

CS958 Project Report 2018-2019

Development of a Scalable Online Booking System for SME Car Rental Systems through implementation of the MERN Stack

Submitted in partial fulfilment for the degree of MSc Software Development

Jack Preston

Registration No.: 201855079

Computer & Information Sciences

Livingstone Tower
26 Richmond Street
Glasgow, G1 1XH

Supervisor: Dr M. Roper

(Dated: August 19th, 2019)

Declaration

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research. I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive. (please tick) Yes ☒ No ☐

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices) is 21,589.

I confirm that I wish this to be assessed as a Type 1 3 4 5 Dissertation (please circle)

Signature: 

Date: 18/08/2019

Abstract

This project seeks to build on the teachings presented during the course lectures throughout the year and bring them together, in a practical example that exercises the principles taught, along with the techniques and technologies presented. It will do this by investigating, designing, building and testing a live web application end-to-end stack using a real world example. This will test not only the theories and techniques presented, but also the practicalities, limitations and challenges of building in the real world. On completion of the project, an analysis of the lifecycle phases, critical decisions and the challenges faced will be presented along with learning's and future development opportunities that could take the project further.

The example used throughout this project will be to develop a real world, scalable, on-line car booking system web application, which will utilise the full MongoDB, Express.js, React.js & Node.js (MERN) stack. This will not only test the configuration, integration and build of the environment components, but also the practicalities and challenges faced by a developer in building real world applications as will be faced in industry.

This project includes all phases of the application development lifecycle, combining requirements, design, construction and test into an overall systems pack that could, with further development, be deployed in the real world. In parallel an analysis of each phase, highlighting the challenges and learnings is documented and discussed within this document.

Combined with the physical application and associated development environment which created the functional MVP, this document pulls together overall conclusions and discusses future improvements which could form the basis of the next phases of this project.

This operational system along with this document will constitute the totality of the project submission.

Acknowledgements

I would firstly like to thank my supervisor, Dr Marc Roper, for his support and guidance over the course of this project. I am grateful and thank him for his ideas and time throughout. I would also like to give a special thanks to my father, James Preston, for providing support, motivation and suggestions throughout the project.

I would finally like to thank my family and friends for their continued support throughout.

Table of Contents

1. Introduction	9
1.1. Background	9
1.2. Review of Literature	11
1.2.1. Front end technologies: Performance and Scalability	11
1.2.2. Database Selection: Challenges of Big Data, Scalability and Performance	14
1.3. Objectives	20
2. Methodology	21
2.1. Requirements	21
2.1.1. Functional Requirements	22
2.1.2. Non-Functional Requirements	23
2.1.3. User Stories	25
2.2. Design	26
2.2.1. System Architecture	26
2.2.2. Database Design	27
2.2.3. Front-end Design	29
2.2.4. Back-end Design	30
2.3. Construction	31
2.3.1. Sprint 1	32
2.3.2. Sprint 2	37
2.3.3. Sprint 3	40
2.3.4. Sprint 4	41
2.3.5. Sprint 5	46
2.4. Testing	49
3. Analysis	55
3.1. Requirements Analysis	55
3.1.1. Absent Requirements	55
3.1.2. Incomplete Requirements	55
3.2. Design Analysis	56
3.3. Construction Analysis	57
3.3.1. User Construction (Mutual functionality of Customer and Employee)	57
3.3.2. Employee Construction	57
3.3.3. Customer Construction	58
3.3.4. Vehicle, Date & Booking Construction	59
3.4 Client Review	59
3.4.1 Employee functionality	59

3.4.2 Customer functionality	59
4. Conclusion.....	60
5. Future work.....	62
6. References	63
Appendix A – User Guide	64
Appendix B – Front-end Testing Document.....	66
Appendix C – Back-end Testing Document.....	77

Table of Figures

Figure 1.1: Application layers of traditional “N-layer” web architectures [2].	11
Figure 1.2: Results for function test 1 [4].	13
Figure 1.3: Result for function test 2 [4].	13
Figure 1.4: Results for function test 3 [4].	14
Figure 1.5: Vertical scaling vs horizontal scaling [9].	15
Figure 1.6: Comparison of Size vs Complexity for types of NoSQL systems [11].	16
Figure 1.7: Diagram of Brewers’ CAP theorem [15].	18
Figure 2.1: The requirements of the system categorized into Must have, Should have, Could have and Won’t have requirements.	24
Figure 2.2: Use Case Diagram displaying the functionality of both types of users within the application	26
Figure 2.3: System architecture of the MERN Stack [16].	27
Figure 2.4: Entities and their attributes held within the system.	28
Figure 2.5: Entity relationship diagram of the system. A user can have multiple bookings, a booking may only have one user associated with it. A booking can only have one vehicle and a vehicle can have multiple bookings. A date can have multiple vehicles and a vehicle can only have one date (as a result of design, each vehicle, although the attributes are the same, is a different object for each date).	29
Figure 2.6: a) Wireframes for Registration Screen b) Wireframes for the Login page.	30
Figure 2.7: UML Diagram displaying the functions required in the back-end of the system.	31
Figure 2.8: Jira board for midway through sprint 1.	32
Figure 2.9: Schema for a User in Mongoose	34
Figure 2.10: Registration form for the application.	35
Figure 2.11: Route to api/cars/addcar receiving a newCar object to be posted to the database	36
Figure 2.12: fetch() request from React to the server	36
Figure 2.13: Route for api/dates/:date_id to retrieve a date object based on the ID provided	37
Figure 2.14: Navigation bar for the employee	38
Figure 2.15: Add Vehicle form	39
Figure 2.16: Card components of each vehicle	39
Figure 2.17: The two Datepicker objects presented to the customer to select their rental dates	41
Figure 2.18: Card view of the vehicle that is shown to the customers	41
Figure 2.19: handleClick() method to locally store the selected vehicle	42
Figure 2.20: Customer payment screen showing the booking details and payment options	43
Figure 2.21: fetch() request for each date in the range to update the vehicle of the specified VIN number	44
Figure 2.22: Route for the PUT request	44
Figure 2.23: Card view of customers current bookings	45
Figure 2.24: Route to DELETE booking from overall booking list using the user email with the start and end dates of the booking as the identifier	45
Figure 2.25: The difference of the two routes	45
Figure 2.26: Table of bookings shown to the employee	47
Figure 2.27: fetch() request for the to GET the list of dates and set the xAxis and yAxis for the graph	47
Figure 2.28: Plot of number of bookings vs dates	48
Figure 2.29: Notification receiving a message property to inform the user their account has been registered	48

Figure 2.30: Account registered notification	48
Figure 2.31: handleSubmit() function for the registration page.....	50
Figure 2.32: Registration form with the successful registration notification	51
Figure 2.33: Console.log() of the user object being sent to the server from the form	51
Figure 2.34: fetch() method to receive a list of vehicles held in the fleet.....	52
Figure 2.35: Results from the fetch() method to receive the list of vehicles held in the fleet	52
Figure 2.36: Route test on Postman testing the GET route to return a user of the specified email .	53
Figure 2.37: Route request to return a user object of the specified email	53
Figure 2.38: Postman query results for the GET request to receive the user object of the specified email	53
Figure 3.1: Example of booking object stored in the overall bookings collection	56
Figure 3.2: Table headers for the table views by the employee when viewing the whole fleet	57

List of Tables

Table 1.1: Table provided by Chęć et al. [4] showing the functionality tests, their objectives and parameters.	12
--	-----------

1. Introduction

1.1. Background

Against a backdrop of increasing operating costs and real time cost comparisons, online booking systems are becoming increasingly popular with middle to small sized companies who are looking to provide an all-in-one experience for not only their users, but also their employees. Online booking systems allow both customers and businesses to easily manage their bookings and also provide an insight into performance, trends and what direction to head toward in the future.

In a world where customers are now expecting instant responses, businesses are looking to provide a platform where the customer can reserve their product with ease, 24/7. Not only do the booking systems provide a better experience for the customer, but they also help in creating more revenue as they are able to take bookings at a time that suits the customer, meaning the customer is far less likely to turn to a competitor. Staff will no longer have to spend time answering calls to take bookings, send reminder emails or manage inventory manually as this will all now be done digitally, allowing them to focus on customer relationships and service.

Companies such as AirBNB are generally recognised as being at the forefront of online booking systems and are paving the way for smaller companies to follow suit. The business model for AirBNB is a great example of an online booking system that makes the most of the data they receive. AirBNB is an online market place that provides its users with a simple online way to rent out their properties or spare rooms to guests. The users of AirBNB have very little to actively manage once their property is listed on their marketplace as the online booking system provided by AirBNB takes care of most of the work. Not only does it provide a quick and easy way for its users to manage their properties bookings, it also provides a great experience for the customers looking to rent. Customers can easily find properties that match their search criteria due to the vast amount of filters they can apply on their searches. Furthermore, users who are posting their properties online to rent can also opt in to a dynamic pricing add-on provided by AirBNB. With this dynamic pricing system, users are prompted to enter the price per night of their property along with the minimum price per night they would like their property to be rented for. The system will then alter the listed price online based on a number of metrics which have been carefully designed by the AirBNB engineers, such as popularity – for example, if a certain property was booked Monday – Wednesday and Friday-Sunday for a certain week, it would be empty on the Thursday night which could potentially leave money on the table, AirBNBs dynamic pricing system would therefore lower the price of the night for the Thursday night to encourage a customer to book. There are many other examples, such as events being held near the property, which could alter the listed price. These algorithms help the user generate more bookings as a result which benefits both the customer and AirBNB.

Hotel companies also make great use of online booking systems, again, allowing staff not to utilise their time answering phones, sending reminder emails etc. instead being able to primarily focus on making the guest experience as enjoyable as they can. With some large hotels having hundreds of rooms, previous manual booking systems could have been

extremely error prone and time consuming, however with online booking software they are able to operate more efficiently. Similar to AirBNB, many hotels that are using online booking software take advantage of the large volumes of data they are receiving, allowing them to spot trends and price rooms accordingly. This results in the customers getting a fair price for their room and the business being able to maximise profits by tuning their operating model to keep every room filled as much as possible.

SkyScanner is possibly one of the most popular online booking systems that are currently on the market and has a similar business model to AirBNB with the difference that their users are airline companies looking to fill seats on their flights. Their booking system like all others provides inventory management, reservation management and 24/7 availability. And similar to the other big players who are currently making use of online booking systems, Skyscanner also has a dynamic pricing system.

Although many smaller to midsize companies have moved in the direction of implementing online booking systems into their business model, very few of them have the ability to develop bespoke systems, to take advantage of the benefits that such systems can provide. With companies, such as those discussed above, gathering data on their reservations, users and trends, machine learning can be applied to maximize efficiency and profits, which is vital for smaller companies who are competing in the same market as some of the bigger players.

A sector that seems to be late adopters with regards to online booking systems and real time customer experience is the car rental industry. The car rental business is currently dominated by a few major players such as Hertz, Europcar and Thrifty. Often, these car rental companies receive poor reviews based on customer satisfaction, be it the long queues at the airport, or the time spent filling out paperwork at the desk before being given the keys to a car. Yet, because these companies are so well established and recognised, the smaller players in the industry don't have the resources or capability to compete, even though they could offer a much more efficient and seamless service. Offering a digitized way for these smaller companies to take bookings and distribute their cars would cause a significant disruptive play in the car rental industry. Nuvven is a TravelTech company with a bold ambition to address this and transform the car rental industry by providing the small and medium sized car rental companies with a suite of applications that not only provide 24/7 customer service, but also provides artificial intelligence and telematics powered technology platforms to create a marketplace offering, which will not only provide a fully digitised experience to customers, but also create more competition against the incumbents.

To explore the opportunities that this type of application could offer, this project, in conjunction with the development team at Nuvven, will be looking to develop a full stack web application which will be used to provide a digitized way for these smaller companies to take bookings and distribute their fleet in the most efficient way possible. The application will be built with scalability and performance in mind, with future works also looking to implement artificial intelligence and machine learning into the application. The application will aim to bring a better more intuitive booking experience for the customer but also provide solutions that will help the fleet partners' better manage their business.

1. 1.2. Review of Literature

1.2.1. Front end technologies: Performance and Scalability

Web 2.0 (the second generation of the world wide web), a term made popular by Tim O'Reilly and his business partner Dale Dougherty in 2004 marked the shift of static HTML webpages to a more dynamic and user friendly experience. [1]. The concepts of Web 2.0 which characterized websites began to be referred to as web applications. A major moment in the process of Web 2.0 was the use of Asynchronous JavaScript and XML (AJAX). AJAX provided a new model of how web applications send and retrieve data from a server allowing web applications to send and retrieve data from a server in the background, therefore the behaviours and display of the current page are not affected. Shortly after, the use of AJAX began to arrive in the early JavaScript libraries such as jQuery.

The traditional “N-layer” architecture of web applications was previously the most common architecture used. The application was split into 3 layers: User Interface, Business Logic and Data Access. [2]

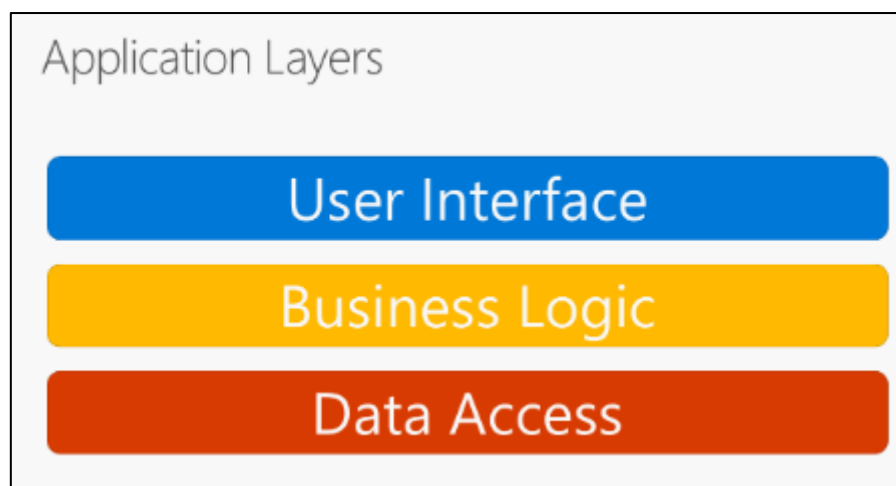


Figure 1.1: Application layers of traditional “N-layer” web architectures [2].

Via the User Interface (UI), the users would make a request which would interact with the Business Logic Layer (BLL). The BLL itself can interact with the Data Access Layer for and data read/writes. With the popularity of the internet increasing rapidly, the architecture evolved to migrate some of the business logic to be processed on the client’s side. In turn this freed up the computational power of the servers, increasing their performance.

However, with this increase in performance server-side, the increase in complexity of the business logic processed on the client’s side became a new issue for developers. In 2013, a group of software engineers at Facebook produced the initial version of the React.js library [3]. The aim of this JavaScript library was to handle a considerable amount of operations performed on the Document Object Model (DOM)* tree, frequently. Due to the tree structure of the DOM, searching through the DOM is easily performed, however easy does not always mean efficient. With many modern web applications having massive DOM trees and being tailored towards being dynamic, the DOM tree updated constantly –resulting in performance issues. The Virtual DOM (VDOM) helps in providing a solution for this issue. The VDOM is an abstraction of the ‘regular’ DOM that is used to improve performance when modifying the DOM. Rather than traversing the DOM to identify which node is to be modified; the VDOM provides an alternate method. The VDOM performs the action of updating the DOM by the comparison of two objects, the first object being the input state of the function (the current state of the DOM) and the second being the state following its completion. The result of the differences in the

two objects will therefore directly point to the elements that are being modified by the web browser [4]. React.js and Vue.js are two web technologies that implement this VDOM strategy for increased in performance.

*a language neutral, cross platform interface that treats HTML/XML documents as a tree structure such that each HTML/XML element becomes a node in the tree. It is an object-oriented representation of the web page which can be modified by with a scripting language such as JS [5].

Paper [4] aimed to analyse the performance of web based applications that implement the VDOM strategy, notably the popular React.js (RRO architecture) library and its descendant the Cycle.js (CLE architecture) framework. Furthermore, Chęć et al. provide an insight into the performance comparison of a web application using JavaScript and the 'regular' DOM (JSD architecture). Each of the 3 web applications was to perform a number of functions that would appear in reality, managing personal data. The functions and their objectives are shown in table 1.1 [4].

Experiment	Objective	Parameters
Loading the list of personal data with a specific number	Measurements of page loading times and DOM/VDOM construction based on the amount of transferred data	Lists with lengths of 100, 200, 500, 1000 components
Searching the full list and displaying cards of persons who meet the criteria	Measurements of execution times of complex DOM/VDOM tree transformation operations – creating and attaching nodes when the action is called	Search phrases entered in the filter fields for a list of 200 and 1000 elements
Deleting data of selected persons – a list of people indicated in random order	Measurements of the DOM/VDOM node removal times from the random location of the previously loaded structure	A one-dimensional, unordered table containing random item identifiers to remove from the 200 and 1000 elements list
Deleting all personal data from the previously loaded list	Measurements of removal times of all DOM/VDOM nodes	One-dimensional, ordered table with a length of 1000

Table 1.1: Table provided by Chęć et al. [4] showing the functionality tests, their objectives and parameters.

To ensure a fair comparison of the three applications, Chęć et al. carried out all experiments on the same computer. Prior to the experiment they also investigated the number of DOM/VDOM tree nodes that were generated by the varying lengths of the list of persons. The number of tree nodes in the DOM/VDOM is directly proportional to the complexity of the task. For the length of the lists of persons 100,200,500 and 1000, the number of DOM/VDOM nodes were 1948, 3848, 9548 and 19048 respectively.

The performance of each of the application architectures was measured using the browser-perf tool. Browser-perf is a NodeJS based tool that allows for developers to test the performance of their web based applications [6]. Chęć et al. broke the performance of the three web applications into 4

measurable metrics, Loading, Scripting, Rendering and Painting. The Loading metric measured the time for each of the application architectures on the conversion and parsing of HTML code to resolve the structure of the DOM. The Scripting metric measures the time for each of the application to interpret, listen and execute the JavaScript code as well as including the browser's allocation control and memory release operations. The Rendering metric measures the time for each event that is involved in processing the page layout and in developing a CSSOM (CSS Object Model – the CSS equivalent of the DOM) and DOM render tree. The final metric, Painting, is used to measure the time for the result page to be converted to pixels and displayed in the web browser.

Each functionality test on the three architectures were performed 15 times and an average taken. Chęć et al. do state that the Loading metrics measured were very low in comparison to the other three and were therefore left out of the analysis.

Results for function test 1 shown in figure 1.2: Loading the list of personal data with a specific number (100, 200, 500 & 1000) concluded that the longest time for a full list of items to be loaded was performed by the React architecture, the Scripting metric being held responsible for this. For length lists of 100 and 200, the JSD application gave the shortest loading times, however as the list length increased to 500 and 1000 and so the number of nodes in the DOM/VDOM, the CLE application performed best. From the results found it can be seen that Rendering and Painting metrics increase **significantly** as the DOM size increases. The Scripting metric which is responsible for the **construction** of DOM, or the VDOM in the case of the RRO and CLE applications show that the construction of the VDOM is **expensive** in comparison to the regular DOM.

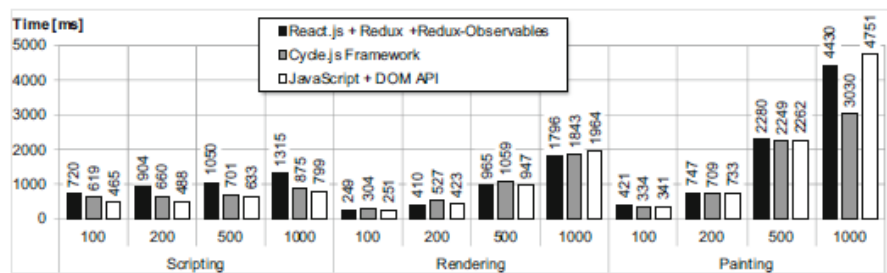


Figure 1.2: Results for function test 1 [4].

The second functionality test: Searching the full list and displaying cards of person who meet the criteria yielded results in favour of the VDOM. This was expected as the VDOM deals with directly modifying specific nodes within the DOM tree much better than the regular DOM. This test was only performed by Chęć et al. with list lengths of 200 and 1000 but the CLE application again came out as the highest performer, closely followed by the RRO application. As seen then figure 1.3 the results show the JSD application did not perform anywhere near those application deploying the use of the VDOM with respect to Scripting and Rendering.

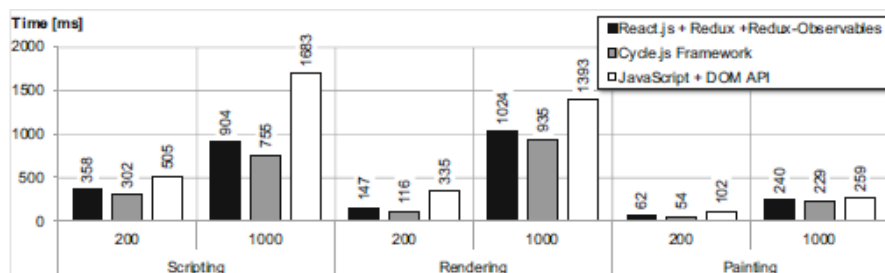


Figure 1.3: Result for function test 2 [4].

The third functionality tests, deleting the data of selected persons, aimed to analyse how well the DOM/VDOM architectures deal with node removal from a structure. As shown in figure 1.4 it can be seen that the CLE and RRO (the applications which implement the VDOM) performed much better with regards to Scripting and Rendering when removing selected persons from a list. This again reinforces the VDOMs ability to outperform regular DOMs when it comes to perform operations on targeted nodes. With regards to the final test, removing all elements from the list, values for each metrics were very low.

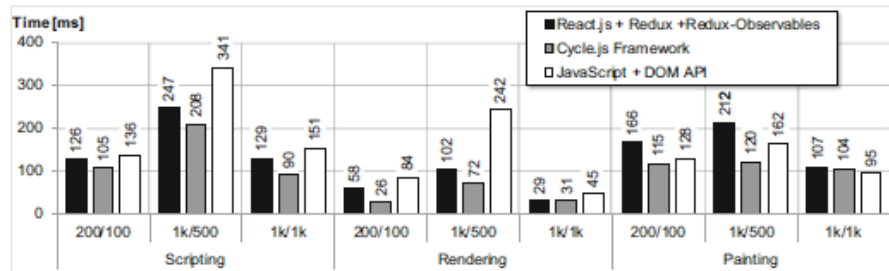


Figure 1.4: Results for function test 3 [4].

Analysis of the results provided from Chęć et al. concludes that the applications which use the virtual representation of the DOM is able to achieve results considerably quicker than those which use the regular DOM. The two applications employing the VDOM provide performance which is becoming a requirement for web applications as more and more business logic is needed to be performed client side. Furthermore, as we enter the era of big data, our applications must be able to scale appropriately, applications using the VDOM will be able to keep up with the curve maintain performance when data and actions become more and more complex.

1.2.2. Database Selection: Challenges of Big Data, Scalability and Performance

Modern applications and businesses are now using big data to allow for machine learning to predict trends, make informed business decisions, recommender systems for customers, pricing systems for products and many more. The goal of deploying the machine learning algorithms is to improve profits and/or the user experience. The term big data refers to large amounts of complex, semi-structured/unstructured data that is generated in a large size and is stored in a database [7 - Research challenges of big data (main paper)]. With the size and variety of the data these modern applications are now using, it is vital that the appropriate database technology is chosen to allow for performance to remain high as the scale increases.

Relational SQL databases versus non-relational NoSQL databases are a contested topic. Both have their pros and cons and the selection between the two is governed by the requirements of the application. With big data in mind, scalability and read/write performance needs to be taken into account.

With scaling being one of the biggest issue databases are facing at this current time, an insight into what SQL and NoSQL offer with respect to scaling is important. There are two types of scaling when it comes to databases, vertical and horizontal. Vertical scaling involves all of the data within the database residing on a single node. In order to scale vertically, extra CPU and RAM are added to the database node to provide more memory and performance. Horizontal scaling however, allows for the addition of more machines which distributes the memory and workload of the machines already involved. Horizontal scaling is an ideal solution for systems that will be using big data as, unlike vertical systems, theoretically there is no upper limit for scaling. Typically, traditional relational databases

have limited ability to scale horizontally, whereas, NoSQL databases can, without much difficulty, due to their flexible nature [8].

