Tax-E - Android App for Local Taxi Bookings

CS39440 Major Project Report

Author: Rhys Evans (rhe24@aber.ac.uk)
Supervisor: Chris Loftus (cwl@aber.ac.uk)

23rd April 2019
Version: 1.0 (Draft)

This report was submitted as partial fulfilment of a BSc degree in Computer Science (G400)

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB
Wales, U.K.

Declaration of originality

I confirm that:

- This submission is my own work, except where clearly indicated.
- I understand that there are severe penalties for Unacceptable Academic Practice, which can lead to loss of marks or even the withholding of a degree.
- I have read the regulations on Unacceptable Academic Practice from the University's Academic Registry (AR) and the relevant sections of the current Student Handbook of the Department of Computer Science.
- In submitting this work I understand and agree to abide by the University's regulations governing these issues.

Name: Rhys Evans

Date: 23rd April 2019

Consent to share this work

By including my name below, I hereby agree to this project's report and technical work being made available to other students and academic staff of the Aberystwyth Computer Science Department.

Name: Rhys Evans

Date: 23rd April 2019

Acknowledgements

I would like to thank the following people for their invaluable support and guidance throughout this project:

My project supervisor, Chris Loftus for allowing me to pursue this project and going above and beyond to support me throughout. His organization of the weekly progress meetings was thoroughly helpful, allowing me to reflect on my progress and receive genuine feedback on various aspects of implementation.

My caring partner, Sara for providing me with continuous support, motivation and inspiration. Her honest and constructive feedback on the app's development was invaluable.

My loving family, John, Davina, Carwyn and Delyth for always being there and believing in me.

Abstract

Web and Mobile Technologies have become increasingly ubiquitous in the travel and tourism industry. These technologies, together with a company's ability to successfully adopt them is often pivotal to the company's success within the industry.

This project's aim was to provide a platform for local taxi companies to interact with and provide a service for their customers. The key aspect of the platform is the creation, viewing, and management of bookings. By handling bookings this way, customers and taxi companies can reliably keep track of their past and present bookings in real time.

This project has produced a fit for purpose, fully functional platform. It is split into 3 key systems: a REST API to interact with the backend database and be consumed by other systems within the platform; a web app to be used by controllers within a company to claim, delegate, and manage bookings; and an Android App to be used by customers and drivers to view, create, and manage bookings. The developed product is flexible, allowing taxi companies to choose the level of adaptation that would best fit their business.

Predictably, however, there is room for improvement and future development within the project. Given the scope and time restrictions provided, the produced platform is only intended as a starting point and proof of concept.

This document will provide an insight into the motivation behind platform's development, address weaknesses and key areas for improvement, and discuss the engineering and project management challenges encountered and how they were overcome.

Contents

List of Figures

List of Tables

Chapter 1

Background & Objectives

The aim of this project was to create a platform to serve as an intermediary between local taxi companies and their customers. The development of such a platform presented many engineering and project management challenges. This chapter aims to discuss the preparations required for the project, together with an analysis of the project's objectives and an in-depth description of the processes used.

1.1 Preparation

1.1.1 Background Research

In order to successfully execute the project, research had to be done into various aspects of the app's development. This includes methodology, technologies, and deployment options.

1.1.1.1 Methodology

Software Development Methodology is a key part of any large project. The effect of the methodology on the product, along with a project's ability to adapt to the methodology should always be considered when making a selection. With this in mind, several options were researched for the project.

The first methodology researched was the Waterfall Model, or 'SDLC' [**?**]. Having already worked on a project using the Waterfall Model, there was less of a focus on researching the methodology but rather to consider its applicability in this project. As the Waterfall Model is a plan-driven approach it would require a lot of upfront documentation and decision making with regards to requirements, thus potentially making it rather inflexible. Given that this project also involved a significant learning curve with regards to the technologies used, it would perhaps be tricky to accurately gauge the ability to implement various functionalities early on.

The next methodology researched was Feature Driven Development, or 'FDD' [**?**]. FDD

is an agile methodology created by Jeff Luca, it is an iterative approach with an emphasis on the timely and successful delivery of features. It is broken down into 5 key processes: developing an overall model, building the feature list, planning by feature, designing by feature, and building by feature. It is clear from these processes that the project's development would be done on a 'per-feature' basis, comparable to stories in other agile methodologies. At first impressions, this methodology seemed much more appropriate for the project than a plan based approach.

Extreme Programming or 'XP' [**?**, **?**], was the final methodology to be researched. It was one of the recommended methodologies provided for the project. XP is described as being ideal in projects with dynamically changing requirements, risks from fixed time projects using new technology, and when using technologies that allow for automated tests [**?**]. The methodology is often identified by its values, which give a great indication to the mission of XP. Reading literature written by one of XP's creators provided a great insight into the application of XP and its key advantages over other approaches.

1.1.1.2 Technologies

Having only experienced Android app development with a local, offline backend; it was essential to the project's success that research was done to explore the tools and technologies available to support a networked mobile app. This research was required for each aspect of the project, including its deployment.

The first and arguably most significant area to research was the API that would support the app(s). It was clear from very early on that there would likely be at least two apps within the platform to consume the API (web app and mobile app). Therefore, an API would need to be developed and a language/framework had to be selected to develop such an API. The researched options were: ASP.NET [**?**], Rails 5 [**?**], Node.js & Express.js [**?**, **?**] and Flask (Flask-RESTful) [**?**]. All of the researched options had fairly similar learning curves (due to the technology they are built on and past experiences), each option was also perfectly fit for purpose to create a REST API to serve a mobile application.

The next technology to research was the database management system (DBMS) that would be used by the REST API for data persistence. The selected DBMS would ideally be able to interface easily with the framework used to create the API itself. The options researched were: PostgreSQL [**?**], MySQL [**?**], and MongoDB [**?**]. The first two options were both traditional relational database management systems, whereas the final option is a NoSQL document-oriented system.

In order to successfully consume the REST API within the Android application, a HTTP Client library had to be chosen. During this selection, two complementary technologies were researched, Retrofit [**?**] and rxJava [**?**]. Retrofit is a HTTP Client library for Java and Android that is primarily used to consume REST APIs, rxJava is an asynchronous programming library with observable streams. This allows requests to be made to the REST API from the mobile application and the responses can be stored in a life-cycle aware Observable object, thus fully supporting asynchronous programming.

It became apparent that a relatively small web app would need to be created as part of

the platform to support the Company Administrator's role. Therefore, a front-end web framework to consume the API had to be chosen. The frameworks that were looked into were: Angular [**?**] and React [**?**]. Although Vue.js [**?**] was initially considered, it became apparent that React and Angular were the two most prominent options and would be the most beneficial technologies to learn.

Due to previous experience, it was already decided that Android Studio would be the chosen IDE used to develop the Android application. However, there was still a decision to be made about developing the API and Web Application. The standard code editor 'Atom' had been used for previous web projects. However, upon further research JetBrains's 'WebStorm' IDE was discovered. Therefore research was done into both solutions in order to select the most appropriate.

Lastly, project management, deployment, and development tools had to be researched. In terms of project management, it was already decided that an online Trello board [**?**] would be used to keep track of the needed tasks. It was also decided early on that GitHub would be the primary version control tool used in the project, due to its familiarity and potential integration with continuous integration tools. In order to fully explore the potential of using an agile development methodology, various continuous integration tools were researched. These included: JetBrains's TeamCity [**?**], TravisCI [**?**], and Jenkins [**?**]. Each CI tool has its strengths, with TravisCI standing out as the most lightweight and easily integrated. TeamCity is developed by JetBrains, several of their products were also considered/used for the project so familiarity would be an advantage. Lastly, Jenkins appeared to be the most widely used tool and therefore would have plenty of documentary and online tutorials. Finally, in order to potentially support any CI tools, the containerization tool 'Docker' [**?**] was researched for use in the API's development.

1.1.1.3 Deployment Options

Deployment of the Android app itself is made fairly straightforward by Android Studio's build options. Allowing for APK signing and Google Play Store support. However, with regards to the REST API and the Web Application, some research was required to decide on the most appropriate deployment solution. The following solutions were researched and considered as options: Heroku [**?**], Amazon Web Services (AWS) [**?**], and self-hosting using Nginx [**?**]. Each of these solutions had its advantages, deploying via Heroku would be made simple due to its in-built Docker support. Using AWS would allow for very fine grain control over server size, resources, and traffic management; it would also make supporting SSL encryption trivial. Lastly, a custom hosting solution using Nginx would be the simplest and easiest to manage, due to pre-existing server setups. However, this solution could have some security implications.

1.1.2 Functionality Research

1.1.2.1 Related Products

A major part of the functionality research required for this project was to investigate applications already on the market that are targetted towards Taxi booking creation and management. This research would go on to help identify the key functionalities required within the platform and the most effective way to implement them. Having already decided on a name, rough purpose, and short description. Researching similar products on the market is regarded as the next key step in developing an app idea.

