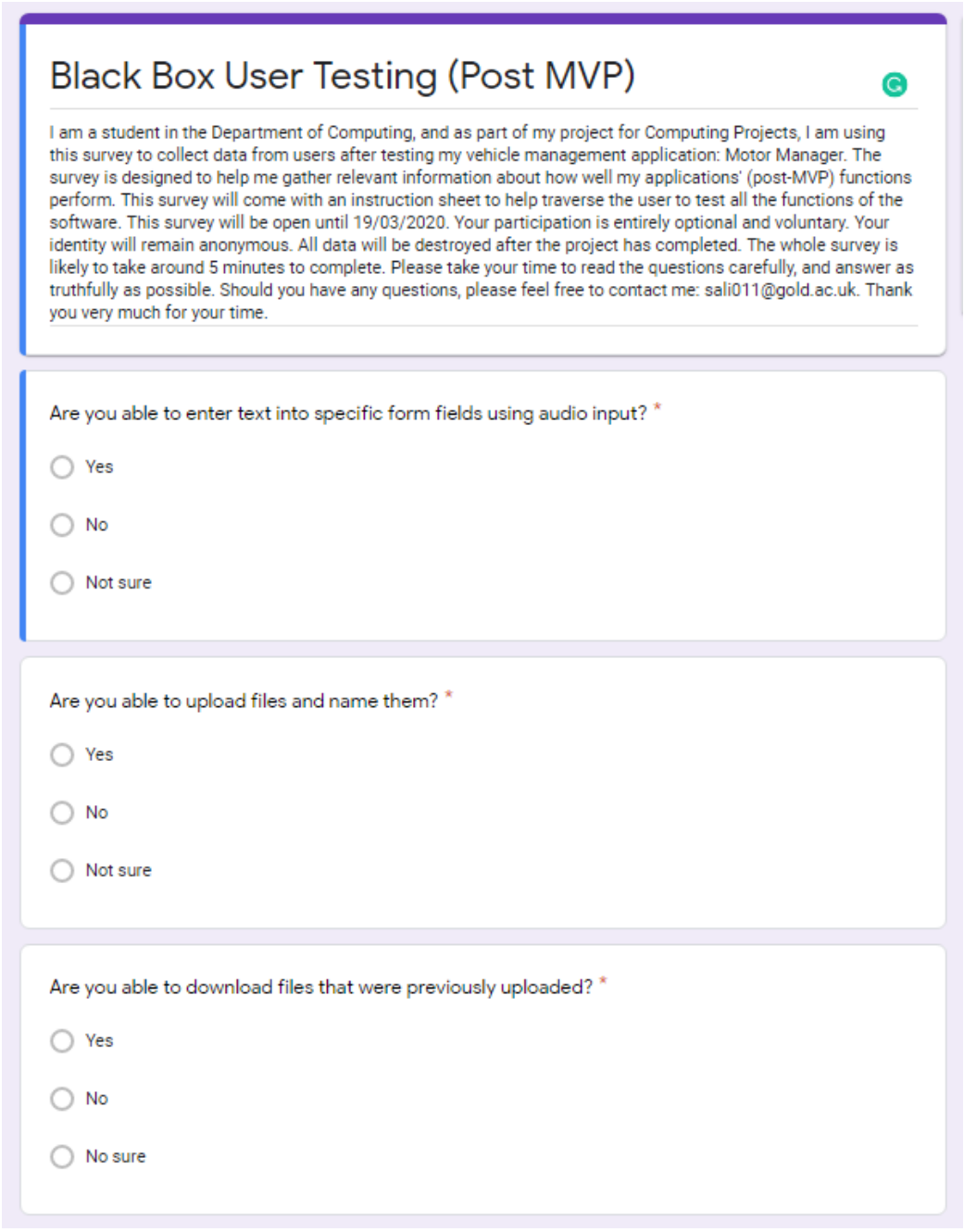
_____

_____

_____

_____

8.17 Appendix Q: Black Box User Testing - Survey (Post-MVP)



Black Box User Testing (Post MVP)

I am a student in the Department of Computing, and as part of my project for Computing Projects, I am using this survey to collect data from users after testing my vehicle management application: Motor Manager. The survey is designed to help me gather relevant information about how well my applications' (post-MVP) functions perform. This survey will come with an instruction sheet to help traverse the user to test all the functions of the software. This survey will be open until 19/03/2020. Your participation is entirely optional and voluntary. Your identity will remain anonymous. All data will be destroyed after the project has completed. The whole survey is likely to take around 5 minutes to complete. Please take your time to read the questions carefully, and answer as truthfully as possible. Should you have any questions, please feel free to contact me: sali011@gold.ac.uk. Thank you very much for your time.

Are you able to enter text into specific form fields using audio input? *

- Yes
- No
- Not sure

Are you able to upload files and name them? *

- Yes
- No
- Not sure

Are you able to download files that were previously uploaded? *

- Yes
- No
- No sure

Are you able to delete existing vehicle data? *

- Yes
- No
- Not sure

Does the 'Dark Mode' work as described? *

- Yes
- No
- Not sure

8.18 Appendix R: Black Box User Testing - Instructions

MVP Instructions

Prerequisite: Ensure the device is connected to the internet (via Wi-Fi or mobile data)