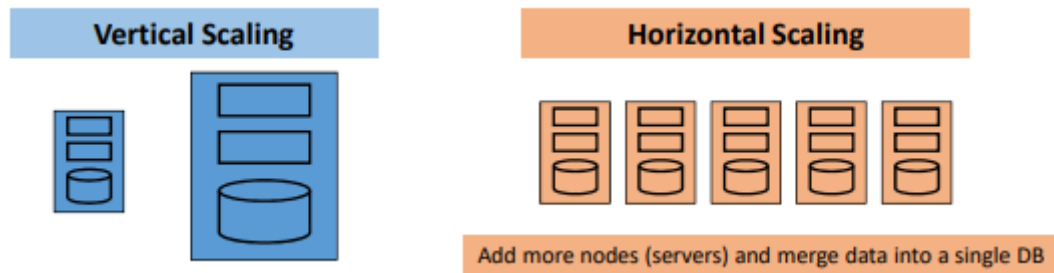


Figure 1.5: Vertical scaling vs horizontal scaling [9].

Cattell states [8] that NoSQL database systems can achieve much higher performance and scalability than its relational counterparts. However, the gain in performance and scalability come with a cost. Relational SQL databases are said to be ACID compliant, such that each transaction (a logical unit of work (data related)) in the database can guarantee four properties [10]:

- Atomicity – Transactions are indivisible and performed in their entirety or not at all
- Consistency – Any transaction must transform the database from one consistent state to another
- Isolation – Transaction execute independently of one another
- Durability – the effects of a successful transaction are permanently recorded in the DB and cannot be lost

These four properties ensure reliability and consistency in SQL databases. NoSQL systems are not ACID compliant, instead they are said to be BASE compliant [8]:

- Basically Available – The system is guaranteed to be available for querying by all users
- Soft state – The values stored in the system may be subject to change as a result of the eventual consistency model
- Eventually consistent – When data is added to the system, the systems state is gradually replicated across all nodes in the system

Although NoSQL systems embrace the 'eventually consistent' many of them provide mechanisms for some degree of consistency such as multi-version concurrency control (MVCC) [8].

Cattell firstly begins by taking a look at the various types of data stores that are currently on offer from developers to use:

- Key-value Stores – systems which stores values along with an index to find them
- Document stores – systems which store documents*. The documents are indexed and simple CRUD operations provided.
- Extensible Record Stores – systems which store extensible records that can be partitioned vertically and horizontally across nodes. Often referred to as 'wide column stores', popular examples of these systems include Googles' Big Table and Apache Cassandra
- Relational Databases – systems which store, index and query tuples (rows in relational DB, where attribute names are pre-defined in schema of system)

*Definition provided by Cattell in [8]: allows values to be nested in documents or lists as well as scalar values, and the attribute names are dynamically defined for each document at run time. Differs from a tuple as document attributes are not defined in a global schema.

Developers often quote Eric Brewer's CAP theorem when talking about NoSQL database implementation, less formally known as the 'no free lunch' theorem [11]. Brewer states that in a NoSQL system, only two of the following attributes can be achieved:

- Consistency – every read sees the most recent write
- Availability – every request receives a response
- Partition tolerance – the system continues to operate despite an arbitrary number of dropped, or delayed, message between nodes.

Graph based database systems, another NoSQL data store, such as Neo4j and OrientDB, are however not included in Cattell's paper. Graph based systems are said to be the best NoSQL system at handling complexity, however they do not scale as well as their key-value or document based equivalents (as can be seen in figure 1.6) [12]. Although, as stated previously, relational databases scale vertically, Cattell provides an insight into some newer relational database systems which offer horizontal scaling. For an application such as the application proposed in this project, a look into the comparison of the scalability of the modern relational systems against the document store systems is necessary.

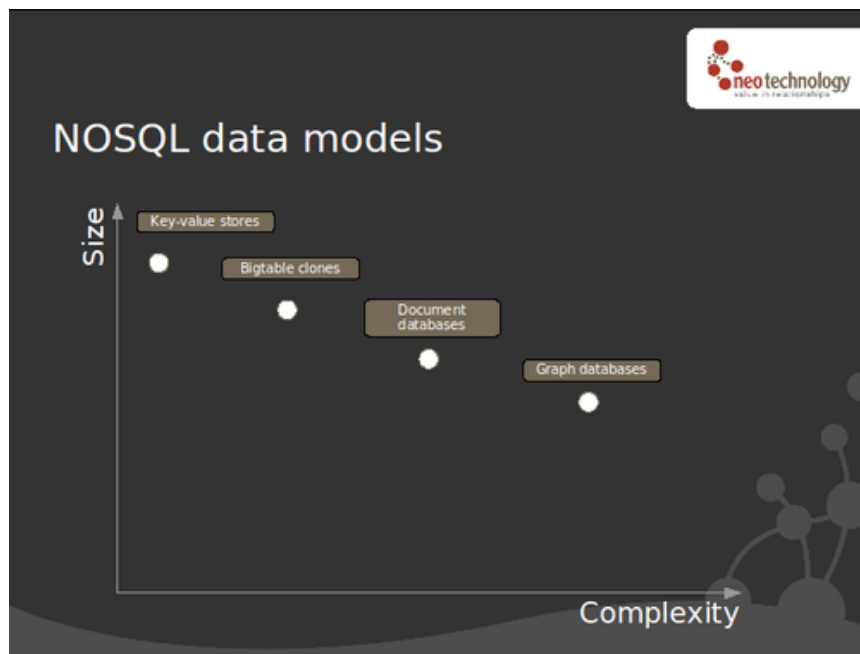


Figure 1.6: Comparison of Size vs Complexity for types of NoSQL systems [11].

Initially, a further look into document based systems is covered, before looking at some relational alternatives. Document based systems provide support for more complex data than key-value store systems as document based systems can support secondary indexes, documents nested within other documents and also multiple types of documents. As the application developed in this project will require support for all of these functions, a document based NoSQL system is a good candidate.

Cattell provides a look at some of the types of documents stores which are available. SimpleDB is reviewed first. SimpleDB is provided as part of Amazon's Web Services and true to its name its model is simple. SimpleDB only provides four operations which can be performed on the documents held, Select, Delete, GetAttributes and PutAttributes. Unlike other document stores, nested documents are

not supported by SimpleDB which limits its use greatly. However SimpleDB does offer more than one grouping in the one database, which as previously mentioned is offered by other document stores but not key value stores [8 - Cattell]. With regards to horizontal scaling, SimpleDB can only offer 10GB maximum per domain which is rather limiting, although for smaller based applications that do not require complex queries, SimpleDB could be a viable option due to its low set up cost and pay as you go expansion [12]. From the information provided by Cattell on SimpleDB, the conclusion can be drawn that SimpleDB has its place as a great database for small scale applications that do not require storage of complex data and execution of complex queries, ideal for startups.

Secondly a look at CouchDB is provided. CouchDB is a document store developed by Apache since 2008. In comparison to SimpleDB, the model provided by CouchDB offers much more functionality. In CouchDB, queries are performed with what are known as 'views'. These views are defined in JavaScript to determine field restraints. However, as these views are PL rather than Declarative, more of a burden is placed on the programmer and if inexperienced room for errors are increased. CouchDB does not use sharding for scalability, instead, scalability is achieved through asynchronous replication* [8 - Cattell]. Reads performed on the system can go to any of the nodes, provided the user does not mind having the latest values. Updates to the database must also circulate through all the nodes. As a result, CouchDB cannot provide a guarantee of consistency on reads, however it does implement multi-version concurrency control. The implementation of MVCC in CouchDB allows the user to be notified that someone else has updated the document since it was fetched, signalling their read may be dirty. Durability is also provided by CouchDB as all updates are flushed to disk on commit, by writing to the end of a file [8]. Combined with the MVCC mechanism, CouchDB can provide ACID transaction support.

*Asynchronous replication is the process in which a write to the database is initially added to the primary node and then copies the data to replication targets.

MongoDB is a free, open source document store which is supported by 10gen. MongoDB is very similar to CouchDB in the sense that they both provide indexes on collections, document query structure and are both lockless but there are some differences [8]. With regards to scaling, MongoDB implements the method of automatic sharding* as opposed to the async replication used by SimpleDB and CouchDB. MongoDB does however use replication but unlike other document stores this replication is not used for scalability but rather for failover**. However, to help with performance the replication performed in MongoDB is asynchronous which may result in some updates being lost if a crash were to occur. Whilst CouchDB provides multi-version concurrency control on its documents, MongoDB implements atomic operations on fields. Whilst CouchDB and SimpleDB can achieve scalability by potential dirty reads, MongoDB can achieve scalability without compromise through the automatic sharding and atomic operations on documents [8 - Cattell]. With respect to Brewer's CAP theorem, both technologies achieve partition tolerance but as can be seen in figure 1.7 [14] MongoDB favours consistency (C) over availability (A) which CouchDB favours.

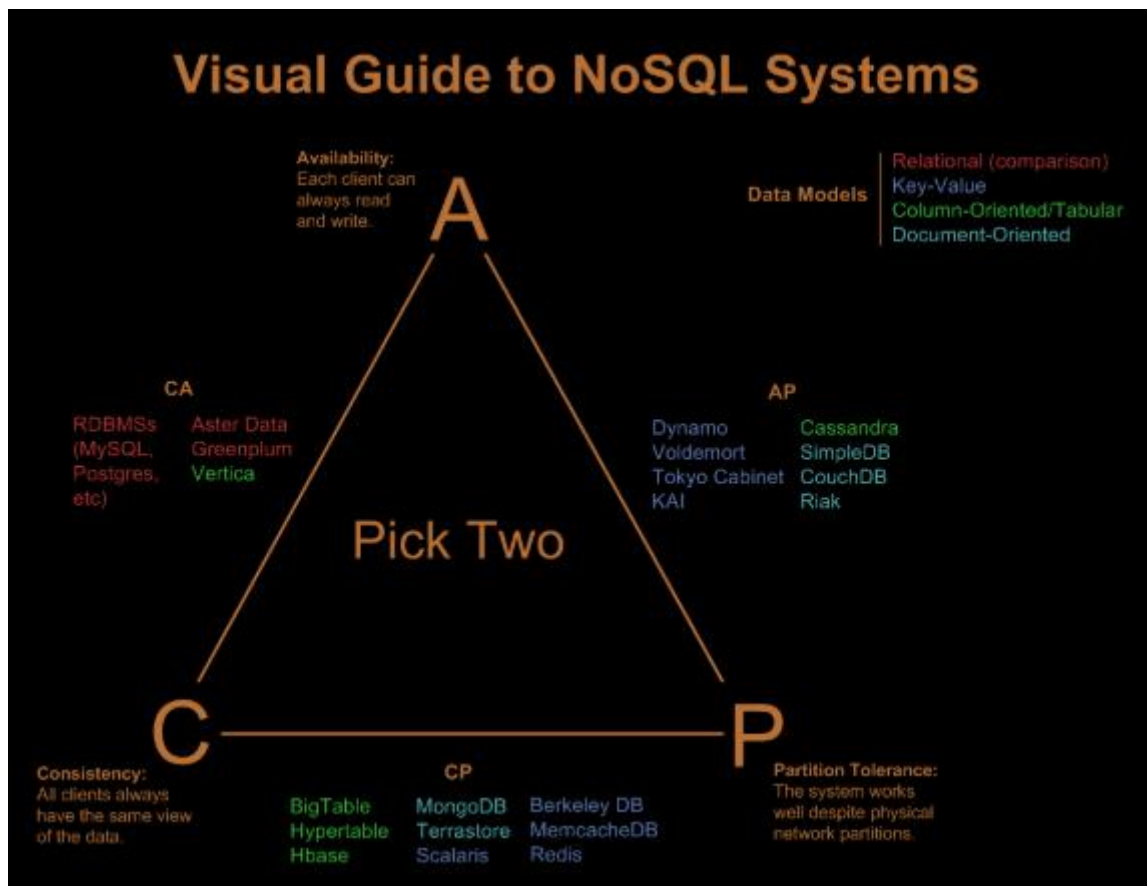


Figure 1.7: Diagram of Brewers' CAP theorem [15].

*Sharding is the process of partitioning a database system horizontally into multiple smaller nodes. This prevents the bottleneck which occurs when scaling vertically.

**Failover is a procedure where a system transfers control to a duplicate system in the event of a fault or failure.

Cattell moves to looking at various technologies for extensible record stores before finally approaching the subject of scalable relational systems. Dissimilar to the NoSQL data stores looked at previously, the relational systems guarantee ACID transactions, must have a pre-defined schema and come with a SQL interface [8 - Cattell]. If RDBMSs were able to achieve scalability similar to NoSQL systems whilst being able to maintain good performance, they would have an advantage over the NoSQL systems due to their high-level SQL language and ACID properties. Furthermore, SQL is a mature technology which is well documented and staff are likely to be well versed in SQL as opposed to the NoSQL alternatives. SQL systems also do come with the overhead of being schema-on-write which means that extra development time must be taken to ensure the database is designed properly and less flexibility than NoSQL systems which are said to be schema-on-read. With the agile method of development coming into fruition, flexible systems are becoming a requirement to allow for quick changes.

MySQL Cluster was one of the earlier scalable relational systems, being part of the MySQL release since 2004. Similar to MongoDB, MySQL Cluster shards data over multiple database servers with replication occurring to support recovery. This method of sharding allows for MySQL to scale to more nodes than its other scalable relational system counterparts, however reportedly, performance bottlenecks occur after a few dozen nodes. For big data projects, this could be a real issue.

The second scalable RDBMS Cattell analyses is VoltDB. VoltDB is designed with scalability and high performance per node in mind. The scalability achieved by VoltDB is competitive with the NoSQL papers covered as well as MySQL Cluster. Tables in VoltDB are distributed over multiple servers and the users can call any server. Selected tables can be replicated over servers to allow for fast access to read-only data and shards are replicated in order for data to be able to be recovered in the event of a crash [8 - Cattell].

Cattell takes a brief dive into some newer scalable RDBMSs, ScaleBase and NimbusDB but however summarizes that the information gathered on both is too limited to come to an educated conclusion as to whether they are viable. He does conclude that, theoretically, provided applications avoid cross node operations (which may require some careful data structure and storage planning) these scalable RDBMS systems such as MySQL Cluster and VoltDB would have an advantage over their NoSQL alternatives. This is due to the simplicity of SQL and transactions being able to be fully ACID compliant which guarantees consistency.

However, Cattell does fail to discuss the impact of changing the schema. With the scalable SQL systems a change in schema could have a large overhead with regards to time and planning where as its NoSQL equivalents as they are schema-on-read can be extremely flexible in how the data stored in the system evolves over time. That being said I do believe the selection of which type of data store to use is situational and dependant on number of factors. With a project such as the one undertaken in this report I believe a NoSQL document based system is suited. Not only, will it help with speed of development as a result of the agile nature of NoSQL, but my data store will also need to store multiple different kind of objects which may vary in attributes. For example, a vehicle in the fleet may have different features to another.

1.3. Objectives

This project aims to develop a full stack web application to provide a fully digitized solution for smaller car rental companies to manage, analyse and distribute their fleet. Due to the wide range of possible requirements, the project will be based on a core set of end-to-end functions that exercise the full depth of a web application stack. Building the application in this way will prove the integrity of the end-to-end design and implementation and would form the basis of a platform that further functionality could be added to at a later date.

Developing on the principles learned throughout the course, and with scalability, performance and future endeavours in mind, the objective is, in addition, to gain a deeper understanding of how to apply a full MERN (MongoDB, Express, React.js and Node.js) stack. The use of a React front-end allows for scalability and performance when dealing with business-logic client side. Node.js with the Express web framework will allow for high performance and scalability in the web application and the node package manager (NPM) will be a great asset as it provides access to thousands of reusable packages. The use of MongoDB will allow for development of the system to begin immediately and will cater for changes in data schemas as the application progresses. With future work looking to implement machine learning and telematics, MongoDB is a great choice as it supports horizontal scaling through sharding, allowing it to cope with the big data that is required to drive these systems.

2. Methodology

2.1. Requirements

The gathering of requirements is one of the most important parts of the development lifecycle and it is vital that all requirements are captured as early on in the lifecycle as possible. According to research, 68% of IT projects fail and it is poor requirement analysis that causes many of these failures (IAG Consulting). Understanding the needs of the client/customers is not a trivial task and from a developer's perspective, a lot of thought must be put into the requirements before development commences. The requirement gathering process, if done correctly, removes ambiguity and allows all that are involved to have a clear, documented, list of what is needed from the application. It is also the case that an accurate set of requirements at the outset leads to cleaner design and implementation, removing the need for expensive rework.

As this project will be looking to provide a digitized platform for smaller car rental companies to manage their bookings and fleet, as well as providing their customers a platform in which they can book vehicles, the requirements of the system are non-trivial and wide ranging.

Unfortunately, due to the short period of time in which this project is to be completed, a thorough investigation into all of the requirements of the application from the Hire companies' perspective was not achievable. Ideally, from a developer's point of view, it would have been extremely useful to have meetings with current small/mid-size rental companies to gather requirements. Prior to the meetings a list of well thought out questions would be written up. These meetings are extremely useful for both client and developers as although the client may know exactly what they want, the developers can also propose some additional features that may be available, as the client may not be well versed in up to date technologies some functionalities may be unknown to them. Although not possible for this project, I had a good idea of what functionality would be required from an employee, through discussions I had with colleagues in Nuvven. Requirement gathering for the customer was far easier; there are plenty of online sources that were available to review. Looking at the booking journeys provided by some of the bigger players in the market, such as Hertz and Enterprise, provided a great insight.

The first stage in the writing up the requirements for this application is to identify the entities which would be involved in the system. Identifying the entities involved in the system allowed for user stories to be developed based on the requirements. For the overall booking system there was 5 entities: employee (member of staff), customer, vehicle, booking and date. As mentioned, due to time constraints I was at the disadvantage of not being able to communicate and gather requirements from a number of small to mid-size fleets that would be the target market for this application, therefore requirements for the business-to-business (B2B) (i.e employee requirements) side of the application were developed through discussion with my peers at Nuvven.

Requirements for a system can be split into two categories; functional requirements and non-functional requirements. Functional requirements define specific behaviours or functions of a system, whereas, non-functional requirements specify benchmarks that can be used to judge how well a system operates. The functional requirements for the application are as follows:

2.1.1. Functional Requirements

2.1.1.1. User (mutual requirements between employee & customer):

- User must be able to register an account
- User must not be allowed to make more than one account under the same email
- User must be able to log in with their correct details
- User, on login must be directed to the correct landing page related to their access level
- User must be able to reset their password if required

2.1.1.2. Employee:

- Employee must be able to view all of the vehicles held within the company's fleet
- Employee must be able to view all of the vehicles held within a specified date
- Employee must be able to filter through the table of vehicle held within fleet via various fields (such as vehicle VIN number, type, model etc.)
- Employee must be able to view the list of vehicles and their availability for specified dates
- Employee must be able to add a new vehicle to the list of vehicles in the fleet (this will involve the employee entering details about the vehicle, price, type, model etc.)
- Employee must be able to view all bookings made by customers
- Employee must be able to filter through the table of bookings made by customers via various fields (such as start date, end date, username etc.)
- Employee must be able to cancel a booking on behalf of a customer
- Employee must be able to remove a vehicle from the fleet
- Employee must be able add a feature to a given date (ex. Mark a date as a holiday which may result in the vehicle prices being altered for given date)
- Employee must be able to view visualisation of booking frequency over a time period
- Employee must be able to remove user accounts
- Employee must be able to view details of customers

2.1.1.3. Customer:

- Customer must be able to specify the date they wish their rental to commence
- Customer must be able to specify the date they wish their rental to finish
- Customer must be shown a list of vehicles that are available for rent on the specific dates – must NOT be shown any vehicles that are unavailable.
- Customer must be able to change the specific start and end dates for their rental
- Customer must be able to make a booking of a vehicle from the list of available vehicles for their chosen dates
- Customer must be able to view their current booking details before making payment
- Customer must be able to select payment option between cash or card
- Customer must be able to view all of their current bookings
- Customer must be able to cancel any of their current bookings
- Customer must be able to amend a current booking
- Customer must be able to update their profile/details

2.1.1.4. Vehicle:

- Vehicle must be able to be marked as unavailable over a given time period
- Vehicle must be able to be marked as available once their booking is complete

- Vehicle must be able to be marked as available over booked dates if booking is cancelled
- Vehicle must only have one booking at a time

2.1.1.5. Dates:

- Date must have a list of available vehicles for that date
- Date must have be able to be marked as a holiday

2.1.1.6. Booking:

- Booking must be able to be cancelled/deleted
- Booking must be able to amended

2.1.2. Non-Functional Requirements

- System must be accessible by any user with an internet connection
- System and its services must be available 24/7
- System must be secure, user passwords to be encrypted locally before submission
- System must be able to display results in real time to user (response time must be feasible for a real time application)
- System must be horizontally scalable to deal with large volumes of data

With the initial requirements having been identified, each requirement was given a prioritization identifier via the MoSCoW method. Using the MoSCoW method for prioritization of requirements is a technique commonly used in the requirement gathering process. This allows the developer to reach an understanding on the importance of each requirement of the system. Although each requirement is important, some must take priority in order to produce a minimal viable product (MVP) as soon as possible. The MoSCoW method is a valuable technique when the developer is working to a fixed deadline such as this project. The term MoSCoW is derived from the first letters of each of the four prioritization groups (excluding the lower case o's):

M – Must have requirement

S – Should have requirement

C – Could have requirement

W – Won't have requirement

The enhanced list of requirements is illustrated in figure 2.1.

Must have:	<p>User must be able to register an account</p> <p>User must be able to make more than one account under the same email</p> <p>User must be able to log in with their correct details</p> <p>User must be directed to the correct landing page related to their access level</p> <p>User must be able to reset their password if required</p> <p>Customer must be able to specify the dates they wish their rental to be over</p> <p>Customer must be shown a list of vehicles that are available for rent on dates specified</p> <p>Customer must be able to make a booking of a vehicle from the list of available vehicles</p> <p>Customer must be able to view their current booking details before making payment</p> <p>Customer must be able to select payment option between cash or card</p> <p>Employee must be able to add a new vehicle to the list of vehicles in the fleet</p> <p>Employee must be able to add a feature to a given date</p> <p>Employee must be able to view all bookings made by customers</p> <p>Employee must be able to view all of the vehicles held within the company's fleet</p> <p>Vehicle must be able to be marked as unavailable over a time period</p> <p>Vehicle must be able to be marked as available after their booking is complete</p> <p>Vehicle must have one booking at a time</p> <p>Vehicle must be able to be marked as available over booked dates if booking is cancelled</p> <p>Date must have a list of available vehicles and be able to be marked as a holiday</p>
Should have:	<p>User must be able to reset their password if required</p> <p>Customer must be able to amend a current booking</p> <p>Customer must be able to change the specific start and end dates for their rental</p> <p>Customer must be able to view all of their current bookings</p> <p>Customer must be able to cancel any of their current bookings</p> <p>Customer must be able to amend a current booking</p> <p>Employee must be able to view visualisation of chosen business analytics</p> <p>Employee must be able to view all vehicles availability on a given date</p> <p>Employee must be able to filter through the table of vehicles held within fleet</p> <p>Employee must be able to filter through the table of bookings made by customers</p> <p>Employee must be able to view visualisation of booking freq. vs time period</p> <p>Bookings must be able to be cancelled or amended</p>
Could have:	<p>Employee must be able to view details of customers</p> <p>Employee must be able to remove a vehicle from the fleet</p>
Won't have:	<p>Customer must be able to update their profile/details</p> <p>Employee must be able to remove user accounts</p> <p>Employee must be able to cancel a booking on behalf of a customer</p>

Figure 2.1: The requirements of the system categorized into Must have, Should have, Could have and Won't have requirements

The prioritization of requirements may change after each iteration of development; 'won't have' requirements could become 'could have' requirements and so on.

As this project is tied to a deadline, all 'must' have requirements are to be fulfilled whilst also completing should have requirements if time allows.