The first and perhaps the most prominent product to be investigated was 'Uber' [**?**]. Uber's mobile app is designed to manage and delegate bookings for registered Uber drivers and not self-contained Taxi companies. However, the interaction and flow of the booking process between the driver and the customer served as a major influence when coming up with the concept for this project. Together with in-app messaging to allow easy communication between customers and drivers, Uber's use of GPS is the ultimate goal for booking creation and tracking. However, it was clear from early on that these well-polished features fell outside of the scope of this project.

Having investigated a commercial solution that did not target Taxi companies directly, the research required a more applicable product. 'My Taxi' [**?**] is a cross-platform, mobile application that allows customers to book, track and pay for their taxi. The app is incredibly well polished and maintained with an intuitive sign-up flow and booking creation process. The app is supported in many major European cities, including Brighton, Edinburgh, London, Manchester, Nottingham, Oxford, Reading, Derby, and Leicester. In addition to their mobile app, they also have a web client to support taxi business administrators.

Lastly, a primarily web-based service, 'minicabit' [**?**] was investigated. Minicabit is a UK based service, that allows users to retrieve taxi booking quotes, it is also possible to book a taxi based on the quotes received. The website's user interface was very friendly and intuitive. The look and feel of the quoting system were very similar to other websites within the travel industry. Although this service provides support for booking taxis, it appeared to have minimal support for any booking management or tracking.

1.2 Analysis

1.2.1 Objective

The primary objective of this project was to create a platform that serves as an intermediary between local taxi companies and their customers. The platform would need to support three types of users: Company Admin, Driver and Customer.

It would be intended that a single instance of the platform be used by many taxi companies, each with their own Company Admins and Drivers. The companies would then use the platform as a service in order to support their internal booking mechanisms. How-

ever, it should also have the ability to be adapted to be used by a single Company as their sole booking management mechanism. This would make the platform very flexible, allowing for varying use cases.

The primary motivation for this project was to streamline the process of making taxi bookings in small, local areas. With the hope that a fully mobile, digital solution would lead to a better experience for both the companies and their customers. For example, being able to view a history of all bookings could be extremely helpful for cases of lost property, criminal damage, etc. In addition to this, by viewing the live status of their booking, and communicating with their driver via notes, customers can make sure they are present for their booking at the correct time, hopefully resulting in fewer wasted Taxis.

1.2.2 Proposed Approaches

Having formalized the objectives of this project, it was required that an approach to fulfilling those objectives be selected. Given the initial topic that this project stemmed from ('Android app of your choice'). It was apparent that a native Android App would be the primary solution to complete the project's objectives. However, there was plenty of freedom to decide on an approach for a backend to serve the application; or perhaps a secondary application to aid in the fulfillment of the objective.

The first approach that was conceived involved a single Android App and a server to provide persistence. The single App would support all three user categories by serving content dynamically based on role. Although an entirely valid approach, it was believed that a mobile application may not be suitable for use by a Taxi Controller in an office environment.

Bearing in mind the realizations gained from the 'single app' approach, a desktop application to support Company Admins was considered. This would likely involve a fairly simple desktop application being developed alongside the mobile application to consume a common backend. This approach certainly addresses the concerns raised by the previous approach. However, it would limit the use of the application to an office environment. There are some taxi companies that operate with shared roles, sometimes the controllers also act as drivers.

The final and accepted approach was to support Customers and Drivers with an Android App and to support Company Admins with a Responsive Web Application. Both of which would still be served by a common backend. This was believed to be the most flexible and fitting solution. It would allow the application to be used from an array of devices and potentially allow multiple systems to be used at once in some very specific use-cases.

1.2.3 Deliverables

As discussed in Section 1.2.1, broadly, the primary objective of this project was to create a fully functioning Taxi booking platform. Six tangible deliverables can be drawn from

this objective.

1.2.3.1 Full set of requirements in the form of User Stories

This deliverable would contain a full set of formally written user stories. Each story would contain four parts: title, description of the feature, 'essential' or 'non-essential' flag, and the acceptance criteria. Below is an example of a User Story for this platform:

Customers can cancel a booking at any time (Essential)
As a customer, I want to be able to cancel my booking so that if I no longer need a taxi I don't waste the taxi company's time driving out.

Acceptance Criteria

- If a customer selects to cancel a booking they are prompted for a confirmation.
- If the customer confirms the cancellation their booking is updated.

Together, all of the 'essential' stories would form the minimum viable product for this project. With various systems partially fulfilling the story, until it has been fully completed. To consider the above example, the cancellation of a booking must be supported both within the API and the mobile application.

1.2.3.2 A fully functioning REST API to behave as the 'core' of the system

Given that each application within this platform would consume the API, it would need to partially meet the majority of the requirements laid out by the User Stories. Therefore, this deliverable would involve the creation and deployment of an API that can fully support each requirement of the platform. Considering the importance and use of this deliverable, security will most certainly be a primary concern.

1.2.3.3 User Interface prototypes and style documentation for the user-facing systems within the platform

This deliverable would involve the creation of UI prototypes and a decision about the general look and feel of the platform to be documented. This would allow the development of the user-facing systems to be informed and consistent with the rest of the platform.

1.2.3.4 A user-friendly and intuitive Android Application to consume the API and be used by Customers and Drivers

In order to support all of the requirements and API functionalities concerning Customers and Drivers, a 'front-end' would need to be created to allow these users to interact with

the platform. This would be the primary user-facing application within the platform. It would also require support for allowing the creation of users and enforce access control for unauthenticated users.

1.2.3.5 A mobile-friendly Web Application to consume the API and be used by Company Admins

Similarly to the mobile application, a system would need to exist to allow Company Admins to interact with the platform. This system would need to satisfy all of the requirements relating to Company Admins. In order to do this, it would need to strictly enforce role-based access control to ensure only registered Company Admins can make use of the system.

1.2.3.6 A set of automated and manual tests to verify the correctness of each system and their success at fulfilling the set requirements

As with any project, correctness and accuracy are imperative. Therefore, thorough testing of the platform is undoubtedly a key deliverable. The testing strategy would also ensure accurate acceptance measurements of the requirements as outlined in each user story.

1.3 Process

Having narrowed down the options for development methodology to those discussed in Section 1.1.1.1 and researched each option it was decided that an adapted version of Extreme Programming would be used for the project. This section will discuss the specific practices and methods that will be applied in this project's development methodology.

Given the restrictions that a plan-based approach such as SDLC would have on the project, with specific regard to the requirements it was decided the approach had to be agile. Due to the relatively small scale and timeframe of the project, it was also thought that it would be very beneficial to focus on high quality, working software over the documentation required by SDLC.

Feature Driven Development was another agile approach that was considered. Although with adaptations, it could have been very well fitted for a single person project, it was apparent that FDD was intended for use in a team-based project. This is seen in its emphasis on roles. Therefore, it was clear that XP was the most appropriate agile methodology for the project.

XP was selected in the hope that its use would help mitigate the increased risk associated with a project where there is a lack of experience and a lot of learning to do with the technologies used. Due to this increased risk, it was believed that creating a static requirement list would prove difficult and likely be very flawed. Therefore, given that XP is an Agile Approach and is intended to support changing requirements it fit perfectly.

1.3.1 Practices

The following practices are all considered primary practices within Extreme Programming [**?**, Ch. 7]. They have been selected due to their relevance to the project. The following inapplicable practices have been omitted: Sit Together, Whole Team, Informative Workspace, Pair Programming, Test First. These practices were aspects of XP targetted towards team-based projects and larger workspaces.

1.3.1.1 Energized Work

Although mainly focused on software development teams to ensure a healthy workplace and work/life balance, this practice also suits individual projects very well. By following this practice, work is undertaken at the right time and location for the right length of time, without burning out. This requires that steps be taken to ensure a focused state can be entered to complete work without distractions.

1.3.1.2 Stories

Stories are a key practice in most agile methodologies, in this project user stories were written for each desired functionality. Each story was labeled as either 'essential' or 'nice to have', this ensured that work was prioritized correctly and essential aspects of the system were built in a timely way. Stories were written on physical, coloured cards and placed on a corkboard in the main work area of the project. They also existed electronically on a Trello [**?**] board, this allowed progress to be shared with the project supervisor and allowed work to be completed away from the main work station. There was also be a static document created with each high-level story written formally, this can be found in the Appendices.

