Development of Cross-platform Booking System for a Hairdressers

by

Hugo Smith

In partial fulfilment of the
requirements for the degree of
MSc
in
Software Development

University of
Strathclyde
Glasgow

Department of
Computer and Information Sciences

August, 2021

Abstract

This report details the development of a cross-platform booking system for a hairdresser's business with the aim of developing a product to fill a gap in the market for a low-cost, customisable application. The project required that the full lifecycle of development take place. First, an evaluation of similar products on the market was conducted to assess the viability of the application and to identify what features would provide the most value. This was followed by requirements gathering and evaluation. Where insights gained from the market evaluation were used as discussion points in interviews and a survey to refine the project's requirements. A review of available technologies was then completed to determine which technologies would be most suitable for the requirements of the project. It was decided that a combination of React, Flask and PostgreSQL were best suited for the project thanks to the reliability and flexibility they offered.

In the next phase of the project development periods were structured into sprints aligned with the Agile methodology. The resulting application fulfilled many of the original requirements but failed to achieve some due to lack of time.

Declaration

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc in Software Development 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 [X] 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 8747

I confirm that I wish this to be assessed as a Type 3 Dissertation.

Signature: [illegible]

Date: 11/08/2024

Acknowledgements

I would like to firstly give thanks to my supervisor Konstantinos Liaskos for his continued support and supervision throughout the duration of the project. I would also like to give special thanks to my mother, Dr Helen Robertshaw who continually supported me and motivated me.

Finally, I would like to give thanks to all my friends and family who supported me throughout the project.

Contents

1 Introduction

1.1 Background

Booking systems have become a staple in the modern digital age as they allow individuals and companies to create, track and manage future events and interactions. They are employed in many industry sectors, ranging from hospitality to healthcare and can provide a streamlined, efficient and convenient process for all users. For customers, they enhance convenience by enabling the creation of reservations without needing to directly contact the service provider and even allow for appointments to be created automatically. For the business, booking systems can boost operational efficiency and ultimately increase revenue. This is accomplished by streamlining resource management, staff scheduling, appointment filling and data analysis of the business.

While booking systems can be an invaluable tool for businesses, there are potentially many challenges to ensuring they are both efficient and reliable. Technical issues can lead to disrupted operations and can heavily impact customer satisfaction. While more complex systems can require extensive staff training to ensure that they are used effectively preventing human error. The ability to dictate the booking times in advance can be a double-edged sword for businesses as it can become complex to manage customer cancellations, without impacting customer satisfaction or business performance.

While a booking system can operate effectively on a single system, it severely limits the ways in which the customers can interact with the system. Expanding the system to be accessible on multiple platforms can greatly enhance the potential customer base, bringing increased revenue to a business. This report will explore how a cross-platform development can enhance a booking system's effectiveness, along with the goals, challenges and considerations when developing one.

1.2 Objectives

The project aims to develop a full-stack application that is designed for compatibility with multiple different platforms. This includes mobile phones that use different types of operating systems, or even across different types of devices such as phones and PCs. An evaluation of the industry and its needs will be conducted to determine the requirements for the minimum viable product. After this, an assessment of available technologies will be carried out to make an informed decision on which options would be best suited to the scope of the project. The application will be designed with scalability in mind, ensuring that it would be suited for real-world use by incorporating features that allow it to handle a growing number of users. The application will also be organised and well-documented to facilitate any future development or maintenance.

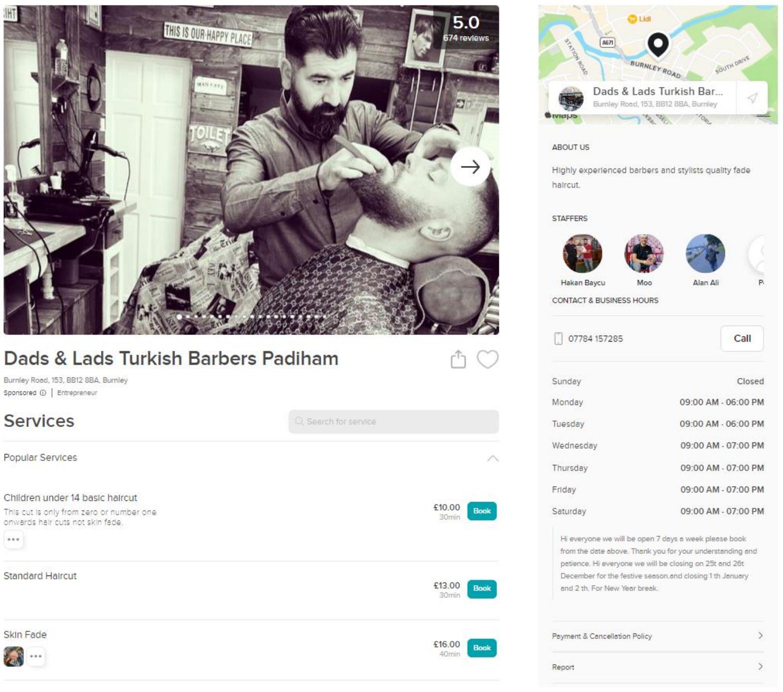
2 Project Requirements

2.1 Initial Requirements

To ensure that the application provides value to its users, the user's requirements must be gathered and refined to set the basis for what must be achieved. The final application will have two types of users, firstly are the customers who will use the system to book appointments and view information about the shop and its services. The second type of user will be the admins, the shop staff, who will manage the system and its customers.

Due to the project's tight timeline, a full requirements evaluation cannot be conducted. For future development, there would be multiple rounds of requirements refining making use of user participation in evaluating low, medium and high-fidelity prototypes. Instead, the process will be divided into two stages; a product evaluation to create initial requirements followed by a user evaluation of these requirements. Initial requirements will be drafted by conducting evaluations of similar available products and services. After which, a survey will be employed to gather general information from a group representing customers, while interviews will be conducted to gain a more in-depth analysis from industry workers, which will compensate for the smaller number of interviewees. This will gather feedback on the set of pre-brainstormed requirements which are to be used as discussion points. This will enable the suggested requirements to be confirmed, built upon or removed and could even stimulate users to suggest new related requirements that had been missed.

One cross-platform service that is already commonly used by many hair businesses is Booksy. It provides a ready-made service where small businesses can pay a monthly fee of £40/per month to gain access to a system that handles, bookings, payments and for additional fees marketing boosts. All businesses are given an identical layout, pictured in Figure 1, which provides information about the shop, its staff and its treatments. While the layout of the page offered to businesses is visually modest, it displays everything in a tidy, easy-to-digest manner.



Dads & Lads Turkish Barbers Padiham
Burnley Road, 153, BB12 8BA, Burnley
Sponsored | Entrepreneur

Services

Search for service

Popular Services

Children under 14 basic haircut This cut is only from zero or number one onwards hair cuts not skin fade.	£10.00	30min
Standard Haircut	£13.00	30min
Skin Fade	£16.00	40min

Dads & Lads Turkish Bar...
Burnley Road, 153, BB12 8BA, Burnley

ABOUT US

Highly experienced barbers and stylists quality fade haircut.

STAFFERS

Hakan Baycu, Moo, Alan Ali

CONTACT & BUSINESS HOURS

07784 157285

Sunday	Closed
Monday	09:00 AM - 06:00 PM
Tuesday	09:00 AM - 06:00 PM
Wednesday	09:00 AM - 07:00 PM
Thursday	09:00 AM - 07:00 PM
Friday	09:00 AM - 07:00 PM
Saturday	09:00 AM - 07:00 PM

Hi everyone we will be open 7 days a week please book from the date above. Thank you for your understanding and patience. Hi everyone we will be closing on 25t and 26t December for the festive season.and closing 1 th January and 2 th. For New Year break.

Payment & Cancellation Policy

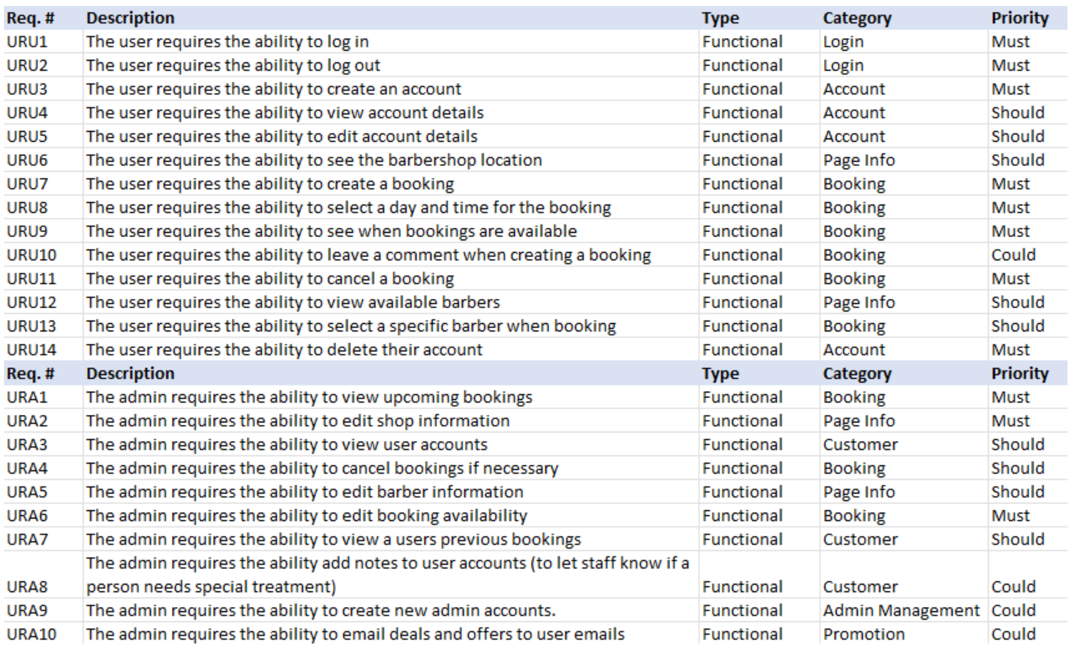
Report

Figure 1 Typical Layout of a Booksy Page

While Booksy does well to cover many of the requirements that could be set, it could be argued that it doesn't provide the bespoke experience that some businesses desire. In a heavily competitive industry, with some providing services over £80, some businesses may feel that the limited nature of Booksy is unable to portray their premium brand to their customers. A custom-built application would provide more customisability and allow the user to experiment with different styles and business tactics to bring in new customers.

One service that provides a more unique experience is BookingKit. While they don't operate in the same industry, instead they mostly focus on events sales, it can still be a helpful exercise to analyse their product. Like Booksy they offer a platform for businesses to manage bookings, transactions and advertise themselves. The key difference being that BookingKit offers a more customisable product. Users have more control over their page and systems aesthetics and can even opt into additional features such as resource management. This comes at a substantially higher cost however as prices to use the service range from £43 - £85+ /monthly with the higher tier payments required to unlock more customisability and admin accounts. Analysis of similar products on the market follows this trend of substantially higher costs for the trade-off of more freedom to customise the system.

Pictured in **Figure 2** are the initial requirements that were drafted prior to the first round of user feedback from analysing existing products on the market. They have been separated into two lists depending on the user they relate to; this was to make management of the requirements easier to process. Requirements were identified as either functional or nonfunctional, where functional requirements typically describe what the system must achieve and its functions. While non-functional requirements describe the system's properties and constraints [1]. At this stage, there is a lack of non-functional requirements as these will be discussed during the interviews. The requirements have been categorised to better organise them for when they are adapted into user stories and improve their traceability. Additionally, the requirements have been ranked using the MOSCOW technique, where requirements are ranked dependant on the amount of value they will provide in the final solution [2].



Req. #	Description	Type	Category	Priority
URU1	The user requires the ability to log in	Functional	Login	Must
URU2	The user requires the ability to log out	Functional	Login	Must
URU3	The user requires the ability to create an account	Functional	Account	Must
URU4	The user requires the ability to view account details	Functional	Account	Should
URU5	The user requires the ability to edit account details	Functional	Account	Should
URU6	The user requires the ability to see the barbershop location	Functional	Page Info	Should
URU7	The user requires the ability to create a booking	Functional	Booking	Must
URU8	The user requires the ability to select a day and time for the booking	Functional	Booking	Must
URU9	The user requires the ability to see when bookings are available	Functional	Booking	Must
URU10	The user requires the ability to leave a comment when creating a booking	Functional	Booking	Could
URU11	The user requires the ability to cancel a booking	Functional	Booking	Must
URU12	The user requires the ability to view available barbers	Functional	Page Info	Should
URU13	The user requires the ability to select a specific barber when booking	Functional	Booking	Should
URU14	The user requires the ability to delete their account	Functional	Account	Must
Req. #	**Description**	**Type**	**Category**	**Priority**
URA1	The admin requires the ability to view upcoming bookings	Functional	Booking	Must
URA2	The admin requires the ability to edit shop information	Functional	Page Info	Must
URA3	The admin requires the ability to view user accounts	Functional	Customer	Should
URA4	The admin requires the ability to cancel bookings if necessary	Functional	Booking	Should
URA5	The admin requires the ability to edit barber information	Functional	Page Info	Should
URA6	The admin requires the ability to edit booking availability	Functional	Booking	Must
URA7	The admin requires the ability to view a users previous bookings	Functional	Customer	Should
URA8	The admin requires the ability add notes to user accounts (to let staff know if a person needs special treatment)	Functional	Customer	Could
URA9	The admin requires the ability to create new admin accounts.	Functional	Admin Management	Could
URA10	The admin requires the ability to email deals and offers to user emails	Functional	Promotion	Could

Figure 2 (Above) Customer Requirements (Below) Admin Requirements

2.2 Requirements Development

Feedback from the survey confirmed that most of the drafted requirements would be either beneficial or necessary for customers when using a booking system. In addition, there were a few points raised that would improve the value provided by the app.

- Customers should be able to re-schedule appointments, not just create and delete them.
- Customers like to be able to view images of the shop and its staff before they make a booking.
- Customers like to be able to see social media links when viewing a business.

The interviews with members from the industry also confirmed many of the suggested requirements but importantly suggested a rethink in how the admin system would work. When asked about having one or multiple staff accounts: It was generally agreed that multiple staff accounts make it easier to track what staff members have done and restrict access to certain features. This also limits the staff to using the app on certain devices, such as the salon computer. When asked about the importance of emails: Most agreed that emails are important or essential for providing password security and sending offers of deals and treatments. Other points raised were that admins should have the ability to add and edit treatments, ‘Edit shop information’ was too vague and could be expanded to provide specifics, and that overall, a web application is preferable over a mobile application, as people generally dislike having to install new applications for every service. When asked about staff numbers and customers, workers said they average from 4-10 staff generally with 3-4 active members and generally see around 20 – 30 customers daily.

2.2.1 Functional Requirements

The following changes were made to the functional user requirements with the updated user requirements being shown below in Figure 3:

URA2 (Admin requires the ability to edit shop information) was deemed as not specific enough. It was changed to describe more specific information (location, deals, treatments, staff, general notices)

Created URA11 -URA22, removed URA2.

It was brought up by both the interviews and surveys that staff and customers should have the ability to reschedule bookings. Currently, only the creation and deletion bookings exist for either.

Created URU15 and URA23

When discussing the functionality of admin accounts, it was brought up that having individual accounts for members of staff is crucial. It allows the system to more easily track which admins have made edits to the system, made financial transactions, and restrict access to certain features.