Once the initial requirements list has been drawn up, is it necessary to create user stories for each case. User stories are an extremely useful tool as it will allow for the developer to acquire a high-level description of the requirements. The user stories should adhere to the following structure in order to clearly portray the purpose of the task: "AS A... I NEED... SO THAT...". The user stories for the system are shown:

2.1.3. User Stories

2.1.3.1. Customer:

- As a customer I need to be able to register my account so that I am able to log in.
- As a customer I need to be able to log in to the application so that I am able to make a booking.
- As a customer I need to be able to make a booking so that I can rent a vehicle.
- As a customer I need to be able to select dates of the rental so that I can be presented with a list of available cars.
- As a customer I need to be able to alter my selected dates before making a booking.
- As a customer I need to be able to view my list of current booking so that I am able to cancel or amend them.

2.1.3.2. Employee:

- As an employee I need to be able to add a vehicle to the fleet of vehicles and dates so that customers can book them.
- As an employee I need to be able to view all bookings and vehicles so that I can monitor and help customers with their inquiries.
- As an employee I need to be able to view all vehicles and their availability for given day so that I can monitor them.
- As an employee I need to be able to see performance graphs of how well the fleet is performing.
- As an employee I need to be able to mark a date as a holiday so that the pricing of the vehicles on that day can be altered.

On completion of the user stories a use case diagram was developed as shown in figure 2.2 and each story is added to the product backlog which will then provide an extensive collection of high level stories that can be targeted in the development process. These high level stories will then be broken down into smaller tasks which will be performed in each sprint during the development of the application.

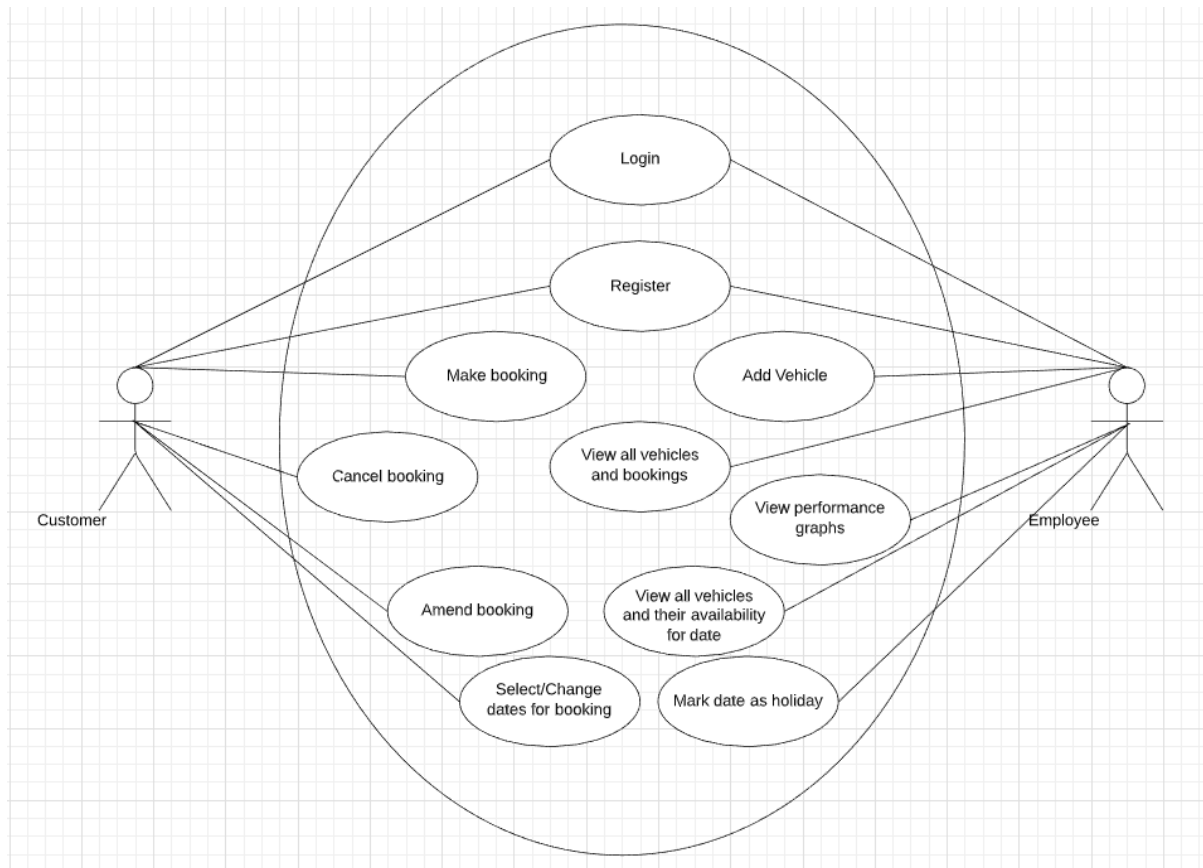


Figure 2.2: Use Case Diagram displaying the functionality of both types of users within the application

As previously stated, it is vital to ensure that the requirement gathering process is performed correctly as it defines the success of the later steps involved in the software lifecycle, it is expected that as development progresses, requirements may change and evolve. With this in mind, it is necessary to point out that development of this application will be taking an iterative approach and the requirement gathering process may be revisited at a later stage.

2.2. Design

2.2.1. System Architecture

The initial step with the design of the application was to establish a system architecture framework. The framework allows for a clear visual of the systems workflow as well as the technologies involved. This application needs to be developed with scalability and performance in mind as well as the future intention of providing machine learning to the platform. As a result, the design of the application plays a crucial role in achieving future success and they need to be designed in at the outset.

As mentioned in the introductory section of this report, it was stated that the technology stack used in this report would consist of MongoDB, Express, React.js and Node.js, more commonly known as the MERN stack. The system architecture shown in figure 2.3 reflects this selection.

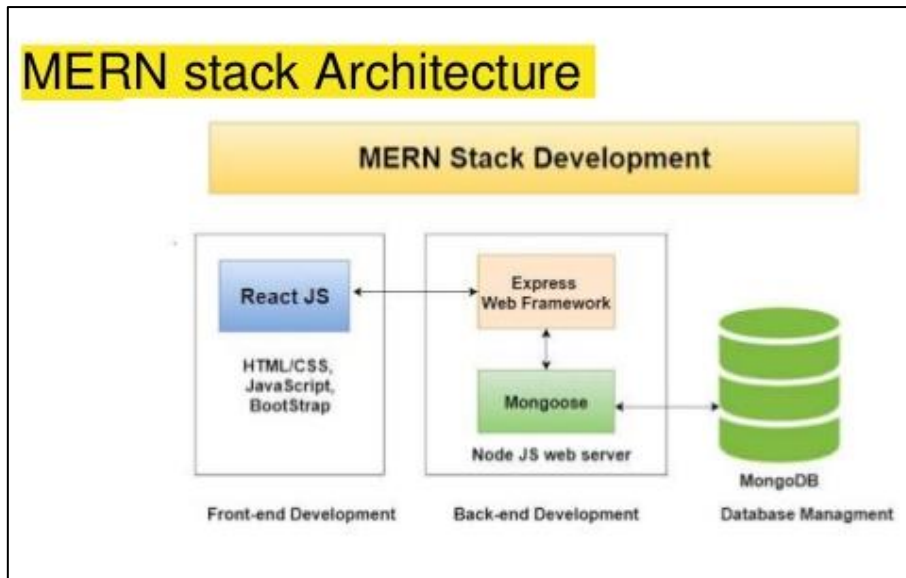


Figure 2.3: System architecture of the MERN Stack [16].

For an application that will handle large volumes of data, the database selection and design is vital. As the application will look to implement machine learning in the future, the database system of choice needs to be one that is both performant and scalable. Furthermore, as this development of this application will be working toward a deadline, it is desirable to choose a database system that does not require a substantial amount of time to configure. As a result, a NoSQL database was selected. NoSQL databases adopt a schema-on-read approach to handling data, which therefore can save a lot of time with respect to design, allowing for construction to begin ASAP. NoSQL also offer the ability to horizontally scale with much greater ease than the SQL alternative, which is more suited for this type of application. The decision is then narrowed down to which type of NoSQL system should be used. As discussed in the section 1.2.2, there are 4 types of NoSQL systems: Key-value, Document, Wide Column (Big table) and Graph based. The objective of the database within this application is to be able to store multiple different kinds of objects that can be searched for via multiple fields (ex. Look up the booking with the given start/end dates and username). Document based NoSQL systems support the requirements needed for this application and therefore a Document based NoSQL system is the database of choice. With respect to the choice of Document based NoSQL database, MongoDB was selected, MongoDB supports sharding for their horizontal scaling unlike CouchDB's asynchronous replication, as well as a free cloud based hosting service via Atlas. CouchDB also does not support a non-procedural query language and therefore implementing a CouchDB database would place more complexity on the developer.

2.2.2. Database Design

Although MongoDB is a non-relational DB and therefore implements the schema-on-read model and does not force a schema, modelling the data is still recommended. The design of an Entity-Relationship-Diagram (ERD) provides a clear model and mapping of the data being held and used in the database. The first step of this process was to determine the attributes for each entity that would be held in the database. With the use of MongoDB, these attributes are not strictly forced and can be altered as requirements change. The entities and their attributes were designed with the use of lucidchart as illustrated in figure 2.4.

User (Customer and Employee)	Vehicle	Date	Booking
_id: ObjectID email : String password: String accessLevel: enum name: String success: enum bookings: [Booking]	_id: ObjectID VIN: String type: String model: String noDoors: String noSeats: String isBooked: boolean price: enum	_id: String count: enum cars : [Vehicle] holiday: boolean	_id: ObjectID start: String end: String car: Vehicle user: String totalPrice: enum

Figure 2.4: Entities and their attributes held within the system.

As mentioned, for NoSQL systems, an ERD is still considered good practise, therefore shown in figure 2.5 is the proposed ERD for the application. There are a few notable features in the design that required some thought. Decisions needed to be made on the implementation of managing the vehicles and bookings over specified dates. With regards to vehicles, there were two possible routes that were possible. The first consideration was for each vehicle registered in the system to have a calendar which would be unique to that one vehicle. Vehicles could then have Boolean flag on each day indicating whether they were available; initially this seemed a viable option although implementation may have been complex. With efficiency and customer experience in mind, this option may have resulted in longer load times for the customer as a result of the page rendering each vehicle component having to iterate through each vehicle owned by the fleet, locate the specified date provided by the customer and query the isBooked status. Therefore, it was necessary to consider an alternate route. The design decision was to have a date itself as an entity in the database. Each individual day will be an object stored in the collection and has a unique identifier as an attribute as well as a list of vehicles that are available for that date. The unique identifier will be the date itself which will allow for calendar components used in the front end to easily provide the search criteria to retrieve the list of vehicles held within a given date. Performance overheads for this approach occur when the employee adds a vehicle to the fleet as the vehicle will be added to the overall fleet of vehicles collection, as well as to each individual date held within the database.

This second approach does create an of redundancy as; each vehicle has to be entered into the database several times as it will be stored in the Fleet collection in addition to each Date object. However, the efficiency and performance issues encountered by this approach will not affect the customer as the use of MongoDB allows for redundancy, within reason, due to its ability to scale well, meaning the impact of adding a vehicle to many lists, will not affect their load times but may affect the employee as they will need to wait until the task is completed. Moreover, with dynamic pricing as a future possibility, it is a requirement that each date be able to be marked as a holiday, therefore this proposed solution appeared to be the best option. To manage the date ranges the addition of a 'count' attribute to allow for date manipulation was also added to the Date entity.

Similar decisions were required with respect to managing Bookings. The approach of having each user entity contain a list of bookings which they have made was adopted. Each booking is generated in MongoDB with a unique identifier although for the purposes of this application the identifier of the booking is a collection of user email, start date and end date as this will be unique (based on the

requirement that a user can only have one booking over a specified time period). Similar to the problem discussed above with the vehicles, a separate list of bookings is kept as a collection of its own within the database. More discussion on this decision will be covered in the analysis part of this report as in retrospect this design choice could have been improved.

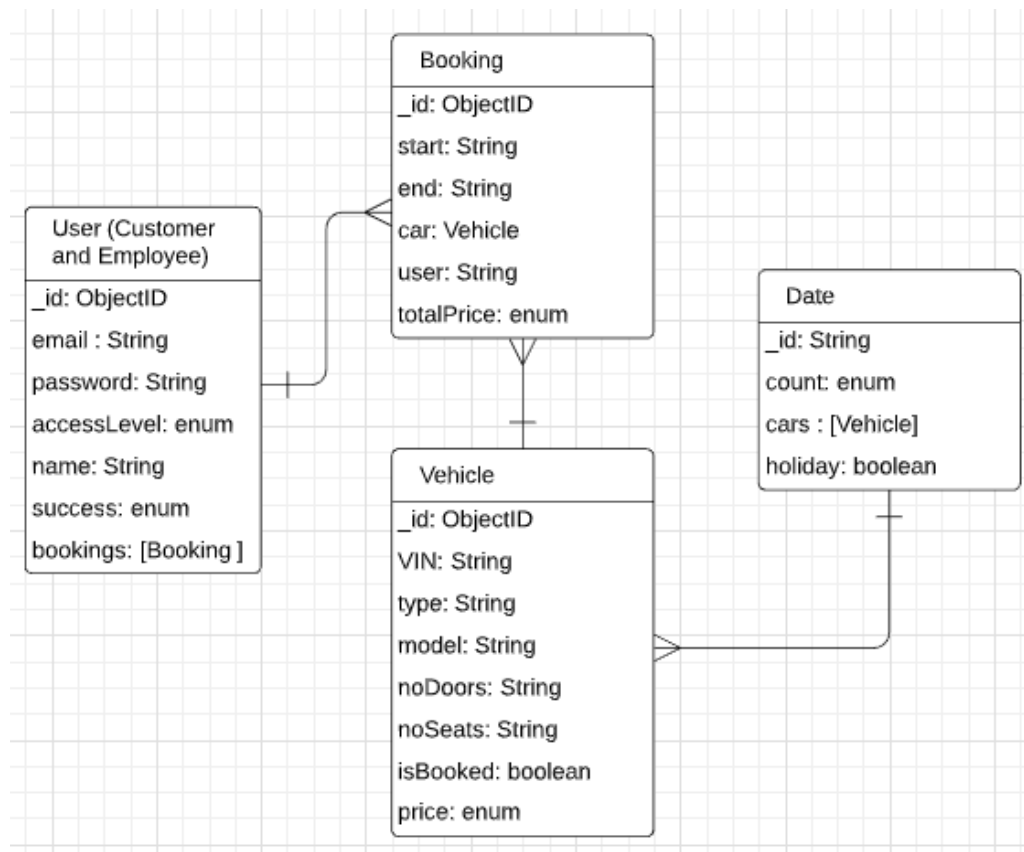


Figure 2.5: Entity relationship diagram of the system. A user can have multiple bookings, a booking may only have one user associated with it. A booking can only have one vehicle and a vehicle can have multiple bookings. A date can have multiple vehicles and a vehicle can only have one date (as a result of design, each vehicle, although the attributes are the same, is a different object for each date).

2.2.3. Front-end Design

With regards to front-end design of the application, a number of steps were involved. Foremost, the decision of the front-end technologies was made. As mentioned in the objective of this report the front-end technology of choice is React.js. React.js is a popular JavaScript library which has come to the fore in the past couple of years. Developed and maintained by Facebook, React.js has become one of the most popular front-end web technologies of the decade. The decision to select React.js as the technology of choice came down to a number of factors. React.js offers a component based approach of designing UI, allowing for single components to be dynamic and reusable. For an application of this type, there will be many reusable components such as the navigation bar for both the employee and the user. React.js is known to have a short learning curve for a developer with experience in JavaScript, however, being at a disadvantage of having no experience in JavaScript, some learning was required before development started. React libraries such as React-Bootstrap were used to help with speed and simplicity of development. One of the major advantages of using a React library such as bootstrap is that it allows for the UI to rescale based on the size of the screen of the device which it is being used

on. It does so by allowing the developer to divide the screen into 12 columns, the developer can then choose which columns are shown on screens of various sizes (xs, sm, md, lg, xl). With many people now opting to use tablets/phones for everyday tasks, this feature will allow the user to use the app on a device of any size. In addition, as covered in the literature review, technologies such as React.js and Cycle.js which implement the VDOM way of manipulation are more scalable and performant when dealing with large sets of data than their alternatives. Although Cycle.js came out as the leader with regards to performance in the report covered by Chęć et al. in [4], Cycle.js is still in its infancy and React.js is a well-documented and mature technology in comparison.

Having selected the technology, the design of front-end screens could then commence. Initial wireframes were developed through the use of Adobe XD to provide a prototype of the screens required for the UI of the application. Not only does Adobe XD provide tools to allow for the developer to visualize the screen designs before development, but it also allows for navigation through pages.

This can allow for usability tests to be performed on the prototype before development begins. Usability tests will allow for a low fidelity prototype to be tested by users and altered into a mid/high fidelity prototype before the development commences, resulting in valuable time not being wasted. Unfortunately due to the limited time of this project, usability tests were not realisable and the initial wireframes designed were therefore used for reference during development.

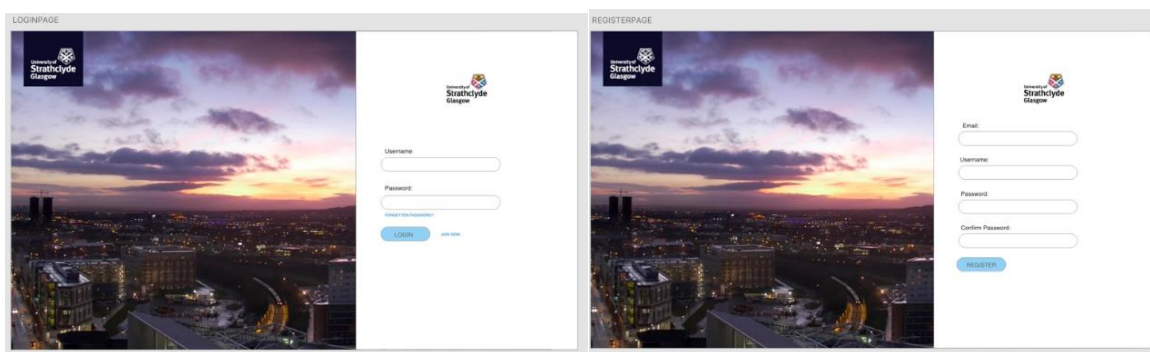


Figure 2.6: a) Wireframes for Registration Screen b) Wireframes for the Login page.

2.2.4. Back-end Design

The requirements of the backend for this application are entirely based on CRUD operations to the database. The back-end will be used for passing data from the front-end via user-input or events, such as logging in, to the database and vice versa. Design aspects of the back-end were drawn up in a UML diagram as shown in figure 2.7 showing the functionality required by the Customer and Employee entities in the back-end of the system. This is valuable as it allows the developer to know exactly what attributes are expected when either sending or receiving data from the database. Again, as mentioned in the objectives, the back-end of choice will be Node.js. Node.js allows for a 'JavaScript everywhere' approach to be used in development, having the server and client side of the application both using a single programming language. Furthermore, MongoDB (the database of choice) supports JavaScript Object Notation (JSON) data, therefore the stack of React.js, Node.js and MongoDB allows for a unified JavaScript development stack. For middleware, between the React.js front-end and Node.js back-end, Express.js was selected. Express.js is a Node.js framework and is the most popular server framework for a Node.js application. Express allows for simple routing between the front and back-end and with the use of Mongoose (a Node.js framework) access to the MongoDB database in an object oriented

way will be possible. A requirement of the system is also for the users' passwords to be securely encrypted locally before being sent to the database.

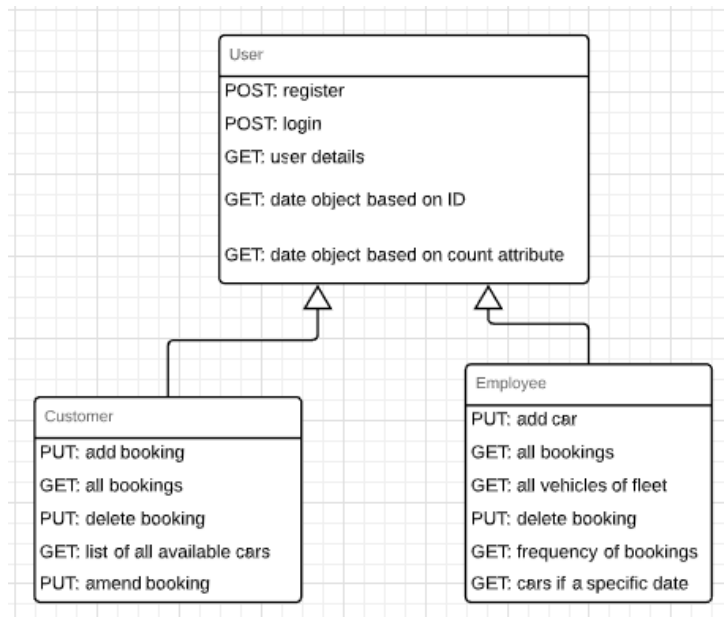


Figure 2.7: UML Diagram displaying the functions required in the back-end of the system.

2.3. Construction

Upon the completion of the design process, development and construction of the application could begin. As discussed in previous sections, construction of the application is to be performed in a dynamic way which will allow for change as the system develops and grows. With this in mind, the construction of the application will be implementing the agile method of development. Alternate methods of development include the 'traditional' waterfall method of development. Waterfall is an outdated method of development that is rarely used in modern development. It involves the same steps as its agile alternative: Requirements, Design, Development, Testing and Maintenance. However, the Waterfall method walks through each step and does not revisit any steps later in development. The result of this approach is an extremely rigid process involved for each step and increased amount of pressure put on the requirement and design steps. The implementation of Agile, mitigates the problems associated with this. Agile uses an iterative approach where each step in the software development lifecycle is revisited multiple times, in order to cater for the changes required to be applied to the system. This can allow for a more dynamic requirement gathering and design process which allows for development to proceed quicker.

The Agile method of software development contains 3 separate methodologies of approaching this iterative way of development: Extreme Programming (XP), Kanban and Scrum. Although this project is an individual effort, the adoption of the Scrum method (generally a method used for teams of developers) of Agile development was implemented. This involved reviewing the product backlog which had been generated via the user stories in the requirements gathering step of development. The overall development of the application was then split into sprints, each sprint aiming to achieve the full development of a number of user stories. With the help of free online software Jira, which provides issue tracking and tools for managing the tasks involved within sprints, each user story was assigned to a sprint in a logical format (it would not be rational to approach the development of a user booking a vehicle when the user being able to register their account had not been implemented). As development time was fairly short, each sprint was approx. 1 week long, more commonly they would

be expected to be 2-3 weeks long. Figure 2.8 shows the Jira board midway through sprint 1 as an example, where it be seen which tasks are to be started, in progress or marked as complete. The use of an Agile board allows for the developer to break down the development of the application, which may seem large and complex, into smaller, more manageable tasks. The time allocated for development of the application was 5 weeks, and therefore development would require 5 sprints in order to achieve a minimum viable product (MVP).

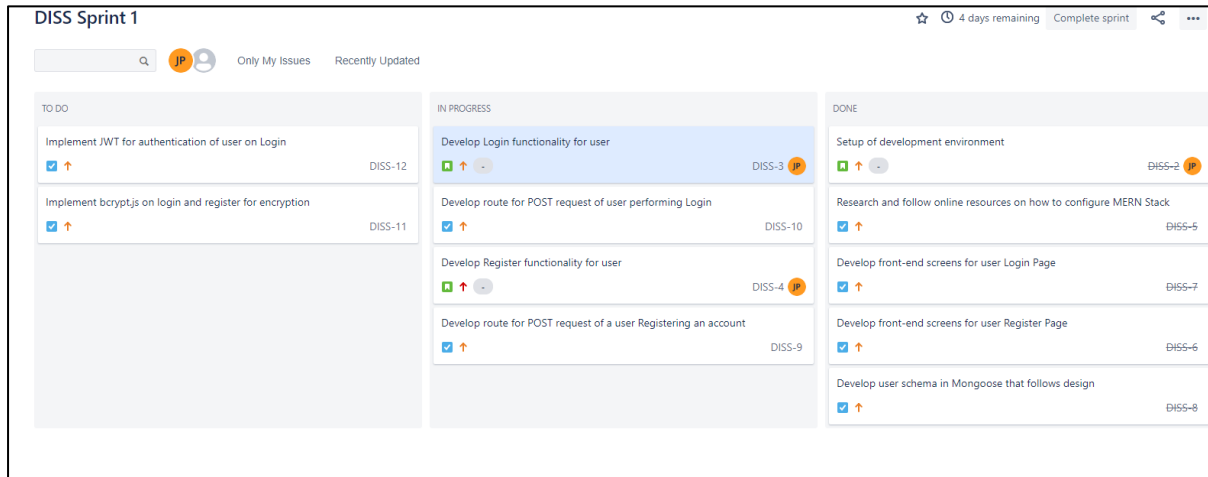


Figure 2.8: Jira board for midway through sprint 1.