1) Select the Motor Manager application (click on the application with the Motor Manager logo).
2) Once the opening screen is present, click the 'Login' button. Check if it takes you to the 'Login' screen. Then go back to the opening screen (click back button).
3) Once back on the opening screen, click the 'Register' button. Check if it takes you to the 'Register' screen. Then go back to the opening screen (click back button).
4) Click on the 'Register' button again, and register an account on the registration page. Ensure you fill in all the fields (check whether the application informs you of empty fields).
5) Once you have registered successfully, click the side-bar navigation button and logout (click 'Sign Out'). Then test if you can log back in, using the credentials you just registered with (on the 'Login' screen).
6) When logged in successfully, check if the default page is the home screen. Click the menu icon, and check if the side-bar navigation comes out, displaying all pages of the application.
7) Click on all pages, and check if their screens are displayed appropriately.
8) Go to the 'Home' screen of Motor Manager (via the side-bar navigation), and then check if you can select the 'add vehicle' button (pencil icon in a circle).
9) Fill in the fields i.e. vehicle image, registration number, etc, then click the upload button (pencil icon). Also, check if the application informs you of empty fields.
10) Check if the 'Home' page updates with the vehicle you just uploaded i.e. thumbnail appear displaying vehicle image and registration number.
11) Click the thumbnail and check if you can view the vehicles' details.

NOTE: MVP (Minimum Viable Product) for Motor Manager includes: Login/Registration System (signing out must also be part of the functionality), Vehicle Data Addition, Vehicle Data Display.

Post-MVP Instructions

1) When adding vehicle details (see MVP instructions above), test if the voice input button allows you to input audio, then convert it correctly into the text fields
2) Next on any stored vehicle and then click the 'file upload' icon (upload image)
3) Then upload a file related to the vehicle e.g. MOT certificate and provide a name for the file
4) Click on the 'file list' icon (list image) and check if the file you uploaded is present with the correct name
5) If the file is present, click on it to view and test whether you have the option of downloading the file
6) Check whether you can delete a vehicle (and its details) using the 'delete' icon (bin image)
7) After, open the burger menu and click 'Settings' to go to the settings page
8) Then click 'Dark Mode' and select 'On' to turn this mode on
9) Check if the application has changed to a 'dark' (black) coloured theme

Definitions:

Intuitiveness - having the ability to understand or know something without any direct evidence or reasoning process

User Interface - how the user and a computer system interact, in particular, the use of input devices and software

Accessibility - hardware and software technologies that help visually or physically impaired people to use the computer

Price - the cost of installing the application, using certain features or having a monthly subscription

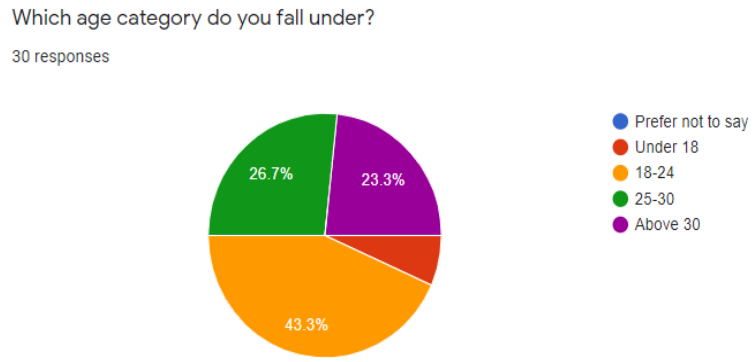
Functions - the actions and abilities of the software e.g. data visualisation, storing and viewing vehicle data

Security - elements which help enable the privacy and integrity of users personal data e.g. hide the password, confirm password

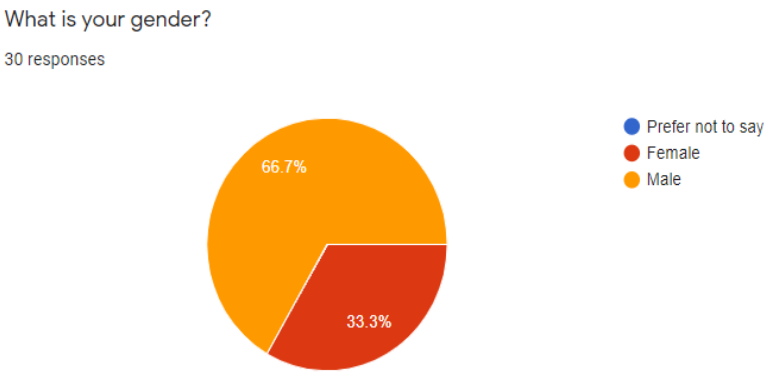
8.19 Appendix S: Black Box User Testing - Survey (MVP) - Analysis

To conduct black box user testing for *Motor Manager*, a survey was created and distributed. This survey also included the original surveyors from the initial primary research survey. There was a total of 30 responses. The survey was made up of 17 main questions, with a further (optional) 5 questions. These extra questions were added to gain user insights and feedback after using the *Motor Manager* application. Additionally, an instruction sheet was provided along with the survey to direct the surveyors to all MVP areas.

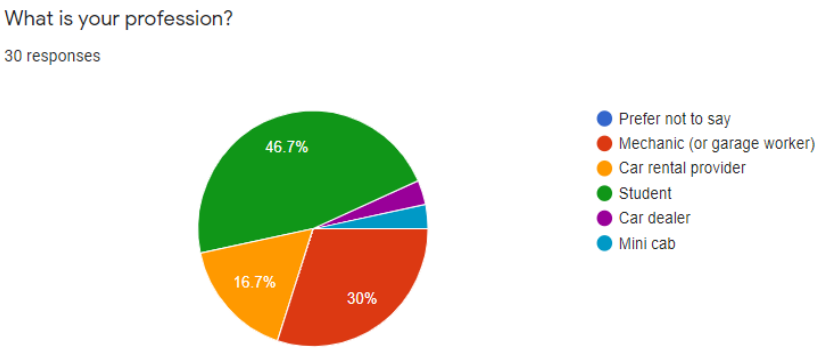
The first four exploratory questions were aimed to extract user profiles from the survey. They asked for the surveyors' age group, gender, profession, and whether they required a vehicle for work or owned a vehicle of their own. The majority of testers were 18-24-year-old, and male. Also, the two largest professions were students and mechanics respectively. Lastly, 90% of users (27 out of 30) owned a vehicle or had a vehicle(s).