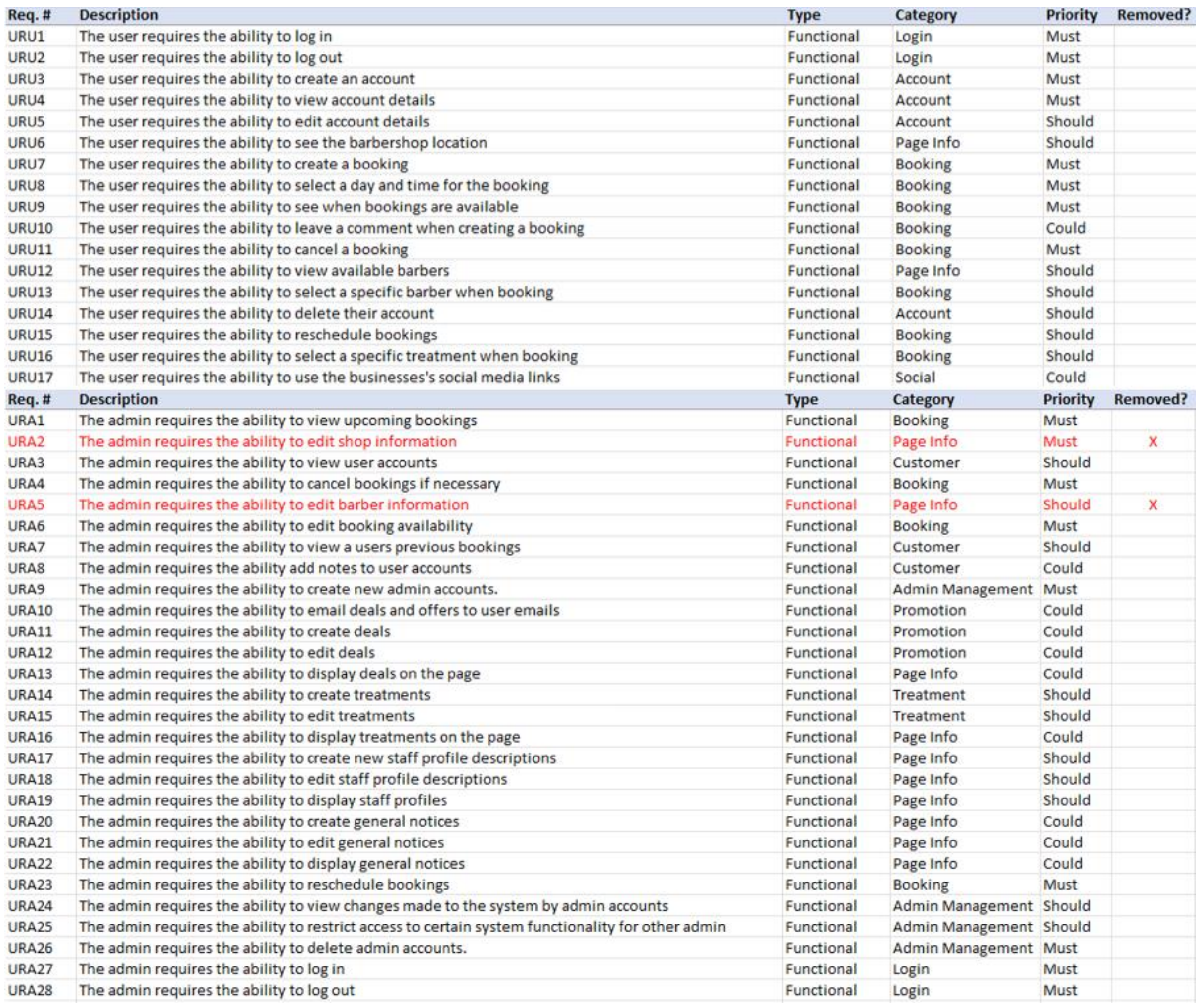
Created URA24, URA25, URA26

The ability to specify a treatment when booking was suggested in the interviews and reinforced by the survey.

Created URU16.

Results from the survey suggested that some users value social media links on a booking page to see further shop images.

Created URU17.



Req. #	Description	Type	Category	Priority	Removed?
URU1	The user requires the ability to log in	Functional	Login	Must	
URU2	The user requires the ability to log out	Functional	Login	Must	
URU3	The user requires the ability to create an account	Functional	Account	Must	
URU4	The user requires the ability to view account details	Functional	Account	Must	
URU5	The user requires the ability to edit account details	Functional	Account	Should	
URU6	The user requires the ability to see the barbershop location	Functional	Page Info	Should	
URU7	The user requires the ability to create a booking	Functional	Booking	Must	
URU8	The user requires the ability to select a day and time for the booking	Functional	Booking	Must	
URU9	The user requires the ability to see when bookings are available	Functional	Booking	Must	
URU10	The user requires the ability to leave a comment when creating a booking	Functional	Booking	Could	
URU11	The user requires the ability to cancel a booking	Functional	Booking	Must	
URU12	The user requires the ability to view available barbers	Functional	Page Info	Should	
URU13	The user requires the ability to select a specific barber when booking	Functional	Booking	Should	
URU14	The user requires the ability to delete their account	Functional	Account	Should	
URU15	The user requires the ability to reschedule bookings	Functional	Booking	Should	
URU16	The user requires the ability to select a specific treatment when booking	Functional	Booking	Should	
URU17	The user requires the ability to use the businesses's social media links	Functional	Social	Could	
Req. #	Description	Type	Category	Priority	Removed?
URA1	The admin requires the ability to view upcoming bookings	Functional	Booking	Must	
URA2	The admin requires the ability to edit shop information	Functional	Page Info	Must	X
URA3	The admin requires the ability to view user accounts	Functional	Customer	Should	
URA4	The admin requires the ability to cancel bookings if necessary	Functional	Booking	Must	
URA5	The admin requires the ability to edit barber information	Functional	Page Info	Should	X
URA6	The admin requires the ability to edit booking availability	Functional	Booking	Must	
URA7	The admin requires the ability to view a users previous bookings	Functional	Customer	Should	
URA8	The admin requires the ability add notes to user accounts	Functional	Customer	Could	
URA9	The admin requires the ability to create new admin accounts.	Functional	Admin Management	Must	
URA10	The admin requires the ability to email deals and offers to user emails	Functional	Promotion	Could	
URA11	The admin requires the ability to create deals	Functional	Promotion	Could	
URA12	The admin requires the ability to edit deals	Functional	Promotion	Could	
URA13	The admin requires the ability to display deals on the page	Functional	Page Info	Could	
URA14	The admin requires the ability to create treatments	Functional	Treatment	Should	
URA15	The admin requires the ability to edit treatments	Functional	Treatment	Should	
URA16	The admin requires the ability to display treatments on the page	Functional	Page Info	Could	
URA17	The admin requires the ability to create new staff profile descriptions	Functional	Page Info	Should	
URA18	The admin requires the ability to edit staff profile descriptions	Functional	Page Info	Should	
URA19	The admin requires the ability to display staff profiles	Functional	Page Info	Should	
URA20	The admin requires the ability to create general notices	Functional	Page Info	Could	
URA21	The admin requires the ability to edit general notices	Functional	Page Info	Could	
URA22	The admin requires the ability to display general notices	Functional	Page Info	Could	
URA23	The admin requires the ability to reschedule bookings	Functional	Booking	Must	
URA24	The admin requires the ability to view changes made to the system by admin accounts	Functional	Admin Management	Should	
URA25	The admin requires the ability to restrict access to certain system functionality for other admin	Functional	Admin Management	Should	
URA26	The admin requires the ability to delete admin accounts.	Functional	Admin Management	Must	
URA27	The admin requires the ability to log in	Functional	Login	Must	
URA28	The admin requires the ability to log out	Functional	Login	Must	

Figure 3 Refined Functional User Requirements

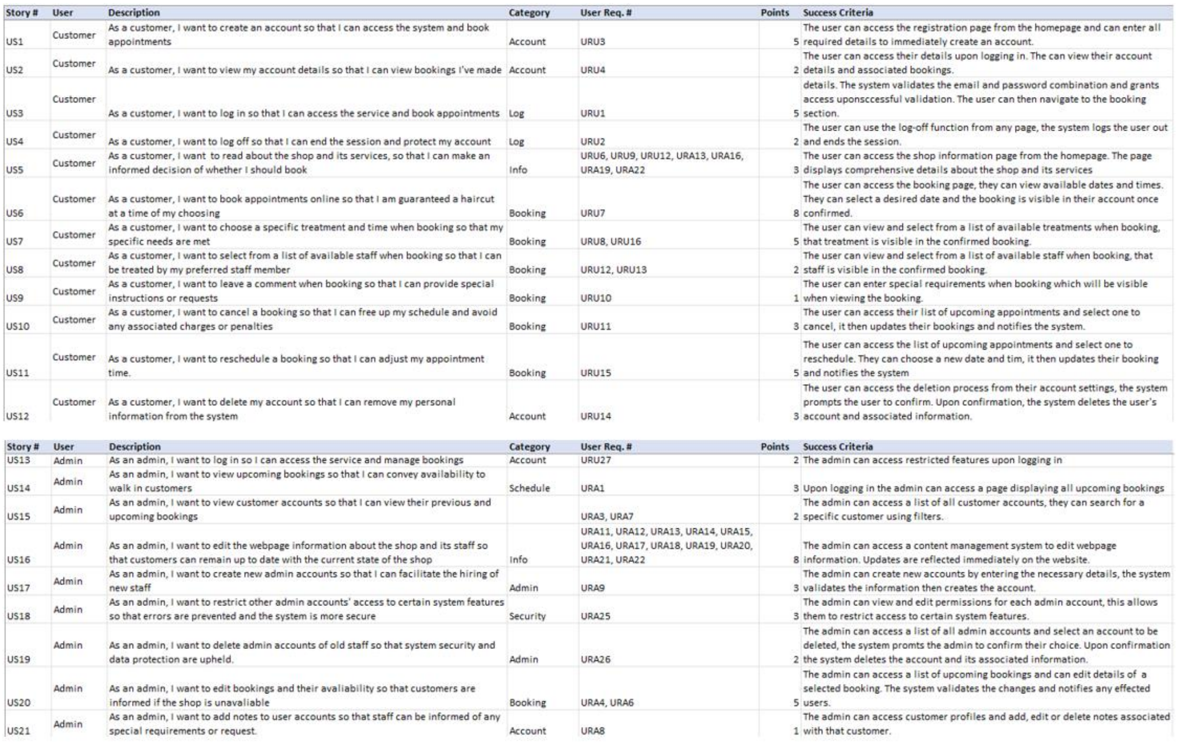
2.2.2 Non-functional Requirements

The following non-functional requirements were established from the feedback gathered in the surveys and interviews.

- The user requires the software handles 10 admin accounts without performance losses.
- The user requires the software handles 30 bookings a day without performance losses.
- The user requires that passwords be encrypted.
- The user requires that the software is a cross-platform web application

2.3 User Stories

User stories provide a more user-centric format that is essential for further development. They also allow for the developer to more accurately estimate the time cost to complete different tasks of the project and so are beneficial to the agile methodology. Later in the project, they will be used to construct user acceptance tests, ensuring that the requirements are correctly implemented. Below, pictured in Figure 4, are the user stories that were drafted from the initial user requirements.



Story #	User	Description	Category	User Req. #	Points	Success Criteria
US1	Customer	As a customer, I want to create an account so that I can access the system and book appointments	Account	URU3	5	The user can access the registration page from the homepage and can enter all required details to immediately create an account.
US2	Customer	As a customer, I want to view my account details so that I can view bookings I've made	Account	URU4	2	The user can access their details upon logging in. The can view their account details and associated bookings.
US3	Customer	As a customer, I want to log in so that I can access the service and book appointments	Log	URU1	5	details. The system validates the email and password combination and grants access uponsccessful validation. The user can then navigate to the booking section.
US4	Customer	As a customer, I want to log off so that I can end the session and protect my account	Log	URU2	2	The user can use the log-off function from any page, the system logs the user out and ends the session.
US5	Customer	As a customer, I want to read about the shop and its services, so that I can make an informed decision of whether I should book	Info	URU6, URU9, URU12, URA13, URA16, URA19, URA22	3	The user can access the shop information page from the homepage. The page displays comprehensive details about the shop and its services
US6	Customer	As a customer, I want to book appointments online so that I am guaranteed a haircut at a time of my choosing	Booking	URU7	8	The user can access the booking page, they can view available dates and times. They can select a desired date and the booking is visible in their account once confirmed.
US7	Customer	As a customer, I want to choose a specific treatment and time when booking so that my specific needs are met	Booking	URU8, URU16	5	The user can view and select from a list of available treatments when booking, that treatment is visible in the confirmed booking.
US8	Customer	As a customer, I want to select from a list of available staff when booking so that I can be treated by my preferred staff member	Booking	URU12, URU13	2	The user can view and select from a list of available staff when booking, that staff is visible in the confirmed booking.
US9	Customer	As a customer, I want to leave a comment when booking so that I can provide special instructions or requests	Booking	URU10	1	The user can enter special requirements when booking which will be visible when viewing the booking.
US10	Customer	As a customer, I want to cancel a booking so that I can free up my schedule and avoid any associated charges or penalties	Booking	URU11	3	The user can access their list of upcoming appointments and select one to cancel, it then updates their bookings and notifies the system.
US11	Customer	As a customer, I want to reschedule a booking so that I can adjust my appointment time.	Booking	URU15	5	The user can access the list of upcoming appointments and select one to reschedule. They can choose a new date and tim, it then updates their booking and notifies the system
US12	Customer	As a customer, I want to delete my account so that I can remove my personal information from the system	Account	URU14	3	The user can access the deletion process from their account settings, the system prompts the user to confirm. Upon confirmation, the system deletes the user's account and associated information.

Story #	User	Description	Category	User Req. #	Points	Success Criteria
US13	Admin	As an admin, I want to log in so I can access the service and manage bookings	Account	URU27	2	The admin can access restricted features upon logging in
US14	Admin	As an admin, I want to view upcoming bookings so that I can convey availability to walk in customers	Schedule	URA1	3	Upon logging in the admin can access a page displaying all upcoming bookings
US15	Admin	As an admin, I want to view customer accounts so that I can view their previous and upcoming bookings		URA3, URA7	2	The admin can access a list of all customer accounts, they can search for a specific customer using filters.
US16	Admin	As an admin, I want to edit the webpage information about the shop and its staff so that customers can remain up to date with the current state of the shop	Info	URA11, URA12, URA13, URA14, URA15, URA16, URA17, URA18, URA19, URA20, URA21, URA22	8	The admin can access a content management system to edit webpage information. Updates are reflected immediately on the website.
US17	Admin	As an admin, I want to create new admin accounts so that I can facilitate the hiring of new staff	Admin	URA9	3	The admin can create new accounts by entering the necessary details, the system validates the information then creates the account.
US18	Admin	As an admin, I want to restrict other admin accounts' access to certain system features so that errors are prevented and the system is more secure	Security	URA25	3	The admin can view and edit permissions for each admin account, this allows them to restrict access to certain system features.
US19	Admin	As an admin, I want to delete admin accounts of old staff so that system security and data protection are upheld.	Admin	URA26	2	The admin can access a list of all admin accounts and select an account to be deleted, the system promts the admin to confirm their choice. Upon confirmation the system deletes the account and its associated information.
US20	Admin	As an admin, I want to edit bookings and their avaliability so that customers are informed if the shop is unavaliable	Booking	URA4, URA6	5	The admin can access a list of upcoming bookings and can edit details of a selected booking. The system validates the changes and notifies any effected users.
US21	Admin	As an admin, I want to add notes to user accounts so that staff can be informed of any special requirements or request.	Account	URA8	1	The admin can access customer profiles and add, edit or delete notes associated with that customer.

Figure 4 Developed User Stories

3 Review of Technologies

This section will provide a discussion of the available technologies that could facilitate the creation of a cross-platform application. It will highlight the benefits to each as well as the potential challenges faced when working with them. The surveys and interviews displayed a preference for a web application over mobile and established this as a non-functional requirement. To accommodate this the technologies reviewed will be focussed on web-based frameworks.

3.1 Front-end Technologies

Front-end development involves writing code that displays information to the client; in the case of a web app, this would be the user's browser.

3.1.1 React

React is a JavaScript library, originally developed by engineers at Facebook to solve challenges involving complex user interfaces that rely on continually changing datasets [3]. React has been designed to achieve high performance and efficiently manage page updates, thanks to its use of the virtual DOM (Document Object Model). The virtual DOM mirrors the structure of the real DOM and allows React to identify differences between the two when components update. This allows it to selectively update the differing components rather than the whole page, which is slower. React is a lightweight framework and requires additional libraries to be integrated to achieve more functionality. While this allows for greater customisation, extra precautions must be taken when implementing new functionality to prevent potential issues. This results in React having a moderate learning curve, some concepts can be harder to get to grips with and implement such as state management and routing.