2.3.1. Sprint 1

Customer:

- As a customer I need to be able to register my account so that I am able to log in.
- As a customer I need to be able to log in to the application so that I am able to make a booking.
- As a customer I need to be able to make a booking so that I can rent a vehicle.
- As a customer I need to be able to select dates of the rental so that I can be presented with a list of available cars.
- As a customer I need to be able to alter my selected dates before making a booking.
- As a customer I need to be able to view my list of current booking so that I am able to cancel or amend them.

Employee:

- As a customer I need to be able to register my account so that I am able to log in.
- As a customer I need to be able to log in to the application so that I am able to manage the fleet.
- As an employee I need to be able to add a vehicle to the fleet of vehicles and dates so that customers can book them.
- As an employee I need to be able to view all vehicles and their availability for given day so that I can monitor them.
- As an employee I need to be able to view all bookings and vehicles so that I can monitor and help customers with their inquiries.
- As an employee I need to be able to see performance graphs of how well the fleet is performing.
- As an employee I need to be able to mark a date as a holiday so that the pricing of the vehicles on that day can be altered.

Sprint 1 consisted mainly of research and set up of the development environment. With the technical stack chosen as MERN, it was now required that the stack was set up and working as expected. As this was the first time for myself as a developer having to set up all the components of the application, mistakes were made and it took a few attempts to establish a functioning end-to-end configured stack. In addition to the Setup, two user stories were approached for the initial sprint: (i) a user being able to register an account and (ii) a user being able to log in using their registered credentials. These two user stories were chosen for a number of reasons. It would be the initial step for both employee and customer when they access the application, and it would also test the full functionality of the stack, as these stories involve writing to, and reading from, the database via user inputs on the front-end. The steps involved in these user stories also required the use of a Node.js framework called Mongoose.js, and a number of libraries. A task that was mutual for both user stories of this sprint was to define the user schema with use of the Mongoose framework. Although our database is schema-less it is useful to define a schema to be loosely followed, Mongoose allows for the developer to specify what attributes are required on the creation of a user. Figure 2.9 shows how the schema is implemented in Mongoose. It can be seen that the bookings attribute of the user entity contains a list which implements another schema (BookingSchema), and these schemas match those shown section **2.2.2** and follow the same format. It is worth noting that the accessLevel field is default: 0, if the user is an employee of the company an admin will be required to alter the access level to a value of 1.

```
// Create Schema for users
const UserSchema = new Schema({
  name: {
    type: String,
    required: true
  },
  email: {
    type: String,
    required: true
  },
  password: {
    type: String,
    required: true
  },
  date: {
    type: Date,
    default: Date.now
  },
  accessLevel: {
    type: Number,
    default: 0,
  },
  success: {
    type: Boolean,
    default: false,
  },
  bookings: {
    type: [BookingSchema],
  }
});
```

Figure 2.9: Schema for a User in Mongoose

Development of the front-end screens for each user story was then required, React-Bootstrap provides a great form component that can be easily styled to cater for the developers needs and was used for both the Register and Login forms. The registration form can be seen in figure 2.10, the login form having a similar aesthetic but only requiring an email address and password.

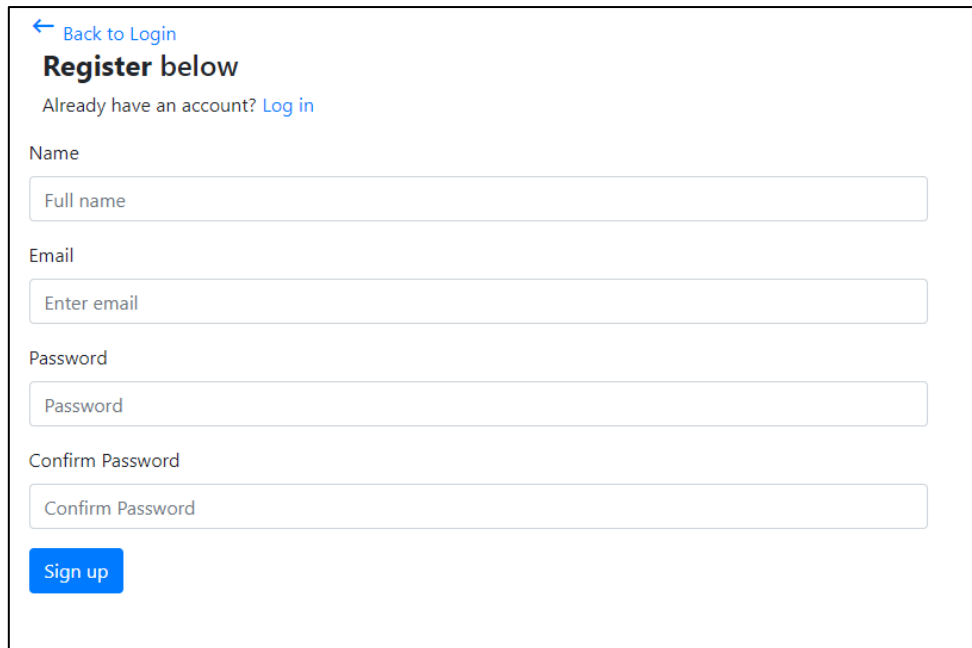
A registration form with a blue 'Back to Login' link at the top left. Below it is the heading 'Register below' and a link 'Already have an account? Log in'. The form contains four input fields: 'Name' with a placeholder 'Full name', 'Email' with a placeholder 'Enter email', 'Password' with a placeholder 'Password', and 'Confirm Password' with a placeholder 'Confirm Password'. At the bottom is a blue 'Sign up' button.

Figure 2.10: Registration form for the application

It was then necessary to write an API route to make a POST request to the MongoDB database from the server. The POST request required the front-end React.js form to provide the back-end with the details provided from the user input fields displayed. It was also necessary that this POST request perform a check on the 'user' collection within the database to ensure that there was not already a user with the same email address registered on the system. This was made simpler by the use of one of Mongooses functions – `collection.findOne()`, which will receive a condition as a parameter and return the first document that satisfies the condition. If the users' email is not held within the system, it is necessary to encrypt their inputted password locally, before posting it to the database. Bcrypt, a Node.js library enables this process to be done automatically and provides a `bcrypt.hash()` function which receives the users entered password as a parameter, hashes it and then outputs the new hashed password. Before encrypting the password it was necessary to include a check on the front-end on the Password and Confirm password field, if they are not equal to not allow for the POST request to continue, else allow for the password to be encrypted. With the new encrypted password now generated, it can be posted to the database along with the name and email inputs provided. Methods of ensuring the routes (a route determines the path of a request from the ProxyEndpoint to the TargetEndpoint [17]) provided by the back-end are functioning properly will be covered in the Testing section (2.4) of the report. Although a simple `console.log(user)` could be performed at the end of the POST method.

Throughout the construction section of this report there will be reference to POST, GET, UPDATE and DELETE requests, therefore it is important to recognise the steps involved in these requests. Having ensured the database is connected to the backend, the back-end is required to provide the routes/end points that are required (used to direct the request to the correct collection in the database) for specific requests. For a POST request, the route will receive data via its parameters and post it to the specified endpoint in the database, figure 2.11 shows the route to add a vehicle to the 'vehicles' collection.

```

// @route POST api/cars/addcar
// @desc Add Car
routerCar.post("/addcar", (req, res) => {
  const newCar = new Car({
    VIN: req.body.VIN,
    type: req.body.type,
    model: req.body.model,
    noDoors: req.body.noDoors,
    noSeats: req.body.noSeats,
    price: req.body.price,
  })
  //save the car to the database
  newCar
    .save()
    .then(car => res.json(car))
})

```

Figure 2.11: Route to api/cars/addcar receiving a newCar object to be posted to the database

As can be seen the post function receives a newVehicle object, the attributes of this new vehicle are received from the front-end. On the front-end, React provides the developer with a fetch API, this fetch API provides a fetch() method that allows for requests to the server to be performed. The fetch() method from the front-end passes the newVehicle object to the server along with the type of request required and the URL of the resource wanting to be 'fetched' as shown in figure 2.12.

```

const newCar = {
  VIN: this.state.VIN,
  type: this.state.type,
  model: this.state.model,
  noDoors: this.state.noDoors,
  noSeats: this.state.noSeats,
  price: this.state.price,
}
fetch('api/cars/addcar', {
  method: 'POST',
  headers: {
    'content-type': 'application/json'
  },
  body: JSON.stringify(newCar)
})

```

Figure 2.12: fetch() request from React to the server

This fetch request will then trigger the route request with the given information and perform the desired functionality.

GET, DELETE and PUT requests all follow in a similar fashion with regards to the fetch() request, the method attribute will reflect the desired functionality (ex. method: 'GET'). The route request on the back-end for the three can take advantage of the functionality provided by the mongoose framework. Mongoose offers functions such as collection.findOne() which will send a request to the specified

collection and return the first document that matches the parameters specified, other functions include `collection.findOneAndDelete()`, `collection.findOneAndUpdate()`, `collection.findById()` and many more. Figure 2.13 shows a simple example of a GET route request which receives the date ID from the front-end via the fetch API and queries the date collection via `Datex.findById()`, if there is no document that matches the query criteria, an error will be returned, if a document does match the query criteria the response will be returned in JSON format. The testing of ensuring these endpoints are working as intended will be discussed in the Testing section (2.4) of this report.

```
// @route get api/dates/:date_id
// @desc allows us to retrieve a date document based on dateID
routerDate.get('/:date_id', (req, res) => {
  Datex.findById(req.params.date_id, function (err, date) {
    if (err) {
      res.send(err)
    }
    res.json(date)
  })
})
```

Figure 2.13: Route for `api/dates/:date_id` to retrieve a date object based on the ID provided

Development of the login functionality was performed in a similar manner. Implementing the login functionality had to also make use of the `bcrypt` library used for securing the password. On receiving an email and password from the front-end of the application, the POST request for login will similarly check the 'users' collection within the database via the `findOne()` function provided by Mongoose, this function will find the user object within the database that has a matching email to the email provided from the user, if no user is found an error will be returned. If the corresponding user is found within the database, the `bcrypt.compare()` function will allow for the comparison of the inputted password from the user and the encrypted password stored against the email. The compare function returns a true or false value based on the comparison of the functions parameters. If the `bcrypt.compare()` function returns a value of true, a `JSONWebToken` is returned through use of the `JWT` library. This token will then provide a time out on the users' login status (default set to a year but is adjustable) and allow for the user to access pages appropriate to their access level. The completion of this user story marked the end of the first sprint.

2.3.2. Sprint 2

Customer:

- ~~As a customer I need to be able to register my account so that I am able to log in.~~
- ~~As a customer I need to be able to log in to the application so that I am able to make a booking.~~
- As a customer I need to be able to make a booking so that I can rent a vehicle.

- As a customer I need to be able to select dates of the rental so that I can be presented with a list of available cars.
- As a customer I need to be able to alter my selected dates before making a booking.
- As a customer I need to be able to view my list of current booking so that I am able to cancel or amend them.

Employee:

- ~~As a customer I need to be able to register my account so that I am able to log in.~~
- ~~As a customer I need to be able to log in to the application so that I am able to manage the fleet.~~
- As an employee I need to be able to add a vehicle to the fleet of vehicles and dates so that customers can book them.
- As an employee I need to be able to view all vehicles and their availability for given day so that I can monitor them.
- As an employee I need to be able to view all bookings and vehicles so that I can monitor and help customers with their inquiries.
- As an employee I need to be able to see performance graphs of how well the fleet is performing.
- As an employee I need to be able to mark a date as a holiday so that the pricing of the vehicles on that day can be altered.

The second sprint placed focus on the user stories related to the employee entity. The majority of the functionality required for a customer involved viewing/booking the vehicles which have been uploaded to the application by the employee, therefore approaching these tasks early in development was necessary. The targeted user stories for sprint 2 were to: develop a functional navigation bar for the employee pages; implement fully, the employee adding a vehicle to each date entity & overall fleet; implement fully, employee to be able to view all vehicles and their availability for specified dates and for the employee to be able to mark a date object as a holiday. It made sense for the front-end pages of the user stories to be developed before approaching the functionality of reading & writing the information to and from the database. The navigation bar was developed as a component within React.js. With future employee requirements in mind (view all bookings, view analytics etc.) the navigation bar was developed and can be seen in figure 2.14.

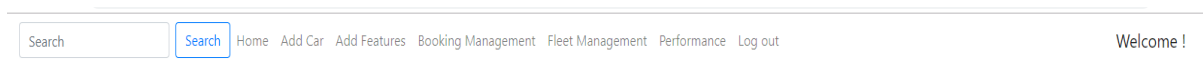
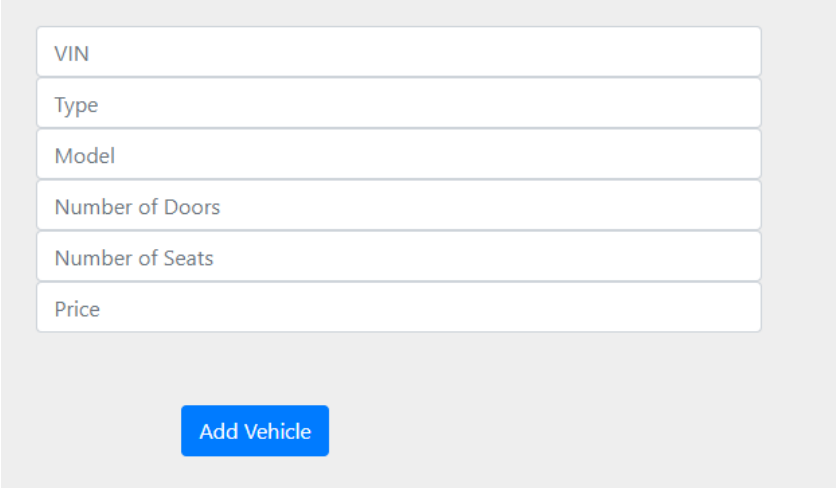


Figure 2.14: Navigation bar for the employee

Each link within the navigation bar would direct the employee to the relevant pages, with log out returning them to the login page and setting the JSONWebToken success status to false.

Logically, the implementation of the employee adding a vehicle was approached first. Similar to the registration process of a user, adding a vehicle would use a POST request to the database in order to store the inputted details to the dates and fleet entities. Once again, a schema which followed the schema provided in the design section **2.2.2** of this report was constructed using Mongoose – enforcing that no vehicle can be posted without adhering to the required fields. The fields provided

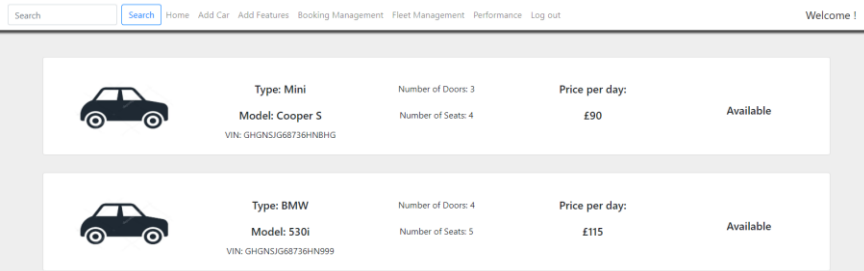
on the 'Add Vehicle' were therefore required to reflect the schema given; ensuring all required inputs were completed with appropriate values (for example a vehicles VIN must be 17 characters). Upon the employee entering the correct details of the vehicle they wish to add into the form developed (shown in figure 2.15) it was then required for two routes to be developed in the back-end, firstly one that would push a vehicle to the list of vehicles associated to a given date, and secondly to add the vehicle to the fleet collection. For appending a vehicle to the list of vehicles associated with a date, an iterative loop was necessary to add the vehicle to all dates in the database.



The image shows a form for adding a vehicle. It consists of six input fields stacked vertically, each with a label: 'VIN', 'Type', 'Model', 'Number of Doors', 'Number of Seats', and 'Price'. Below these fields is a blue button labeled 'Add Vehicle'.

Figure 2.15: Add Vehicle form

Following the completion of the employee adding a vehicle, a decision was made that for the 'Home' page of the employee, the employee would be able to view all the vehicles in the fleet and their status for a selected day. Another route was required to perform a GET operation on the database in order to return the list of vehicles for a given day. A list view of card components is used to dynamically display the vehicles. Taking advantage of Reacts mapping feature which allows for components to be rendered in real time, iterating through data received via the database, in this case the list of vehicles. Figure 2.16 shows how the card component of each vehicle is displayed to the employee.



The image shows a screenshot of a web application's 'Home' page. At the top, there is a navigation bar with a search input, a 'Search' button, and links for 'Home', 'Add Car', 'Add Features', 'Booking Management', 'Fleet Management', 'Performance', and 'Log out'. A 'Welcome !' message is on the right. Below the navigation bar, there are two vehicle cards. Each card displays a car icon, the vehicle's type and model, its VIN, the number of doors and seats, the price per day, and its availability status.



Vehicle Icon	Type	Model	VIN	Number of Doors	Number of Seats	Price per day	Availability
	Mini	Cooper S	GHGNSJG8736HN8HG	3	4	£90	Available
	BMW	530i	GHGNSJG8736HN8999	4	5	£115	Available

Figure 2.16: Card components of each vehicle

The implementation of the 'Home' page was successfully completed; however, the functionality of an employee being able to select a date to view the vehicles and their availability was pushed to a later sprint.

The 'Add Feature' page involved implementing functionality that would allow the employee to select a date and use a check box to mark it as a holiday; this would require a PUT request to the database

to allow for the selected date object to be updated. This was achieved through the use of React.js library, Datepicker. Datepicker provides a selectable calendar for React as shown in fig x.x, the selected date is then passed as a parameter into the constructed PUT request. As noted in design, each date object has a Boolean isHoliday attribute, the PUT request will therefore alter the isHoliday status of the selected date corresponding to the state of the checkbox provided.

2.3.3. Sprint 3

Customer:

- ~~As a customer I need to be able to register my account so that I am able to log in.~~
- ~~As a customer I need to be able to log in to the application so that I am able to make a booking.~~
- As a customer I need to be able to make a booking so that I can rent a vehicle.
- As a customer I need to be able to select dates of the rental so that I can be presented with a list of available cars.
- As a customer I need to be able to alter my selected dates before making a booking.
- As a customer I need to be able to view my list of current booking so that I am able to cancel or amend them.

Employee:

- ~~As a customer I need to be able to register my account so that I am able to log in.~~
- ~~As a customer I need to be able to log in to the application so that I am able to manage the fleet.~~
- ~~As an employee I need to be able to add a vehicle to the fleet of vehicles and dates so that customers can book them.~~
- ~~As an employee I need to be able to view all vehicles and their availability for given day so that I can monitor them.~~
- As an employee I need to be able to view all bookings and vehicles so that I can monitor and help customers with their inquiries.
- As an employee I need to be able to see performance graphs of how well the fleet is performing.
- ~~As an employee I need to be able to mark a date as a holiday so that the pricing of the vehicles on that day can be altered.~~

Focus was shifted from employee functionality to development of the customer functionality for the third sprint. Goals for this sprint were to firstly develop the navigation bar for the customer pages and to also achieve a customer being able to: select the dates they are wishing to rent a vehicle, alter the selected dates, and be presented with the vehicles which are available for the specified dates. As with development of the employee functionality, a navigation bar was initially developed. The navigation bar is exactly the same as the employees' apart from the fact that the links to various pages are altered to reflect the functionality required by a user (Home, Change Dates, My Bookings and Log out). Similarly, the front-end pages were developed before approaching the back-end routing and database calls. Upon logging in, the customer is to be presented with a screen that enables them to select the dates on which they wish to book their rental. Implementing the Datepicker library, as used previously, two Datepicker components are presented on the page, as shown in figure 2.17.

Figure 2.17: The two Datepicker objects presented to the customer to select their rental dates

When the dates have been selected, the customer is required to click the 'Select Dates' button provided. On click of the button, the selected dates will be added to local storage. The usage of local storage will allow for the dates to easily be accessed and manipulated when necessary.

The customer will then be directed to their 'Home' page, this page is required to display a list of all the vehicles that are available for rent over the specified dates. The vehicles will be displayed, again by using a listview of card components. The components will show all of the relevant details for the booking as well as provide a button to allow for the customer to proceed with the booking and payment screen. Implementing the logic behind ensuring that the customer was only presented with vehicles that were available involved some work. With each date there is an associated count that corresponds to the position of the date in the 'dates' collection (added in chronological order). Therefore a GET request that retrieves a date from the database based on its count is required, for each date between and including the start and end dates specified by the customer checking what vehicles are have an isBooked status equal to False.

The vehicles are displayed on the page in a similar manner to the employees homepage, view the card component with a couple of subtle differences (VIN number is not displayed to the customer and a 'Book' button is added). Making use of the mapping function provided by React.js again, the cardss can dynamically render in real time based on the list received from the database. An example of the Customer Vehicle View Card component is shown in fig 2.18.

Figure 2.18: Card view of the vehicle that is shown to the customers

Upon the completion of the user being able to view the available vehicles for the specified dates, the task of developing the functionality for the user to be able to alter the dates in which they are wishing to rent. As a result of the implementation of the user choosing their initial dates, the development of this task involved few steps. When the user selected the 'Changes Dates' link provided on the user navigation bar, they will be redirected back to the 'Select Dates' page which they were initially presented with after successfully logging in. The customer is then prompted to select the start and end dates of their rental, as done previously. The new dates selected by the user are then stored in local storage where they will overwrite the previous dates locally stored. As a result of the start and ends dates being updated via local storage, the implementation of the user being presented with the list of available vehicles for the dates remains the same.

2.3.4. Sprint 4

Customer:

- As a customer I need to be able to register my account so that I am able to log in.

- ~~As a customer I need to be able to log in to the application so that I am able to make a booking.~~
- As a customer I need to be able to make a booking so that I can rent a vehicle.
- ~~As a customer I need to be able to select dates of the rental so that I can be presented with a list of available cars.~~
- ~~As a customer I need to be able to alter my selected dates before making a booking.~~
- As a customer I need to be able to view my list of current booking so that I am able to cancel or amend them.

Employee:

- ~~As a customer I need to be able to register my account so that I am able to log in.~~
- ~~As a customer I need to be able to log in to the application so that I am able to manage the fleet.~~
- ~~As an employee I need to be able to add a vehicle to the fleet of vehicles and dates so that customers can book them.~~
- ~~As an employee I need to be able to view all vehicles and their availability for given day so that I can monitor them.~~
- As an employee I need to be able to view all bookings and vehicles so that I can monitor and help customers with their inquiries.
- As an employee I need to be able to see performance graphs of how well the fleet is performing.
- ~~As an employee I need to be able to mark a date as a holiday so that the pricing of the vehicles on that day can be altered.~~

Sprint 4's primary aim was to complete the final steps of the user being able to make a booking and for the user to also be able to view their current bookings. With the conclusion of sprint 3 allowing the user to specify the dates they wish to travel and to be presented with the list of available vehicles for those specified dates, it is now necessary for the user to be able to select a vehicle from the chosen list, proceed to a payment screen where they can review the details of their booking and confirm their vehicle.