1.3.1.3 Weekly Cycle

Weekly Cycles are synonymous with iterations in other agile methodologies. In this project weekly cycles ran from Saturday-Friday, meaning each cycle began on a Saturday and ended on a Friday. The first day of the cycle mainly consisted of planning and choosing which stories to tackle; there was also an overall reflection and a look at the project's current progress. The goal was that by the end of the cycle all stories had been realized and are now implemented, integrated and tested features within the system. The final day of the cycle involved writing an entry into the project's weekly blog [**?**] reflecting on the week's work and documenting the goals for the next week. By using a weekly cycle, software was constantly being produced, allowing for a regular opportunity to analyze and get feedback.

1.3.1.4 Quarterly Cycle

Quarterly Cycles are synonymous with releases. Given the scale of this project (roughly 12 weeks), a quarterly cycle consisted of 3 weekly cycles. This meant that the first quarterly cycle passed just in time for the mid-project demonstration, thus ensuring there was at least a partially working system to be delivered. At the beginning of each quarterly cycle, a plan was drafted for which stories should be completed in that cycle. The quarterly cycle plan was re-visited briefly at the beginning of each weekly cycle.

1.3.1.5 Slack

The purpose of slack is to add low priority or non-essential stories to weekly and quarterly cycles. This allowed tasks or stories to be discarded if time became an issue, thus accounting for inaccurate estimates.

1.3.1.6 Ten-Minute Build

The 10-minute build practice is intended to enforce a quick and easy build process, if a build process is long and arduous it is less likely to be done often. Therefore an easy, automatic build process ensured that builds were done frequently, allowing for more feedback and less time between errors.

1.3.1.7 Continuous Integration

Continuous Integration supports the idea of a fast build process. By immediately testing and integrating code systems into the overall system, integration issues were caught much sooner and with less consequence. Continuous Integration also encouraged and enforced frequent and reliable automatic testing.

1.3.1.8 Incremental Design / Refactoring

Incremental Design suggests that some spike work be done up front to understand how much effort is required and any issues that might arise. This approach reduced risk and allowed for a better estimation for a given story. This practice allowed for informed design decisions to be made when necessary and be made with the most current information available. The practice of incremental design also calls for the use of frequent refactoring to ensure correct design.

Chapter 2

Design

This chapter will discuss the overall architecture and higher level aspects of the platform's design. It is only intended to provide an overview of the design. A more in-depth discussion of specific design decisions will be presented on a per-iteration basis later in the report.

2.1 Overall Architecture

The platform is comprised of 3 key systems: a REST API that interfaces with a MongoDB database; an Android App to consume the API and serve content to Customers and Drivers; a Responsive Web App to consume the API and serve content to Company Admins. The overall architecture of the system can be seen in Figure 2.1.

Both the Android and Web App are categorized as user-facing systems. Meaning the platform's users are never intended to interact with the API directly, but rather through one of these two Applications (depending on their role). This also means that the two user-facing Applications never interact with each other, they exclusively send and receive data from the API.

Data received from the API is in JSON format, which is natively deserialized by Javascript and Typescript. However, Android's GSON library is used to support JSON conversion to and from 'plain old Java objects' (POJOs).

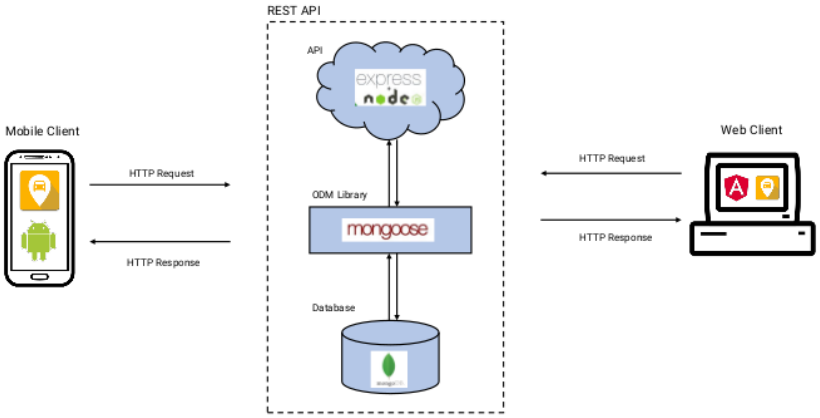

Figure 2.1: The Overall Architecture of the Platform

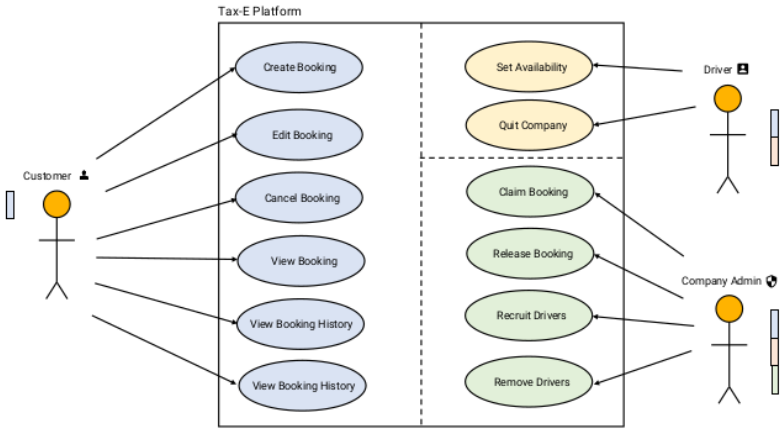

Figure 2.2: The Use-Cases of the Platform for each User type

2.2 REST API Architecture Overview

Broadly, the REST API is a modular system split up into the following: models, services, controllers, routes, middlewares, and helpers.

The model module contains all the models for the system. These directly map onto a collection within the MongoDB database. Some models may reference each other using MongoDB ObjectIDs, comparable to SQL's foreign key constraint. Any model file can be referenced from anywhere within the system and used as a 'type' to initialize objects.

The services module is the lowest level of abstraction for interfacing with the backend database. It exposes several necessary public functions to the rest of the system. Typically, it will implement CRUD functions via mongoose. The service files are only intended to be accessible from their respective controllers.

Controllers are the next level up from services in the API's abstraction layers. Controller

files are called by the API's router to pass information from the HTTP request to the required service. Then execute the relevant middleware. The functions within a controller file are typically very simple and simply serve as an intermediary between the router and the services themselves.

Routes are used to capture information from the HTTP request and pass it to the controllers so the relevant operations can be completed (via the service). These include retrieving values from request headers, the request body, and any URL parameters or queries. The various layers of abstraction within the API can be seen in Figure 2.3

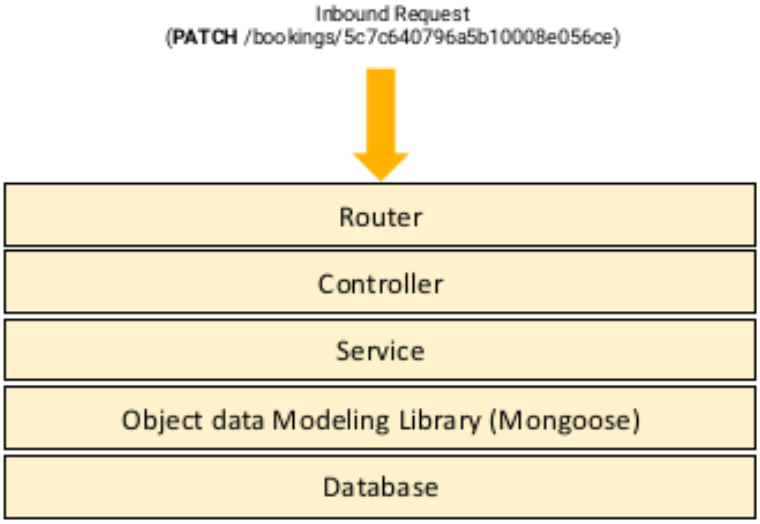

Figure 2.3: The layers of abstraction within the API.

Several middleware files were created for the API, middleware is executed by the router before any code from the controller is executed. Middleware can be configured to run on every route or to be executed on a per-route basis. For example, the error handling middleware needs to be executed on every route. However, a piece of middleware such as access control or authentication may only need to be executed on certain protected routes. The order of execution of middleware for a route that requires every piece of middleware is depicted in Figure 2.4.

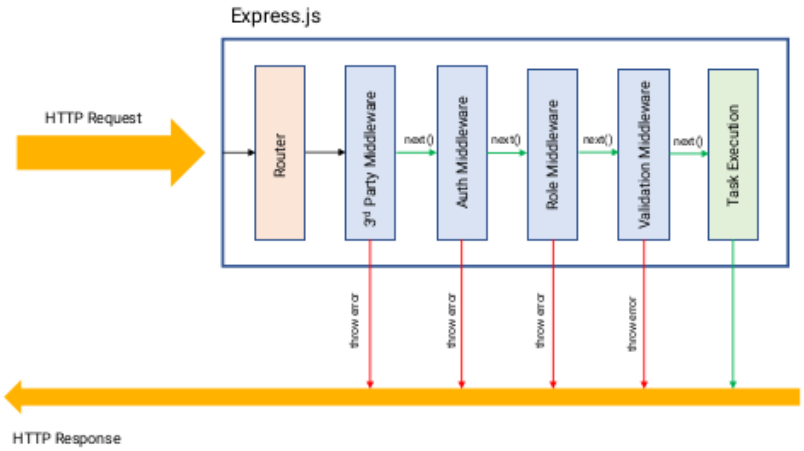

Figure 2.4: Diagram to display the order of execution of middleware.