3.1.2 Angular

Angular is another powerful and popular framework that was developed and is still maintained by Google. It's commonly used by developers when creating web applications and as a testament to its worth it is the backbone of Google’s Cloud Platform services and ad management platform. Like React, Angular applications are also single page applications (SPA), allowing developers to provide a more dynamic and interactive experience to users. Having been released in 2010 Angular has had plenty of time to develop a rich community and ecosystem, resulting in extensive support and documentation along with many tools and libraries. However, as Angular is a full-fledged framework it is considered to have a steeper learning curve. This could impact development time and result in incomplete requirements. Lack of proper knowledge can also lead to performance issues if the code is not correctly optimised.

3.1.3 Vue.js

Vue.js is open-source JavaScript framework, like React it also a virtual DOM to efficiently update components on the page. Vues learning curve is generally considered to be very gentle, especially if the developer is familiar with HTML, CSS and JavaScript. Like React Vue is extremely flexible, allowing it to be used for a wide range of applications. Although Vue is well established it still is less widely adopted than the other technologies mentioned. This results in Vue having a smaller community and so there are less available plugins, libraries and online documentation.

3.1.4 Technology Comparison

Each technology provides unique advantages and challenges to be considered when designing an application. React offers great flexibility thanks to its large selection of potential libraries that can be imported, but this same reason can result in inexperienced developers writing code that causes performance issues. Angular is a powerful framework that offers many built-in features, allowing developers to build software that maintains performance at large scales. However, like react the steeper initial learning curve can result in un-optimised applications and issues further down the line. Vue.js offers a middle ground between the two, combining some of the best aspects of the two previous technologies with a gentler learning curve.

As established earlier in the project, a driving requirement is that the application must be able to accommodate a bespoke experience to users to provide value over similar products on the market. The scale of the application is also predicted to be small as shops only see on average 20 -30 customers a day and generally have less than 10 staff. Given these requirements, React's potential for providing a truly bespoke application and more extensive documentation and community make it a strong choice for the front-end technology.

3.2 Back-end Technologies

Incorporating backend technologies into an application can enable server-side processing and seamless interaction with a database. This can greatly enhance the user experience by updating view layer components reactively to information provided by the user providing a more unique experience.

3.2.1 Django

Django is a free open-source Python framework, released in 2005, it has since gained popularity and is the backbone of many popular services such as Instagram, Pinterest and Dropbox. It boasts faster development times thanks to incorporating many commonly used features, reducing the need for third-party integrations. This includes built-in security measures to protect against many common web vulnerabilities. Its monolithic architecture means that all parts of the application are tightly bound, making it less suitable for the microservices architecture. While the abundance of built-in features is beneficial for experienced developers, it is generally considered to make for a steeper learning experience for new developers. While the framework is scalable and has good high-traffic performance, it is less suitable for smaller lightweight applications.

3.2.2 Flask

Released in 2010, Flask is a micro web framework that is also written in the Python language. Unlike Django Flask comes with no built-in features instead relying on third-party libraries to provide common functions. This allows developers to cherry-pick certain libraries to suit the specific needs of the application and can allow for the creation of lightweight specialised applications. Like the other technologies, it boasts a developed community and extensive documentation and is generally considered to have a gentle learning curve making it ideal for new developers. It is less suitable for developing larger applications due to the lack of essential built-in tools. Applications can also experience performance problems if components are structured incorrectly. The lack of inbuilt security features can lead to security issues if developers aren’t rigorous with their security measures.

3.2.3 Node.js with Express.js

Node.js is a runtime that allows you to run JavaScript code on the server side. When building web applications, it is commonly paired with express.js, a web application framework. It uses an event-driven non-blocking input/output (I/O) model, this essentially means that it can have multiple operations running in parallel without having to wait for other operations to complete. This makes it suitable for developing applications that require real-time data processing, such as analytics visualisers. It supports the microservices architecture pattern, making it scalable and maintainable. Developed documentation and a large active community, facilitate easier issue resolution and availability of learning sources for new technologies

3.2.4 Technology Comparison

As mentioned in the previous comparison the driving requirement of the application is that it is flexible and customisable to the business's needs to provide maximum value. Django’s structured format and built-in libraries make it a strong contender as these allow for rapid development, security and the ability to handle high-traffic loads. While these are all desirable traits, they result in Django applications being limited in their flexibility and are potentially unnecessary for what is predicted to be a lightweight application. Both Node and Flask are suitable back-end frameworks with both having the potential to create flexible, scalable applications. By using Node JavaScript would be a unified language across the application, and while this is beneficial for an experienced JS developer it may hinder development when compared to using Python, a familiar language. Flask's effectiveness, simplicity, and the fact that it uses a familiar language make it a preferable choice considering the time pressure on the project.

3.3 Database Technologies

When it comes to choosing a database technology first a developer must decide between SQL (Structured Query Language) and NoSQL. SQL databases are relational databases that use a fixed schema and so work better with structured data. In contrast NoSQL / non-relational databases better suit unstructured data as they are not required to have a fixed schema. While the database design hasn’t been conducted yet, it’s likely that the data will be structured as customers, staff and bookings could all be represented by structured objects. Using an SQL technology would also allow for simpler querying, resulting in there being less chance of errors.

3.3.1 SQLite

SQLite is a serverless, self-contained database framework which doesn’t require advanced setup or configuration benefitting rapid development and testing. However, it's not designed for large-scale deployments resulting in poor scalability. SQLite also lacks some of the more advanced features that other SQL technologies offer, which can result in less efficient data validation and handling.

3.3.2 MySQL

MySQL is a popular, open-source relational database framework. It excels in read-heavy applications and supports both horizontal and vertical scaling. It also has reasonable Flask library support with libraries like Flask-MySQL. However, it lacks some the advanced features of PostgreSQL which can limit the potential of the application.

3.3.3 PostgreSQL

PostgreSQL has many advanced features to enable more complex queries. It supports both vertical and horizontal scaling and is highly compatible with Flask and SQLAlchemy. However, this results in a more complex setup and configuration when compared to SQLite and MySQL.

3.3.4 Technology comparison

While all three technologies can integrate with Flask well, SQLite was found not to be suitable due to its scalability limitations. Although the remaining two perform very similarly, PostgreSQL was chosen for its reliability and ability to integrate well with SQLAlchemy.

4 Design

4.1 System Architecture

With the chosen technologies established the architecture design can now be described. React-Flask applications follow a layered architecture, which is generally a Three-Tier Architecture pattern. The Three-Tier Architecture is a commonly used design that involves separating the application into three distinct computing tiers. Each tier has its own responsibilities, which enables the separation of logic and can allow for each tier to be scaled independently. The separation of the layers also allows for improved component classification, makes code easier to test, and easier to identify errors when debugging. These factors reduce the complexity and time of development which in turn lowers overhead costs. The architecture design for the app can be seen in Figure 5.

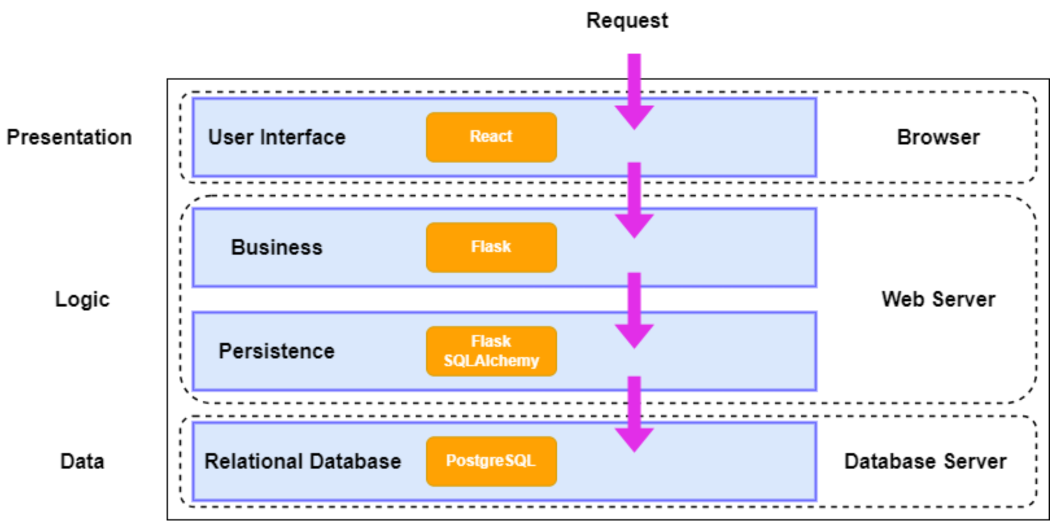


Figure 5 Diagram of the Three Tier Architecture of the Application

The presentation layer encompasses the front-end and user interface of the application. It defines the application's outward appearance and handles users' interactions with the software. It communicates with the back-end through HTTP requests, to fetch or manipulate data. These requests (GET, POST, PUT or DELETE) will be received using RESTful APIs.

The logic layer also called the application layer, encompasses the main body of the application and acts as the intermediary between the presentation and data layers. It takes inputted information from the presentation layer and processes it using business logic. The business logic of an application is a strict set of rules that data is processed by, that relate to the operations handled by the application's business. In this application this layer also includes the persistence layer, which acts a protective barrier containing all the necessary code to access the data layer.

Finally, is the data layer where the system stores its data in relations. This layer handles CRUD (Create, Read, Update, Delete) operations that are used to manipulate the data. By abstracting the data layer away from the rest of the application, it restricts interactions to an interface and so simplifies the process of accessing data and enables optimised operations.

It can be helpful to describe how the application will act when a user interacts with it. Visualised below in **Figure 6** is a UML sequence diagram displaying the processes involved in a successful user login.

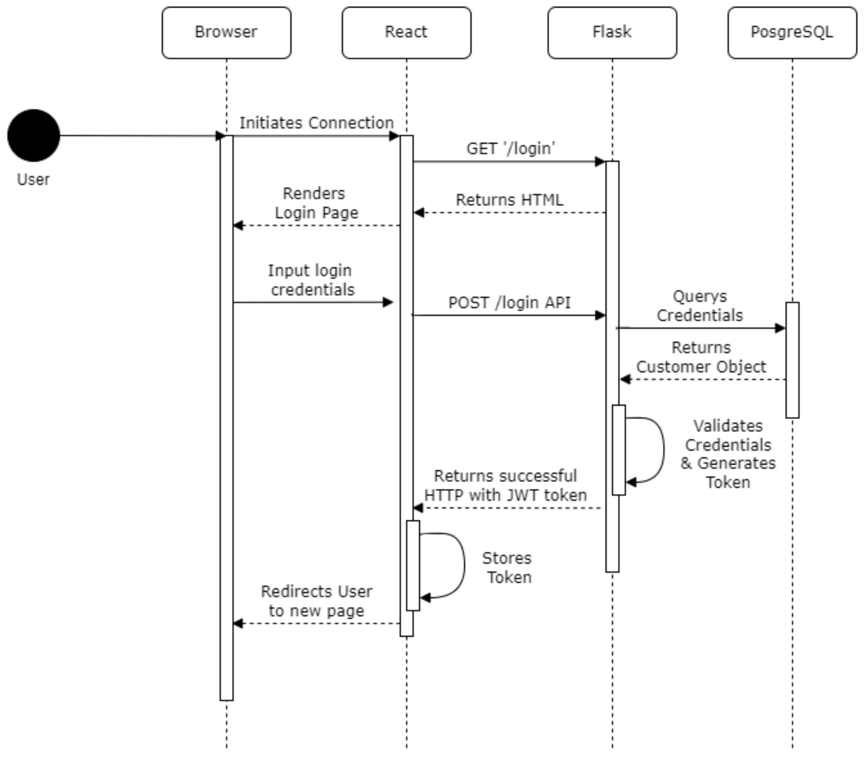


Figure 6 UML Sequence Diagram Displaying the Customer Login Process

4.2 Front-end Design

Creating a front-end/user interface design encourages a developer to consider the structure of the application and can promote development of more organised and cohesive code. It also ensures that the application is easy to navigate and visually appealing to the user. A low fidelity prototype of the UI can be found at Appendix A where wireframes were used to make a fast but effective model of the application.

4.3 Back-end Design

4.3.1 Route Design

Flask uses routes. These are functions that are called by an associated URL. These functions are where the majority of the application data is processed, they are responsible processing inputs from the front-end, making calls to the database and returning the processed data to the front-end. As such the class diagram will be grouped into related routes rather than classes.

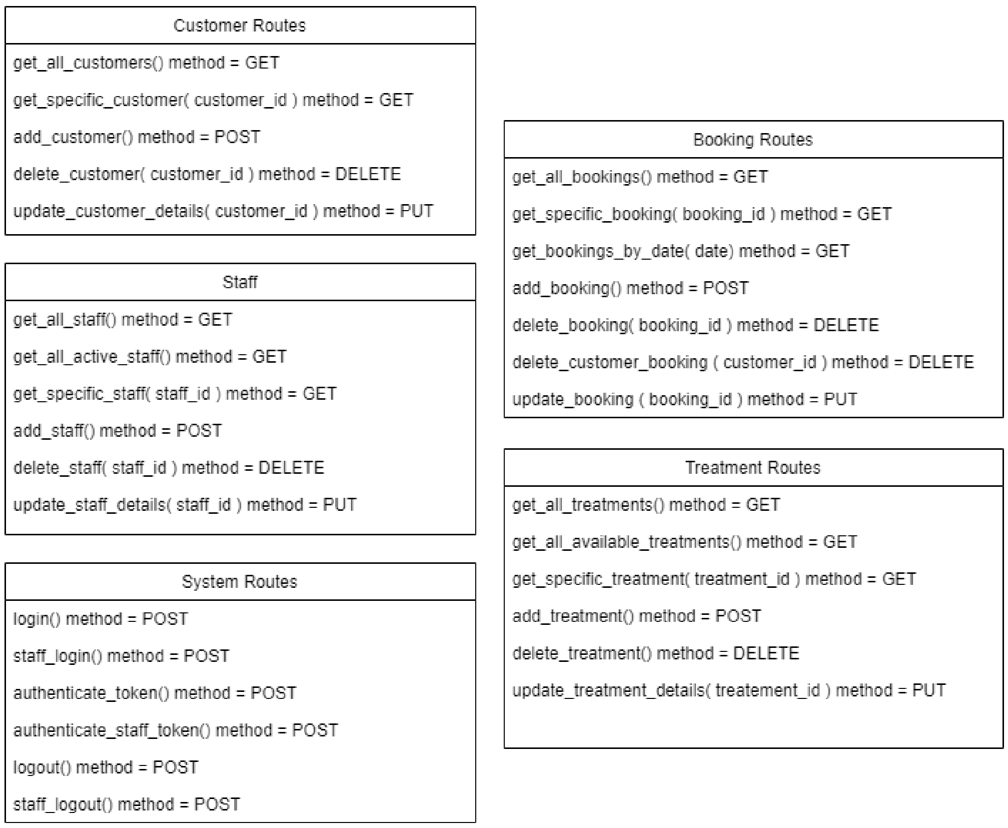

Customer Routes
get_all_customers() method = GET
get_specific_customer(customer_id) method = GET
add_customer() method = POST
delete_customer(customer_id) method = DELETE
update_customer_details(customer_id) method = PUT
Staff
get_all_staff() method = GET
get_all_active_staff() method = GET
get_specific_staff(staff_id) method = GET
add_staff() method = POST
delete_staff(staff_id) method = DELETE
update_staff_details(staff_id) method = PUT
System Routes
login() method = POST
staff_login() method = POST
authenticate_token() method = POST
authenticate_staff_token() method = POST
logout() method = POST
staff_logout() method = POST
Booking Routes
get_all_bookings() method = GET
get_specific_booking(booking_id) method = GET
get_bookings_by_date(date) method = GET
add_booking() method = POST
delete_booking(booking_id) method = DELETE
delete_customer_booking (customer_id) method = DELETE
update_booking (booking_id) method = PUT
Treatment Routes
get_all_treatments() method = GET
get_all_available_treatments() method = GET
get_specific_treatment(treatment_id) method = GET
add_treatment() method = POST
delete_treatment() method = DELETE
update_treatment_details(treatement_id) method = PUT

Figure 7 Route Design for the Logic Layer

4.3.2 Database Design

Constructing an entity-relationship (ER) diagram can be a useful exercise when designing a backend system. It encourages the designer to consider how the relations should be formatted as well as their various relationships with other relations. When developing databases, relation normalisation is an important process to improve the database's efficiency and reduce data redundancy. Due to the simplicity of the database, very little was necessary, but the model meets the requirements of 3rd normal form (NF) by satisfying both 1NF and 2NF while also having every non-key column be non-transitively fully dependent on the primary key. This model can be seen below in Figure 8.