As shown in figure 2.18 the card component that is rendered to display each available vehicle to the user has a 'Book' button on the right hand side of the card component. As with previous tasks, the front-end screen of the payment/finalize booking page was developed before any back-end logic was implemented. The final screen of the booking journey consists of two components, one of which the user can select to pay either by card or cash on collect, the other to display the overall details of the selected booking. On click of the 'Book' button, the vehicle which is related to the specific card is stored into local storage as shown in figure 2.19, the user is then also redirected to the final screen as discussed above, allowing them to finalize their booking.

```
handleClick() {
  let selectedVehicle = this.state.car
  localStorage.setItem('selectedVehicle',
    JSON.stringify(selectedVehicle))
  this.setState({
    toPayment: true,
  })
}
```

Figure 2.19: handleClick() method to locally store the selected vehicle

Similar to the way the change dates functionality worked as implemented in sprint 3, if the user were to return to the 'Home' page and select another vehicle (as they were not happy with the final booking details), they could easily do so as selecting another vehicle would overwrite the stored vehicle in local storage.

The booking details are made visible to the user via a card component similar to the styles as previously encountered in the booking journey, shown in figure 2.20. The details that are displayed within the card component are retrieved via the locally stored start/end dates and selected vehicle. The total price of the booking is the price per day of the vehicle multiplied by the number of days the rental is over.

The screenshot displays a web interface for a car rental booking. At the top, there is a navigation bar with a search input field, a 'Search' button, and links for 'Home', 'Change Dates', 'My Bookings', and 'Log out'. A 'Welcome !' message is on the right. The main content area is titled 'Booking Details:' and contains a white card with the following information: a car icon, 'From: 11082019', 'To: 13082019', 'No. of days: 3', 'Type: Mini', 'Model: Cooper S', 'Price Per Day: £90', and 'Total Price: £270'. Below the card, it states 'Cash Payment Selected' and 'Please bring the specified amount with you on collection'. There are two buttons: 'Credit/Debit Card' and 'Cash Payment', with the latter being selected. A blue 'Submit' button is at the bottom right.

Figure 2.20: Customer payment screen showing the booking details and payment options

Figure 2.20 shows the full final booking page. If the user is happy with their booking details and price, they can select their payment method, fill out the appropriate details and click the submit button. On click, the submit button is required to perform a number of functions. The first requirement of the submit button is for the vehicles `isBooked` status to be set to false for the days in which the vehicle is out for rent, this will ensure that only the vehicles flagged as available are presented to future customers. This involved developing a PUT request in which the vehicle corresponding to the provided VIN number is located within each Date of the rental. For each date the vehicles `isBooked` status is updated from False to True. The fetch request from the React.js file is shown in figure 2.21 as well as one of the more complex API routes from the Node.js in figure 2.22.

```

//remove the car from being available for the dates it is booked
let v = this.state.car.VIN
for (let i = this.state.sDate.count; i < this.state.eDate.count + 1; i++) {
  fetch('api/dates/datelist/book/' + i + '/' + v, {
    method: 'PUT',
    headers: {
      'content-type': 'application/json'
    },
  })
}
}

```

Figure 2.21: fetch() request for each date in the range to update the vehicle of the specified VIN number

```

// @route update api/dates/:date_id/carlist/:vin
// @desc allows the status of a car of a certain VIN nested within a dates status to be changed
routerDate.put('/:date_id/carlist/:vin', (req, res) => {
  Datex.findById(req.params.date_id, function (err, date) {
    let d = date.toObject()
    console.log(d)
    let cars = []
    cars = d.cars
    cars.forEach(element => {
      if (element.VIN === req.params.vin) {
        Datex.update({ '_id': req.params.date_id, 'cars.VIN': req.params.vin }, {
          '$set': {
            'cars.$.isBooked': true,
          }
        }, function (err, car) {
          res.send(car)
        })
      }
    })
  })
});
}
}

```

Figure 2.22: Route for the PUT request

Secondly, the submit button must create a booking and append it to the list of the customers bookings and also to the overall booking collection. As per the design, each booking contains a start and end date, the vehicle which is being rented and the total price of the rental, the booking which is added to the overall booking also has a user email associated with it (a design flaw which will be discussed in analysis). A PUT request is required to append to the users booking list and a POST request to add the booking to the list of overall bookings in the database.

With the user story of a customer being able to make a booking fully functional, the remainder of the sprint was spent completing the required functionality of a customer being able to view all of their current bookings. As can be seen on the customers navigation bar in figure 2.20 (image of full page) there is a link labelled 'My Bookings'. On click of this link the user is redirected to a page where they will be able to view all bookings. Again, similar to the design of the card style components used in other areas of the application, a card component was developed to allow for the user to view the booking details; each card would represent each individual booking as shown in figure 2.23.

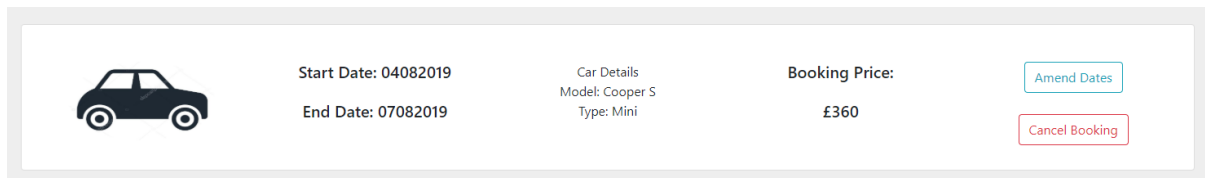


Figure 2.23: Card view of customers current bookings

The implementation of the customer being able to view their bookings involved a GET request which returned the list of bookings associated to the specific customer, then, once again, making use of Reacts' mapping feature, to display each list entry on a newly rendered card. It is also worth noting, on the right hand side of each booking card, there are two buttons, one which will cancel the specific booking and one which will allow the customer to amend the dates for their booking. The implementation of the second task of the sprint was completed with time to spare, therefore the 'Cancel Booking' functionality was appended to the current sprint. The functionality of this task required was that on click of the 'Cancel Booking' button, the booking corresponding to the button is removed from the overall bookings list, the users individual bookings list and also that the vehicle that had been booked is then made available again over the specified dates. The first two functions were simple to achieve, by identifying the booking ID and username of the customer, a DELETE request can be executed to remove the corresponding booking from the customers booking list. Removing the booking from the overall bookings collection followed a similar DELETE request to the database, however, due to a design flaw (discussed in **section 3.2**), the identifier of the booking specified for deletion was required to be the user email, start and end dates of the bookings.

```
routerBooking.delete('/deletebooking/:start/:end/:user', (req, res) => {
  Booking.findOneAndDelete({'start': req.params.start, 'end': req.params.end, 'user': req.params.user}, function(err, booking){
    if (err){
      res.send(err)
    }
    res.json(booking)
  })
})
```

Figure 2.24: Route to DELETE booking from overall booking list using the user email with the start and end dates of the booking as the identifier

The implementation of the final functionality required of the 'Cancel Booking' button followed the same process of the vehicle being made unavailable over the specified dates, which was achieved earlier in the sprint. The only difference of between the two functionalities is that for the Cancel booking, the attribute of the specified vehicle needs to be updated from True to False. This required the establishment of a route request that was identical to the one shown in figure 2.24 with the only change being:

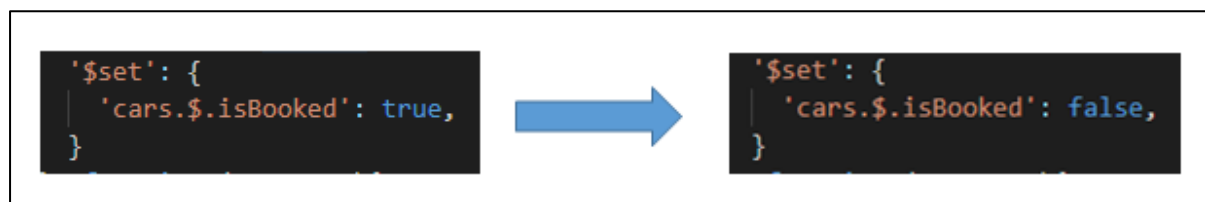


Figure 2.25: The difference of the two routes

Completion of cancel booking functionality marked the end of the 4th sprint.

2.3.5. Sprint 5

Customer:

- ~~As a customer I need to be able to register my account so that I am able to log in.~~
- ~~As a customer I need to be able to log in to the application so that I am able to make a booking.~~
- ~~As a customer I need to be able to make a booking so that I can rent a vehicle.~~
- ~~As a customer I need to be able to select dates of the rental so that I can be presented with a list of available cars.~~
- ~~As a customer I need to be able to alter my selected dates before making a booking.~~
- ~~As a customer I need to be able to view my list of current booking so that I am able to cancel or amend them.~~

Employee:

- ~~As a customer I need to be able to register my account so that I am able to log in.~~
- ~~As a customer I need to be able to log in to the application so that I am able to manage the fleet.~~
- ~~As an employee I need to be able to add a vehicle to the fleet of vehicles and dates so that customers can book them.~~
- ~~As an employee I need to be able to view all vehicles and their availability for given day so that I can monitor them.~~
- As an employee I need to be able to view all bookings and vehicles so that I can monitor and help customers with their inquiries.
- As an employee I need to be able to see performance graphs of how well the fleet is performing.
- ~~As an employee I need to be able to mark a date as a holiday so that the pricing of the vehicles on that day can be altered.~~

The aim of sprint 5 was to complete development of the MVP of the application. This meant the completion of any Must Have requirements that remained incomplete in the product backlog. The remaining Must Have requirements; employee must be able to view all of the vehicles held within the company's fleet, employee must be able to view visualisation of chosen business analytics and employee must be able to view all bookings made by customers. A further requirement of the first two requirements was that the employee must be able to filter through the tables showing all bookings and vehicles.

The first two requirements will have similar front-end and back-end functionality (both will require a filterable table containing the details of the bookings or vehicles), and so these two tasks were targeted first. React-Table is a React.js library that provides a lightweight, filterable datagrid on the front-end. The only requirements when using React-Table is when instantiating a table, the columns along with their header name and accessor are provided. The table itself will then receive a list of the data to be displayed. The data is provided via a GET request from the database. For the view all bookings table, the GET request receives a list of all bookings in the bookings collection, and for the view all vehicles in the fleet, the GET request receives a list of all vehicles in the vehicles collection. React-Table provides the filter functionality required for both tables and therefore no development work is required. React-Table also provides a page selection and number of rows per page option, the default is set to 10 rows of data per page but this can be altered via the drop down menu provided. Figure 2.26 shows the table view of all bookings in the system.

As can be seen in fig x.x, the `fetch()` will execute a GET request which will return a list of dates. As per the design, each date has an ID (its date in string format) and a list of vehicles. When the request returns the list of dates, marked as data in figure 2.27, the data is then split into two lists, `xAxis` and `yAxis`. It is vital that both lists are coherent, such that the date specified on the X-Axis corresponds to the correct booking count for that date. For each date in the list received from the data, the dates ID is push directly into the `xAxis` list and the number of booked vehicles which are found via a nested for loop is added to `yAxis`. The result of this function is two lists, X and Y which is received by the graph component and rendered to show the plot as shown in figure 2.28.

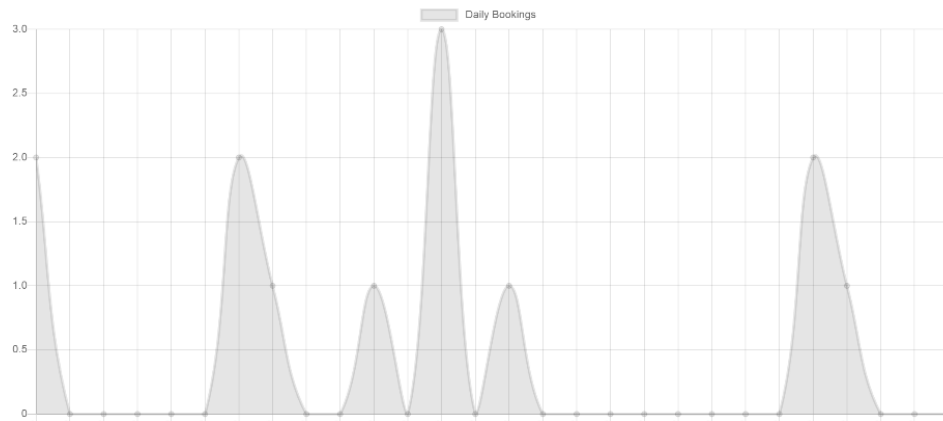


Figure 2.28: Plot of number of bookings vs dates

The remaining time of the final sprint was spent on CSS styling and adding notifications. Notifications were required to be added on a number of pages to allow the users to be alerted when their action is completed. Functions such as the employees adding a vehicle need to prompt the user if the execution was successful or not. Customers when they have confirmed their booking need to know it has gone through without having to navigate to the 'My Booking' page and so on. A notification component was developed which will receive a message property depending on which page the component is rendered, the instantiation of a Notification component receiving a message property can be seen in figure 2.29. The display of the notification is determined by a conditional render, which is a useful tool provided by React.js. The conditional render allows the developer to display specific components on a page, based on actions that have occurred.

```
{
  this.state.success && <Notification message="Account successfully registerd, please return to login page" />
}
```

Figure 2.29: Notification receiving a message property to inform the user their account has been registered

Figure 2.29 shows the conditional render of the notification based on the pages 'success' attribute. The pages default success status is Boolean False. When the registration function is complete, the success status is updated to True, and therefore the component to the right of the `&&` operator is rendered. The rendered component in this case is shown in figure 2.30.

Account successfully registerd, please return to login page

Figure 2.30: Account registered notification

A timeout can be added which will make the notification disappear after a specified amount of time, however for situations such as the registration; it is not necessary for there to be a timeout.

The final task remaining was the action of the customer being able to amend the dates of their booking. This is done through the 'My Bookings' page by clicking the 'Amend Dates' button. The implementation of this task wasn't trivial but was successfully completed. On clicking the 'Amend Dates' button the customer is taken to a new page where on the left hand side they see their previous booking dates. In the centre of the page there are two Datepicker objects which allow the customer to pick the two new dates they wish to have their rental on. The updated booking details are displayed on the right hand side of the screen and a submit button below. When the customer is happy with the booking and clicks the select button, there are a number of processes that follow. The customers current booking is deleted, this is to allow for a check to be done in the database to see if the specified vehicle is available for the new dates, for example if a vehicle was book from the 17th to the 20th of August and the customer wants to amend these dates to the 17th to the 21st if the booking is not removed initially, the vehicle status will come back as unavailable as the vehicle will be flagged as booked. Upon removing the current booking, its details are stored locally. There is then a check done on the database to check if the vehicle is now available over the new specified dates. If so the new booking is then made and added to the customers bookings, if not the customer is notified that this amendment is not possible as there is another booking that involves this vehicle over the chosen dates and re-adds their previous booking to their list.

2.4. Testing

Testing of a full stack application involves a number of various techniques, all aspects of the system are required to be tested carefully to minimise the risk of bugs and to ensure that the system satisfies the requirement set. As mentioned previously, an agile approach to development was undertaken and therefore testing of the system was done in a continuous manner. At the end of each sprint, discussed in construction section (2.3), it was required that each completed task had been tested; this included front-end testing and back-end testing.

The majority of the functionality performed in this application is either a read or write request to or from the database. Therefore, it is crucial to ensure that the routes in the back-end are directed to the correct collections/documents in the database and that the values entered by a user in the front-end are being correctly sent to the database.

Front-end testing covers a wide variety of testing strategies as there is a need for visual and usability testing as well as functionality tests. As discussed in the design section 2.2.3 with the use of AdobeXD the initial wireframes were developed. User demos of the initial wireframes would have been extremely useful, on completion of the demo a questionnaire on the usability and style of the screens for the user would have provided a great insight on changes required. This would result in the wireframes being of a higher-fidelity for the beginning of development. However, as the deadline for an MVP to be developed was pressing, these tests were not manageable as development was required to begin ASAP. Testing of the front-end during development made use of Google Chromes DevTools, which provides a set of web development tools directly built into Chrome. Chrome DevTools, allows for the developer to manipulate the DOM or CSS directly through the browser whilst also allowing for console log reads. The ability to view the console as actions take place on the front-end enable the ability to ensure that the correct information is being received or sent to/from the server. DevTools also provides additional tools that can be used for testing front-end, such as a performance monitor which allows the developer to record runtime performance of page load times, including individual times for metrics such as Scripting, Rendering and Painting. Although this is out of the scope of the

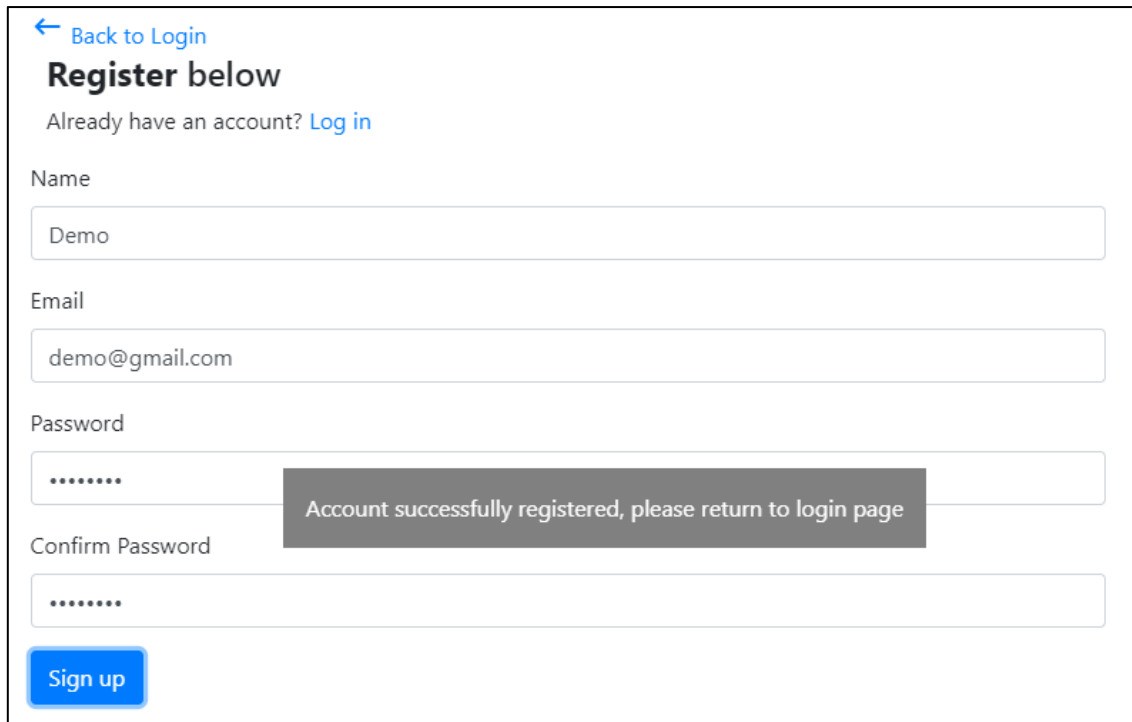
testing performed in this project, for ensuring the system is scalable and optimized this tool would be extremely useful.

An example of a common test on the front end is considered. The user has entered their details in order to register an account for the application. It is crucial that the information of the user is sent to the server in order to ensure that when the user attempts a log in they can be successful.

```
handleSubmit(e) {
  e.preventDefault();
  const newUser = {
    name: this.state.name,
    email: this.state.email,
    password: this.state.password,
    password2: this.state.password2
  };
  //console log the user to ensure the details are correct
  //passwords will not be hashed at this point - as fetch() performs the hash
  console.log(newUser)
  fetch('/api/users/register', {
    method: 'POST',
    headers: {
      'content-type': 'application/json'
    },
    body: JSON.stringify(newUser)
  })
  .then(response => response.json())
  .then(data => this.setState({
    success: true,
  }))
};
```

Figure 2.31: handleSubmit() function for the registration page

Fig 2.31 shows the handleSubmit() function of the registration page, as the user fills out the input fields provided, the state of the corresponding attribute is updated, therefore the newUser object that is being sent, reflects what is entered by the user. To ensure this, the console.log(newUser) will show, via the DevTools provided, what is being sent to the server before the fetch() method sends it. Fig x.x shows the filled out input fields for a demo user and fig x.x shows the console.log() reflecting what is entered.



[← Back to Login](#)
Register below
 Already have an account? [Log in](#)

Name

Email

Password

Confirm Password

[Sign up](#)

Account successfully registered, please return to login page

Figure 2.32: Registration form with the successful registration notification

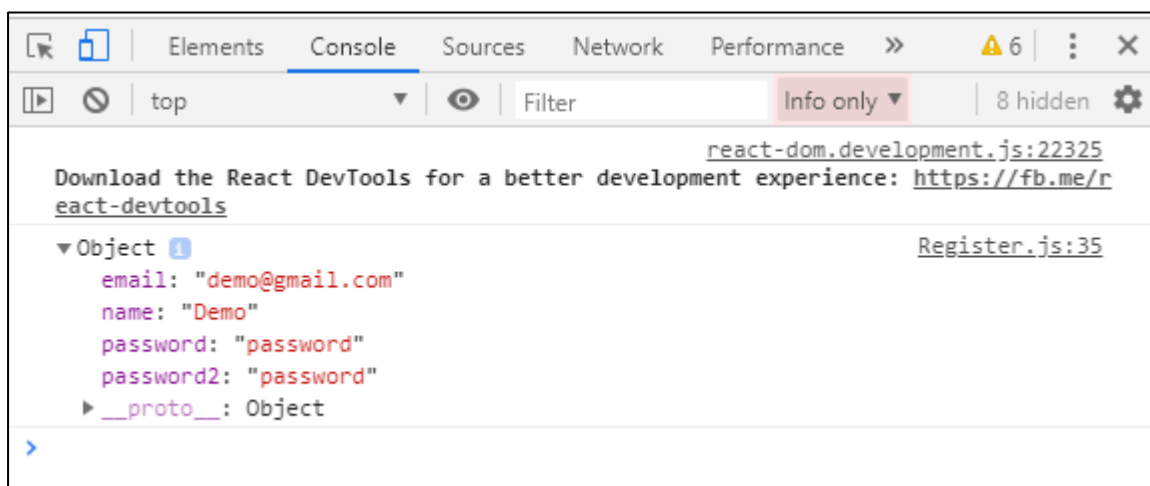


Figure 2.33: Console.log() of the user object being sent to the server from the form

As can be seen via the Console of the web page, the object being sent to the server by the `fetch()` method reflects the details that have been provided in the input fields. From the result of this test, it can be ensured that the object being sent to the server reflects what has been entered by the user. For all cases where a write from the front-end to the database is required, this technique of testing is used.

Furthermore, testing that the front-end is receiving the correct objects from the server depending on the `fetch()` request is also required. This is performed in similar manner, making use of the Console provided in Chrome, but does require that the back-end endpoints are functioning and have been tested properly, which will be discussed below. However, by way of example, assuming the back-end endpoint for retrieving the list of vehicles of the fleet has been tested and is working correctly, the data received by the front-end is logged to the console. Figure 2.34 shows the `fetch()` method to

receive the list of vehicles held within the fleet, expected results are a list of vehicle objects that contain the attributes provided in the vehicle schema.

```
componentDidMount(){
  fetch('api/cars/carlist', {
    method: 'GET',
    headers: {
      'content-type': 'application/json'
    },
  })
  .then(response => response.json())
  .then(data =>
    console.log(data)
  )
}
```

Figure 2.34: fetch() method to receive a list of vehicles held in the fleet

The results from the fetch request shown above are displayed in figure 2.35 where it can be seen that a list of vehicle objects has been returned from the database (labelled as data) and logged to the console.