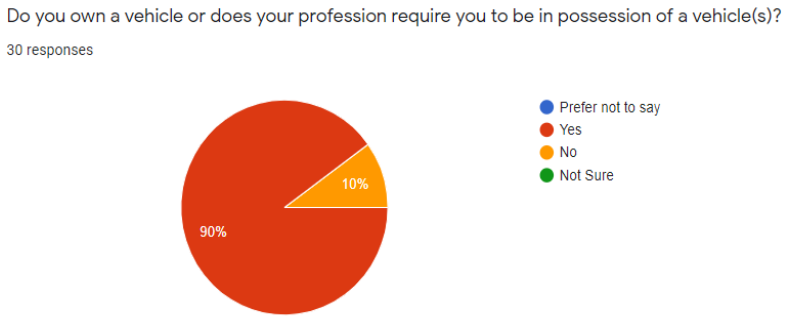
Question 1 - Pie Chart



Question 2 - Pie Chart

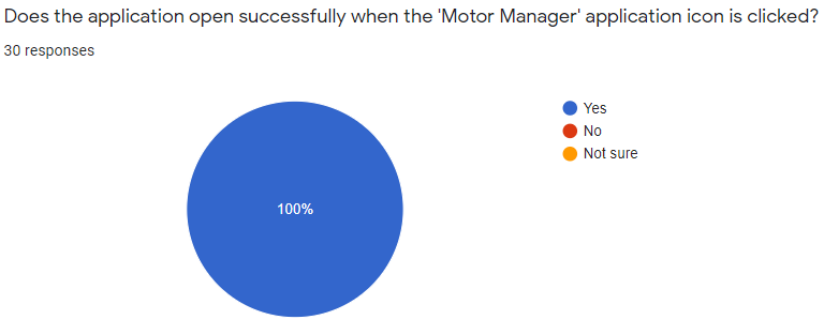


Question 3 - Pie Chart

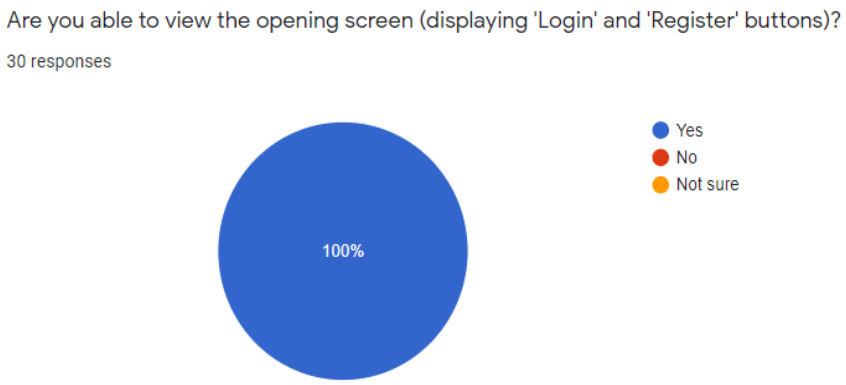


Question 4 - Pie Chart

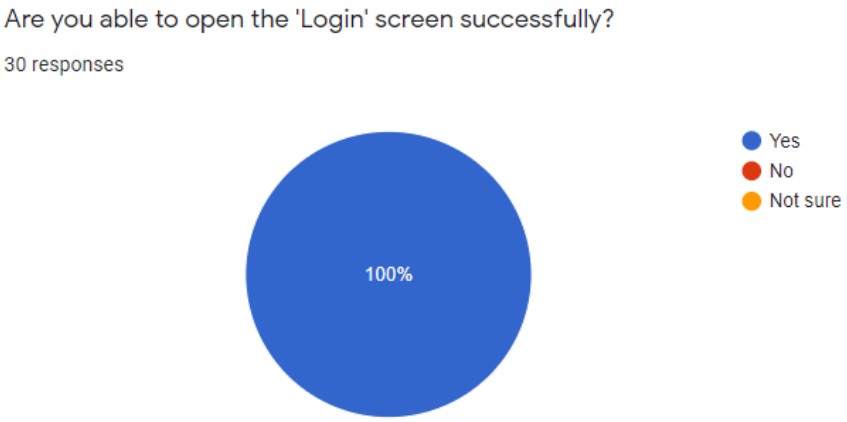
The next thirteen questions were simultaneously being answered by the surveyors as they used the instruction sheet to traverse *Motor Manager*. All 30 surveyors found that the MVP for the application worked accordingly, as all thirteen questions were answered with ‘Yes’, 100% of the time. As a result, this indicates that no issues arose in the functionality of the *Motor Manager* MVP. Consequently, no bugs have to be rectified. Therefore, the first version of this application has been successful.