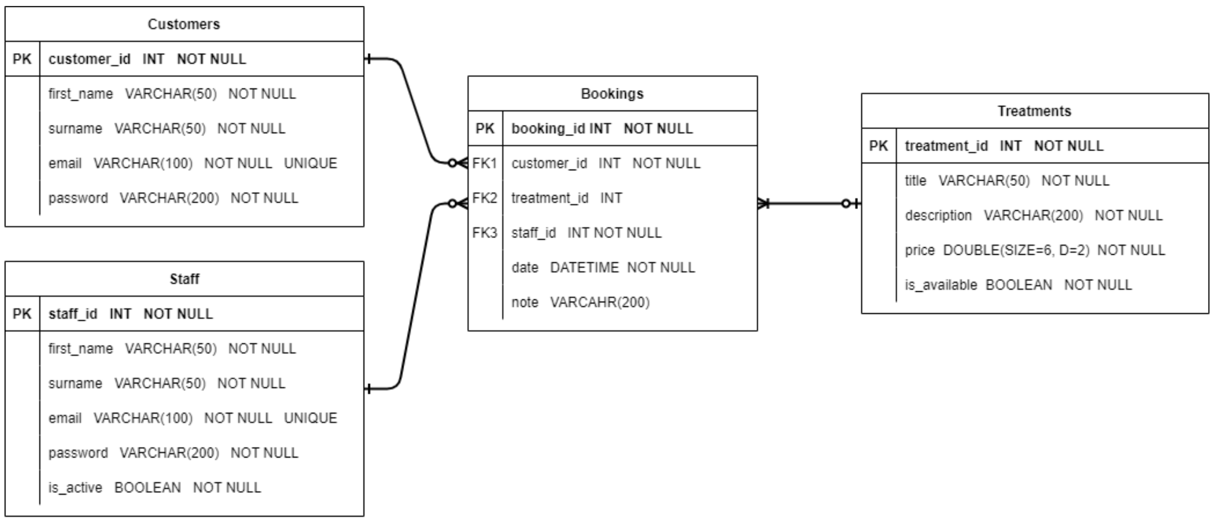

Customers
PK customer_id INT NOT NULL
first_name VARCHAR(50) NOT NULL
surname VARCHAR(50) NOT NULL
email VARCHAR(100) NOT NULL UNIQUE
password VARCHAR(200) NOT NULL
Staff
PK staff_id INT NOT NULL
first_name VARCHAR(50) NOT NULL
surname VARCHAR(50) NOT NULL
email VARCHAR(100) NOT NULL UNIQUE
password VARCHAR(200) NOT NULL
is_active BOOLEAN NOT NULL
Bookings
PK booking_id INT NOT NULL
FK1 customer_id INT NOT NULL
FK2 treatment_id INT
FK3 staff_id INT NOT NULL
date DATETIME NOT NULL
note VARCAHR(200)
Treatments
PK treatment_id INT NOT NULL
title VARCHAR(50) NOT NULL
description VARCHAR(200) NOT NULL
price DOUBLE(SIZE=6, D=2) NOT NULL
is_available BOOLEAN NOT NULL

Figure 8 Entity Relationship Diagram for the Application

5 Sprints

With a design for the application established the production of the code could begin. The following sprints describe how the software was developed and provide examples of code snippets and UI to help illustrate the process. There are sections of the development that repeat certain design patterns, and so descriptions and snippets will be left out if their general design has already been covered but will still be mentioned.

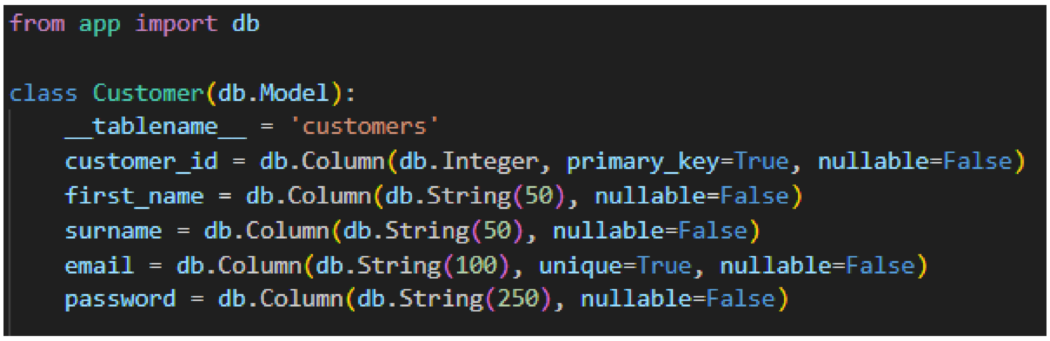
5.1 Sprint 1

The initial sprint mainly constituted becoming familiar with the new technologies required to build the application. As a first-time developer in both Flask and React a good understanding of both frameworks was essential to prevent bugs and performance issues further in the project's development. First, the technologies to be used were installed and set up, all coding was completed in Microsoft's Visual Studio and the Flask framework and React framework had to be imported using the pip and NPM installers. A virtual environment was configured to better manage dependencies and prevent conflicts. Due to the time taken to correctly implement this the amount of code written during this sprint, is significantly less than others.

The initial user story to be accomplished in this stage was:

- US1 – As a customer, I want to create an account so that I can access the system and book appointments

Implementing this user story would allow the testing of the full stack of the application as it required all three layers to accomplish. It would also make use of several key design methods such as state management, form validation, error handling and the RESTful API design. First, the model, pictured in **Figure 9**, for the customer was created and committed to the database, PGAdmin, a PostgreSQL GUI software was used to verify that the model had been added correctly. SQLAlchemy allows for the definition of the column's data types and values to better structure the model. This is shown by 'customer_id' being established as a primary key (PK), and several columns being prevented from containing null values. This ensures data consistency and prevents query errors.

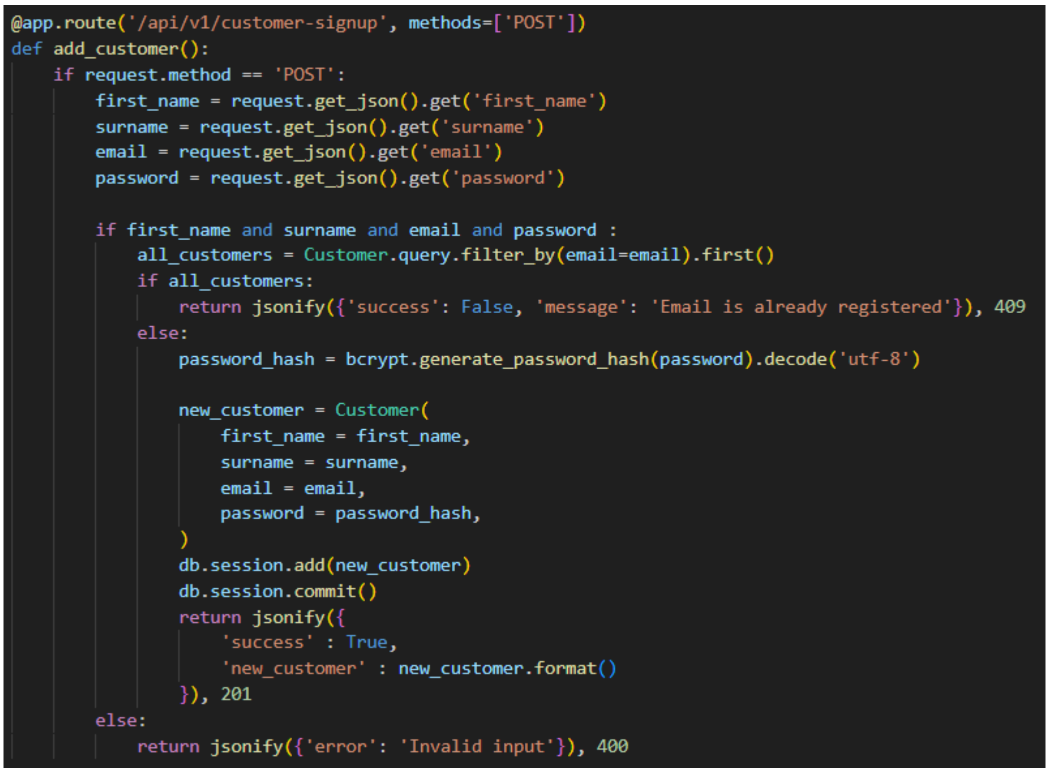


```python
from app import db

class Customer(db.Model):
    __tablename__ = 'customers'
    customer_id = db.Column(db.Integer, primary_key=True, nullable=False)
    first_name = db.Column(db.String(50), nullable=False)
    surname = db.Column(db.String(50), nullable=False)
    email = db.Column(db.String(100), unique=True, nullable=False)
    password = db.Column(db.String(250), nullable=False)
```

Figure 9 SQLAlchemy Data Model for Customer Objects

As shown in **Figure** 10, the API endpoint was created which consists of the following: an '@app.route' decorator which declares the URL path where the endpoint will be accessible, along with the accepted request methods. In the case of this route the URL is'/api/v1/customer-signup' and method = 'POST'. Under this is the associated function where the operations on the data posted to the endpoint are handled.

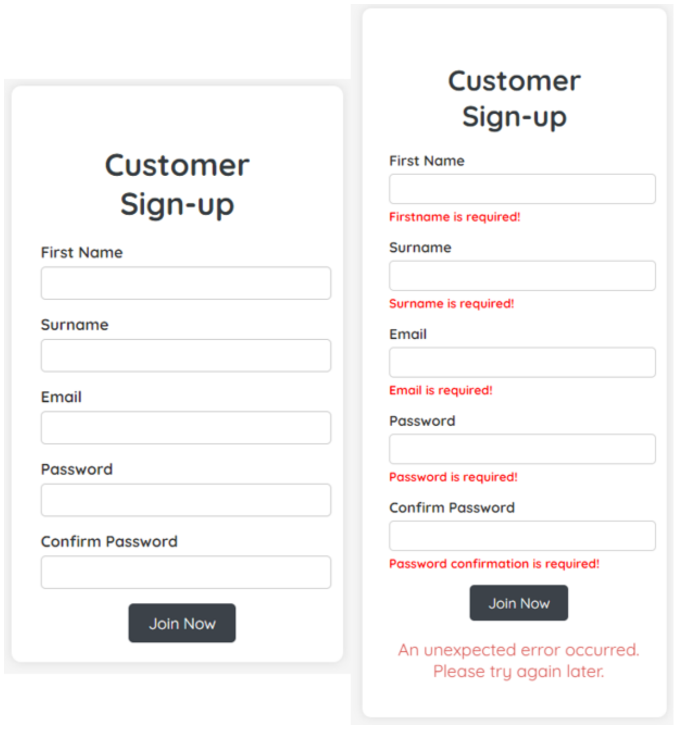
The user inputs are extracted from the JSON data and assigned to variables. The entered email is compared against the database to ensure the customer doesn't already exist. If the email is unique and all the input information is correct a new customer object is created and committed to the database. Bcrypt is used as a necessary security measure to hash user passwords before they are stored in the database, ensuring the passwords are securely protected. The route then responds with the appropriate status code which communicates the outcome of the request to the front end and enables easier debugging.



```python
@app.route('/api/v1/customer-signup', methods=['POST'])
def add_customer():
    if request.method == 'POST':
        first_name = request.get_json().get('first_name')
        surname = request.get_json().get('surname')
        email = request.get_json().get('email')
        password = request.get_json().get('password')

        if first_name and surname and email and password :
            all_customers = Customer.query.filter_by(email=email).first()
            if all_customers:
                return jsonify({'success': False, 'message': 'Email is already registered'}), 409
            else:
                password_hash = bcrypt.generate_password_hash(password).decode('utf-8')

                new_customer = Customer(
                    first_name = first_name,
                    surname = surname,
                    email = email,
                    password = password_hash,
                )
                db.session.add(new_customer)
                db.session.commit()
                return jsonify({
                    'success' : True,
                    'new_customer' : new_customer.format()
                }), 201
        else:
            return jsonify({'error': 'Invalid input'}), 400
```

Figure 10 API Endpoint for Customer Object Creation

With the route and API endpoint established, development of the front-end began. To collect the user inputs for the signup page a React form was used. When changes are made in the input forms, an onChange function is called that collects the new value and stores it in a state. When the 'join now' button is clicked the onSubmit function is called that validates that the state values have been entered and are formatted correctly. If any values don't meet the requirements an appropriate error message is displayed under the input field and the form submission is cancelled. The code for the state management, form validation and input handling can be found in Appendix B.1.1, B.1.2 & B.1.3

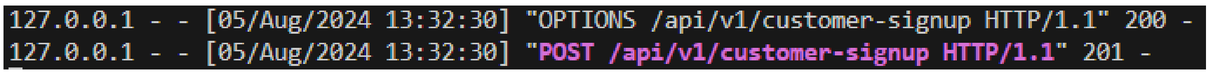


Firstname is required!
Surname is required!
Email is required!
Password is required!
Password confirmation is required!
An unexpected error occurred. Please try again later.

Figure 11 Customer Signup component (Left),

Customer Signup Component with Error Messages Displayed (Right)

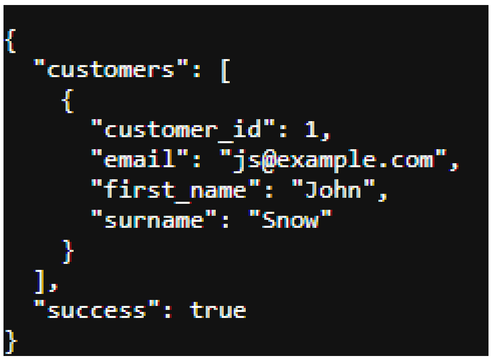
If all the values are entered correctly, the submit function attempts to post the values to the API endpoint and awaits the response. This can be seen below in **Figure 12** where the server receives an ‘OPTIONS’ request from the client checking if it can perform a POST request to the API endpoint, followed by a successful POST confirming the customer object creation.



```
127.0.0.1 - - [05/Aug/2024 13:32:30] "OPTIONS /api/v1/customer-signup HTTP/1.1" 200 -
127.0.0.1 - - [05/Aug/2024 13:32:30] "POST /api/v1/customer-signup HTTP/1.1" 201 -
```

Figure 12 Server Messages Upon Form Submission

As a final measure to ensure the customer object has been successfully created, a second API endpoint was created to get a formatted JSON of all the customer objects in the database. Connecting to the API endpoint returns the JSON, shown in **Figure 13**, which confirms the creation route performed as expected.



```
{
  "customers": [
    {
      "customer_id": 1,
      "email": "js@example.com",
      "first_name": "John",
      "surname": "Snow"
    }
  ],
  "success": true
}
```

Figure 13 JSON Displaying Formatted Customer Object

This provided confirmation that all three layers of the application were working and communicating correctly, paving the way for more pages and routes to be created.

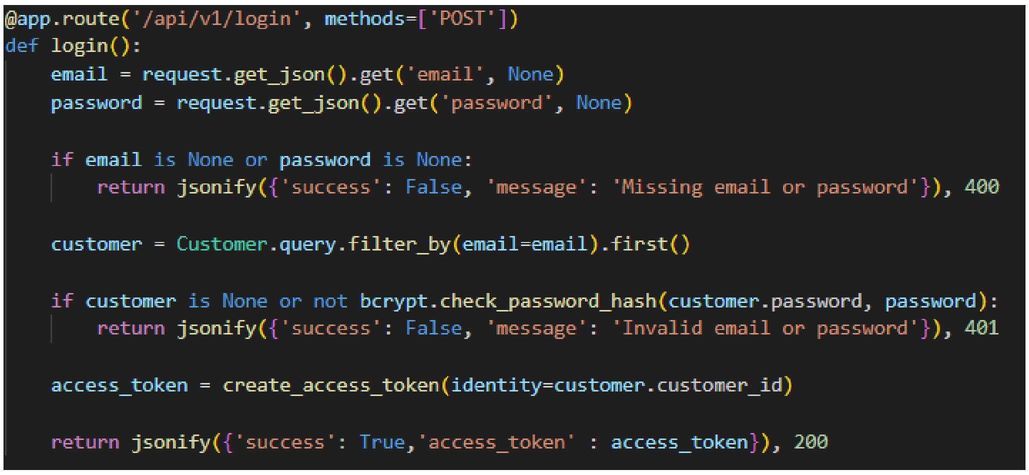
5.2 Sprint 2

User stories covered in this sprint:

- US3 - As a customer, I want to log in so that I can access the service and book appointments.
- US4 - As a customer, I want to log off so that I can end the session and protect my account.
- US6 - As a customer, I want to book appointments online so that I am guaranteed a haircut at a time of my choosing.
- US7 - As a customer, I want to choose a specific treatment and time when booking so that my specific needs are met.
- US8 - As a customer, I want to select from a list of available staff when booking so that I can be treated by my preferred staff member.
- US9 - As a customer, I want to leave a comment when booking so that I can provide special instructions or requests.