Helpers do not have a designated place in the order of execution of requests. Rather, they are intended to be used by other modules within the API to perform frequent, utility tasks or to enumerate types. For example, there are two enumeration types within the helper module 'role' and 'status'. These are used across many different modules within the API to reduce code duplication.

2.3 Data Model

The data model for the platform is fairly straight forward, the simple, flexible nature of MongoDB certainly assisted in this simplicity. To preface, a 'collection' in this context is comparable to a 'relation' in a traditional relational DBMS. A 'document' is comparable to a 'tuple' or 'record', and a 'field' is comparable to a 'column' or 'attribute'.

The data model contains three collections 'User', 'Booking', and 'Company'. All collections have unique Object IDs and each collection references another. The 'User' collection is likely the largest, it is a general model for Customers, Drivers, and Company Admins. The decision to use a single, larger model for all users, as opposed to a specific model for each type of user was made early on, due to its simplicity. This is implemented by leaving the type-specific fields as null if they are not applicable to the user. A user's type is stored using the 'role' field. The data model is depicted in the Entity Relationship seen in Figure 2.5.

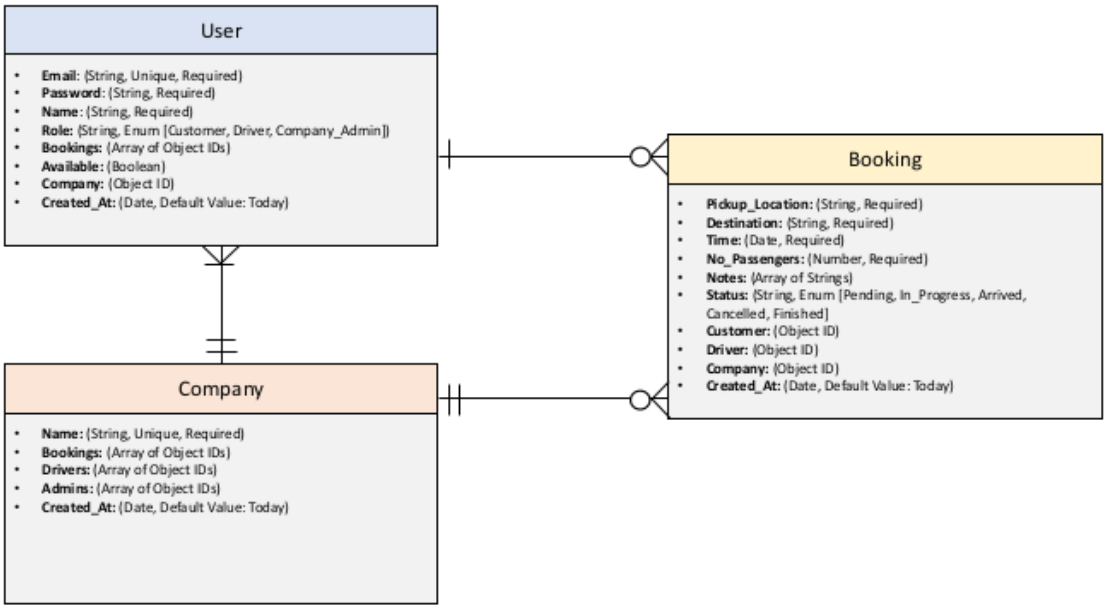

Figure 2.5: An Entity-Relationship diagram to accurately depict the Data Model used

2.4 Android App Architecture Overview

Much of the Android App's design is boilerplate, with the main package split into java code and app resources. There are no significant differences between the app's build variants as it is intended to offer different functionality dynamically, depending on the user's role. It was considered early on to provide two different variants of the app, one for customers and another for drivers. However, as the differences between the two app's functionalities would be very minor it was decided that a single app would be the best approach.

The app's main java package is comprised of five sub-packages: model, network, ui, util, and viewmodel.

The model package contains 5 model classes, each representing a type of JSON response that can be received from the API. They are required to enable serialization/de-serialization of JSON objects. The 'User', 'Booking', and 'Company', and 'Response' models are the ones uses most frequently. With 'LoginResponse' only being used in a single special case. The Response model is returned anytime the API is expected to return a generic response with no particularly important data. It is very useful for Error Handling for retrieval of JSON values.

The network package contains the classes required to interface with the API using Retrofit. It contains the 'RetrofitInterface' interface, which is comparable to the Repository interface that would be used in Android's data persistence library. The 'NetworkUtil' class has several overloaded methods that each return a differing instance of the RetrofitInterface. For example, if the method is called with no parameters it returns a Retrofit instance with no HTTP authentication headers. Whereas, if the method is called with an access

token as a parameter, it returns a Retrofit instance with the access token in the HTTP headers.

All of the app's activity and fragment classes are located in the ui package. The package itself is split into 5 sub-packages, which are intended to relate to each of the app's screens and dialogs. The app has two main activities Authentication and Main. This is how access control is enforced in the app, the user can only access the main activity if they have successfully logged in via the authentication activity. There are of course other activities within the app, but they are always called on top of one of the two main activities.

The util package contains all of the classes that contain shared code that is intended to be used all across the app. This includes the app's error handler, validator, shared preference manager, customer de-serializers, enum types, and more. The use of a util package with these classes is intended to reduce code duplication.

The Model View ViewModel (MVVM) design pattern is implemented in this app via the viewmodel package. The view models within this package provide a layer of abstraction between the networking and UI classes. In the case that the API's routes were updated, supporting this update in the app is made considerably easier using view models. Therefore increasing maintainability and encouraging good object orientation practices.

2.5 Web App Architecture Overview

The Web Application is likely the simplest system within the platform. Similarly to the Android Application, the main directories are fairly similar to a boilerplate Angular Application. The main directory of the app contains seven sub-directories: '_guards', '_helpers', '_models', '_services', 'components', 'layouts', 'modules'.

The '_guards' directory currently contains a single guard file to enforce authentication within the app.

The '_helpers' directory contains two HTTP interceptors, an error interceptor, and an access token interceptor. The error interceptor is intended to intercept incoming responses from the API and handle any errors appropriately. If the application was of a larger scale, it would likely be more appropriate to also have a designated error handling file. However, due to the scale of the application currently, this solution was selected due to its simplicity. The 'jwt' interceptor is intended to intercept outgoing requests to the API and append the user's access token to the request headers. This means that the token doesn't have to be manually added each time a request is made.

The '_services' directory contains all of the services required by various views. The services are intended to add a layer of abstraction between the view and the HTTP requests. There is also a service file for displaying notifications in the view, it works by injecting a HTML element into the view and running jQuery code to animate the element.

All of the app's angular components reside in the 'components' directory. An angular component represents a 'page' within the app, with components belonging to a module. For example, the 'drivers' page is a component that belongs to the 'main' module. Each component contains a view file, style file, and controller file. These are typically html,

scsss, and typescript files respectively.

Any recurring layouts within the app reside in the 'layouts' directory. This includes things such as headers, footers, navbars, etc. Similarly to components, each layout has a view, style, and controller file. However, the layout directory is treated as its own Angular module.

Lastly, the modules directory contains the app's two main modules 'auth' and 'main', these modules create a scope in which components can exist. Each module requires a router and module file, in addition to the files required by standard components. A module's view file will typically contain a placeholder for whichever component the module's router serves.

2.6 User Interface

The User Interface for both the Android Application and the Responsive Web Application is intended to conform to Google's Material Design guidelines [**?**]. By attempting to implement Material Design in all of the platform's user-facing applications, a consistent and familiar style is created. Together with a consistent colour scheme and typeface, it strives to provide the best user experience.

Implementing Material Design in the Android Application was made trivial with the tools that Android Studio provides. Once a colour scheme was selected, the hex values were simply entered into the App's relevant resource file. Any components subsequently created were coloured accordingly. A bottom navigation bar was used for the app's main navigation, the bar is rendered differently depending on the user's role.

Due to time restrictions, a third party UI template was used for the Web App. The template, 'Material Dashboard Angular' [**?**], uses Bootstrap, Sass, and HTML5 to create a productivity dashboard template that conforms to Google's Material Design. The template also provides some boilerplate Angular files for routing and layout displays, these were adapted from the provided code for the needs of the application.

2.7 Technologies Used

As the developed platform would contain several systems, it was clear that a diverse array of technologies would need to be adopted. Both to aid project management and for use in implementation. This section will discuss the selected technologies for this project.