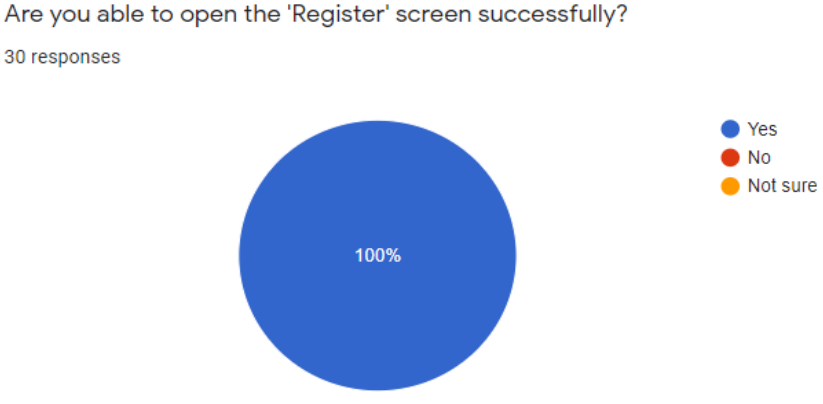
Question 5 - Pie Chart



Question 6 - Pie Chart



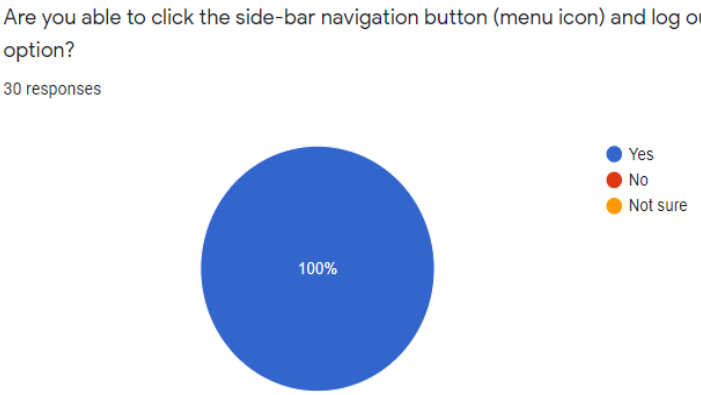
Question 7 - Pie Chart



Question 8 - Pie Chart

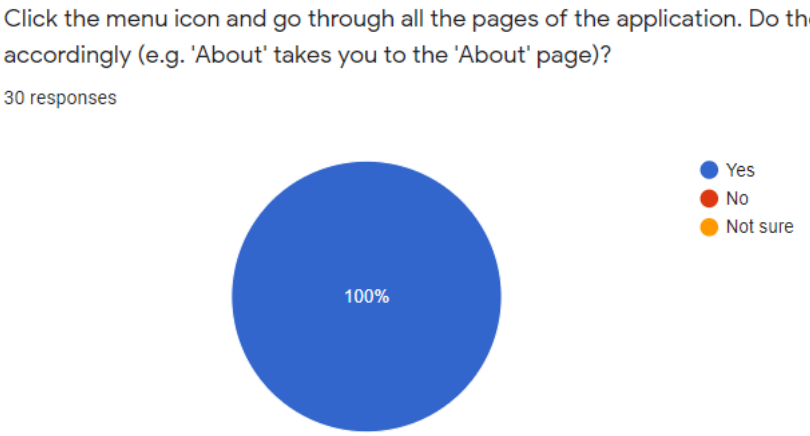
Question 9 - Pie Chart

Question 10 - Pie Chart

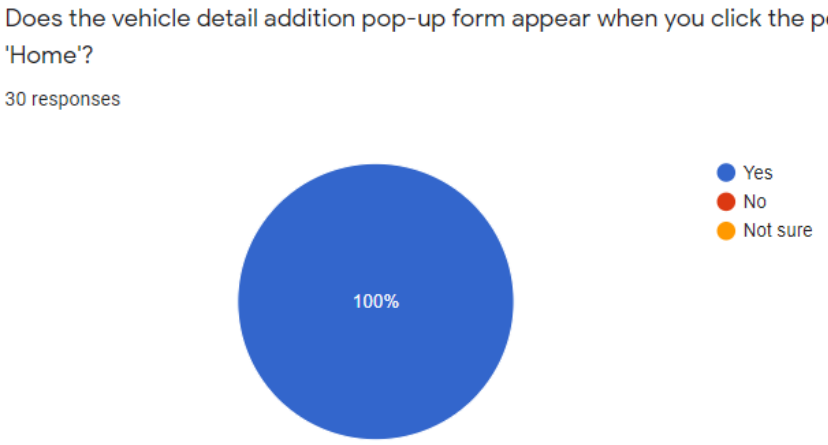


Question 11 - Pie Chart

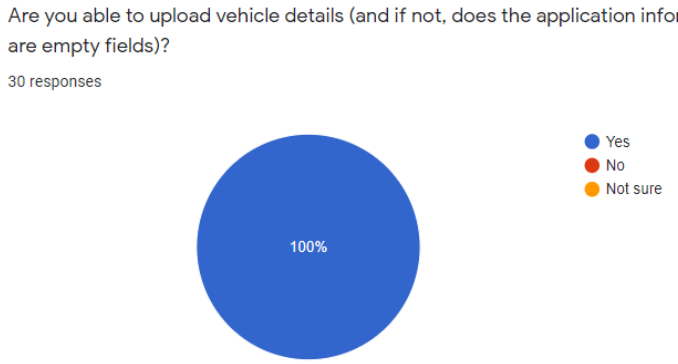
Question 12 - Pie Chart



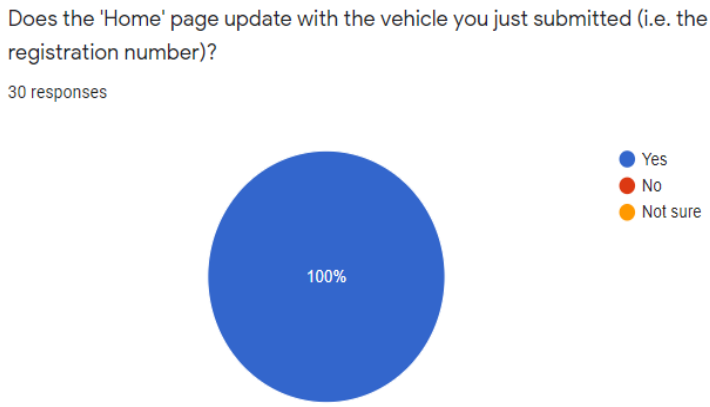
Question 13 - Pie Chart



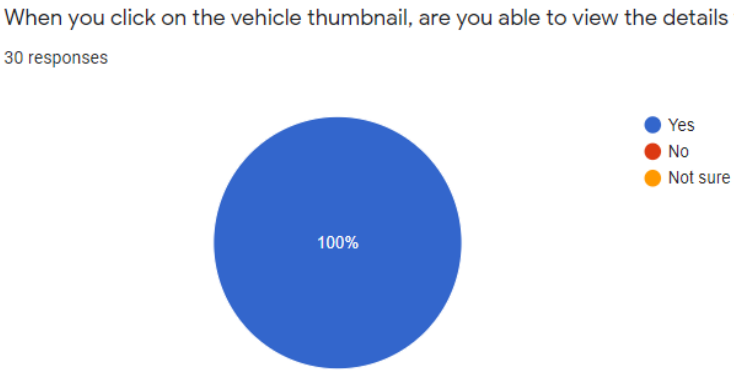
Question 14 - Pie Chart



Question 15 - Pie Chart



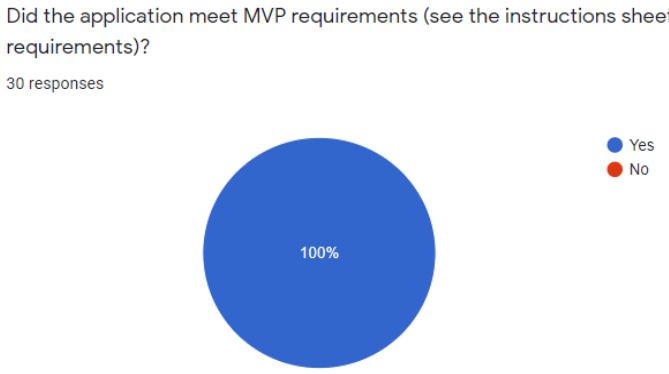
Question 16 - Pie Chart



Question 17 - Pie Chart

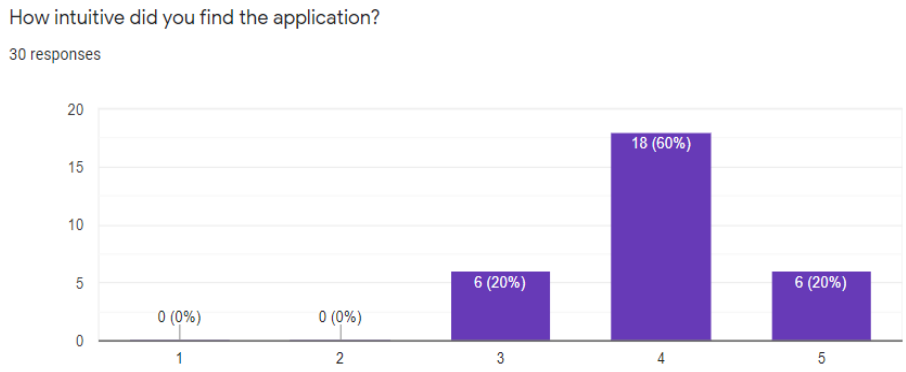
Furthermore, after the conclusion of the first section of the survey (MVP black box user testing), the additional (optional) questions followed in section two. All 30 surveyors answered the optional questions and provided feedback that can be used for future versions of *Motor Manager*.