5.2.1 Customer Login & Logout

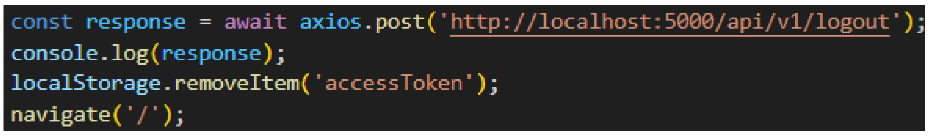
When creating user authentication in Flask there are two popular options: Session-based authentication and token-based authentication. JSON Web Tokens (JWT) were chosen as they are quick and easy to implement with React and scale well. They also enable further layers of authentication to be implemented relatively easily, which will be implemented in a later sprint. The endpoint, visualised in **Figure** 14, first verified the provided data was correct before comparing it with customer objects in the database. It then returns the corresponding response and HTTP status code depending on the success of the operation. If the user provided the correct credentials, a token was created with their customer_id acting as the identity. This token is then passed back to the front-end where it's stored locally, while convenient this is one of the cons of using JWT as it makes the application vulnerable to cross-site scripting attacks.



```
@app.route('/api/v1/login', methods=['POST'])
def login():
    email = request.get_json().get('email', None)
    password = request.get_json().get('password', None)

    if email is None or password is None:
        return jsonify({'success': False, 'message': 'Missing email or password'}), 400

    customer = Customer.query.filter_by(email=email).first()

    if customer is None or not bcrypt.check_password_hash(customer.password, password):
        return jsonify({'success': False, 'message': 'Invalid email or password'}), 401

    access_token = create_access_token(identity=customer.customer_id)

    return jsonify({'success': True,'access_token' : access_token}), 200
```

Figure 14 Customer Login API Endpoint

When using JWTs logging out a user is fairly simple as it can be handled on the front end as shown in **Figure 15.** It is accomplished by removing the access token from the user's local storage. The POST request to the endpoint isn't to process any data rather it's used as a confirmation that the user is still connected to the service. It's set to respond with a 200 status for debugging purposes.



```
const response = await axios.post('http://localhost:5000/api/v1/logout');
console.log(response);
localStorage.removeItem('accessToken');
navigate('/');
```

Figure 15 Removal of the Access Token Upon Successful Connection

5.2.2 Token Verification

With a token system implemented, they could now be used to restrict access to certain routes and pages. This is thanks to a feature of the JWT library where routes can be given the '@jwt_required' decorator, which restricts access to a route unless a valid token is provided. This was used to create a Flask function and associated React component which could be imported into other components to be used as a user authenticator shown in Appendix B.2.1. **U**pon verifying a valid token the function returns a user object which would be stored in a state and used to hide components or pages from the user.

5.2.3 Customer Bookings

To fulfil US6, US7, US8 and US9 for this sprint, customers would need to be able to create bookings customised for their needs. The predicted time to complete this story was underestimated as it would involve not just creating booking objects but also specifying the staff and treatments. This would require them to be added to the database before being requested and displayed to the customer doing the booking. This resulted in technical debt which had to be picked up in the next sprint. To save time, instead of coding the endpoints and front-end pages for creating Staff and Treatment objects, they were instead added directly to the database using console commands.

As shown below the booking page consisted of several components. The components were designed separately and imported to improve code organisation and abstraction. The general plan for the file structure is displayed in **Figure 16**.

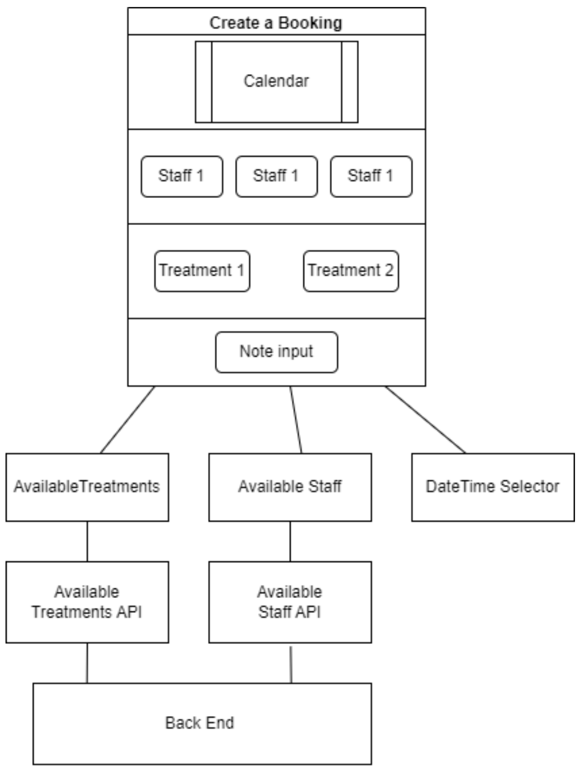
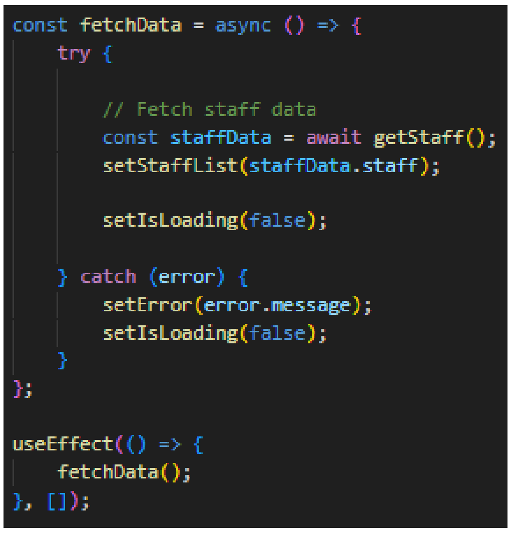


Figure 16 Planned File Structure for Booking Creation Page

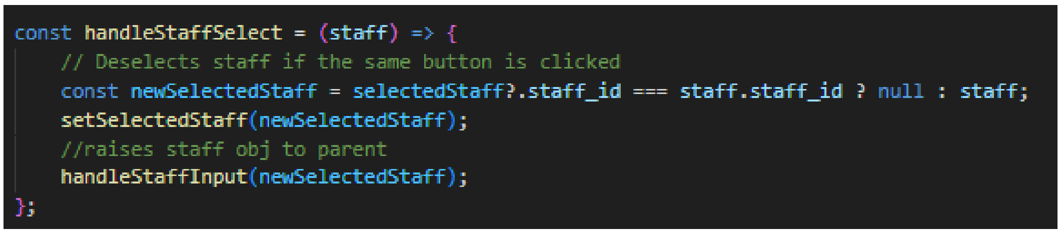
The available staff list and treatments list use the same method to gather and display information from the database. So, the following process description applies to both. First, an endpoint was created that queries the database for Staff objects with filter applied for the ‘is_available’ value set to true. An AvailableStaffAPI.js file was then created to fetch staff data from the API endpoint. This file processes any error handling by displaying appropriate error messages and assigns headers to assign the content as JSON. An example of this API file can be found in Appendix B.3.1. This was abstracted away from the component file to improve the organisation and readability of the code. The AvailableStaff.js component then imports the functionality from API to get the data from the backend. This is accomplished using asynchronous functions and React’s UseEffect Hook. In React asynchronous functions are used for performing operations that take time to complete. They allow the rest of the code to continue running while the operation finishes, which ensures that the UI remains responsive to the user. UseEffect is a React Hook that allows for operations to be performed that affect something outside the scope of the function, in this case, it’s fetching data from the API. These come together to enable the list of staff members to load smoothly while the data is being fetched. An example of how they are used is shown below in **Figure 17**.



```
const fetchData = async () => {
    try {

        // Fetch staff data
        const staffData = await getStaff();
        setStaffList(staffData.staff);

        setIsLoading(false);

    } catch (error) {
        setError(error.message);
        setIsLoading(false);
    }
};

useEffect(() => {
    fetchData();
}, []);
```

Figure 17 Example Usage of the React UseEffect and Asynchronous Function to Fetch Data

An additional function was created to handle the selection of a staff member, displayed in **Figure 18**. It's called when one of the list items is clicked and assigns the selected staff object to a UseState which is passed up to the parent component, which in this case is the booking page.



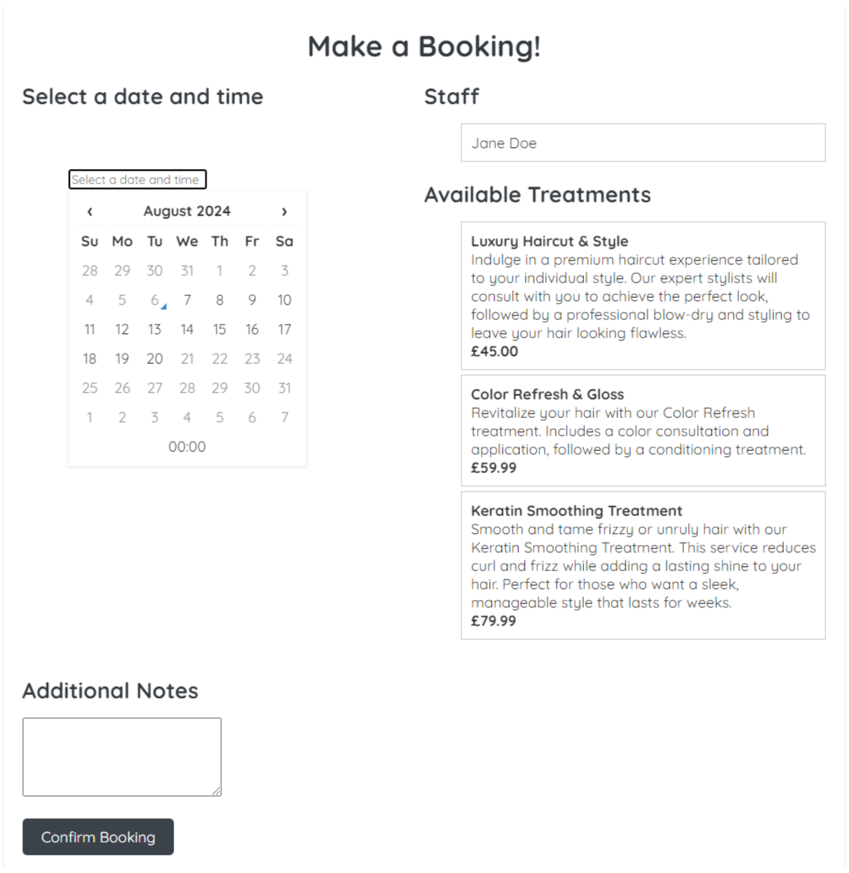
```
const handleStaffSelect = (staff) => {
    // Deselects staff if the same button is clicked
    const newSelectedStaff = selectedStaff?.staff_id === staff.staff_id ? null : staff;
    setSelectedStaff(newSelectedStaff);
    //raises staff obj to parent
    handleStaffInput(newSelectedStaff);
};
```

Figure 18 Staff Selection Function

The remainder of AvailableStaff.js and the JSX it returns can be found in Appendix B.3.2 for further review.

To create the date and time selector component, a library called reactDateTime was imported. It was chosen for its ability to configure the available dates and times that the user can select, and as it can format the returned date string. It was configured to only allow the selection of days from the current date to two weeks in advance, and only between the times of 9 am-5 pm. The output was formatted to 'YYYY-MM-DD HH: MM' to ensure it was compatible with SQLAlchemy's DateTime type. The full component code can be found in Appendix B.4.1.

The finished components were then imported to the AddBooking.js file for use. The file used a similar form design to the customer creation file, where the form inputs would be stored in a state before being processed for errors upon submission. If there are no errors with the values, a POST call would be made to the API to create the booking object. The resulting booking page can be seen below in **Figure 19.**


Make a Booking!

Select a date and time

Select a date and time

‹			August 2024			›
Su	Mo	Tu	We	Th	Fr	Sa
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
1	2	3	4	5	6	7

00:00

Staff

Jane Doe

Available Treatments

Luxury Haircut & Style
Indulge in a premium haircut experience tailored to your individual style. Our expert stylists will consult with you to achieve the perfect look, followed by a professional blow-dry and styling to leave your hair looking flawless.
£45.00

Color Refresh & Gloss
Revitalize your hair with our Color Refresh treatment. Includes a color consultation and application, followed by a conditioning treatment.
£59.99

Keratin Smoothing Treatment
Smooth and tame frizzy or unruly hair with our Keratin Smoothing Treatment. This service reduces curl and frizz while adding a lasting shine to your hair. Perfect for those who want a sleek, manageable style that lasts for weeks.
£79.99

Additional Notes

Figure 19 Booking Creation Page

Since a customer ID is required to create the booking object and the ID is pulled from the access token, it was necessary to implement authentication to restrict access to the booking page if no token was detected. The token verification function discussed earlier in the sprint was used to display a popup component, illustrated in **Figure 20**, preventing the user from creating a booking if they had not logged in. This component would be re-used on any page that required a token, for example, the account details page.

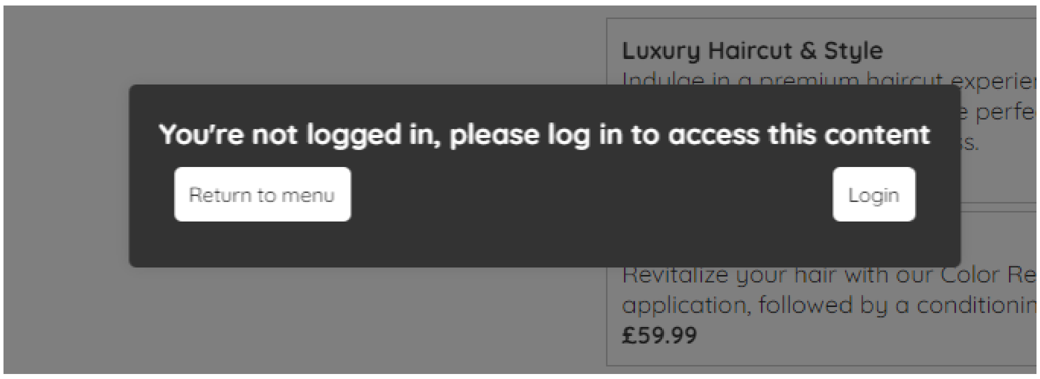


Figure 20 Popup Component Preventing Access to Booking Page

5.3 Sprint 3

As mentioned before, a portion of sprint 3 was dedicated to dealing with the technical debt resulting from the booking creation overshooting its deadline. This somewhat reduced the number of stories that could be completed.

In this sprint the stories to be completed were:

- US2 - As a customer, I want to edit my account details so that my personal details remain up to date.
- US10 - As a customer, I want to cancel a booking so that I can free up my schedule and avoid any associated charges or penalties.
- US12 - As a customer, I want to delete my account so that I can remove my personal information from the system.

5.3.1 Home & Menu

To enable the user to move between the different pages a homepage, menu and navigation bar were created. The navigation bar used the token authenticator function to switch between two states, illustrated in **Figure 21**, to allow the user to more easily login and logout.