2.7.1 Implementation

With regards to the technology that would be used to develop the REST API, it was a close decision between Rails 5 and Node.js / Express.js. Although Flask and ASP.net were most certainly fit for purpose and had plenty of documentation to get started. There was a distinct lack of experience in the underlying languages (Python & C#). The project

supervisor had a level of expertise in Rails, this would have been an advantage for the adoption of this framework. However, due to personal preference and a slight bias towards Javascript, Node.js together with Express.js was selected. This choice showed great promise, with a considerable amount of online resources and available testing libraries. It would also mean the development of the API would integrate seamlessly with Node.js's default package manager (NPM). This would allow access to hundreds of thousands of javascript packages.

Having researched both Angular and React it was clear they were both equally appropriate for this task. However, Angular's traits were increasingly more appealing after in-depth research. It offered a substantial amount of in-built functionality compared to React. Examples of these functionalities are an In-built XSS protection, a native Router (eliminating the need for a third party library), a form builder, and SCSS compiling. It also supported Google's Material Design, through pre-built components, this was ideal to ensure consistency between the platform's applications. The only considerable drawback was that Angular is based in Typescript, a syntactically strict superset of Javascript; this was an unfamiliar technology. However, given the considerable advantages and previous experience with Javascript, the benefits certainly outweighed the drawbacks. It should also be mentioned that the adoption of Angular would loosely complete the project's adoption of the MEAN stack [**?**].

2.7.2 Data Persistence

Although the research was certainly a key part of making a fully informed decision; a JSON-like document-oriented DBMS seemed like an obvious fit, having already decided to use Node.js for the API. Therefore, MongoDB was selected. One disadvantage of a NoSQL solution is that complicated queries are typically considered more difficult compared to a traditional SQL DBMS. However, given the simple nature of the queries for this project, this didn't seem overly relevant.

2.7.3 Development Environments

Having researched JetBrains' WebStorm, it was decided it would be a better fit for the development of both the platform's REST API and Web Application. With substantial language-specific support, code completion, syntax highlighting, and a preview of markdown files, it was an easy decision. WebStorm also provides in-built integration with docker, GitHub, and NPM.

2.7.4 Project Management

After researching many project management solutions, a decision was made to use Docker for containerization of the REST API. This conclusion was mainly motivated by the ability to easily replicate the development environment across multiple machines. Due to Extreme Programming being the project's chosen development methodology, a Continuous Integration tool had to be selected. Having researched several options, it was eventually

decided that TravisCI would be used. It was by far the most lightweight and easy to understand tool of those researched. It seamlessly integrates with GitHub through the use of a single YAML file and OAuth, making adoption relatively easy for inexperienced users.

2.7.5 Deployment

In order to make an informed decision regarding the deployment method for the API and Web Application, sample applications were deployed using both Heroku and AWS. It was discovered that the process was actually a lot more in-depth than initially anticipated. Together with having to register an account and provide banking details this was somewhat discouraging. Therefore the API and Web Application were both deployed to a home-based headless ubuntu-server box. The API is deployed using Nginx as a reverse proxy and Node.js's process manager, PM2. The Web Application is deployed using an Apache web server.

Although there are certain security implications with a 'homemade' deployment solution. For example, the lack of reliable SSL encryption and physical security concerns due to the location of the server itself. The deployment is only intended for demonstration purposes and will likely only contain fictitious test data. If the platform were ever to be deployed for production, a fully supported commercial solution would be sought out.

Chapter 3

Implementation

The implementation phase of this project spanned across 9 weeks. Given that the software development methodology adopted was Extreme Programming. The platform was implemented with an iterative approach or a series of 'Weekly Cycles'. In this chapter, an overview of the work done in each weekly cycle will be provided, including a discussion of any problems encountered and how they were overcome.

As discussed in Section 1.3, at the beginning of each week it was decided which stories would be completed, with some non-essential stories included to add slack.

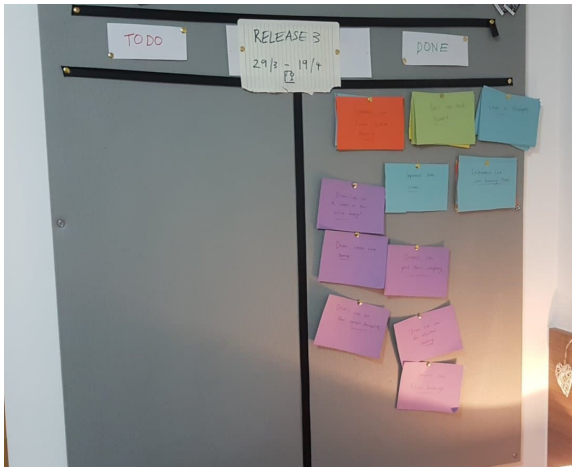

Figure 3.1: The physical Story Board used throughout the project

3.1 Week 1 (15/02 - 22/02)

All the required prototyping for the platform took place in the first week of development. This included UI Prototypes for the Android App's screens, a prototype Authentication API, and a prototype Android App to consume the Authentication API.

3.1.1 Prototype Authentication API

During the creation of the prototype Authentication API, research into third-party authentication libraries was also done, with libraries such as Okta [**?**] and Passport.js [**?**] being considered. It was eventually decided, however, that a custom solution would allow for more fine-grain control over the API's authentication and access control.

3.1.2 Prototype Android App

Using the created API, the next task was to create a prototype Android App to consume the API. A major part of this task was to set up HTTP requests and JSON response deserialization. The second tutorial in Raj Amal's series of walkthroughs [**?**] proved essential in understanding Retrofit, rxJava, and Google's GSON. The prototype App allowed users to register, login, and view their profile. It enforced access control to ensure unauthenticated users couldn't view the main screen of the App. The Android aspect of this implementation was fairly trivial due to previous experience. For example, storing the user's access token in shared preferences and limiting access by requiring the presence of the token.

3.1.3 User Interface Prototypes

Now that there were prototypes in place to lay the foundation, it was time to prototype the User Interface for the actual platform. The first step in this process was to establish a general style and branding. This involved deciding on a colour scheme, choosing a typeface, and creating the first draft of the platform's logo. The remainder of the work involved actually creating the prototype for each of the App's screen. Prototypes were created with varying degrees of fidelity, the first pass simply involved sketching out some of the screens on paper. The final produced prototypes used Material Design components provided for Adobe XD [**?**]. This meant that the produced prototypes would closely resemble the finished product produced in Android Studio. Examples of some of the created prototypes can be found in the report's appendices.

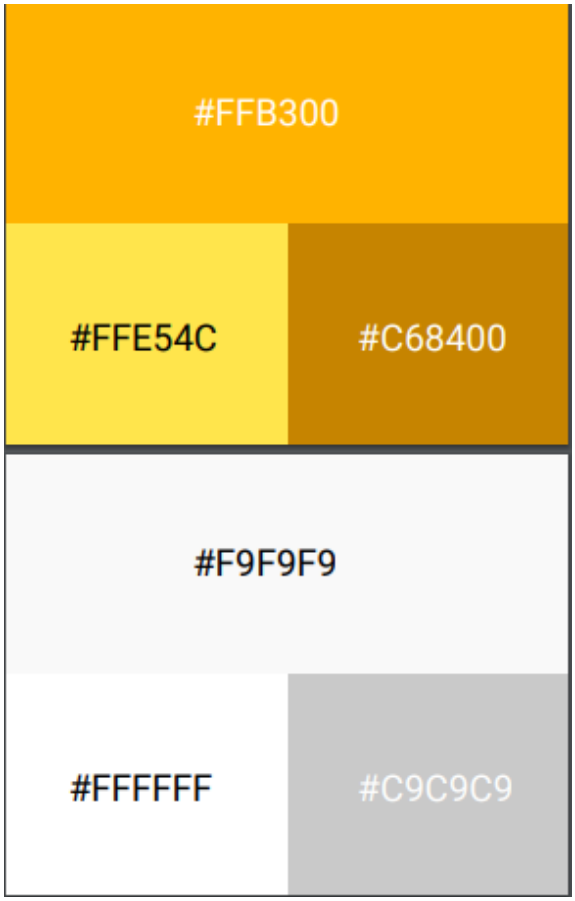

Figure 3.2: The Colour Scheme Selected for the Platform

3.2 Week 2 (23/02 - 01/03)

The focus of the second week of implementation was to get the REST API up and running and supporting all User related tasks: Registration, Authentication, and Account Management.

3.2.1 Node.js Project Creation

The very first step to implementing the API was to create the Node.js project and configure it to run in a Docker container alongside MongoDB. In order to create the Node.js project. An excellent GitHub repository [**?**] was discovered that outlined the best practices and standards in JavaScript projects.