The first additional question asked if all functions worked as required. The pie chart below demonstrates that all surveyors believe the functions were working. Therefore, no code changes need to be applied to the MVP.



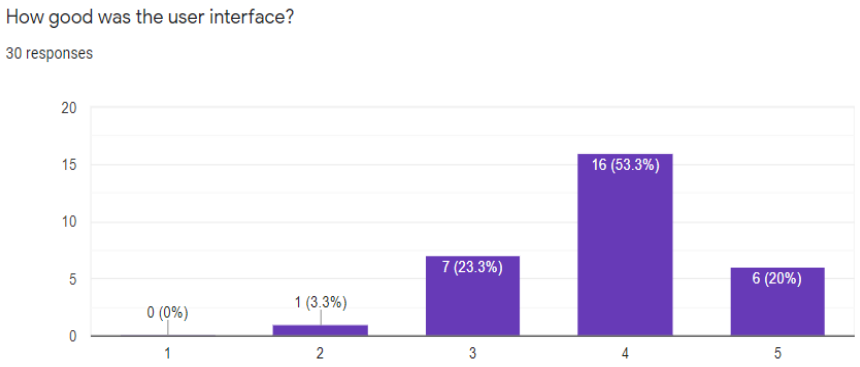
Optional Additional Questions - Question 1 - Pie Chart

The following question asked how intuitive the application was on a scale of 1-5 (1 being not intuitive at all). The bar chart below portrays that the views regarding the intuitiveness of *Motor Manager* are very positive. All surveyors believed the application to be intuitive (24 ratings it a '4' or '5'). As a result, minimal work needs to be carried out to improve the intuitiveness of the application. However, this will now need to be maintained in later versions of the application.