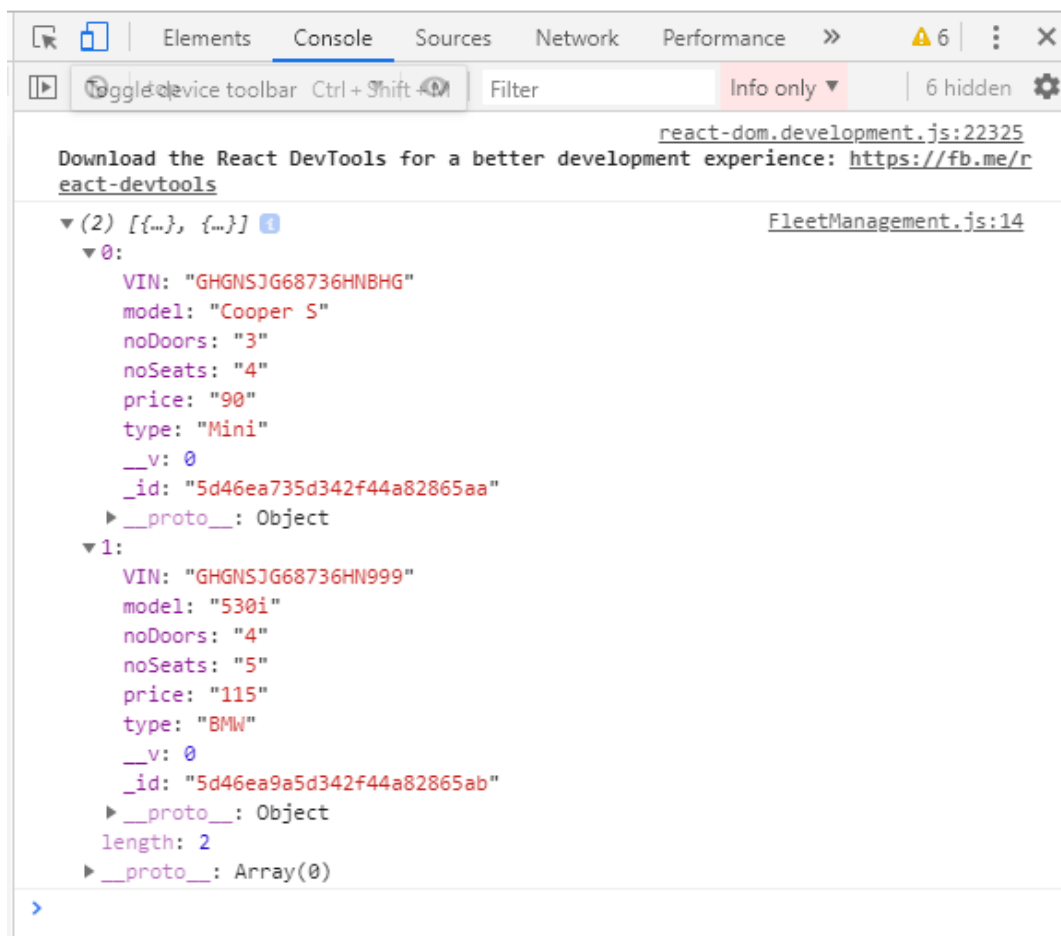


Figure 2.35: Results from the fetch() method to receive the list of vehicles held in the fleet

Again, as for all writes from the front-end, all reads to the front-end are tested in the manner shown.

The previous example of testing the front-end made the assumption that the back-end routes were functioning correctly, to ensure the back-end routes are providing the requests to the correct endpoints, testing is required. An application called Postman is used for testing these routes. Postman is a headless browser that provides a method of testing a web service without the use/interaction of a front-end. Each route on the back-end executes a query to the database, Postman allows the developer to manually execute the route they wish to test by entering the routes request in the search box provided, an example is shown in figure 2.36.

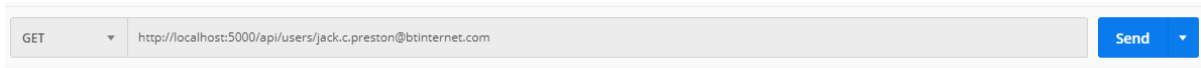


Figure 2.36: Route test on Postman testing the GET route to return a user of the specified email

Figure 2.36 shows the GET request that would be executed by the following route on the back-end to retrieve details of the user with the provided email (the server is running on localhost:5000). The route being tested is shown in figure 2.37.

```
// @ GET api/users/:email
// @ desc returns the user with the matching email
routerUser.get('/:email', (req, res) => {
  User.findOne({ email: req.params.email }, function (err, user) {
    if (err){
      res.send(err)
    }
    res.json(user)
  })
})
```

Figure 2.37: Route request to return a user object of the specified email

Postman provides a response of this query in the raw data form received from the database. What is expected from this request is the response of a user object with the email: jack.c.preston@btinternet.com in JSON format. The Postman query results is shown in figure 2.38 and is exactly as expected, concluding that the route shown in figure 2.37 is retrieving the information that is expected.

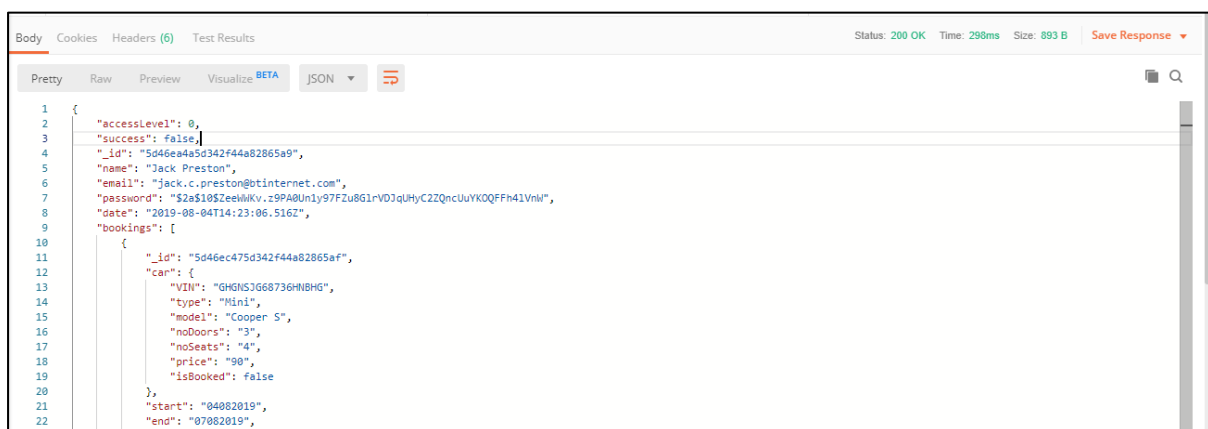


Figure 2.38: Postman query results for the GET request to receive the user object of the specified email

It is also worth noting that Postman provides some details about the request, the Status 200 signifies the request was successful and the Time and Size fields provide an insight to the developer on how

long the request took and the size of the response. All routes developed in the back-end were tested in this manner to ensure the requests required by the application were all working correctly.

All tests for both front and back-end functionalities can be found in Appendix B & Appendix C.

3. Analysis

On the completion of the final sprint, marking the end of construction, and therefore the end of development of the application, a fully functional MVP of an online booking system was produced. However, it is necessary to analyse each step in the development lifecycle independently in order to learn and improve the system in future iterations. Analysis is used to provide an insight into what went well over the lifecycle and what did not. In addition, it will identify weaknesses which can then be revisited and rectified in the future and will reinforce good practise for the developer.

3.1. Requirements Analysis

Before analysing how well each requirement was implemented in the system, it is worth reflecting on the requirements themselves. As discussed in requirements section (2.1) it would have been valuable to be able to interview some rental companies and customers in order to ensure the specified requirements of the system reflected what the users would need. However, as mentioned, this was not possible and the requirement gathering process was predominately based on peer feedback and research. Therefore, it is valuable to identify requirements that are noticeably missing and also discuss requirements of the system that were not completed in the development of the MVP.

3.1.1. Absent Requirements

Requirements absent from the initial specified requirements can be added and targeted in future iterations based on their importance. On completion of the construction, an important requirement was noted to be missing; a customer must not be able to select a date to book a vehicle that has already occurred. In hindsight, this requirement should've been a must have priority and will be of marked as high priority for the next iteration. Other requirements that were absent from the initial specification was that the past bookings from customers must be removed from their list of active bookings, if a customer makes bookings frequently their 'My Bookings' list could become large if their past bookings are present. Finally, the last requirement absent, is the customer being able to filter through the list of vehicles that are available, from a usability perspective this requirement is crucial if the fleet offers a wide range of vehicles.

3.1.2. Incomplete Requirements

There were also a number of requirements which were not completed in the development process. Some of these requirements were placed in to the 'Won't Have' category of this iteration such as, a customer being able to update their profile/details, and were therefore not expected to be included in the application MVP. However, there was one requirement in the 'Should Have' category and a couple in the 'Could Have' category that remained incomplete at the end of development, this was mainly due to time constraints but in retrospect the incomplete 'Should Have' requirement should have taken priority over some of the 'Could Have' requirements that were implemented.

The incomplete 'Should Have' requirement was that a user must be able to reset their password if they have forgotten it. The functionality of this requirement would involve the implementation of a 'Forgot Password' link on the login page which would direct the user to a separate page where they would be prompted to enter their email address for a reset link. A GET request to the database would then be implemented to ensure there is a user with the matching email entered by the user. If the database finds the user, an email will be sent containing a password link. Nodemailer, is a library for Node.js that allows for the sending of template emails, using this with JWT authentication, the reset functionality can be achieved.

The first incomplete 'Could Have' requirement was the requirement that employees must be able to view details of all registered customers. The execution of this requirement would be very similar to

the table views developed using React-Table as discussed in the construction section (2.3.5). Rather than performing a GET request to retrieve the list of vehicles or bookings held in the database, the GET request will receive a list of all customers and be mapped to a table.

Secondly, the 'Could Have' requirement that the Employee must be able to remove a vehicle from the fleet was incomplete. Implementation of this requirement would involve the Employee choosing which vehicle they wish to delete and with a DELETE request to the database remove the vehicle from the overall fleet collection and also from each date. The identifier of which vehicle to delete would be the VIN number of the chosen vehicle.

3.2. Design Analysis

Analysis of design is also required, during development poor design choices may become clear and have the need to be revisited. As agile was the development method of choice, changes to design can be altered when they are revisited on the next iteration of the application, therefore it is extremely useful to perform an analysis on the design. A number of design flaws were encountered during the development of this application and should be altered for future development. Most noticeably, the design of the booking entity stood out. The requirements specified that each user must have a list of bookings and that the employee must be able to view all bookings made by customers. The design resulted in the customer having an attribute of bookings, which was an array of booking objects, and there also being a separate collection in the database which held a list of all bookings. This design has resulted in redundant data being held within the database, and although through the use of horizontal scaling, redundancy isn't a major issue in MongoDB it is still worth removing if possible. A proposed solution for this redundancy is to remove the booking attribute from the customer. For the customer to review all of their current bookings a GET request to the database can be used to retrieve all bookings that correspond to the email address of the customer. Figure 3.1 shows an example of a booking objects stored in the overall bookings collection within the database.

```
_id: ObjectId("5d46ec475d342f44a82865ae")
start: "04082019"
end: "07082019"
> car: Object
totalPrice: 360
user: "jack.c.preston@btinternet.com"
__v: 0
```

Figure 3.1: Example of booking object stored in the overall bookings collection

As can be seen, the booking object has a user attribute which holds the email of the customer related to the booking. As a result, it will be possible to use this user field as a condition on the GET request to allow for the user to view all of their current bookings, and negate the need of the user entity having an array of booking held within their document.

There is a similar issue of redundancy concerning the storage of vehicles within the fleet. As mentioned in the design each date has a list of vehicles that can be rented on that day, any time a vehicle is added to the fleet it is also appended to every date within the system as they need to be booked against dates. Due to the design of how bookings within the system were to be implemented, it is necessary for each date to contain a list of vehicle (discussed in depth in design section 2.2.2). Therefore, the redundant collection in this scenario is the 'fleet' collection which contains all the vehicles in the fleet. The only requirement that makes use of this collection is the employee being able to view all vehicles

held within the company's fleet. However, as figure 3.2 shows the table headers of the table used to display the vehicles in the fleet the vehicles isBooked status is not relevant.

ID	Type	Model	No. Doors	No. Seats	Price

Figure 3.2: Table headers for the table views by the employee when viewing the whole fleet

As each date will have an attribute containing a list of all vehicles held within the fleet, the requirement of the employee being able to view all vehicles in the fleet can receive the list of vehicles via a GET request to the vehicle list of any date in the 'dates' collection.

The final design flaw is a flaw regarding the date object. To allow for date manipulation such as finding the number of days a booking was made for in order to provide the total price of a booking, the date object was assigned a count attribute. The count attribute of a date reflected its chronological order, for example, if the date 13/08/2019 was the first date added to the collection it was assigned count: 1, 14/08/2019 would therefore be assigned count: 2 and so on. This not only enabled the functionality of being able to calculate the number of days a rental is over but also enabled the functionality of adding a vehicle to all dates and updating the isBooked attribute of vehicles over a number of date as the count value can be used as the start and end values of the for loop required. Although this implementation works, it was not necessary and is a fault of my inexperience of dealing with external libraries. A library such as moment.js provides the developer with a simple way to manipulate and display dates and times in JavaScript. The implementation of a library such as moment.js would require a lot of work but for future implementation this may be a better solution.

With regards to the design of the system architecture of the system, I am happy with the technologies selected, React.js and Node.js provided useful libraries and the service provided by MongoDB: Atlas was easy to implement and use. The selected technology stack should enable to application to scale well whilst still maintaining performance which will allow for the introduction of ML in future.

3.3. Construction Analysis

An analysis of the requirements which were met in the development of the application is also necessary. Although the requirements have been completed, reviewing the implementation and construction of each requirement will provide an insight in to what can be improved in future iterations.

3.3.1. User Construction (Mutual functionality of Customer and Employee)

The construction for the User requirements is initially analysed. The functionality required of the User (mutual requirements between the employee & customer) were all related to registration and logging. Construction of these requirements went well. Learning how to use the JWT library provided for authentication and bcrypt.js for encryption was extremely valuable and enabled the details of the user to be stored securely. However, I would have liked to have spent more time exploring the functionality provided by JWT and make use of the timeout feature it provides.

3.3.2. Employee Construction

Construction for the Employee requirements was also positive. The majority of the requirements were met excluding the two discussed above. The use of React-Table to allow for the employees to view the list of all vehicles and bookings was made very simple by the functionality provided by the library and I am satisfied with the result. The requirement of the employee being able to add a vehicle to the fleet

collection and also each date is one that could have been constructed in a better way with respect to both front and back-end. As discussed earlier in **3.2**, the need for the fleet collection itself is unnecessary, but there are other issues that appear as a consequence of this design decision. When the employee fills in the details of the vehicle they wish to add and submit the form, the vehicle is required to be added to several dates which takes quite a bit of time (so much so that a notification of 'Adding...' was developed to show the user the request was processing). As the collection of dates gets bigger, this wait time is going to also increase. With the way the system is designed, this may be something that cannot be avoided and therefore it may lead to a change in the design in the future. Furthermore, on completion of the submission, the employee will receive a notification informing them that the vehicle was added successfully, when this occurs the fields in which the employee has filled out to add the vehicle remain with the details of the vehicle that has just been added. The result of this is that the user has to reload the page in order to clear the fields or manually clear them all, from usability perspective this implementation needs improved such that the fields reset to empty on completion and the notification is removed via a click outside of its box. The construction of the employee being able to view a plot of the booking frequency against dates went well, using the React-Chartjs library; this was able to be done quick and effectively. The option for the employee to be able to specify start and end points of the graph would have been useful from a business analysis point of view, but as this was implemented in the final sprint, there was no time. The requirement for the employee to be able to view the vehicles and their availability and their availability was implemented well, however, as the fleet grows in size, the use of the card components to display each vehicle may result in a large page. It may be a better option to have a table such as the tables used in viewing bookings and the fleet as a whole. Implementation of the employee marking a date as a holiday was done successful, although if the employee selects a date that has already been marked as a date it would be useful for them to be notified informed via a notification. Furthermore, the ability for the employee to mark a selection of dates in one click would be much more user friendly, currently only a single date can be marked as a holiday.

3.3.3. Customer Construction

The implementation of Customer requirements was also successful. Most of the requirements for the customer involved the use of dates, which as discussed earlier in the design section did not implement a date library such as moment.js. Without the use of a date library, the implementation of the customer being able to select the dates they wish to rent for etc. was well implemented. Although the complexity of the solution was made more difficult by not using a library, from a customer perspective there is no impact on load times. The requirement that the customer must not see unavailable vehicles for the specified dates was implemented well, it required some logic to be applied on render which could have impacted performance but due to using a technology that used the VDOM representation, performance issues are suppressed. The customer views the available vehicles via a card component and if the list of available vehicles is large this may cause some usability issues, this suggests the need for a filter function on the vehicles the customer can view, this requirement should be added to the list of requirements for the next iteration. The requirement of the customer being able to select their payment type went well, using a conditional render to show the correct component. Although the card payment was an option, there was no connection made to a payment service to allow for the customer to actually make a payment. Of course this functionality would not be expected for the submission application but it is worth noting if development is to continue. Implementation of the customer being able to view/cancel/amend their current bookings was successful, however, if the changes suggested in the **3.2** are applied (removing the booking list from the user object) they will need to be reimplemented to perform GET, PUT and DELETE requests on the overall bookings list and not the list within the user object. The use of two lists caused some issues in the implementation of

these requirements as when the customer makes, amends or cancels a booking, a request to both lists need to be made. Both lists contained the same values in their attributes but their ObjectIDs were different as they were two different instances of the same booking, this meant that the identifier of the booking, in order to be able to perform actions on both lists was required to be the start date, end date and users email. The complications caused by this design issue means that it should be marked as a high priority change that needs to occur in the future.

3.3.4. Vehicle, Date & Booking Construction

Requirements of the Vehicles were all taken care of through functionality of other requirements. For example, a vehicle must be able to be marked as unavailable over a given time period, this requirement is satisfied when the user performs a booking, the chosen vehicles' isBooked status is set to true for the days involved in the booking. The same holds true for the requirements of Dates and Bookings, the requirement of a date having a list of available vehicles for that date is accomplished by the functionality of the employee adding vehicles to a specific date and the user making bookings on the vehicles, the requirement of a booking to be able to be cancelled is attained when the user cancels their booking.

3.4 Client Review

For the final section of the analysis, the application was demonstrated to a fellow employee at Nuvven and their feedback captured. This client review allows for an external view point on the application and its functionalities and provide a future opportunity for improvements or requirements that can be added to the system that the developer may have not covered. The process was to walk through the user stories from an employee and customer perspective and capture areas of feedback which warrants improvement. The feedback is summarised below.

3.4.1 Employee functionality

Employee

- Employee landing page with news and stats (how many cars are in etc)
- Car images
- Visual navigation cues to be added to each page to guide user through story
- Automatic page reloading – when the employee adds a car they need to refresh themselves
- Currency in fleet management table (Pound Sterling, Euros etc.)
- Attribute of vehicle age – to allow employee and customers to view the age of car
- User entity to have contact details in addition to email (phone number etc.)

3.4.2 Customer functionality

- Sort function for list of available cars
- Car images
- Type of credit cards supported
- Amend booking is cumbersome, too many clicks
- Customer has to login before they can view any cars – can't browse cars without registering/logging in

4. Conclusion

This project set out to provide full end-to-end functionality for a core set of requirements through the implementation of a MERN stack in order to design a functioning online booking service for small rental companies. The development of the system focused on 4 key steps of the software development lifecycle: Requirement gathering, Design, Construction and Testing. The method chosen for the development of the system was Agile, allowing for an iterative approach to be taken in all of the steps specified above.

Requirement gathering for the system was the first step in the development and required some research into the typical functionalities offered from an online booking system with both the customer and Hire companies' in mind. From the customer's perspective, research into typical car rental online booking systems was done. There were a large number of online sources available for review on the web providing the functionality required from a customer in order to make a booking. As discussed in requirement section (2.1) a thorough investigation into all of the requirements from the Hire companies' perspective was not possible. Ideally, meetings would have been held with some Hire companies in order to capture all functionality they would expect from a booking management system. However, discussion with colleagues in Nuvven led to the list of Employee requirements being developed. On reflection of the requirement gathering process, the captured requirements covered most of the functionality necessary for customers and employees. However, the process did fail to include some requirements that would be crucial if the application was to be released, such as the customer must not be able to make a booking on a date that has already passed, see 3.1.1 for others. These requirements would be appended to the list of requirements and given a high priority for the next iteration.

The design of the application required that the system was devised in such a way that it could be both performant and scalable. With the technologies selected (MERN), both of these attributes can certainly be obtained but the design of the entities in the system would play a vital role. Section 3.2 discusses the poor design choices that occurred in this step of development. Notably, the design of each user having a list of bookings was flagged as a poor design choice, as there is already a list of all bookings made by all customers this design leads to redundancy that is not necessary. Similarly, the design decision made in managing a separate collection of all vehicles held within the fleet was analysed as redundant also. The design of the date entity works well, however, much less work would have been required if a library such as moment.js was implemented, one of the main lessons from this section. Overall, removing/altering the poor design choices indicated in 3.2, will ensure the system will remain performant as it scales.

Construction of the required functionality of the system followed the design process. Development of this stage went well. Almost all requirements in the 'Must Have', 'Should Have' and 'Could Have' categories were implemented well with the exception of a user being able to reset their password if required and the employee being able to view details of all registered customers. The use of breaking the user stories into tasks within the 5 sprints helped in reducing complexity of the system and ensured development remained on track with respect to deadlines. The requirements that were implemented were executed successfully, the only issues occurring in the construction of these requirements were a result of their design. This is discussed in 3.3, issues in performance occur when the employee adds a car to each date object (as it has to add it to each individual date object) which is a direct result of the design choices made. Construction also saw the use of many libraries and frameworks which was extremely useful for myself as a developer who had not been exposed on how to implement and use these and also for speed of development. The use of React-Table, React-Chartjs, and Mongoose.js etc allowed for difficult tasks to be developed with ease.

Testing was a continuous process throughout the construction phase of the application. For front-end testing, user demos would have been useful in design of screens but, were not possible due to time constraints. Front-end testing that was within the scope of the project involved the use of using Google Chromes DevTools to allow for console logs of the data being entered and received by the front-end is as expected. Chromes DevTools also provides a performance monitor which would have been useful in order to measure load times of pages as their size increased, something that could be done in the future. Back-end testing made use of the application Postman which is a headless browser, allowing for the testing of the routes and endpoints without the need of a user interface. Postman was used in a continuous manner in the development of the back-end with tests being ran on the routes as soon as they had been developed in order to ensure they were working correctly.

As a whole, the objectives of this project were achieved. The application developed exercises the full depth of the stack and implements end-to-end functionality of a core set of requirements. With the changes required in the design, as indicated in the analysis, being implemented in a future iteration the application should be able to remain performant as scale increases. The development process provided myself with some important learnings, and from the point of view of a developer with little experience in full-stack development, this project provided a very useful insight into the challenged and pitfalls.

5. Future work

This section will briefly cover possible future works for this application. As mentioned, the application developed in this project is an MVP. The analysis flagged issues that will be required to be changed in future iterations of the application in order to ensure performance and scalability. Further testing should be performed with user demos before the development of the second iteration is to commence, this will allow for feedback on usability from both an employee and customers' perspective which may result in requirement/design changes.

Furthermore, the addition of machine learning (ML) has also been introduced within the report. With the use of MongoDB, the implementation of machine learning should be possible. ML relies on big data in order to train its model and therefore a database that can scale well horizontally is crucial. The ML 'container' can be connected to the database out with the application. This will allow for the ML to not affect the performance of the application itself and work in isolation. The machine learning can be used in a number of ways to help with customer experience and also increase revenue for the Hire companies'. Features such as dynamic pricing can be introduced, where prices of vehicles can vary based on a number of factors, such as the date being a holiday. With the date entity in the system already having a holiday attribute which can be altered by the employee, this would be realisable.