3.2.2 Containerization of Project

Containerization of the API itself proved to be very straightforward, as it is a well-documented process and simply involved the creation of two files: Dockerfile and docker-

compose.yml. The difficulty arose when attempting to connect the API container with MongoDB in order to persist data. The solution appeared to be to include a second container dedicated to MongoDB. This was working and a successful connection to the database could be made from within the API. However, in order to support data persistence after the containers were terminated, Data Volumes were required. A very helpful Medium tutorial [**?**] was used to correctly set up the data volumes and the setup was now working as intended.

3.2.3 Authentication

During the prototyping phase of development, an authentication API was created as a piece of spike work. Alongside this spike work, research into third-party authentication libraries was also done, with libraries such as Okta [**?**] and Passport.js [**?**] being considered. It was eventually decided, however, that a custom solution would allow for more fine-grain control over the API's authentication.

In order to provide the best security for the API, a token-based approach to authentication was used. It was also believed this would make the consumption of the API easier for the mobile application, by removing the need for sessions; thus making the API truly RESTful. The core principle behind this approach is that the user would send their email and password to a specific route within the API via Basic Authentication (Base64 encoding). The credentials would then be decoded and verified against the password hash stored in the database. If valid, the API would then provide the user with a 24-hour access token, to be sent in the header of any subsequent requests. Although initially requiring a bit of thought and research, implementation of this approach didn't provide any considerable difficulties. An excellent and in-depth tutorial written by Raj Amal [**?**] was used as a reference to assist in the implementation.

3.3 Week 3 (02/03 - 08/03)

Testing of the API's current routes and consumption of the API by the Android App was the main focus of the third week.

3.3.1 API Testing

With the API already running inside a Docker container, the testing process was much more portable. Two testing libraries were used: Mochajs [**?**] as the primary testing framework and Chaijs [**?**] as the assertation library. Whilst gaining familiarity with the testing libraries, Samuele Zaza's tutorial on Scotch.io [**?**] was invaluable. The API was tested on a per-route basis, with several different cases for each route being tested. There was a bit of difficulty getting the tests to run seamlessly within the Docker container in TravisCI's environment. However, improved Docker Compose configuration eventually solve these issues, with assistance from Joe Cieslik's article in Hackernoon [**?**].

3.3.2 Implementation of Android App

Consumption of the API by the Android App was where the bulk of the work took place during this week of development. Having already created a prototype App to consume an API using Retrofit and rxJava, much of the concepts and the boilerplate code was already in place. The difficulty came with refactoring the code to ensure reliability and good design practices. A major part of this was the implementation of the Model, View, ViewModel design pattern. An article on Medium.com by Ahmad Shubita [**?**] was used to get to grips with the design pattern and its implementation via Retrofit.

Another essential aspect of the consumption of the API by the Android App was Error Handling, this is especially true for the registration process. In order to fully support internationalization/localization within the app, it was important that English error messages from the API not be displayed directly to the UI. Rather that error responses from the API come with four attributes: 'name' - the name given to a particular error; 'code' - an arbitrary number associated with a given error; 'message' - the error's name displayed in a more human-readable way; and 'description' - a brief description of the nature of the error. This way, the mobile app that consumes the API can simply read the code associated with a given error and fetch the matching string resource. One notable downside to this approach is that it increases the maintenance required. For example, if updates are made to the API that affects its error handling (a new error is added), this change must be accounted for in the App's Error-Parser.

[INSERT ERROR-HANDLING DIAGRAM]

3.4 Week 4 (09/03 - 15/03)

The fourth week of development was mainly focused on preparations for the mid-project demonstration. Therefore, implementation of the platform itself was halted.

3.4.1 App Deployment

A major part of the preparations required for demonstrating the app's progress so far was the deployment of the API. Up to this point, the App had only been usable within an emulator ran on the same machine as the Node.js application. In order to properly demonstrate the app's development, it had to be able to access the API from an actual mobile device.

As discussed in Section 2.7.5, it was decided that the API would be deployed on a local server box using the Node.js process manager and Nginx's reverse proxy. Although there are certainly benefits to using Docker in production, for example, reduced setup times and improved portability. Much of the research done heavily discouraged its use if inexperienced, especially in a self-hosted environment. Therefore, the decision was made to deploy the API without docker, however, it will continue to be utilized for development and testing environments.

The majority of the steps required to deploy the API were fairly straightforward and very

well documented [**?**]. As discussed previously, there are certainly security implications associated with the chosen deployment approach. Such as lack of decent support for SSL, physical security concerns, inexperience with system administration, and the server itself being used for other, unrelated web applications. However, as a solution for demonstration purposes with fictitious personal data, it served its purpose excellently.

3.5 Week 5 (16/03 - 22/03)

With authentication already completed and supported both in the mobile app and on the REST API, the next logical step was to work on the implementation of booking creation. This task took surprisingly less time than anticipated, therefore there was time to complete some of the 'slack tasks'.

3.5.1 Booking Data Model support whithin the API

Having already implemented user authentication, creation, and management within the API, much of the foundations required for booking creation, management, and viewing were already in place. The main steps required were the following:

- Create booking model
- Interact with booking model via service
- Create a controller to allow the router to interact with the service
- Implement all the necessary routes for bookings (CRUD)
- Add all the required input validations and error handling
- Write unit tests for each booking route

The booking model, service, controller, and router are all fairly standard and similar to those for the user data model.

Input validation for booking creation was fairly trivial, with the exception of the 'time' field, as it had some special constraints:

- Booking time cannot be further than 3 hours in the future
- Boking time cannot be sooner than 10 minutes away
- Booking time cannot be in the past

In order to easily compare time values, the ISO 8601 time stamp had to be parsed into a plain Javascript date object.

3.5.2 Booking Creation support within the Android App

Similarly to the API, having full support for the user data model within the App already made the implementation of booking creation very straightforward. Aspects of the App such as error handling and validation had to be updated to support the changes in the API.

The main challenge for this aspect of the implementation was the UI. A dedicated Activity was created for booking creation, with standard text inputs for pickup location, destination, and notes. The number of passengers and booking time fields each have a custom dialog that prompts the user for an input. The number of passengers dialog is a custom dialog that contains a number-picker, with the parent activity implementing its value listener. Therefore, whenever the value of the number picker is changed, the parent activity is kept updated. The time picker, however, is a default android time picker with the dialog all created programmatically. Therefore, tracking the selected value was trivial.

An interesting challenge faced when implementing the booking creation activity was deciding how to force the App to navigate to the 'booking viewing' screen upon successful creation of a booking. A considered option was simply to force the navigation whenever the booking creation activity was closed. However, this would prompt navigation even when the booking creation was canceled, which was an undesired behavior. The solution eventually implemented was to launch the booking creation activity with Android's 'startActivityForResult()' method. This would allow the activity to provide a return code when exited, meaning the activity could either exit with success or failure. If exited with success, a toast message is shown and the navigation is changed.

3.5.3 Account Management within the Android App

As mentioned at the start of this section, the primary tasks for this week's implementation took less time than anticipated. Therefore there was time to implement some of the less essential features.

Given that the support for account management was already supported in the API, it made sense to create the UI for account management. The main features of the account screen are a circular avatar with the user's first name initial and a list of account actions. All of the account actions were implemented with the exception of changing password. This required further security precautions within the API.

3.5.4 Support Dynamic Orientation within the Android App

The next functionality to be addressed was landscape support, up until this point the app had been locked into portrait mode for simplicity. Most screens ported over to landscape fairly easily, given that the Constraint Layout was used for most screens and is typically well supported in both landscape and portrait modes. However, the account overview screen required its own layout file specifically for landscape mode. Luckily, Android makes this very easy with the use of the 'land' suffix in resource folder names. A new

resource folder named 'layout-land' was created to contain the landscape specific layout.

Allowing the app to be rotated did present some unexpected complications, specifically concerning asynchronous API calls. When a fragment or activity is destroyed, any outstanding asynchronous subscriptions are terminated (to avoid dangling). This does, however, mean that whenever a view is destroyed and re-created due to orientation change, any outstanding API calls are discarded. A simple, yet arguably lazy solution to this issue was simply to lock the orientation in place whenever an API call is made, then to unlock the orientation upon completion.

3.5.5 Welsh Language Support within the Android App

Having gone to the effort to support internationalization/localization within the app, it made sense to add string values for another language. This was trivial and simply involved translating all the existing string values and adding them to a '-cy' resource directory.

[INSERT SCREENSHOT]

3.6 Week 6 (23/03 - 29/03)

This week's development was almost entirely based on the Android App. It primarily involved full support for bookings via implementation of the following: Active Booking Overview, Booking Cancelation, Booking History. Password changing was also revisited.