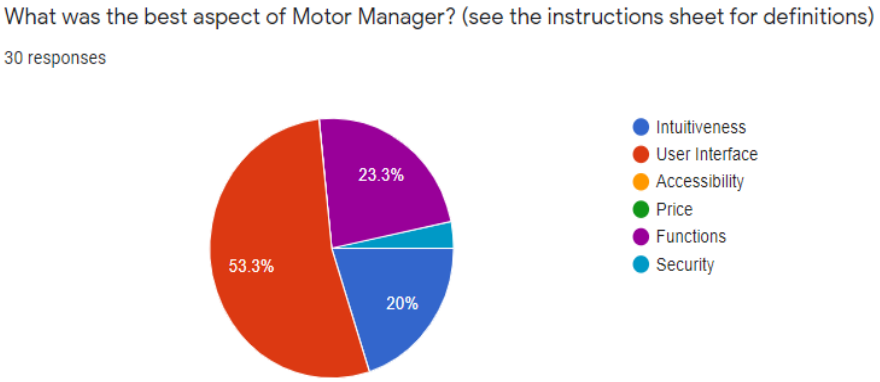
Optional Additional Questions - Question 2 - Bar Chart

Moreover, the next question was similar to the previous, however, instead of intuitiveness, surveyors were asked to rate the user interface quality. The bar chart below demonstrates how the majority of surveyors (almost 73.3%) perceived the user interface to be of good quality. Of the 30 testers, 7 believed that the user interface was of a satisfactory level (rating of 3) and 1 tester rated it a '2'. An inference can be made that the overall user interface is good, however, a few improvements will be required to be made, to ensure the satisfaction of all future intended users.



Optional Additional Questions - Question 3 - Bar Chart

The penultimate question asked surveyors what their favourite aspect of *Motor Manager* was. Over half the responses (53.3%) believed 'User Interface' was the best aspect. The next two aspects were: 'Intuitiveness' (20%) and 'Functions' (23.3%). As a result, an inference can be made here that different sections of the application were appealing to different testers. Therefore, in future versions of the application, improvements to aspects, especially the three aforementioned, will have to be made, whilst maintaining (or increasing) the level of quality.



Optional Additional Questions - Question 4 - Pie Chart

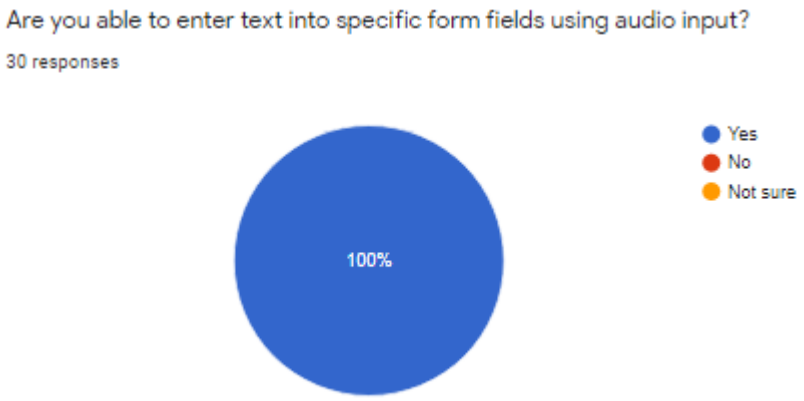
Lastly, the final question asked surveyors what they thought could be improved or added in future releases of *Motor Manager*. There were a total of 30 suggestions. The list below states the most common answers:

- Make the user interface more specific to vehicle management e.g. vehicle wallpaper
- Edit/Delete vehicle details
- Customisable displayable
- Audio input/output functions
- Attach files to vehicles e.g. MOT certificate
- Image recognition e.g. camera opens up and detects registration plate number
- Dark mode

Overall, most ultimately, the results from the black box user testing survey confirm that the MVP of *Motor Manager* was successful. The results of the middle initial thirteen questions affirm that all required functions for the MVP work as expected. Consequently, no issues need to be resolved semantically. Also, the feedback has been provided by surveyors regarding the experience they had with the application. Therefore, an aim will be to integrate these improvements and ideas into future releases of *Motor Manager*.

8.20 Appendix T: Black Box User Testing - Survey (Post-MVP) - Analysis

After the conclusion of the MVP survey, a *post-MVP* survey was carried out. The survey consisted of five questions and received 30 responses (the same individuals from the first survey). This survey had the aim of confirming that the post-MVP functions of *Motor Manager* worked correctly. The instructions on how to test these functions were described in the same instructions sheet from the first survey. As reflected by the pie charts below, all testers found that the functions worked as required. Consequently, these results matched the *MVP* analysis, and no significant changes were required to be applied to this version of the application.

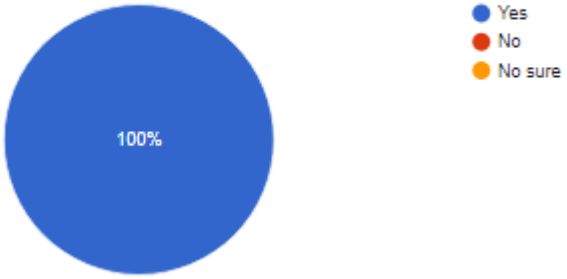


Question 1

Question 2

Are you able to download files that were previously uploaded?