6. References

1. Graham P. *Web 2.0*. Available from: <http://www.paulgraham.com/web20.html> [Accessed July 2019].
2. Microsoft. *Common Web Application Architectures*. Available from: <https://docs.microsoft.com/en-us/dotnet/standard/modern-web-apps-azure-architecture/common-web-application-architectures> [Accessed July 2019].
3. Occhino, Tom, Walke, Jordan. *JS Apps at Facebook*. Available from: <https://www.youtube.com/watch?v=GW0rj4sNH2w> [Accessed July 2019].
4. Chec D, Nowak Z. The Performance Analysis of Web Applications Based on Virtual DOM and Reactive User Interfaces, In: Kosiuczenko P, Zielinski Z. (eds.) *Engineering software systems: research and praxis*. Advances in intelligent systems and computing. Switzerland: Springer International; 2019. P.119-134.
5. Moz DB. *Introduction to the DOM*. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction [Accessed July 2019].
6. AxeMcLion. *Browser-perf gethub*. Available from: <https://github.com/axemclion/browser-perf/wiki#why-browser-perf> [Accessed July 2019].
7. Younas M. Research challenges of big data. *Service oriented computing and applications*. 2019; 13(2): 105-107.
8. Cattell R. *Scalable SQL and NoSQL data stores*. *ACM SIGMOD Record*. 2010; 39(4): 12-27.
9. Revie C. *Lecture 10* [Lecture] CS992 Database Development. University of Strathclyde. March 2019.
10. Revie C. *Lecture 5* [Lecture] CS992 Database Development. University of Strathclyde. March 2019.
11. Revie C. *Lecture 9* [Lecture] CS992 Database Development. University of Strathclyde. March 2019.
12. Eifrem E. *NOSQL – Scaling to size and scaling to complexity*. Available from: <https://neo4j.com/blog/nosql-scaling-to-size-and-scaling-to-complexity/> [Accessed July 2019].
13. Amazon. *Amazon SimpleDB developer guide*. Available from: <https://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/SDBLimits.html> [accessed July 2019].
14. Gorst D. *MongoDB vs CouchDB*. Available from: blog.scottlogic.com/2014/08/04/mongodb-vs-couchdb.html [Accessed July 2019].
15. Wake up Code. *Brewer's CAP theorem*. Available from: <https://wakeupcode.wordpress.com/2015/09/08/brewers-cap-theorem/> [accessed July 2019].
16. Roy V. *Top 10 best mern stack tutorials*. Available from: <https://www.slideshare.net/VickyRoy17/top-10-best-mern-stack-video-tutorials> [Accessed July 2019].
17. Apigee. *Understanding roots*. Available from: <https://docs.apigee.com/api-platform/fundamentals/understanding-routes> [Accessed July 2019].

Appendix A – User Guide

Installation –

Requirements: Visual Studio Code, Default browser set to Chrome.

Unzip folder into Users/name/

Open in VSC – can do so by running code . when in the directory of the file within Users/name/MERNProject.

Database setup: The user may need to whitelist their IP address within the database, please go to <https://cloud.mongodb.com/user#/atlas/login> and enter the following details.

Username: jack.c.preston@btinternet.com

Password: merndissertation1

When within the database locate the 'Network Access' link on the left hand panel and add your current IP address.

To run the application:

The server and react need to be started via the terminal within VSC.

Ensure there is nothing running on localhost:3000 or localhost:5000

In terminal within VSC enter: **npm run dev**

This will start up the full application and automatically run it in the default browser (Chrome)

Employee –

Logging in:

There is currently one employee account in the system which will allow for access to the employee side of the application.

To login as the employee, enter the following details:

Email: employee@aol.com

Password: password

Functions:

Upon logging in you will be taken to the home page which will show the vehicles and their availability for the current day.

On the navigation bar you can see links to specific pages. Add car will allow for you to add a vehicle to the system. Booking management will allow for you to see all bookings in the system. Fleet management will allow you to see all vehicles in the fleet. Add feature will allow for you to mark a date as a holiday.

Customer –

Logging in:

Logging in will require you to register an account. Once registers, log in user email and password the account was registered with.

Upon logging in you will be faced with dates to select a booking for. Please do not pick any dates later 30/09/2019 and earlier than the current date, date objects for dates further than a month in advance have not been added yet.

When the dates have been selected you will be displayed a list of vehicles which are available for rent, choose your car of choice and click the 'Book' button to be taken to payment screen. If you would like to change your dates the 'Change Dates' link on the nav bar will return you to the date picking page.

Select the booking option of choice, however there is no need to fill in card details. Click the submit button to make your booking. You can view your bookings in the 'My Bookings' page in the link provided in the nav bar. From here you can cancel or amend the dates for your booking.

Appendix B – Front-end Testing Document

As discussed in **2.4 (Testing)** the front-end testing of the application was done so with the use of Google Chrome DevTools, allowing for the developer to view console.logs as actions are performed. As the front end of this application is sending (POST, PUT, DELETE) and receiving data (GET) from the server, it is necessary to ensure both the data sent is as expected and the data being received is as expected.

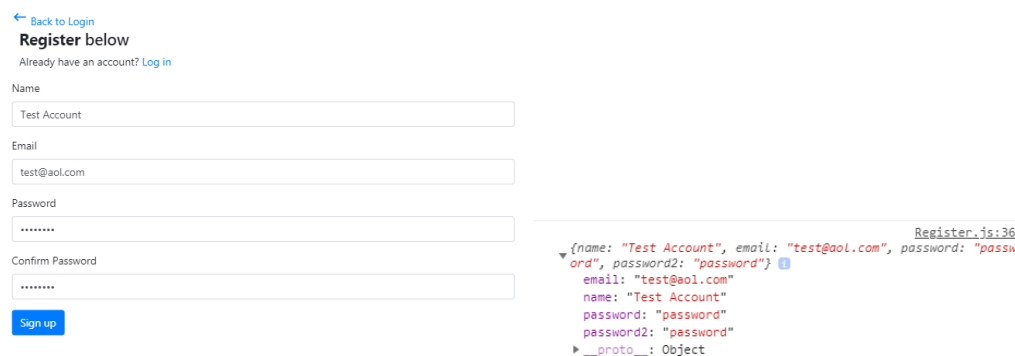
Below will show the front-end tests for each POST, PUT, DELETE and GET operation in the system.

Register:

Testing: Information being sent in POST

Expected: Details matching input fields

Test Results:



The screenshot shows a web form titled "Register below" with a link "Back to Login" and "Log in". The form has four input fields: Name (Test Account), Email (test@aol.com), Password (password), and Confirm Password (password). A "Sign up" button is at the bottom. To the right, the Chrome DevTools console shows the log for Register.js:36, displaying a JSON object with the form data: {name: "Test Account", email: "test@aol.com", password: "password", password2: "password"}.

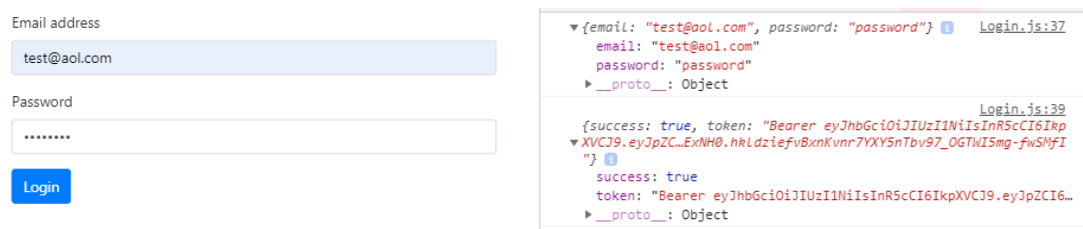
Passed

Login:

Testing: Information being sent in POST

Expected: Details matching inputted fields & JWT token of success or failure

Test Results:



The screenshot shows a web form titled "Login" with two input fields: Email address (test@aol.com) and Password (password). A "Login" button is at the bottom. To the right, the Chrome DevTools console shows two logs. The first log for Login.js:37 shows the input data: {email: "test@aol.com", password: "password"}. The second log for Login.js:39 shows the successful response: {success: true, token: "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZC5kLziefuBxNkVnr7YXY5nTbv97_OGTWISmg-fwSMfI"}.

Status Passed

Customer Pick Dates:

Testing: Selected Dates are passed into localStorage

Expected: Dates selected to be stored in local storage as the date_id (ddmmyyyy) – 17082019 & 19082019

Test Results:

Start date:

End date:

Select Dates

```
> localStorage.startDate  
< ""17082019""  
> localStorage.endDate  
< ""19082019""  
> |
```

Status: Passed

Customer Car Selection Page:

Testing: GET of all available cars for dates specified

Expected: A list of vehicles that are available for the dates to be logged to the console

Test Results:

```
▼ cars: Array(10)  
  ▶ 0: {VIN: "GHGNSJG68736HNBHG", type: "Mini", model: "Coo...  
  ▶ 1: {VIN: "GHGNSJG68736HNBHG", type: "BMW", model: "530i...  
  ▶ 2: {VIN: "ZYHNSJG68736HNBHG", type: "Audi", model: "A4"...  
  ▶ 3: {VIN: "ZYHNSJG68736HNBHG", type: "Audi", model: "A4"...  
  ▶ 4: {VIN: "GHGNSJG68736HNBHG", type: "Jaguar", model: "F...  
  ▶ 5: {VIN: "ZYHNSJG68736HNBHG", type: "Suzuki", model: "S...  
  ▶ 6: {VIN: "ZYHNSJG12736HNBHG", type: "Mini", model: "Clu...  
  ▶ 7: {VIN: "JCPNSJG68736HNBHG", type: "Fiat", model: "500...  
  ▶ 8: {VIN: "JCPNSJG68736HNBHG", type: "Vauxhall", model: "...  
  ▶ 9: {VIN: "LMNNSJG68736HNBHG", type: "BMW", model: "M4", ...  
  length: 10
```


Status: Passed

Customer to Payment Screen:

Testing: When Book of a card is clicked, selected Vehicle is added to local storage to be access in payment page.

Expected: Clicking on book of vehicle card shown below, the details on the card are stored in local storage as selectedVehicle.

Test Results:



Type: Mini
Model: Clubman

Number of Doors: 3
Number of Seats: 4

Price per day:
£95

Book

```
> localStorage.selectedVehicle  
< '{"VIN":"ZYHNSJG12736HNBHG","type":"Mini","model":"Clubman","noD  
  oors":"3","noSeats":"4","price":"95","isBooked":false}'  
>
```

Status: Passed


Customer Booking Submit:

Testing: When the customer fills out payment forms and submits, the booking correct booking details (as shown on card below) are sent to the server.

Expected: A booking with dates, details and price matching the details shown in card below and the logged users email. The vehicle selected is Booked status to be True over all dates chosen (read directly from DB).

Test Results:

Booking Details:



From: 17082019	Type: Mini	Total Price: £285
To: 19082019	Model: Clubman	
No. of days: 3	Price Per Day: £95	

```
{_id: "5d57ff4b989dc5186489fd9a", start: "17082019", end: "19082019", car: {...}, totalPrice: 285, ...}
  car: {VIN: "ZYHNSJG12736HNNXYZ", type: "Mini", ...
    end: "19082019"
    start: "17082019"
    totalPrice: 285
    user: "test@aol.com"
    __v: 0
    _id: "5d57ff4b989dc5186489fd9a"
  }
```

```
_id: "17082019"
count: 15
cars: Array
  0: Object
  1: Object
  2: Object
  3: Object
  4: Object
  5: Object
  6: Object
    VIN: "ZYHNSJG12736HNNXYZ"
    type: "Mini"
    model: "Clubman"
    noDoors: "3"
    noSeats: "4"
    price: "95"
    isBooked: true
```

```
_id: "18082019"
count: 16
cars: Array
  0: Object
  1: Object
  2: Object
  3: Object
  4: Object
  5: Object
  6: Object
    VIN: "ZYHNSJG12736HNNXYZ"
    type: "Mini"
    model: "Clubman"
    noDoors: "3"
    noSeats: "4"
    price: "95"
    isBooked: true
```

```
_id: "19082019"
count: 17
cars: Array
  0: Object
  1: Object
  2: Object
  3: Object
  4: Object
  5: Object
  6: Object
    VIN: "ZYHNSJG12736HNNXYZ"
    type: "Mini"
    model: "Clubman"
    noDoors: "3"
    noSeats: "4"
    price: "95"
    isBooked: true
```


Status: Passed

Customer View Bookings:

Testing: When the customer navigated to the page that a list of booking objects is returned corresponding to their bookings made.

Expected: As the test user only has the one booking made which is shown above – expected that this booking object will be returned and displayed.

Test Results:



Start Date: 17082019
End Date: 19082019

Car Details
Model: Clubman
Type: Mini

Booking Price:
£285

[Amend Dates](#)
[Cancel Booking](#)

```
▼ [{"...}] ⓘ  
  ▼ 0:  
    ▶ car: {VIN: "ZYHNSJG12736HXYZ", type: "Mini...  
      end: "19082019"  
      start: "17082019"  
      totalPrice: 285  
      _id: "5d57ff4b989dc5186489fd99"  
    ▶ __proto__: Object  
    length: 1  
    ▶ __proto__: Array(0)
```

Status: Passed

Customer Cancel Booking:

Testing: When the customer cancels the booking the booking is removed from the customer bookings list and also made available over the days it was booked for.

Expected: The customers list of bookings to be empty as the only one has been removed, and the Mini Clubman to be available for dates 17/08/2019, 18/08/2019 & 19/08/2019 (checked by a direct read from DB)

Test Results:

```
CustomerViewBookingsPage.js:26
▼ [] 1
  length: 0
  __proto__: Array(0)
```

<pre>_id: "18082019" count: 16 ▼ cars: Array > 0: Object > 1: Object > 2: Object > 3: Object > 4: Object > 5: Object > 6: Object VIN: "ZYHNSJG12736HNNXYZ" type: "Mini" model: "Clubman" noDoors: "3" noSeats: "4" price: "95" isBooked: false > 7: Object > 8: Object > 9: Object holiday: false</pre>	<pre>_id: "19082019" count: 17 ▼ cars: Array > 0: Object > 1: Object > 2: Object > 3: Object > 4: Object > 5: Object > 6: Object VIN: "ZYHNSJG12736HNNXYZ" type: "Mini" model: "Clubman" noDoors: "3" noSeats: "4" price: "95" isBooked: false > 7: Object > 8: Object > 9: Object holiday: false</pre>	<pre>_id: "17082019" count: 15 ▼ cars: Array > 0: Object > 1: Object > 2: Object > 3: Object > 4: Object > 5: Object > 6: Object VIN: "ZYHNSJG12736HNNXYZ" type: "Mini" model: "Clubman" noDoors: "3" noSeats: "4" price: "95" isBooked: false > 7: Object > 8: Object > 9: Object holiday: false</pre>
---	---	---


Customer amend booking:

Testing: When the customer amends a booking, if the vehicle is available for a new booking that the booking is amended. If the vehicle is not available for the amended dates that the users booking remains as the previous.


Expected: For first test it is expected that the amendment goes through as the selected vehicle is not booked out on the extended dates.

Test Results:

Original booking

	Start Date: 20082019 End Date: 23082019	Car Details Model: X5 Type: BMW	Booking Price: £480	Amend Dates Cancel Booking
---	--	---------------------------------------	------------------------	---

Dates amended to 20th – 24th. The BMW X5 is not booked on the 24th therefore should go through.


	Start Date: 20082019 End Date: 24082019	Car Details Model: X5 Type: BMW	Booking Price: £480	Amend Dates Cancel Booking
--	--	---------------------------------------	------------------------	---

Status: Passed

Expected: Customer to make a booking for BMW 116i from 25th-29th and then another booking for the same car is made from 20th to 24th. The booking made from 20th to the 24th attempts to amend the booking to include the 25th. They should be notified this is not possible.

Test results:

Original booking.

	Start Date: 25082019 End Date: 29082019	Car Details Model: 116i Type: BMW	Booking Price: £475	Amend Dates Cancel Booking
---	--	---	------------------------	---

Notification

Booking amendment unavailable for chosen vehicle, please try new dates.

Status: Passed

Employee Add Car:

Testing: The employee POST a Vehicle to database, ensuring what is being sent matches the inputted fields. And ensuring the car has been added to dates (3 random dates DB read). And added to overall fleet

Expected: Vehicle object to match input fields

Test Results:

NMIU6754FT112HJBH

Ferrari

F50

2

2

300

Add Vehicle

AddCar.js:65

{VIN: "BMKJGVC98YH653D45", type: "Ferrari", model: "F50", noDoors: "2", noSeats: "2", ...}

VIN: "BMKJGVC98YH653D45"

isBooked: false

model: "F50"

noDoors: "2"

noSeats: "2"

price: "350"

type: "Ferrari"

__proto__: Object

_id: "29082019"

count: 27

cars: Array

0: Object

1: Object

2: Object

3: Object

4: Object

5: Object

6: Object

7: Object

8: Object

9: Object

10: Object

VIN: "BMKJGVC98YH653D45"

type: "Ferrari"

model: "F50"

noDoors: "2"

noSeats: "2"

price: "350"

isBooked: false

holiday: false

_id: "25082019"

count: 23

cars: Array

0: Object

1: Object

2: Object

3: Object

4: Object

5: Object

6: Object

7: Object

8: Object

9: Object

10: Object

VIN: "BMKJGVC98YH653D45"

type: "Ferrari"

model: "F50"

noDoors: "2"

noSeats: "2"

price: "350"

isBooked: false

holiday: false

_id: "19082019"

count: 17

cars: Array

0: Object

1: Object

2: Object

3: Object

4: Object

5: Object

6: Object

7: Object

8: Object

9: Object

10: Object

VIN: "BMKJGVC98YH653D45"

type: "Ferrari"

model: "F50"

noDoors: "2"

noSeats: "2"

price: "350"

isBooked: false

holiday: false

VIN	Brand	Model	Doors	Seats	Price
LMNNSJG68736HNNPL	BMW	M4	2	5	140
BMKJGVC98YH653D45	Ferrari	F50	2	2	350

Previous

Page 1 of 1

10 rows

Next

Status: Passed

72



Employee Dashboard View Cars for Day:

Testing: The employee GET a list of cars for a given date

Expected results: Console log of a list of car objects and to be displayed on screen in cards matching object details.

Test Results:

```
EmpDash.js:31
{cars: Array(11), holiday: false, _id: "17082019", count: 15}
  cars: Array(11)
    0: {VIN: "GHGNSJG68736HNBHG", type: "Mini", ...}
    1: {VIN: "GHGNSJG68736HNN999", type: "BMW", ...}
    2: {VIN: "ZYHNSJG68736HNBHG", type: "Audi", ...}
    3: {VIN: "ZYHNSJG68736HNBHG", type: "Audi", ...}
    4: {VIN: "GHGNSJG68736HNBHG", type: "Jaguar...", ...}
    5: {VIN: "ZYHNSJG68736HNNJL", type: "Suzuki...", ...}
    6: {VIN: "ZYHNSJG12736HNNXYZ", type: "Mini", ...}
    7: {VIN: "JCPNSJG68736HNBHG", type: "Fiat", ...}
    8: {VIN: "JCPNSJG68736HNNPL", type: "Vauxha...", ...}
    9: {VIN: "LMNNSJG68736HNNPL", type: "BMW", ...}
    10: {VIN: "BMKJGVC98YH653D45", type: "Ferra...", ...}
    length: 11
    __proto__: Array(0)
  count: 15
  holiday: false
  _id: "17082019"
  __proto__: Object
```

	Type: BMW Model: 530i VIN: GHGNSJG68736HNN999	Number of Doors: 4 Number of Seats: 5	Price per day: £115	Booked
	Type: Audi Model: A4 VIN: ZYHNSJG68736HNBHG	Number of Doors: 4 Number of Seats: 5	Price per day: £100	Available

Status: Passed

Employee View All Bookings:

Testing: Employee GET a list of all bookings made by customers

Expected results: Console log of a list of all bookings made by customers and displayed in table.

Test Results:

```
EmpViewAllBookings.js:14
▼ (3) [{...}, {...}, {...}]
  ▼ 0:
    ▶ car: {VIN: "GHGNSJG68736HNBHG", type: "Jagu...
      end: "20082019"
      start: "16082019"
      totalPrice: 750
      user: "jack.c.preston@btinternet.com"
      __v: 0
      _id: "5d56fbda6938448f48fa5a8a"
      __proto__: Object
    ▼ 1:
      ▶ car: {VIN: "GHGNSJG68736HNBHG", type: "BMW"
        end: "20082019"
        start: "17082019"
        totalPrice: 460
        user: "jack.c.preston@btinternet.com"
        __v: 0
        _id: "5d5703c5322bf24f5430b04b"
        __proto__: Object
      ▼ 2:
        ▶ car: {VIN: "ZYHNSJG12736HNBHG", type: "Mini...
          end: "19082019"
          start: "17082019"
          totalPrice: 285
          user: "test@aol.com"
          __v: 0
          _id: "5d58027c3a1e7314cca17c9a"
          __proto__: Object
        length: 3
        __proto__: Array(0)
```

Booking ID	User	Start Date	End Date	Car Type	Car Model	Total Price
5d56fbda6938448f48...	jack.c.preston@btinte...	16082019	20082019	Jaguar	F-Type	750
5d5703c5322bf24f54...	jack.c.preston@btinte...	17082019	20082019	BMW	530i	460
5d58027c3a1e7314cc...	test@aol.com	17082019	19082019	Mini	Clubman	285

Status: Passed

Employee View All Vehicles:

Testing: Employee GET a list of all vehicles in fleet

Expected results: Console log of list corresponding to the list of cars in the fleet and displayed in table.

Test Results:

```
FleetManagement.js:14
(9) [{...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}, {...}]
  0: {
    VIN: "GHGNSJG68736HNBHG"
    model: "Cooper S"
    noDoors: "3"
    noSeats: "4"
    price: "90"
    type: "Mini"
    __v: 0
    _id: "5d46ea735d342f44a82865aa"
    __proto__: Object
  }
  1: { _id: "5d46ea9a5d342f44a82865ab", VIN: "GH..."
  2: { _id: "5d532df234c2043d046523d0", VIN: "ZY..."
  3: { _id: "5d53354e13371842887e97c4", VIN: "ZY..."
  4: { _id: "5d53366201913f2e6c07214b", VIN: "ZY..."
  5: { _id: "5d533762bbb5973a605efaf5", VIN: "JC..."
  6: { _id: "5d5337a2ab3fd32ab05ee35d", VIN: "JC..."
  7: { _id: "5d533a612f0eb51a9462dbd6", VIN: "LM..."
  8: { _id: "5d5804542bb59022d4a25f4e", VIN: "BM..."
    length: 9
    __proto__: Array(0)
```

ID	Type	Model	No. Doors	No. Seats	Price
GHGNSJG68736HNBHG	Mini	Cooper S	3	4	90
GHGNSJG68736HN999	BMW	530i	4	5	115
ZYHNSJG68736HNBHG	Audi	A4	4	5	100
ZYHNSJG68736HNNJL	Suzuki	Swift	3	4	80
ZYHNSJG12736HNXYZ	Mini	Clubman	3	4	95
JCPNSJG68736HNBHG	Fiat	500	3	4	80
JCPNSJG68736HNNPL	Vauxhall	Corsa	3	5	80
LMNNSJG68736HNNPL	BMW	M4	2	5	140
BMKJGVC98YH653D45	Ferrari	F50	2	2	350
PreviousPage 1 of 110 rowsNext					

Status: Passed

Appendix C – Back-end Testing Document

The following tests are strictly to ensure the routes developed in the back-end perform functionality on the correct endpoints and return/send the correct data. As discussed in **2.4** (Testing) a headless browser was used for this called Postman.

GET, PUT, POST requests will return the response in RAW JSON format and DELETE requests will be provided with details on how long the DELETE took to perform.

Each Route will be shown and the Postman response.