3.6.1 Active Booking Overview

The user's currently active booking could be retrieved by simply retrieving their most recent booking. This is guaranteed to be accurate as new bookings cannot be created whilst an active booking already exists. The booking overview screen simply contained a card view with information populated by the API request. The card view is populated using Android's Data Binding library, allowing for stricter conformity to the MVVM design pattern. Unfortunately, the view does need to be manually refreshed to fetch up to date data. However, this is made intuitive through the use of Android's SwipeRefreshLayout. The view is also refreshed whenever the user changes navigation.

3.6.2 Booking Cancelation

After implementing the booking overview screen, adding support for booking cancelation was trivial. A text-button was added to the bottom of the card view. Pressing the button prompts a simple confirmation dialog to the user. If confirmed, a request is sent to the API to update the booking's status to 'Cancelled'.

3.6.3 Booking History and Custom JSON De-Serialization

As seen in Figure 2.5, the way bookings are stored under a user's record in the database is as an array of MongoDB Object IDs. Therefore when a user is retrieved from the API, their bookings are shown simply as a list of Object references. Whereas the booking history list is required to display a limited amount of information about the booking. Although technically each booking could be retrieved fully by making a GET request to the API for that specific booking, this is an undesirable solution and would require significantly more API requests. Therefore a new route was implemented within the API specifically to retrieve a full list of a user's bookings. Mongoose's 'populate' function proved very useful for this purpose.

The newly created route does not populate the 'driver' and 'customer' fields of the booking. Meaning they are still represented as strings (object IDs), whereas the booking model within the App stores the driver and customer fields as User objects. This caused some unexpected complications when it came to de-serialization of the JSON booking objects. When the booking JSON object was returned from a GET request to the /bookings/ route, it would be de-serialized just fine (due to driver and customer being populated). Whilst, if the booking JSON objects were returned from the /users/id/bookings route, a de-serialization exception was thrown. Upon further research, it was discovered that custom GSON de-serializers could be created that accounted for dynamic JSON fields. Although it took some time, this was done relatively easily and the problem was resolved.

The remainder of the implementation for booking history went off without a hitch, a boilerplate Recycler View Adapter was created to handle the list of bookings. With a click listener to open an Activity and display the booking's full details. The booking overview fragment created earlier in this section was re-used here.

3.6.4 Password Changing

All of the UI elements required for supporting password changing were already implemented. An update to the user updating route on the API was required in order to force the user to provide a valid previous password before the changes could be approved.

[INSERT BOOKING OVERVIEW SCREENSHOT]

3.7 Week 7 (30/03 - 05/04)

This week's primary focus was adding support for the Driver role within the platform. There was also some time to implement the app's home screen and re-assess how notes were being handled within bookings.

3.7.1 Supporting the Driver Role

In order to fully support drivers within the API, a few changes needed to be made. The User model had to be updated to include fields for the driver's availability and company. This was fairly straightforward and simply required the addition of the fields to the model file and support for their update in the PATCH route. Some extra authorization checks were required for the user PATCH route, to ensure a user would only be able to change their own role under specific circumstances (resignation from a company).

All other aspects of the user model were already designed to accommodate users of different roles. For example, the 'bookings' field required no change as it could serve different purposes depending on the user's role.

It was also predicted that certain routes in the future would need to be 'role-protected'. Therefore a very simple role-based access control middleware was created. The middleware could be applied to any route and either a single role or a list of roles could be authorized for that given route.

3.7.2 Redesign of the Booking Notes System

Having initially opted that a booking's notes be a single, optional string. It was decided that the support for communication this provided was very minimal. It's only purposed so far was to allow a Customer to make 'special requests' along with their booking. However, it was realized that a collection of notes within a booking that could be added by both the Customer and the Driver could potentially serve as a very simple communication mechanism. Notes would also be added by the system itself to inform users when changes to their booking have occurred, for example, status changes.

Implementing the new booking notes system on the API was very straightforward, it simply required a different type within the Booking model; together with support for appending notes to a Booking via its PATCH route.

3.7.3 The App's Home Screen

The last bit of development for this week of implementation was the fairly small task of implementing a home screen for the app. The layout and contents of this screen were decided a long time ago in the prototyping phase. Therefore the only required work was creating the screen's XML layout for both portrait and landscape orientations and implementing the API requests required. The information needed for the home screen required two requests be made to the API: one for the user's information (name, role, number of bookings) and another for the user's most recent booking information.

3.8 Week 8 (06/04 - 12/04)

This was the last week of development on the Android App, quite a lot of functionality was added. Primarily, complete support for Drivers within the App and updates to the booking overview to support the new notes system. In addition to developments within the App itself, it was also required that preparation work be done in the API, to support the Company Admin Web App.

3.8.1 Supporting the new Notes System within the App

In order to support the new notes system that was developed in the previous week, a new Activity was created to serve as a 'notes view'. This new activity also contained a Floating Action Button that opened a text input for the user to append a new note to the booking. The only minor obstacle encountered in this implementation was that notes were simply an array of strings with no date stamp. Therefore in order to display them in semi-chronological order, the returned array was reversed (as they were being appended to the back of the array within the database).

3.8.2 Implementing all the Driver Featuress within the App

In order to implement all the functionality required for drivers within the app, surprisingly little work had to be done. The main tasks involved: The addition of an availability switch; an option to resign from their company; removal of 'booking creation'; support for viewing several active bookings; and support for updating a booking's status.

The addition of an availability switch and resignation option was fairly straightforward, the most difficult aspect was displaying an alternative 'home screen' and 'account overview' depending on the user's role. This was done by storing the user's role in the App's shared preferences and using this value to determine what version of each dynamic screen was loaded.

The API had already been updated to prevent drivers from creating bookings, however, the navigation option and booking creation screen were still very much available. It was decided that simply removing the 'new booking' option within the app's navigation for Drivers was the best solution. This was achieved with the addition of a second XML navigation menu, the required menu was then loaded into the Navigation Bar as needed. If the user opted to resign as a driver from within the App itself, the dynamic aspects of the app such as the Navigation Bar, Home Screen, and Account Screen were automatically changed to 'Customer Mode'. However, if the user was forcibly 'demoted' by their Company's Management, the App would not automatically update the required screen.

Although still not ideal, the best conceivable solution was to query the user's role whenever a request is made to the API. This is when the most up-to-date information regarding the user is retrieved and the App could then select the correct 'mode' to display. However, unavoidably, the user's token would still contain a payload that indicated their role was 'Driver'. Therefore any attempt at the booking creation would be rejected. This problem would also persist if the order of role change was the other way around (Customer

to Driver). It was ultimately decided that if the App detected any change in the user's role, it would consider the user's 'session' as expired and log them out.

The final step required in order to fully support Drivers within the app was implementing the ability to view multiple active bookings at once and edit their status. Whilst implementing the Customer aspects of the app it was assumed that only one active booking could exist at any given time. However, there was already support for 'booking lists' (due to the History screen). Therefore the approach was to re-use the Recycler View Adapter used by the Booking History list and implement a new view for each 'entry'. Upon clicking on an entry in the active booking list, the original Booking Overview fragment could also be re-used to display details about the booking. However, some amendments had to be made to the Booking Overview fragment for it to be usable by Drivers, namely: Replacing the 'Driver' field with 'Customer', and the addition of an 'edit' button by the booking's status. Of course, these changes would only be available to Drivers. The Booking Overview fragment would use the same approach as the other 'dynamic' screens within the app in order to display the correct content based on the user's role.

3.8.3 Implementing the Company Model within the API

In preparation for the development of the Web Application that is targeted towards Company Admins, the API had to be adapted to support Companies. Because most of the design for Company support had already been conceptualized, implementing the model for the Company was fairly straightforward. The same steps required for implementing Users and Bookings were taken, including the creation of a model, controller, and service file. With all the required /companies/ routes also implemented.

The bulk of the work took place in deciding how Companies would receive bookings from Customers. In the early stages of development, it was hoped that bookings would be delegated to companies automatically, by the system. However, for simplicity and to give full control to the companies themselves, a virtual 'booking market' approach was selected. This would include a list of all unallocated bookings that have been created by Customers. A Company Admin can then view this list and chose to 'claim' a booking. Once claimed, the booking would be removed from the list and would have all the object references required to 'belong' to that company. The same principle would apply for allowing companies to 'release' a booking, back into the 'booking market'. This would effectively undo all of the object references from the 'claiming' process.

The above mechanisms were implemented through the addition of three new routes: GET /bookings/, PATCH /bookings/id/release, PATCH /bookings/id/claim. The routes have all the necessary access protections to ensure only authorized users can perform the actions (for example, bookings can only be released by the company that owns it).

[INSERT SEQUENCE DIAGRAM OF BOOKING CLAIM / RELEASE]

3.9 Week 9 (13/04 - 19/04)

The final week of development focused solely on the implementation of the Responsive Web App. In order to serve some of the created screens within the Web App, some minor updates to the API were also required.