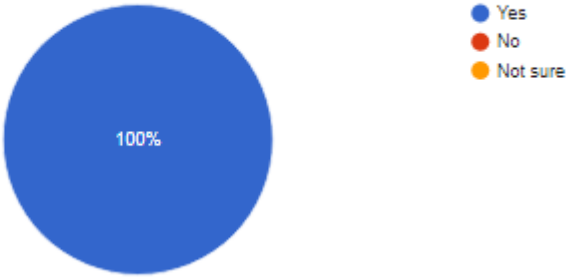
30 responses



Question 3

Are you able to delete existing vehicle data?

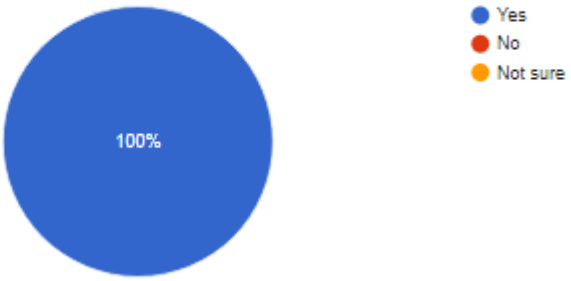
30 responses



Question 4

Does the 'Dark Mode' work as described?

30 responses



Question 5

8.21 Appendix U: Firebase Robo Testing

The *Firebase Development Platform* encompasses a testing environment called *Test Lab*. *Test Lab* has a tool for testing known as *Robo Test*. This type of test interrogates and explores the structure of an application's user interface. It simulates activities acting as a human user. The test traverses the application chronologically and values for fields can be supplied beforehand. To run a Robo test, *Motor Manager's APK* (Android Application Package) file had to be generated and uploaded to the *Firebase* console.

Figure U.1 displays the *Test Lab* console and labels have been assigned to specific areas. The labels are as follows:

1. *Robo Test* name and emulation device
2. Test status
3. Date test performed
4. Duration of test
5. The orientation of the device during the test
6. Language
7. Robo test outputs:
    - **Test Issues** - Any issues that occurred during the duration of the test
    - **Robo** - Displays statistics on actions carried out, activities and screens traversed, and notifications encountered (generates a crawl graph as seen in Figure U.2)
    - **Logs** - log files (actions) generated can be downloaded
    - **Screenshots** - images of the screens during testing
    - **Videos** - video of the automated testing being completed
    - **Performance** - displays performance-related statistics e.g. memory, network and CPU usage, test transactions and emulation device details