User Routes:

1. Register, passwords not matched example, successful example and user of same email example

```
// @route POST api/users/register
// @desc Register user
// @access Public
routerUser.post("/register", (req, res) => {
```

POST http://localhost:5000/api/users/register

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL BETA

KEY	VALUE	DESCRIPTION
name	test1	
email	test1@aol.com	
password	password	
password2	wrongpassword	
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 400 Bad Request Time: 4ms Size: 257 B Save

Pretty Raw Preview JSON

```
1
2 "password2": "Passwords must match"
3
```

POST http://localhost:5000/api/users/register

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL BETA

KEY	VALUE	DESCRIPTION
name	test1	
email	successemail@aol.com	
password	password	
password2	password	
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 125ms Size: 455 B Save

Pretty Raw Preview JSON

```
1
2 {"accessLevel": 0,
3  "success": false,
4  "_id": "5d58183471bf152418529be1",
5  "name": "test1",
6  "email": "successemail@aol.com",
7  "password": "52a5185h1v25LHfu5QzV5N3H.1nOPQm31vEmIEGozPVaoQlyC41cN6OUwa",
8  "date": "2019-08-17T14:33:24.078Z",
9  "bookings": [],
10  "__v": 0}
11
```

Untitled Request

POST

http://localhost:5000/api/users/register

Send

Params

Authorization

Headers (13)

Body

Pre-request Script

Tests

Cookies

Code

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL BETA

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> name	test1	
<input checked="" type="checkbox"/> email	test@aol.com	
<input checked="" type="checkbox"/> password	password	
<input checked="" type="checkbox"/> password2	password	
Key	Value	Description

Body

Cookies

Headers (6)

Test Results

Status: 400 Bad Request

Time: 32ms

Size: 253 B

Show

Pretty

Raw

Preview

JSON

```

1 {
2   "email": "Email already exists"
3 }

```

2. Login, wrong password, successful login

```
// @route POST api/users/login
// @desc Login user and return JWT token
// @access Public
routerUser.post("/login", (req, res) => {
```

POST http://localhost:5000/api/users/login Send

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL BETA

	KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/>	email	successemail@aol.com	
<input checked="" type="checkbox"/>	password	password	
	Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 95ms Size: 438 B Sav

Pretty Raw Preview JSON

```
1 {  
2   "success": true,  
3   "token": "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ2ZC1lZjVkbktgdW90dXMzFjZeHJqcyQyODUwZWJSISIm5hdWI0LjBZXm9SImlhbnN1AhJfodtK3hjASHzkz0g.cEO38-F4Lvhrn1Ahpjs6xr767gQ5mmv0P-PJ7fxI"  
4 }
```

POST

http://localhost:5000/api/users/login

Send

Params

Authorization

Headers (13)

Body

Pre-request Script

Tests

Cookies

Code

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL BETA

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> email	sucessemail@aol.com	
<input checked="" type="checkbox"/> password	rwrong	
Key	Value	Description

Body

Cookies

Headers (6)

Test Results

Status: 400 Bad Request

Time: 96ms

Size: 263 B

Save

Pretty

Raw

Preview

JSON

1

2

3

```
{
  "passwordincorrect": "Password incorrect"
}
```

3. Get all users

```
// @route GET api/users/userlist
// @desc returns a list of all users on DB
// @access Public
routerUser.get('/userlist', (req, res) => {
```

GET http://localhost:5000/api/users/userlist

Params Authorization Headers (13) Body Pre-request Script Tests

none form-data x-www-form-urlencoded raw binary GraphQL BETA

KEY	VALUE	DESCRIPTION
email	successemail@aol.com	
password	rwrong	
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 31ms Size: 2.52 KB Save

Pretty Raw Preview JSON

```
1 {
2   "accessLevel": 0,
3   "success": false,
4   "_id": "5d46ea4a5d342f44a82865a9",
5   "name": "Jack Preston",
6   "email": "jack.c.preston@btinternet.com",
7   "password": "52a51857eeakKv.z9PA0Unly977Zu861rV03qUyC22QncUuYKQOFH41VnH",
8   "date": "2019-08-04T14:23:06.516Z",
9   "bookings": [
10     {
11       "_id": "5d5703c5322bf24f5430b04c",
12       "car": {
13         "VIN": "GKHNSJG68736H0999",
14         "type": "BMW",
15         "model": "530i",
16         "noDoors": "4",
17         "noSeats": "5",
18         "price": "115",
19         "isBooked": false
20       },
21       "start": "17082019",
22       "end": "20082019",
23       "totalPrice": 460
24     }
25   ],
26   "_v": 0
27 }
```

Can't show all.

4. GET user details of specified email

```
// @ GET api/users/:email
// @ desc returns the user with the matching email
routerUser.get('/:email', (req, res) => {
```

GET http://localhost:5000/api/users/test@aol.com

Params Authorization Headers (13) Body Pre-request Script Tests

none form-data x-www-form-urlencoded raw binary GraphQL BETA

KEY	VALUE	DESCRIPTION
email	successemail@aol.com	
password	rwrong	
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 32ms Size: 668 B Save

Pretty Raw Preview JSON

```
1 {
2   "accessLevel": 0,
3   "success": false,
4   "_id": "5d57f01c80993f1c1878ad01",
5   "name": "Test Account",
6   "email": "test@aol.com",
7   "password": "52a51857eeakKv.z9PA0Unly977Zu861rV03qUyC22QncUuYKQOFH41VnH",
8   "date": "2019-08-17T12:42:04.365Z",
9   "bookings": [
10     {
11       "_id": "5d58027c3a1e7314cca17c9b",
12       "car": {
13         "VIN": "ZYHNSJG12736H0Y2",
14         "type": "Mini",
15         "model": "CLL0man",
16         "noDoors": "3",
17         "noSeats": "4",
18         "price": "95",
19         "isBooked": false
20       },
21       "start": "17082019",
22       "end": "18082019",
23       "totalPrice": 205
24     }
25   ],
26   "_v": 0
27 }
```


5. GET all bookings for a user email

```
// @ GET api/users/bookings/:email
// @ desc retrieve all the bookings a specific user has
routerUser.get('/bookings/:email', (req, res) => {
```

GET Send

Params Authorization Headers (13) **Body** Pre-request Script Tests Cookies Code

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 35ms Size: 426 B Save

Pretty Raw Preview JSON Copy

```
1 {
2   "_id": "5d58027c3a1e7314cca17c9b",
3   "car": {
4     "VIN": "ZYHNS3G12736H0XZ",
5     "type": "Wlsni",
6     "model": "Clubman",
7     "noDoors": "3",
8     "noSeats": "4",
9     "price": "95",
10    "isBooked": false
11  },
12  "start": "17082019",
13  "end": "19082019",
14  "totalPrice": 285
15 }
16
17
```

6. DELETE a booking from users booking list of certain ID

```
// @ PUT api/users/:email/:bookingID
// @ desc delete the booking of a specific booking ID from a users booking list
routerUser.put('/bookings/:email/:booking_id', (req, res) => {
```

PUT Send

Params Authorization Headers (13) **Body** Pre-request Script Tests Cookies Code

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 50ms Size: 499 B Save

Pretty Raw Preview JSON Copy

```
1 {
2   "n": 1,
3   "nModified": 1,
4   "opTime": {
5     "ts": "6726147187601833985",
6     "t": 4
7   },
8   "electionId": "7fffffff8000000000000004",
9   "ok": 1,
10  "operationTime": "6726147187601833985",
11  "clusterTime": {
12    "clusterTime": "6726147187601833985",
13    "signature": {
14      "hash": "3k25UCYPd8uPahlt9HyZQpfzp53Q=",
15      "keyId": "6706416558815576065"
16    }
17  }
18 }
```

Date Routes:

1. GET all dates in DB

```
// @route GET api/dates/dateslist
// @description returns list of all DATES in DB
routerDate.get('/dateslist', (req, res) => {
```

GET http://localhost:5000/api/dates/dateslist

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 69ms Size: 38.09 KB Save

Pretty Raw Preview JSON

```
1 {
2   "cars": [
3     {
4       "VIN": "GHGNSJG68736HMBHG",
5       "type": "Mini",
6       "model": "Cooper S",
7       "noDoors": "3",
8       "noSeats": "4",
9       "price": "90",
10      "isBooked": false
11    },
12    {
13      "VIN": "GHGNSJG68736H1999",
14      "type": "BMW",
15      "model": "530i",
16      "noDoors": "4",
17      "noSeats": "5",
18      "price": "115",
19      "isBooked": true
20    },
21    {
22      "VIN": "ZYHNSJG68736HMBHG",
23      "type": "Audi",
24      "model": "A4",
25      "noDoors": "4",
26      "noSeats": "5",
27      "price": "100",
28      "isBooked": false,
29      "success": false
30    },
31  ],
32  {
33    "VIN": "ZYHNSJG68736HMBHG"
34  }
35 }
```

2. PUT date as holiday

```
// @route update api/dates/:date_id
// @desc allows the user to update a specific date - CHANGING FEATURES etc.
routerDate.put('/:date_id', (req, res, next) => {
```

PUT http://localhost:5000/api/dates/20082019

Pretty Raw Preview JSON

```
65 {
66   "noSeats": "4",
67   "price": "90",
68   "isBooked": false
69 },
70 {
71   "VIN": "JCPHGJG68736HMBHG",
72   "type": "Fiat",
73   "model": "500",
74   "noDoors": "3",
75   "noSeats": "4",
76   "price": "80",
77   "isBooked": false
78 },
79 {
80   "VIN": "JCPHGJG68736HMBPL",
81   "type": "Vauxhall",
82   "model": "Corsa",
83   "noDoors": "3",
84   "noSeats": "5",
85   "price": "80",
86   "isBooked": false
87 },
88 {
89   "VIN": "LYHNSJG68736HMBPL",
90   "type": "BMW",
91   "model": "1M",
92   "noDoors": "2",
93   "noSeats": "3",
94   "price": "140",
95   "isBooked": false
96 },
97 {
98   "VIN": "BHKJG68736HMB3045",
99   "type": "Ferrari",
100  "model": "F50",
101  "noDoors": "2",
102  "noSeats": "2",
103  "price": "350",
104  "isBooked": false
105 },
106 },
107 "holiday": true,
108 "id": "20082019",
109 "count": 18
110 }
```

3. GET date based on its count

```
// @route get api/dates/:count_id
// @desc allows us to access a car list based on its count returns date of count specified
routerDate.get('/datelist/:count', (req, res) => {
```

GET http://localhost:5000/api/dates/datelist/18

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 36ms Size: 1.56 KB Save

Pretty Raw Preview JSON

```
1 {
2   "cars": [
3     {
4       "VIN": "GHGNS3G68736HHBHG",
5       "type": "Mini",
6       "model": "Cooper S",
7       "noDoors": "3",
8       "noSeats": "4",
9       "price": "90",
10      "isBooked": false
11    },
12    {
13      "VIN": "GHGNS3G68736HH999",
14      "type": "BMW",
15      "model": "530i",
16      "noDoors": "4",
17      "noSeats": "5",
18      "price": "115",
19      "isBooked": true
20    },
21    {
22      "VIN": "ZYHNS3G68736HHBHG",
23      "type": "Audi",
24      "model": "A4",
25      "noDoors": "4",
26      "noSeats": "5",
27      "price": "100",
28      "isBooked": false,
29      "success": false
30    },
31    {
32      "VIN": "ZYHNS3G68736HHBHG",
33      "type": "Audi"
```

4. PUT add car to date based on count

```
// @route update api/dates/:count_id
// @desc allows the user to add a car based on its count - required for add all date car lists
routerDate.put('/datelist/:count', (req, res, next) => {
```

PUT http://localhost:5000/api/dates/datelist/18

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL BETA

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> VIN	BNHJGK879JK876908	
<input checked="" type="checkbox"/> type	BMW	
<input checked="" type="checkbox"/> model	116i	
<input checked="" type="checkbox"/> noDoors	5	
<input checked="" type="checkbox"/> noSeats	5	
<input checked="" type="checkbox"/> price	90	
<input checked="" type="checkbox"/> isBooked	false	
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 64ms Size: 1.56 KB Save

Pretty Raw Preview JSON

```
1 {
2   "cars": [
3     {
4       "VIN": "GHGNS3G68736HHBHG",
5       "type": "Mini",
6       "model": "Cooper S",
7       "noDoors": "3",
8       "noSeats": "4",
9       "price": "90",
10      "isBooked": false
11    },
12    {
13      "VIN": "GHGNS3G68736HH999",
14      "type": "BMW",
15      "model": "530i",
16      "noDoors": "4",
17      "noSeats": "5",
18      "price": "115",
19      "isBooked": true
```

5. PUT to change the isBooked status of a car within a date based on its VIN

```
// @route update api/dates/:date_id/carlist/:vin
// @desc allows the status of a car of a certain VIN nested within a dates status to be changed
routerDate.put('/:date_id/carlist/:vin', (req, res) => {
```

PUT `http://localhost:5000/api/dates/20082019/carlist/GHGNSJG68736HNBHG` Send

Params Authorization Headers (13) **Body** Pre-request Script Tests Cookies Code

☐ none ☐ form-data ☒ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL BETA

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> VIN	BNHJGK879K876908	
<input checked="" type="checkbox"/> type	BMW	
<input checked="" type="checkbox"/> model	116i	
<input checked="" type="checkbox"/> noDoors	5	
<input checked="" type="checkbox"/> noSeats	5	
<input checked="" type="checkbox"/> price	90	
<input checked="" type="checkbox"/> isBooked	false	
Key	Value	Description

body Cookies Headers (6) Test Results Status: 200 OK Time: 67ms Size: 499 B Sa

Pretty Raw Preview **JSON** ⌵

```
1 {
2   "n": 1,
3   "nModified": 1,
4   "opTime": {
5     "ts": "6726151254935863297",
6     "t": 4
7   },
8   "electionId": "7fffffff0000000000000004",
9   "ok": 1,
10  "operationTime": "6726151254935863297",
11  "$clusterTime": {
12    "clusterTime": "6726151254935863297",
13    "signature": {
14      "hash": "XLrrE5czR7Kq8UfeU5h12oLmM=",
15      "keyId": "6706416558815576065"
16    }
17  }
18 }
```

6. PUT to change isBooked of cart within date count based on its VIN

```
// @route update /api/dates/:count/carlist/:vin
// @desc allows when a booking is submitted for the date with the given count to update the vehicle that is being booked to true
routerDate.put('/:date_id/book/:count/:vin', (req, res) => {
```

PUT `http://localhost:5000/api/dates/dateid/book/18/ZYHNSJG68736HNBHJL` Send

Params Authorization Headers (13) **Body** Pre-request Script Tests Cookies Code

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

body Cookies Headers (6) Test Results Status: 200 OK Time: 93ms Size: 499 B Save

Pretty Raw Preview **JSON** ⌵

```
1 {
2   "n": 1,
3   "nModified": 1,
4   "opTime": {
5     "ts": "67261523508152523778",
6     "t": 4
7   },
8   "electionId": "7fffffff0000000000000004",
9   "ok": 1,
10  "operationTime": "67261523508152523778",
11  "$clusterTime": {
12    "clusterTime": "67261523508152523778",
13    "signature": {
14      "hash": "63xwpc79Mc3u/BMH12d041PqQ=",
15      "keyId": "6706416558815576065"
16    }
17  }
18 }
```

```
{
  "_id": "20082019"
  count: 18
  cars: Array
    > 0: Object
    > 1: Object
    > 2: Object
    > 3: Object
    > 4: Object
    > 5: Object
      VIN: "ZYHNSJG68736HNBHJL"
      type: "Suzuki"
      model: "Swift"
      noDoors: "3"
      noSeats: "4"
      price: "80"
      isBooked: true
    > 6: Object
    > 7: Object
    > 8: Object
    > 9: Object
    > 10: Object
  holiday: true
}
```

7. Cancel Booking based on car VIN and Date count

```
// @route update /api/dates/datelist/cancelbook/:count/:vin
// @desc allows when a booking is cancelled for it to be cancelled based on its VIN and date count
routerDate.put('/datelist/cancelbook/:count/:vin', (req, res) => {
```

PUT http://localhost:5000/api/dates/datelist/cancelbook/18/ZYHNSJG68736HNNJL Send

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 59ms Size: 499 B Save

Pretty Raw Preview JSON Copy

```
1 {
2   "n": 1,
3   "modified": 1,
4   "opTime": {
5     "ts": "6726153630052777986",
6     "t": 4
7   },
8   "electionId": "7fffffff0000000000000004",
9   "ok": 1,
10  "operationTime": "6726153630052777986",
11  "$clusterTime": {
12    "clusterTime": "6726153630052777986",
13    "signature": {
14      "hash": "Q/Fnk1MFIvYEXr3npVOVZn7cS9u=",
15      "keyId": "6706416558815576065"
16    }
17  }
18 }
```

_id: "20082019"
count: 18
cars: Array
 > 0: Object
 > 1: Object
 > 2: Object
 > 3: Object
 > 4: Object
 > 5: Object
 VIN: "ZYHNSJG68736HNNJL"
 type: "Suzuki"
 model: "Swift"
 noDoors: "3"
 noSeats: "4"
 price: "80"
 isBooked: false
 > 6: Object
 > 7: Object
 > 8: Object
 > 9: Object
 > 10: Object
holiday: true

8. GET list of cars based on dates count

```
// @route get api/dates/:count_id
// @desc allows us to access a car list based on its count returns date of count specified
routerDate.get('/datelist/:count', (req, res) => {
```

GET http://localhost:5000/api/dates/18 Send

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 37ms Size: 1.56 KB Save

Pretty Raw Preview JSON Copy

```
1 {
2   "cars": [
3     {
4       "VIN": "09HNSJG68736HNNJL",
5       "type": "Suzuki",
6       "model": "Cougar S",
7       "noDoors": "3",
8       "noSeats": "4",
9       "price": "80",
10      "isBooked": true
11    },
12    {
13      "VIN": "09HNSJG68736HNNJL",
14      "type": "Suzuki",
15      "model": "Cougar S",
16      "noDoors": "3",
17      "noSeats": "4",
18      "price": "80",
19      "isBooked": true
20    },
21    {
22      "VIN": "09HNSJG68736HNNJL",
23      "type": "Suzuki",
24      "model": "Cougar S",
25      "noDoors": "3",
26      "noSeats": "4",
27      "price": "80",
28      "isBooked": false
29    },
30    {
31      "VIN": "09HNSJG68736HNNJL",
32      "type": "Suzuki",
33      "model": "Cougar S",
34      "noDoors": "3",
35      "noSeats": "4",
36      "price": "80",
37      "isBooked": false
38    }
39  ]
40 }
```

10 Bootstrap Build Run

9. GET list of cars based on the dates ID

```
// @route get api/dates/:date_id/carlist
// @desc used to return the list of cars for a given date
routerDate.get('/carlist/:date_id', (req, res) => {
```

GET http://localhost:5000/api/dates/carlist/20082019 Send

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 33ms Size: 1.56 KB Save

Pretty Raw Preview JSON

```
1 {
2   "cars": [
3     {
4       "VIN": "GHGNS3G68736HNBHG",
5       "type": "Mini",
6       "model": "Cooper S",
7       "noDoors": "3",
8       "noSeats": "4",
9       "price": "90",
10      "isBooked": true
11    },
12    {
13      "VIN": "GHGNS3G68736HNB999",
14      "type": "BMW",
15      "model": "530i",
16      "noDoors": "4",
17      "noSeats": "5",
18      "price": "115",
19      "isBooked": true
20    },
21    {
22      "VIN": "ZYHNS3G68736HNBHG",
23      "type": "Audi",
24      "model": "A4",
25      "noDoors": "4",
26      "noSeats": "5",
27      "price": "180",
28      "isBooked": false,
29      "success": false
30    },
31  ]
}
```

Car Routes:

1. POST a car to the overall fleet list

```
// @route POST api/cars/addcar
// @desc Add Car
routerCar.post("/addcar", (req, res) => {
```

POST http://localhost:5000/api/cars/addcar Send

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL BETA

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> VIN	BNHJGK879JK876908	
<input checked="" type="checkbox"/> type	BMW	
<input checked="" type="checkbox"/> model	116i	
<input checked="" type="checkbox"/> noDoors	5	
<input checked="" type="checkbox"/> noSeats	5	
<input checked="" type="checkbox"/> price	90	
<input checked="" type="checkbox"/> isBooked	false	
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 38ms Size: 350 B Save

Pretty Raw Preview JSON

```
1 {
2   "_id": "5d581b5cb94ac1280cedf1a",
3   "VIN": "BNHJGK879JK876908",
4   "type": "BMW",
5   "model": "116i",
6   "noDoors": "5",
7   "noSeats": "5",
8   "price": "90",
9   "isBooked": false,
10  "success": true
}
```

2. GET carlist of fleet

```
// @route GET api/cars/carlist
// @description returns list of all cars in DB
// Access public - may need to change to emp only...
routerCar.get('/carlist', (req, res) => {
```

GET http://localhost:5000/api/cars/carlist

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 33ms Size: 1.58 KB Sav

Pretty Raw Preview JSON

```
1 {
2   {
3     "_id": "5d46ea735d342f44a82865aa",
4     "VIN": "GHEH5JG68736HVBHG",
5     "type": "Bent",
6     "model": "Cooper S",
7     "noDoors": "3",
8     "noSeats": "4",
9     "price": "98",
10    "__v": 0
11  },
12  {
13    "_id": "5d46ea9a5d342f44a82865ab",
14    "VIN": "GHEH5JG68736HVBHG",
15    "type": "Bent",
16    "model": "530i",
17    "noDoors": "4",
18    "noSeats": "5",
19    "price": "115",
20    "__v": 0
21  },
22  {
23    "_id": "5d532df234c2043d046523d0",
24    "VIN": "ZYHHSJG68736HVBHG",
25    "type": "Audi",
26    "model": "A4",
27    "noDoors": "4",
28    "noSeats": "5",
29    "price": "100",
30    "__v": 0
31  },
32  {
33    "_id": "5d5335da13371847887e97c4"
```

3. GET car based on its ID

```
// @route get api/cars/:carid
routerCar.get('/:car_id', (req, res) => {
  // Car find by id function
```

GET http://localhost:5000/api/cars/5d46ea9a5d342f44a82865ab

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 32ms Size: 351 B Sav

Pretty Raw Preview JSON

```
1 {
2   "_id": "5d46ea9a5d342f44a82865ab",
3   "VIN": "GHEH5JG68736HVBHG",
4   "type": "Bent",
5   "model": "530i",
6   "noDoors": "4",
7   "noSeats": "5",
8   "price": "115",
9   "__v": 0
10 }
```

Booking Routes:

1. GET list of all Bookings

```
// @route GET api/bookings/bookinglist
// @desc list of booking
// may want to change this to the list of bookings looking into the future
routerBooking.get('/bookinglist', (req, res) => {
```

GET http://localhost:5000/api/bookings/bookinglist Send

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

Query Params

KEY	VALUE	DESCRIPTION
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 31ms Size: 974 B Save

Pretty Raw Preview JSON

```
1 {
2   "id": "5d56fdae938448f48fa5a8a",
3   "start": "16082019",
4   "end": "20082019",
5   "car": {
6     "VIN": "GHEH5JG6873GHI8HG",
7     "type": "Jaguar",
8     "model": "F-Type",
9     "noDoors": "2",
10    "noSeats": "2",
11    "price": "150",
12    "isBooked": false
13  },
14  "totalPrice": 750,
15  "user": "jack.c.preston@btinternet.com",
16  "__v": 0
17 },
18 {
19   "id": "5d5703c5322bf24f5430b04b",
20   "start": "17082019",
21   "end": "20082019",
22   "car": {
23     "VIN": "GHEH5JG6873GHI899",
24     "type": "BMW",
25     "model": "530i",
26     "noDoors": "4",
27     "noSeats": "5",
28     "price": "115",
29     "isBooked": false
30   },
31   "totalPrice": 460,
32   "user": "jack.c.preston@btinternet.com"
33 }
```

2. POST a booking to booking list

```
// @route POST api/bookings/addbooking
// @desc add booking to DB
routerBooking.post('/addbooking', (req,res) => {
```

POST http://localhost:5000/api/bookings/addbooking Send

Params Authorization Headers (13) Body Pre-request Script Tests Cookies Code

none form-data x-www-form-urlencoded raw binary GraphQL BETA

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> start	17082019	
<input checked="" type="checkbox"/> end	20082019	
<input checked="" type="checkbox"/> car	{}	
<input checked="" type="checkbox"/> totalPrice	550	
<input checked="" type="checkbox"/> user	test@aol.com	
Key	Value	Description

Body Cookies Headers (6) Test Results Status: 200 OK Time: 33ms Size: 341 B Save

Pretty Raw Preview JSON

```
1 {
2   "id": "5d581dc9b94ac1280cedfeb",
3   "start": "17082019",
4   "end": "20082019",
5   "car": {},
6   "totalPrice": 550,
7   "user": "test@aol.com",
8   "__v": 0
9 }
```


3. DELETE booking based on start/end/username

```
routerBooking.delete('/deletebooking/:start/:end/:user', (req, res) => {
```

The screenshot shows a REST client interface with the following details:

- Method:** DELETE
- URL:** http://localhost:5000/api/bookings/deletebooking/17082019/20082019/test@aol.com
- Body Type:** x-www-form-urlencoded
- Body Data:**

KEY	VALUE	DESCRIPTION
start	17082019	
end	20082019	
car	0	
totalPrice	550	
user	test@aol.com	
Key	Value	Description
- Status:** 200 OK
- Time:** 28ms
- Size:** 214 B
- Response Body:** 1 null