3.9.1 Web App Implementation

Although the primary focus during the prototyping phase was on the API and the Android App itself. Some low-fidelity pen and paper UI prototypes were also created for the Web App. It was eventually decided, however, that the best choice for the Web App's User Interface was to use a template [**?**]. This shaved off a lot of implementation time (that was becoming scarce at this point of development) and also provided a professional, consistent look and feel. Having used a Material Design template, it also matched the style that was established throughout the development of the Mobile App.

Thankfully, most use-cases for the Web App required only boilerplate Angular code to make requests to the API and display the results on the screen. Bootstrap tables were mostly used to display information. Authentication and Access Control mirrored what had been done on the mobile app, with the user's access token being stored in local storage and attached to any HTTP request headers. Error Handling was also very straightforward, with any errors simply being displayed to the user via a notification element. The only downside to this approach is that error messages were being shown directly from the HTTP response, making the Web App unfriendly for internationalization. This is an area that given an opportunity for further development, would certainly be addressed.

3.9.2 Required Updates to the API

Some aspects of the Web App's development called for changes to the API in order to be implemented successfully. The two most prominent examples are listing of a Company's Admins and recruiting drivers via their email. There was no dedicated mechanism within the API to retrieve a populated list of a Company's Admins, this was required for the Web App's 'company profile' screen. This addition was very simple and simply involved the creation of a new route 'GET /companies/id/admins' that retrieved a list of admins.

Another required change was supporting the addition of drivers to a company by their Email. Up to this point, Drivers had been added to a company by their Object IDs. However, this was not ideal, as Object IDs are long, complex strings that a user is unlikely to memorize or have access to. Therefore the 'PATCH /companies/drivers' route was adapted to support user emails as opposed to Object IDS.

[INSERT SCREENSHOT OF ADD DRIVER]

Chapter 4

Testing

Detailed descriptions of every test case are definitely not what is required here. What is important is to show that you adopted a sensible strategy that was, in principle, capable of testing the system adequately even if you did not have the time to test the system fully.

Provide information in the body of your report and the appendix to explain the testing that has been performed. How does this testing address the requirements and design for the project?

How comprehensive is the testing within the constraints of the project? Are you testing the normal working behaviour? Are you testing the exceptional behaviour, e.g. error conditions? Are you testing security issues if they are relevant for your project?

Have you tested your system on "real users"? For example, if your system is supposed to solve a problem for a business, then it would be appropriate to present your approach to involve the users in the testing process and to record the results that you obtained. Depending on the level of detail, it is likely that you would put any detailed results in an appendix.

Whilst testing with "real users" can be useful, don't see it as a way to shortcut detailed testing of your own. Think about issues discussed in the lectures about until testing, integration testing, etc. User testing without sensible testing of your own is not a useful activity.

The following sections indicate some areas you might include. Other sections may be more appropriate to your project.

4.1 Overall Approach to Testing

4.2 Automated Testing

4.2.1 Unit Tests

4.2.2 User Interface Testing

4.2.3 Stress Testing

4.2.4 Other types of testing

4.3 Integration Testing

4.4 User Testing

Chapter 5

Evaluation

Examiners expect to find a section addressing questions such as:

- Were the requirements correctly identified?
- Were the design decisions correct?
- Could a more suitable set of tools have been chosen?
- How well did the software meet the needs of those who were expecting to use it?
- How well were any other project aims achieved?
- If you were starting again, what would you do differently?

Other questions can be addressed as appropriate for a project.

The questions are an indication of issues you should consider. They are not intended as a specification of a list of sections.

The evaluation is regarded as an important part of the project report; it should demonstrate that you are capable not only of carrying out a piece of work but also of thinking critically about how you did it and how you might have done it better. This is seen as an important part of an honours degree.

There will be good things in the work and aspects of the work that could be improved. As you write this section, identify and discuss the parts of the work that went well and also consider ways in which the work could be improved.

In the latter stages of the module, we will discuss the evaluation. That will probably be around week 9, although that differs each year.

Appendices

The appendices are for additional content that is useful to support the discussion in the report. It is material that is not necessarily needed in the body of the report, but its inclusion in the appendices makes it easy to access.

For example, if you have developed a Design Specification document as part of a plan-driven approach for the project, then it would be appropriate to include that document as an appendix. In the body of your report you would highlight the most interesting aspects of the design, referring your reader to the full specification for further detail.

If you have taken an agile approach to developing the project, then you may be less likely to have developed a full requirements specification. Perhaps you use stories to keep track of the functionality and the 'future conversations'. It might not be relevant to include all of those in the body of your report. Instead, you might include those in an appendix.

There is a balance to be struck between what is relevant to include in the body of your report and whether additional supporting evidence is appropriate in the appendices. Speak to your supervisor or the module coordinator if you have questions about this.

Appendix A

Third-Party Code and Libraries

If you have made use of any third party code or software libraries, i.e. any code that you have not designed and written yourself, then you must include this appendix.

As has been said in lectures, it is acceptable and likely that you will make use of third-party code and software libraries. If third party code or libraries are used, your work will build on that to produce notable new work. The key requirement is that we understand what your original work is and what work is based on that of other people.

Therefore, you need to clearly state what you have used and where the original material can be found. Also, if you have made any changes to the original versions, you must explain what you have changed.

The following is an example of what you might say.

Apache POI library - The project has been used to read and write Microsoft Excel files (XLS) as part of the interaction with the client's existing system for processing data. Version 3.10-FINAL was used. The library is open source and it is available from the Apache Software Foundation [**?**]. The library is released using the Apache License [**?**]. This library was used without modification.

Include as many declarations as appropriate for your work. The specific wording is less important than the fact that you are declaring the relevant work.

Appendix B

Ethics Submission

This appendix includes a copy of the ethics submission for the project. After you have completed your Ethics submission, you will receive a PDF with a summary of the comments. That document should be embedded in this report, either as images, an embedded PDF or as copied text. The content should also include the Ethics Application Number that you receive.

Appendix C

Code Examples

For some projects, it might be relevant to include some code extracts in an appendix. You are not expected to put all of your code here - the correct place for all of your code is in the technical submission that is made in addition to the Project Report. However, if there are some notable aspects of the code that you discuss, including that in an appendix might be useful to make it easier for your readers to access.

As a general guide, if you are discussing short extracts of code then you are advised to include such code in the body of the report. If there is a longer extract that is relevant, then you might include it as shown in the following section.

Only include code in the appendix if that code is discussed and referred to in the body of the report.

3.1 Random Number Generator

The Bayes Durham Shuffle ensures that the psuedo random numbers used in the simulation are further shuffled, ensuring minimal correlation between subsequent random outputs [**?**].

```
#define IM1 2147483563
#define IM2 2147483399
#define AM (1.0/IM1)
#define IMM1 (IM1-1)
#define IA1 40014
#define IA2 40692
#define IQ1 53668
#define IQ2 52774
#define IR1 12211
#define IR2 3791
#define NTAB 32
#define NDIV (1+IMM1/NTAB)
#define EPS 1.2e-7
```

```
#define RNMX (1.0 - EPS)

double ran2(long *idum)
{
  /*-------------------------------------------------*/
  /* Minimum Standard Random Number Generator        */
  /* Taken from Numerical recipies in C              */
  /* Based on Park and Miller with Bays Durham Shuffle */
  /* Coupled Schrage methods for extra periodicity   */
  /* Always call with negative number to initialise  */
  /*-------------------------------------------------*/

  int j;
  long k;
  static long idum2=123456789;
  static long iy=0;
  static long iv[NTAB];
  double temp;

  if (*idum <=0)
  {
    if (-(*idum) < 1)
    {
      *idum = 1;
    }else
    {
      *idum = -(*idum);
    }
    idum2=(*idum);
    for (j=NTAB+7;j>=0;j--)
    {
      k = (*idum)/IQ1;
      *idum = IA1 *(*idum-k*IQ1) - IR1*k;
      if (*idum < 0)
      {
        *idum += IM1;
      }
      if (j < NTAB)
      {
        iv[j] = *idum;
      }
    }
    iy = iv[0];
  }
  k = (*idum)/IQ1;
  *idum = IA1*(*idum-k*IQ1) - IR1*k;
  if (*idum < 0)
  {
```

```
    *idum += IM1;
  }
  k = (idum2)/IQ2;
  idum2 = IA2*(idum2-k*IQ2) - IR2*k;
  if (idum2 < 0)
  {
    idum2 += IM2;
  }
  j = iy/NDIV;
  iy=iv[j] - idum2;
  iv[j] = *idum;
  if (iy < 1)
  {
    iy += IMM1;
  }
  if ((temp=AM*iy) > RNMX)
  {
    return RNMX;
  }else
  {
    return temp;
  }
}
```