Figure U.1: Firebase - Test Lab

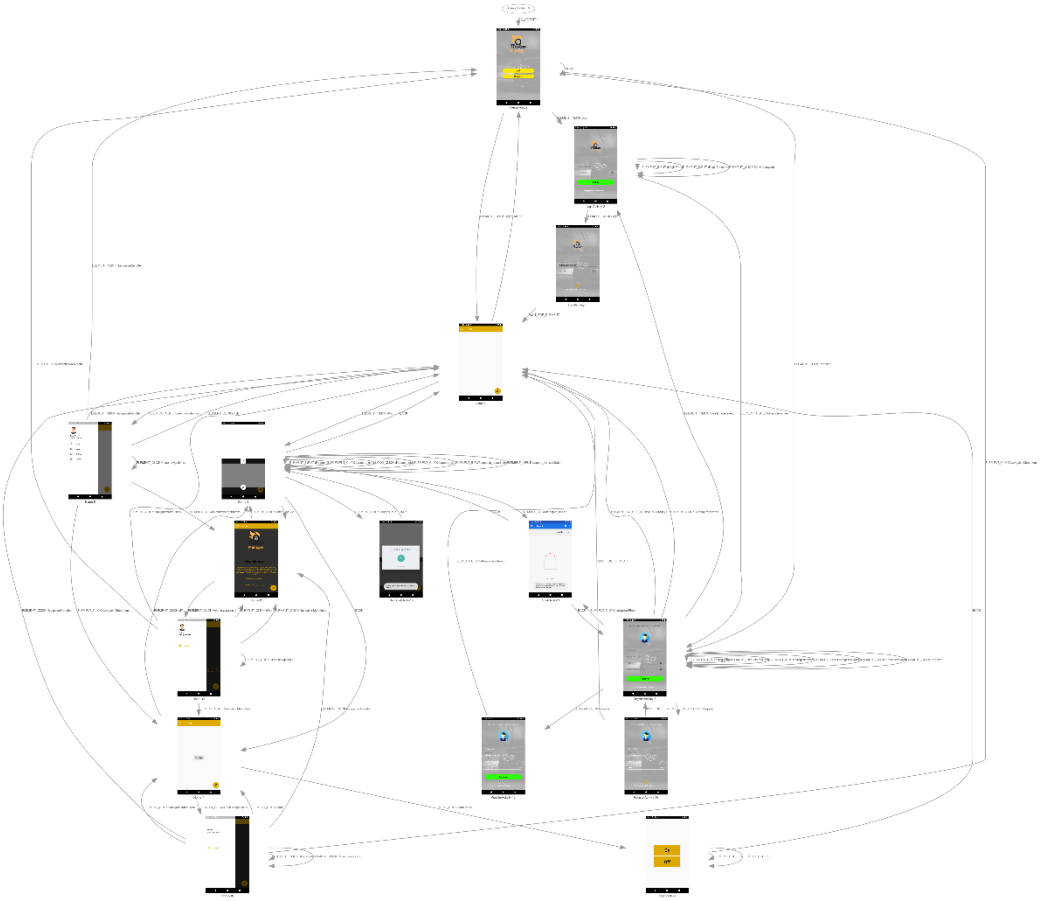
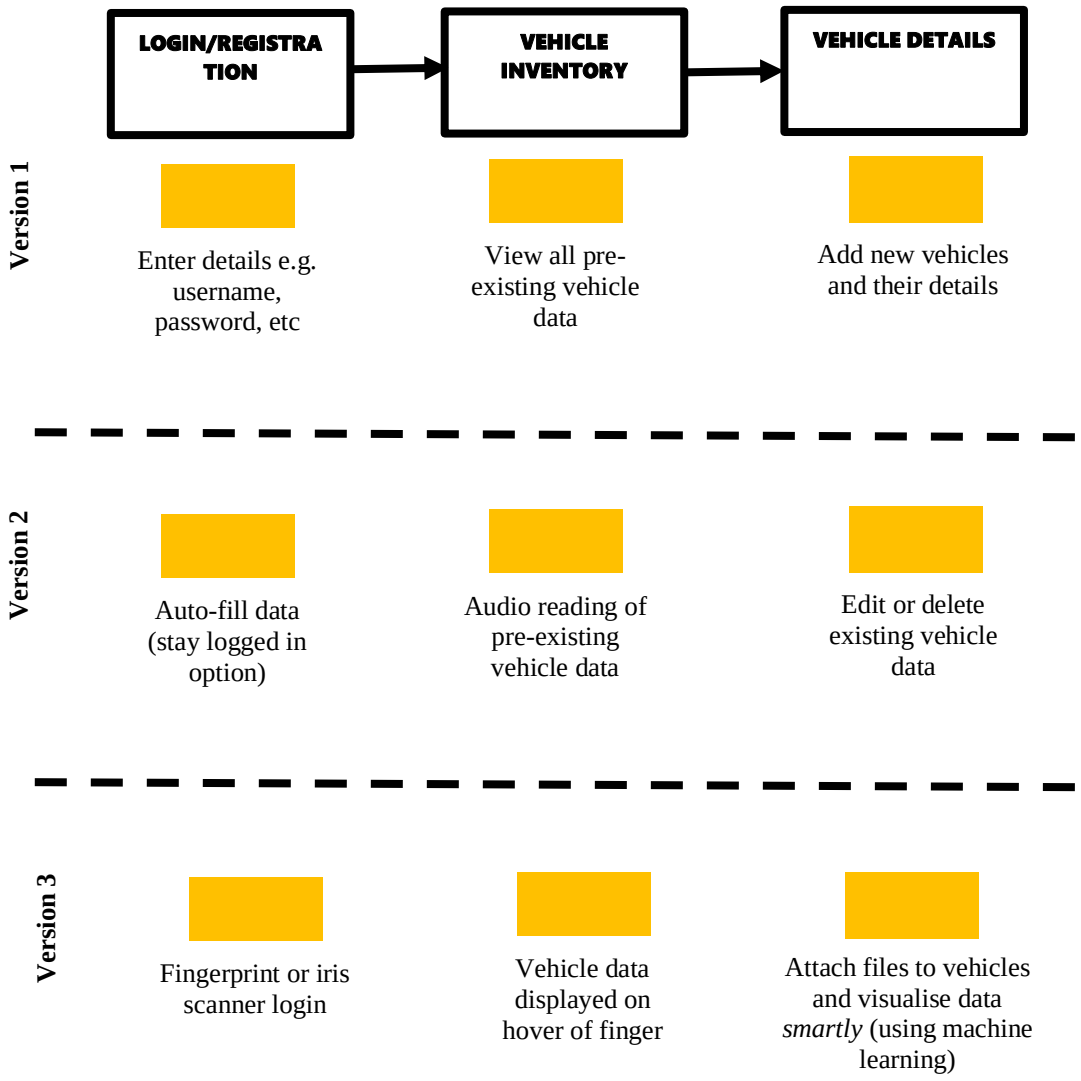


Figure U.2: Robo Test - Crawl Graph

Table U.1 below displays the non-functional requirements for this project and whether they were met by *Motor Manager*.

Non-functional Requirements		
ID	**Requirement**	**Status (Pass/Fail)**
001	Application must launch as soon as the user opens it	PASS
002	Application must inform the user of incomplete fields	PASS
003	Application must be able to handle 100 users at a time	PASS
004	Application must be portable (can download on any Android device)	PASS
005	The application must have the same user experience on all compatible devices	PASS
006	Application must allow storage of at least three vehicles	PASS
007	Application must ensure data is secure	PASS

8.22 Appendix V: Backlog



8.23 Appendix W: Project repository and related links

GitHub (Project Code Repository) - https://github.com/Shakil-Ali/Vehicle-Service-Management-Application.git

Tumblr - https://shakilali-goldsmiths.tumblr.com/

Google Drive - https://drive.google.com/open?id=1QgkaBOPPVAOaFd6P5_K9fkL5kkDvcMcs

User Guide - https://drive.google.com/open?id=1Y8guydSmKVJDaaaetmjx-DaS-qF-q3We

Trello - https://trello.com/invite/b/cS9E5WWf/45c83cd4dc562843ecdd8f125d44acde/shakil-ali-final-year-project

Motor Manager Video - https://www.youtube.com/watch?v=B8PP7tUWWEg