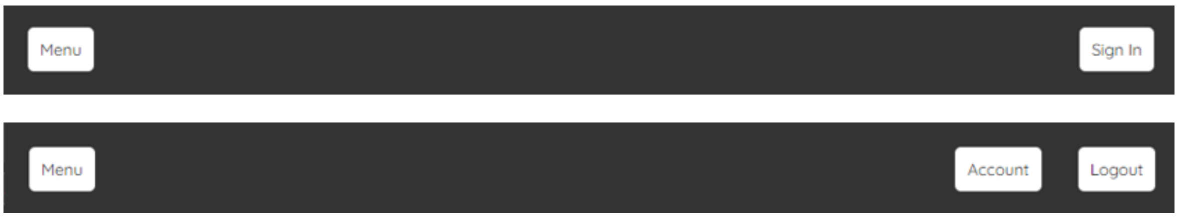
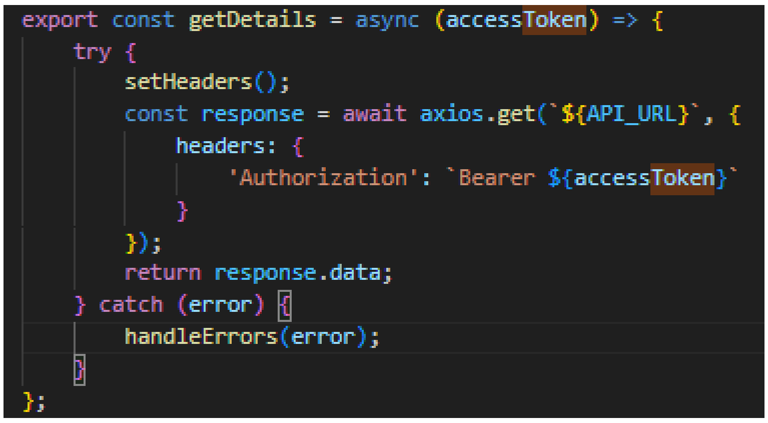


Figure 21 NavBar Component when Logged Out (Top), NavBar Component when Logged in (Bottom)

5.3.2 Customer Account Details

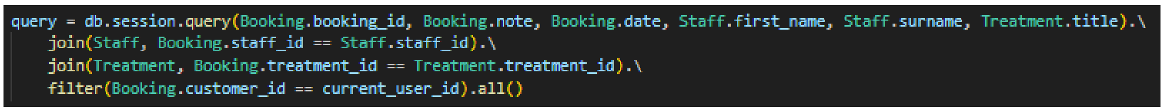
To fetch and display the customers' details, the same process was applied as the AvailableStaff.js component from sprint 2. Where an asynchronous function in AccountDetails.js would await a response from an API for the information before displaying it. One key difference was that the API endpoint was given a '@jwt_required' decorator for added security. This meant that to access the endpoint the data would need to be configured to attach an authorisation header to the HTTP request with the access token as shown in **Figure 22.**



```javascript
export const getDetails = async (accessToken) => {
    try {
        setHeaders();
        const response = await axios.get(`${API_URL}`, {
            headers: {
                'Authorization': `Bearer ${accessToken}`
            }
        });
        return response.data;
    } catch (error) {
        handleErrors(error);
    }
};
```

Figure 22 API Call with Access Token

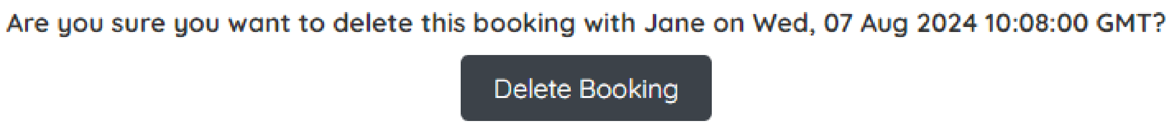
The function associated with the route then decodes the access token to identify the customer ID which is used to query their information. Since the account details should also include the customer’s bookings, further work was necessary to modify the route to also query for bookings with the user’s customer ID. As booking records only store the staff and treatment IDs, rather than their corresponding names and titles, a join operation had to be created between the bookings, treatment and staff tables to return the desired information. This join operation is illustrated in **Figure 23,** where Bookings and Staff are joined via the ‘staff_id’ value, and Bookings and Treatments were joined via the ‘treatment_id’ value.



```
query = db.session.query(Booking.booking_id, Booking.note, Booking.date, Staff.first_name, Staff.surname, Treatment.title).\
    join(Staff, Booking.staff_id == Staff.staff_id).\
    join(Treatment, Booking.treatment_id == Treatment.treatment_id).\
    filter(Booking.customer_id == current_user_id).all()
```

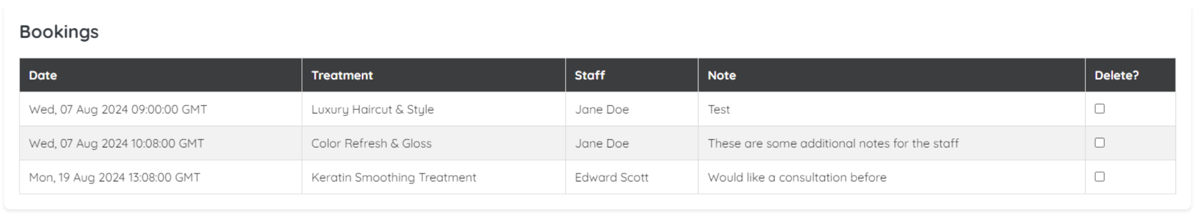
Figure 23 Booking Table Query with Joins

The list of bookings was then returned to the frontend in JSON format where they were mapped to a table to be displayed. An example of this table is displayed in **Figure 25.** To satisfy US10 bookings would also need to be removable, this was accomplished including a check box for booking selection. Upon selection, the booking’s data would be passed to an imported component, DeleteBookingButton.js. The component, illustrated in **Figure 24**, provided a message displaying information about the booking alongside a confirmation button. When pressed the button makes a request to an API endpoint to delete the object from the database. As additional security, the endpoint is JWT protected and requires that either the token’s identity matches the booking’s customer ID or it is recognised as a staff identity. Upon successful deletion route would respond with a 200 status-code and message, else if the request was rejected due to invalid values or authorisation it would respond with the appropriate code and message.



Are you sure you want to delete this booking with Jane on Wed, 07 Aug 2024 10:08:00 GMT?

Figure 24 Booking Deletion Confirmation Message and Button

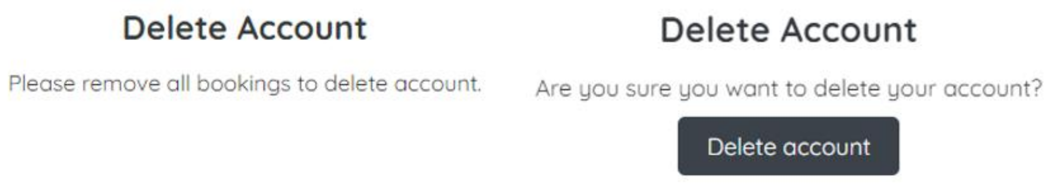


Bookings

Date	Treatment	Staff	Note	Delete?
Wed, 07 Aug 2024 09:00:00 GMT	Luxury Haircut & Style	Jane Doe	Test	☐
Wed, 07 Aug 2024 10:08:00 GMT	Color Refresh & Gloss	Jane Doe	These are some additional notes for the staff	☐
Mon, 19 Aug 2024 13:08:00 GMT	Keratin Smoothing Treatment	Edward Scott	Would like a consultation before	☐

Figure 25 Table Containing Booking Information on the Account Details Page

Finally, to satisfy US12 the customer would also need to be able to remove their account from the system. This utilised a similar process to the booking deletion where an imported button component would handle the deletion request to the API. However, there was one key change, as deleting an account while their bookings were on the system could cause errors when querying the data. For this reason, the component would only be accessible if no booking objects associated with the customer account were found. To indicate this to the user, the component would flip between the 2 states shown in **Figure 26**.



Delete Account

Please remove all bookings to delete account.

Delete Account

Are you sure you want to delete your account?

Figure 26 Component View while Bookings remain (Left), Component View with no Bookings (Right)

5.4 Sprint 4

With the application fulfilling the customer user stories the focus moved to the admin/ staff side of the application. Many of the processes that were used to develop the customer side of the application could be reworked into features required by the admin user stories, speeding up development.

User stories completed:

- US13 – As an admin, I want to log in so I can access the service and manage bookings.
- US14 – As an admin, I want to view upcoming bookings so that I can convey availability to walk in customers

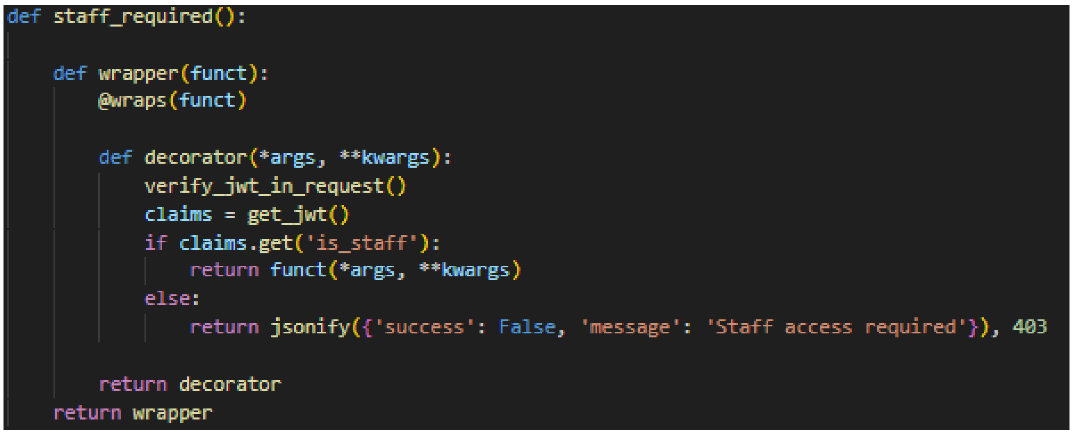
- US15 – As an admin, I want to view customer accounts so that I can view their previous and upcoming bookings

5.4.1 Staff Registration

The staff registration process is adapted from the customer registration created in sprint 1. It again uses a form to collect the inputs from the user, before validating there are no errors. The data is then posted to a Flask route which queries the database to prevent duplicates before returning an appropriate status code depending on whether the process was successful or not. The password is again encrypted using Bcrypt. One difference is that the newly created staff member can be assigned a Boolean value of ‘is_active’ which is used to determine whether their name appears in the booking creation process.

5.4.2 Staff Login & Authentication

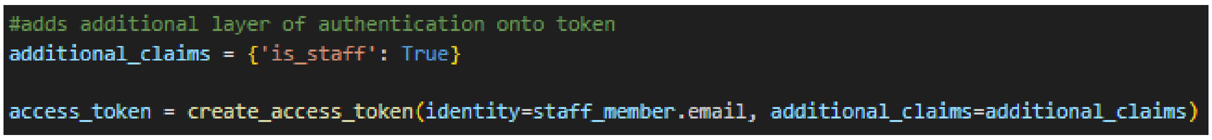
Like the customer login page, StaffLogin.js also used a form to validate inputs before posting the information to an API endpoint to be verified. However, a change to the authentication was required to provide additional security. While staff tokens can still access routes protected by the ‘@jwt_required’ decorator, it was necessary to create an additional decorator to restrict access to administrative pages. This involved creating a new decorator called ‘staff required’ illustrated in **Figure 27**. When the ‘@staff_required()’ decorator is applied to a function, that function is passed to the wrapper which is responsible for setting up the decorator’s behaviour. In this case, the decorator checks the Boolean value ‘is_staff’ is present in the token and is set to true. If ‘is_staff’ is true, the decorator allows the function to operate as normal. If not, it prevents the function from executing and returns the JSON response with an error message.



```python
def staff_required():

    def wrapper(funct):
        @wraps(funct)

        def decorator(*args, **kwargs):
            verify_jwt_in_request()
            claims = get_jwt()
            if claims.get('is_staff'):
                return funct(*args, **kwargs)
            else:
                return jsonify({'success': False, 'message': 'Staff access required'}), 403

        return decorator
    return wrapper
```

Figure 27 The staff_required() Decorator Function

To include the 'is_staff' value in the token, an additional step was added in the 'staff_login' route. The value was declared as an additional claim in the 'create_access_token()' function before the access token was returned to the front-end (**Figure 28**). This staff decorator was then applied to all future admin-specific routes to restrict access.



```python
#adds additional layer of authentication onto token
additional_claims = {'is_staff': True}

access_token = create_access_token(identity=staff_member.email, additional_claims=additional_claims)
```

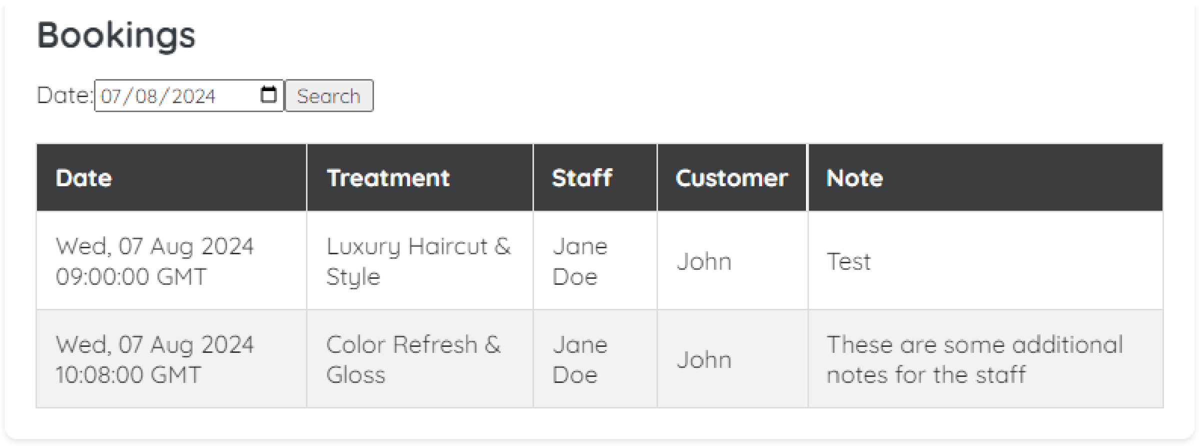
Figure 28 The is_staff value being assigned to Staff access tokens

To log out of the admin side of the app the logout component was re-used which removes the access token from the local storage and loaded the home page.

5.4.3 View Upcoming Bookings

To satisfy US14 admins would need to be able to see upcoming bookings in advance. As planned in the low-fidelity prototype, this would require the user to enter a date after which the page would return all the appointments booked for that day.

The dateTimePicker library, previously used for the calendar in the create booking page, wasn't deemed necessary. Instead, a single input of type 'date' was used along with a submit button. When clicked the submit button would make a get request to a 'booking_search' API endpoint which took the date as a string parameter. The flask route converts the date to dateTime value before using it to filter a query search of the booking table. A join between the booking table, staff table, treatment table and customer table was also completed to ensure the returned information was informative to the reader. This list of configured booking objects was then converted to JSON format and returned to the front-end along with a status code. The returned information was then mapped to a table to be displayed to the user as seen in **Figure()**.



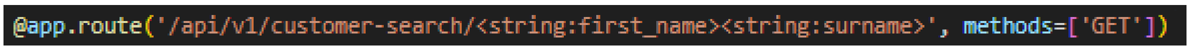
Date	Treatment	Staff	Customer	Note
Wed, 07 Aug 2024 09:00:00 GMT	Luxury Haircut & Style	Jane Doe	John	Test
Wed, 07 Aug 2024 10:08:00 GMT	Color Refresh & Gloss	Jane Doe	John	These are some additional notes for the staff

Figure 29 Results Table from Booking Search

5.4.4 Customer Search

US15 required that admins would be able to search for customers to view bookings associated with them. As discussed in the interviews this would used for customers who phoned in or walked in allowing staff to assign information for them and create bookings.

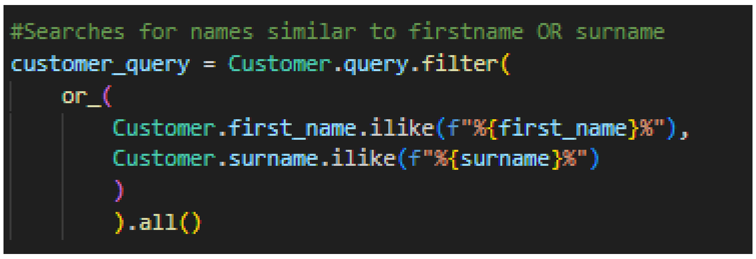
Building on the design from the low-fidelity prototype the search would take two inputs: the customer's first and surname. The values of inputs are passed to the API file which then makes a get request to the API endpoint with the names used as parameters as shown in Figure 30.



```python
@app.route('/api/v1/customer-search/<string:first_name><string:surname>', methods=['GET'])
```

Figure 30 Customer Search API Route

The API function was set to query the customer table for strings that contained the same pattern of characters as the entered names, as shown in Figure 31.



```python
#Searches for names similar to firstname OR surname
customer_query = Customer.query.filter(
    or_(
        Customer.first_name.ilike(f"%{first_name}%"),
        Customer.surname.ilike(f"%{surname}%")
        )
        ).all()
```

Figure 31 Query for Similar Names

5.5 Sprint 5

In the final sprint, major challenges were realised with some of the remaining user stories. Firstly, with US16, which relates to customising the data on the main page, where the unstructured nature of the data didn't fit in with the current database design and would require it to be redesigned to accommodate for this. Additionally, with US18 where restricting some staff from accessing features would require an additional wrapper. This was certainly achievable but would also require changes to be made to the staff table to accommodate for it. With the limited time remaining it was decided that the focus should be on completing stories that would not require large changes to be made to the system.

- US17 - As an admin, I want to create new admin accounts so that I can facilitate the hiring of new staff.

6 Testing

Testing is a crucial part of app development as it ensures the app's functionality and identifies bugs that can impact the user experience. Multiple testing methods were required to ensure that testing covered the full scope of the application. This would confirm that the application was stable and had fulfilled the user stories.

6.1 Unit Testing

Unit testing was used to verify that the API endpoints and their associated functions behaved as expected. Unit tests were written using the Pytest library which provides a simple but effective means of testing Python code. Fixtures were utilised to arrange input data before testing. This included the configuration of the database URL to a separate test database, as well as the creation of mock data such as sample customer and staff objects and their access tokens. Initially, unit tests were written for only positive results but after this uncovered few bugs it was expanded to also test for negative scenarios to ensure the error handling was correctly implemented. The results showed that there were unexpected status codes raised when tokens weren't correctly assigned. Examples of the unit tests used to test the customer routes can be seen in Appendix C.1.

6.2 User Acceptance Testing

User acceptance testing was used to ensure that the success criteria of the user stories had been met. It was also used to test the user experience and the front-end code was working as expected, as it is generally harder to write unit tests for React components. The user acceptance test questions can be found in Appendix C.2. One set of tests was conducted on a Linux System to ensure that the application performed similarly on multiple platforms.

6.3 Manual Testing

Manual testing that was completed during development was also documented and can be found in a table in Appendix C.3.

7 Analysis

It’s beneficial to analyse what was accomplished during the project’s development to better understand the application's strengths and flaws, as well as to gauge if the user requirements were fulfilled by the minimum viable product (MVP). This will allow for improvements to be made on future iterations of the application and prevent any mistakes from being repeated.

7.1 Requirements Analysis

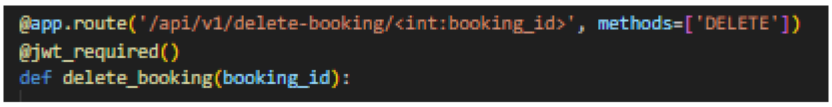
While it was beneficial to get insights from industry members directly, the feedback's reliability could have been improved by a larger number of interviewees. This would reduce the chance that crucial requirements are missed. As mentioned in section 2.1, multiple rounds of user participation using interactive prototypes would have allowed for a more refined design prior to the production of code and would ensure requirements were accurately represented.

7.1.1 Incomplete Requirements

Analysis of requirements that weren’t completed can be a helpful exercise as it can identify mistakes in the development process that can be avoided in future iterations. There were several stories that were not able to be completed during the development time. There was no rescheduling of bookings for either customers or staff implemented in the final product. With more time this could have been implemented by editing the booking object, similar to how the treatment edit page functions. Though this may have not been satisfactory for real-world use as will be discussed in the next section. While it's possible for staff to see upcoming bookings this feature was not available to customers, this also could have been implemented in a similar fashion to the staff page. With more time these stories could have been implemented, I believe part of the issue was my own hesitation to work on them. I realised too late in development that the design of the system time slots was ineffective and needed a rework. Another major story that wasn’t completed was the customisation of the main home page

7.2 Design Analysis

While generally the code is well structured and features good design choices to prevent errors from arising there was still room for improvement. Identifying poor design choices will prevent them from being made in future iterations and will enable easier code refactoring. Potentially due to my inexperience with the framework, I was hesitant to utilise props to their full potential. This resulted in some pages having multiple components at a time independently checking for an access token as they were each calling the checkAuth function. This could have been improved by only the parent component calling the function or retrieving the token and then passing the information to the children components. Another improvable design choice was the API URLs naming structure. An example, shown in **figure()**, uses '/api/v1/delete-booking' when using '/api/v1/booking' with a delete method would be enough to distinguish it from other similarly named routes. Again, this was a mistake due to inexperience that was realised late in development.



```
@app.route('/api/v1/delete-booking/<int:booking_id>', methods=['DELETE'])
@jwt_required()
def delete_booking(booking_id):
```

One design issue with using JWT for authentication is that storing them in the local storage results in a cross-site scripting (XSS) attack vulnerability. This would need to be addressed in future iterations, or it could result in a data breach.

Another flaw was how the bookings are being handled, currently attempting to double book at the same time does prevent the user from doing so. However, there is no visual cue prior to attempting the booking. This could be improved by altering the booking process, this may require only a rethink of the visual design or could require new libraries to be researched and imported.

8 Future Work

As mentioned in the previous section the main two flaws in the current application are the approaches to the current booking time selection process and the method for displaying information in the main page. I’m fairly confident that with more time the bookings could be re-worked without making changes to the database, it require a revision of what libraries are available as there may be a more suitable one than the one currently used. With regards to the home page, several additional tables would need to be added to the database to store the information set out by the requirements.

9 Conclusion

In conclusion, the project set out to create a cross-platform booking system, and the resulting MVP successfully fulfils the majority of the requirements. The system is operational across multiple platforms and provides a solid foundation for users to book appointments. However, there is still room for improvement, specifically in the design of the booking creation process and the application's general aesthetics. Future iterations of the application could make these the main focus without requiring significant changes to existing code which would allow for reduced development times. With these improvements, the booking system would be better equipped to function smoothly and provide the value that was set out by the requirements.

References

[1] Z. Kurtanović and W. Maalej, “Automatically Classifying Functional and Non-functional Requirements Using Supervised Machine Learning,” IEEE, 2017.

[2] T. Kravchenko, T. Bogdanova and T. Shevgunov, “Ranking Requirements Using MoSCoW Methodology in Practice,” in *Cybernetics Perspectives in Systems*, 2022.

[3] C. Gackenheimer, Introduction to React, Apress, 2015.

[4] A. M. Turing, “Computer machinery and intelligence,” *Mind,* vol. 59, no. 236, pp. 433-460, 1950.

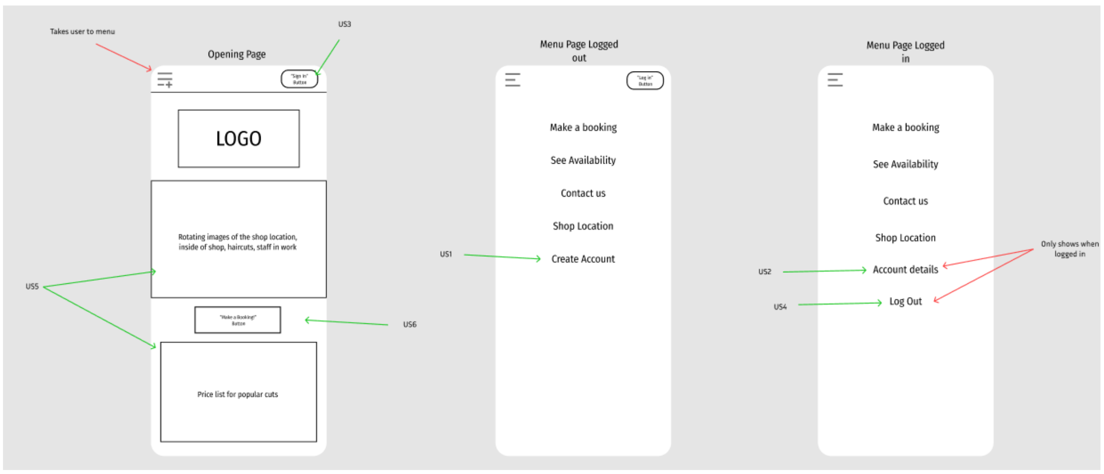
10 Appendix

A Low Fidelity Prototpye

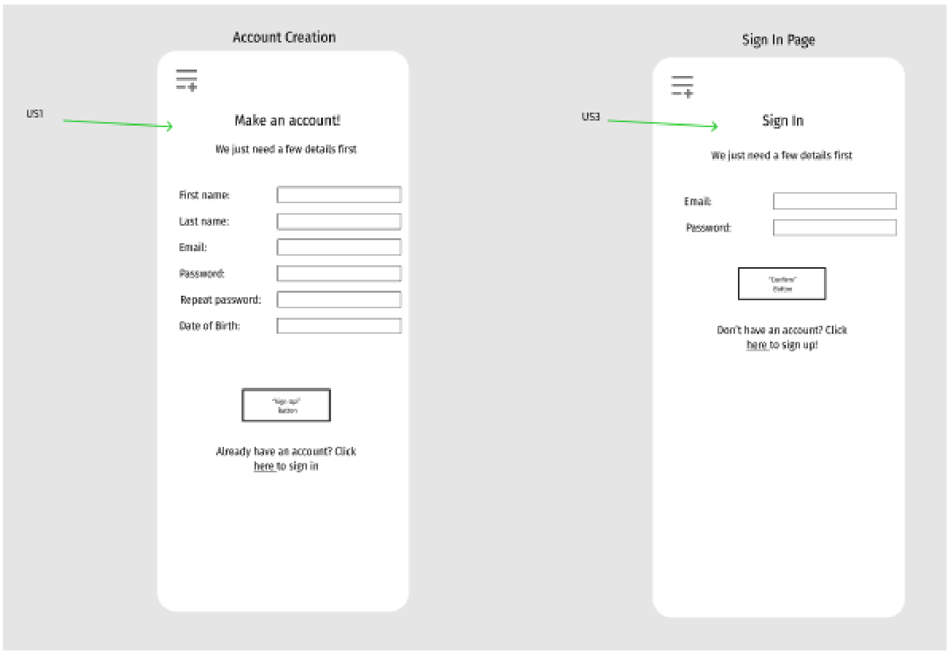
The low fidelity prototype was created during the design phase of the project.

A.1 Customer Pages

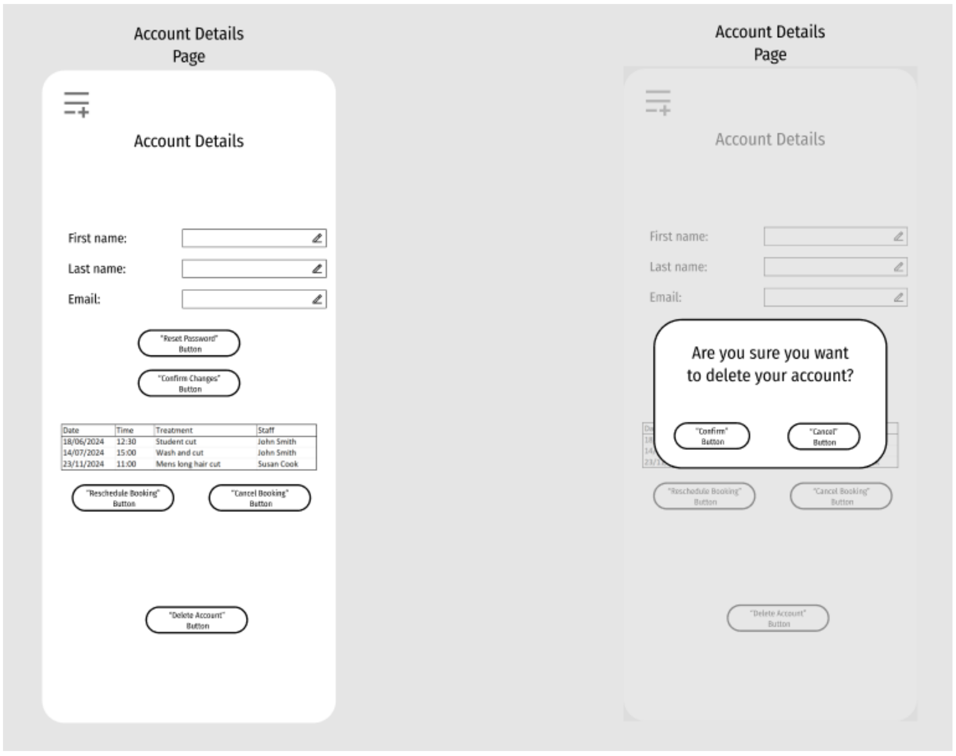
A.1.1 Opening and Menu pages



A.1.2 Account Creation and Sign-in Pages

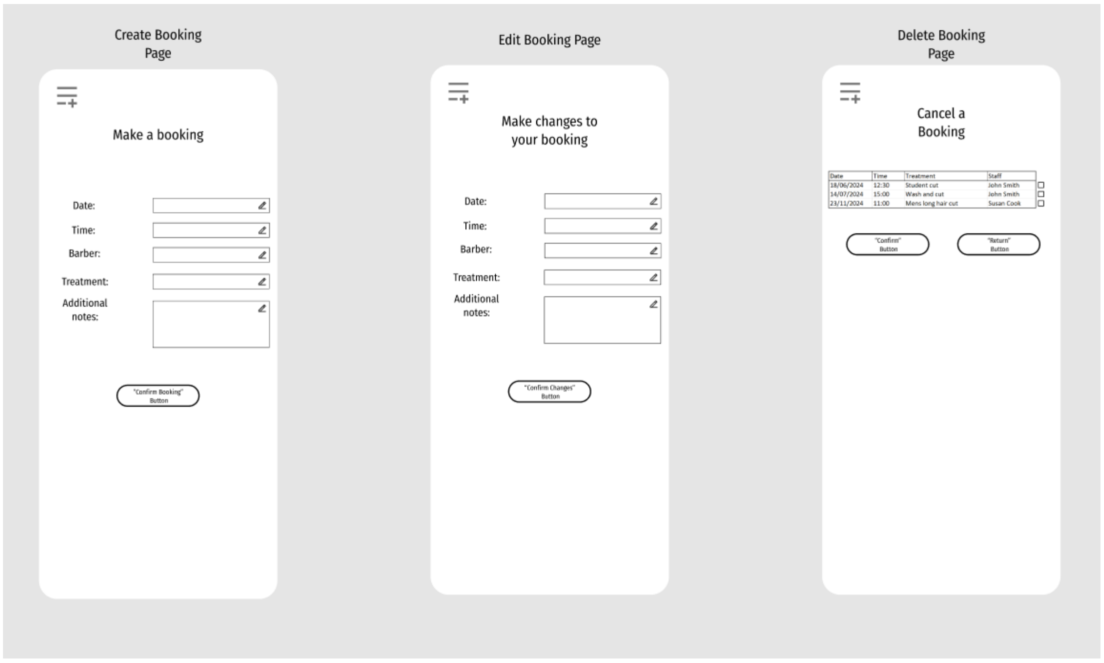


A.1.3 Account Details Page and Popup



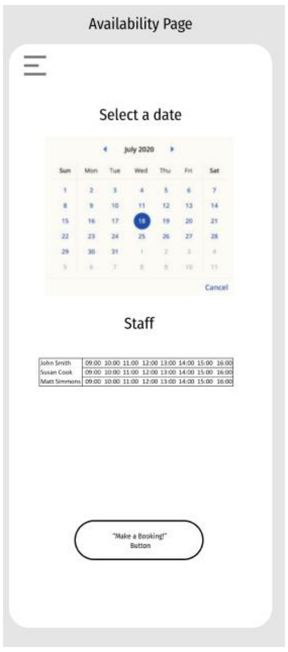
Date	Time	Treatment	Staff
18/06/2024	12:30	Student cut	John Smith
14/07/2024	15:00	Wash and cut	John Smith
23/11/2024	11:00	Mens long hair cut	Susan Cook

A.1.4 Booking Pages



Date	Time	Treatment	Staff	
18/06/2024	12:30	Student cut	John Smith	☐
14/07/2024	15:00	Wash and cut	John Smith	☐
23/11/2024	11:00	Mens long hair cut	Susan Cook	☐

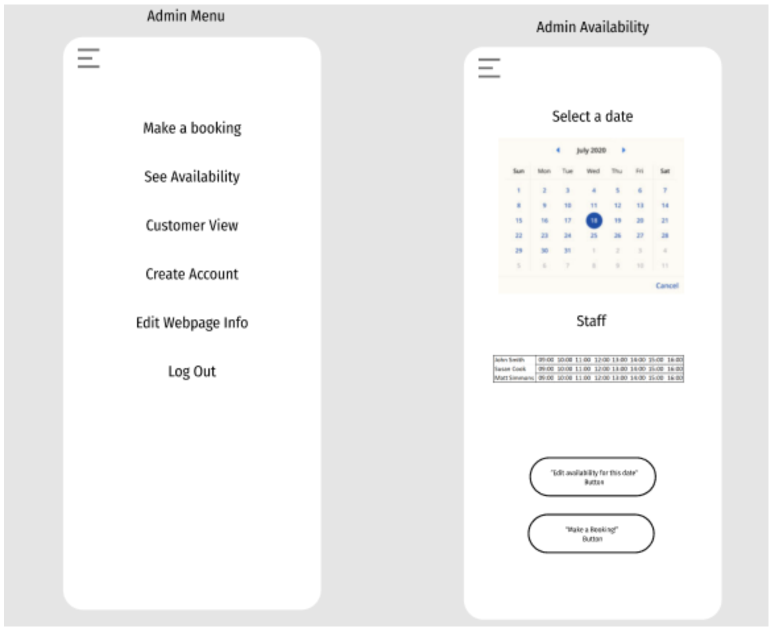
A.1.5 Availability Page



John Smith	09:00	10:00	11:00	12:00	13:00	14:00	15:00	16:00
Susan Cook	09:00	10:00	11:00	12:00	13:00	14:00	15:00	16:00
Matt Simmons	09:00	10:00	11:00	12:00	13:00	14:00	15:00	16:00

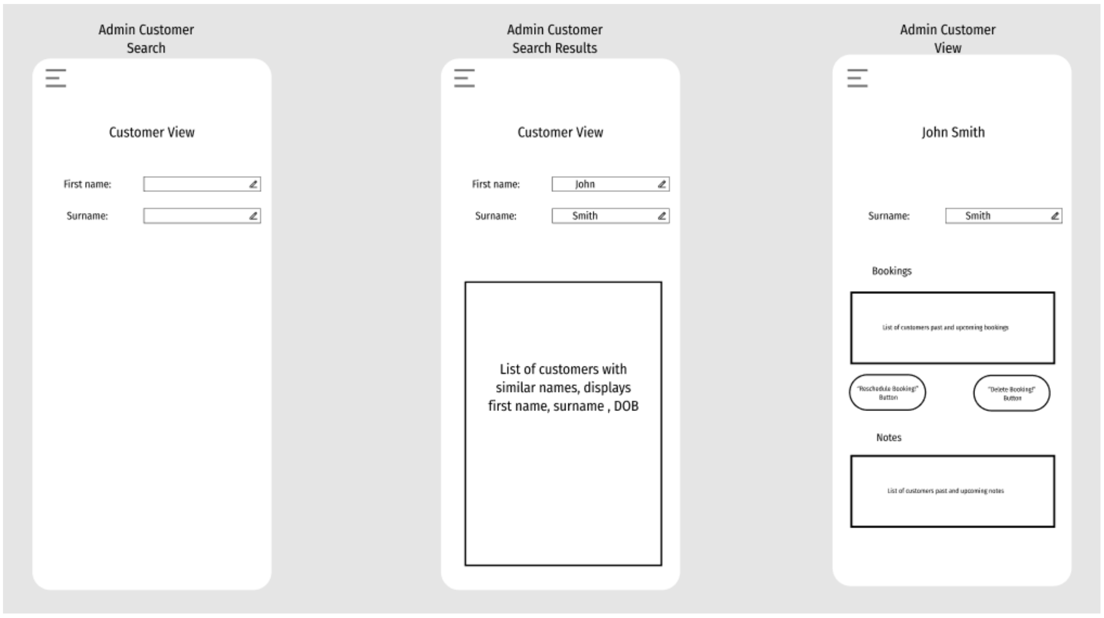
A.2 Admin LFP Pages

A.2.1 Admin Menu and Availability Page



John Smith	09:00	10:00	11:00	12:00	13:00	14:00	15:00	16:00
Susan Cook	09:00	10:00	11:00	12:00	13:00	14:00	15:00	16:00
Matt Simmons	09:00	10:00	11:00	12:00	13:00	14:00	15:00	16:00

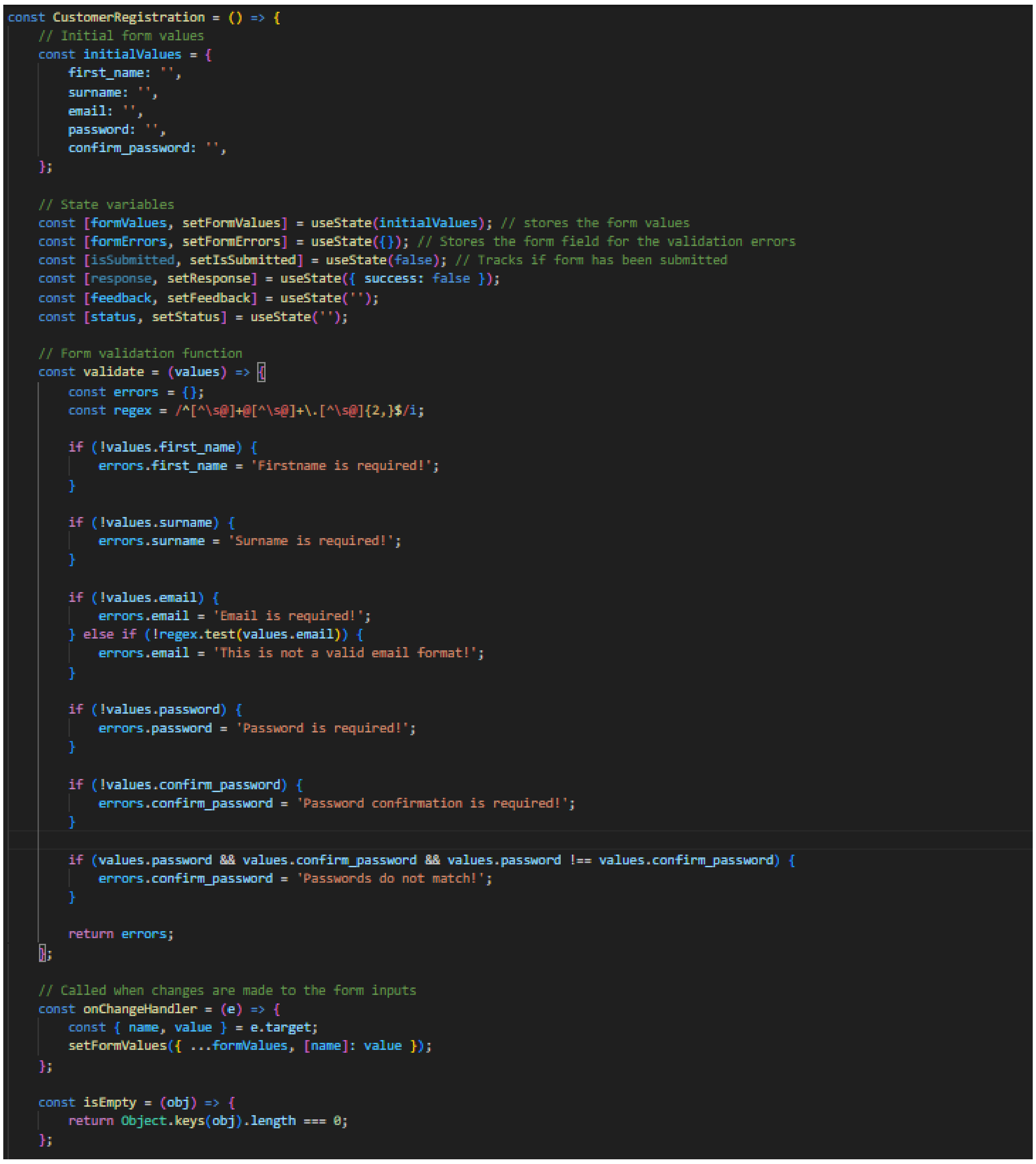
A.2.2 Admin Customer SearchPage


Admin Customer Search
Customer View
First name:
Surname:
Admin Customer Search Results
Customer View
First name: John
Surname: Smith
List of customers with similar names, displays first name, surname , DOB
Admin Customer View
John Smith
Surname: Smith
Bookings
List of customers past and upcoming bookings
"Reschedule Booking" Button
"Delete Booking" Button
Notes
List of customers past and upcoming notes

B Code Snippets

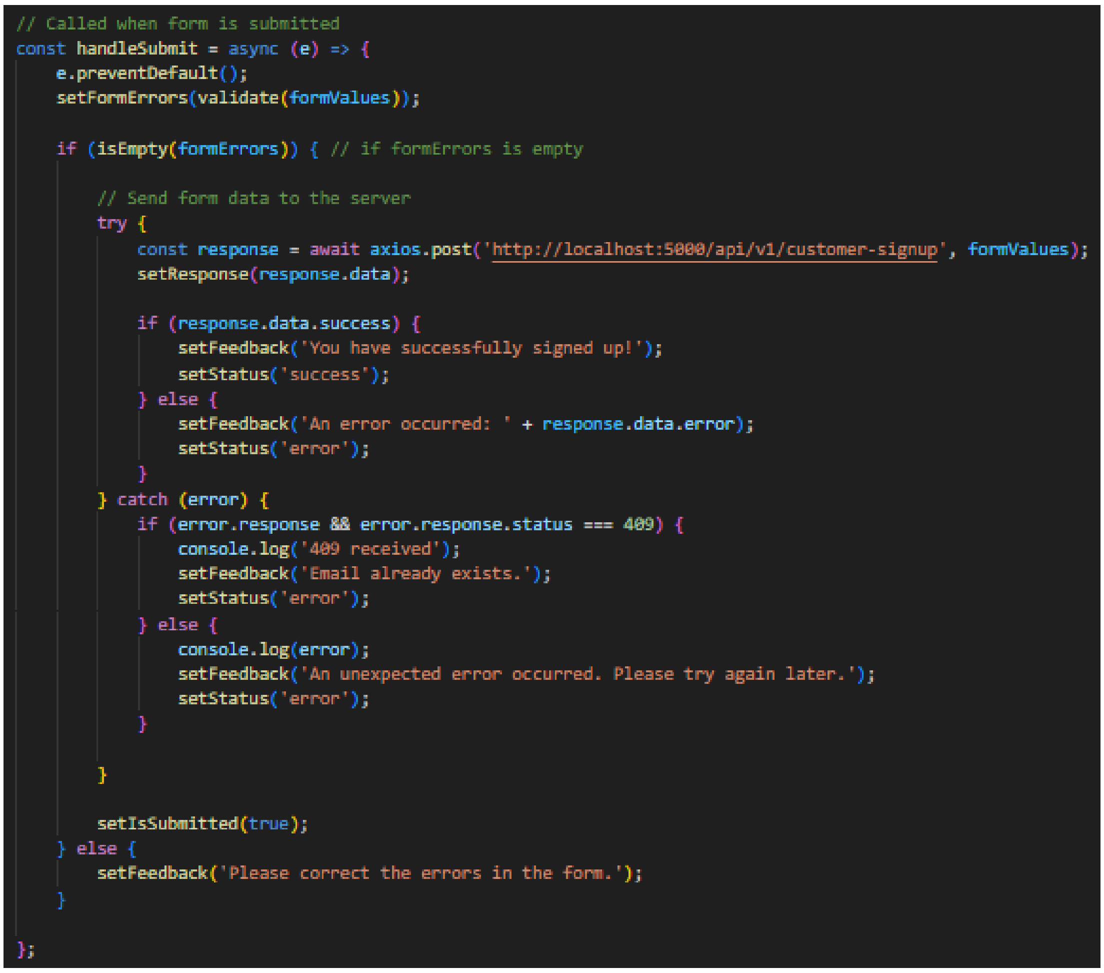
B.1 Customer Creation

B.1.1 Example of State Management and Form Validation



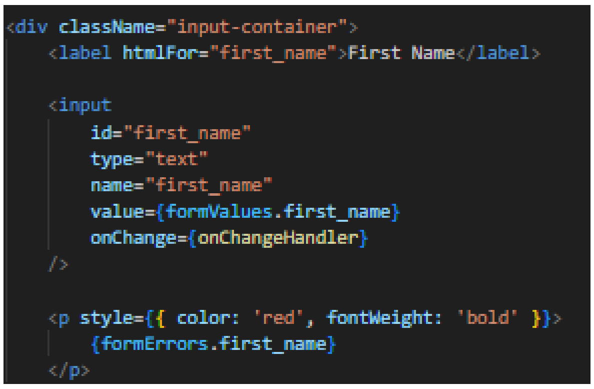
```javascript
const CustomerRegistration = () => {
    // Initial form values
    const initialValues = {
        first_name: '',
        surname: '',
        email: '',
        password: '',
        confirm_password: '',
    };

    // State variables
    const [formValues, setFormValues] = useState(initialValues); // stores the form values
    const [formErrors, setFormErrors] = useState({}); // Stores the form field for the validation errors
    const [isSubmitted, setIsSubmitted] = useState(false); // Tracks if form has been submitted
    const [response, setResponse] = useState({ success: false });
    const [feedback, setFeedback] = useState('');
    const [status, setStatus] = useState('');

    // Form validation function
    const validate = (values) => {
        const errors = {};
        const regex = /^[^\s@]+@[^\s@]+\.[^\s@]{2,}$/i;

        if (!values.first_name) {
            errors.first_name = 'Firstname is required!';
        }

        if (!values.surname) {
            errors.surname = 'Surname is required!';
        }

        if (!values.email) {
            errors.email = 'Email is required!';
        } else if (!regex.test(values.email)) {
            errors.email = 'This is not a valid email format!';
        }

        if (!values.password) {
            errors.password = 'Password is required!';
        }

        if (!values.confirm_password) {
            errors.confirm_password = 'Password confirmation is required!';
        }

        if (values.password && values.confirm_password && values.password !== values.confirm_password) {
            errors.confirm_password = 'Passwords do not match!';
        }

        return errors;
    };

    // Called when changes are made to the form inputs
    const onChangeHandler = (e) => {
        const { name, value } = e.target;
        setFormValues({ ...formValues, [name]: value });
    };

    const isEmpty = (obj) => {
        return Object.keys(obj).length === 0;
    };
```

B.1.2 Example of Custom Submit Operation



```
// Called when form is submitted
const handleSubmit = async (e) => {
    e.preventDefault();
    setFormErrors(validate(formValues));

    if (isEmpty(formErrors)) { // if formErrors is empty

        // Send form data to the server
        try {
            const response = await axios.post('http://localhost:5000/api/v1/customer-signup', formValues);
            setResponse(response.data);

            if (response.data.success) {
                setFeedback('You have successfully signed up!');
                setStatus('success');
            } else {
                setFeedback('An error occurred: ' + response.data.error);
                setStatus('error');
            }
        } catch (error) {
            if (error.response && error.response.status === 409) {
                console.log('409 received');
                setFeedback('Email already exists.');
                setStatus('error');
            } else {
                console.log(error);
                setFeedback('An unexpected error occurred. Please try again later.');
                setStatus('error');
            }

        }

        setIsSubmitted(true);
    } else {
        setFeedback('Please correct the errors in the form.');
    }

};
```

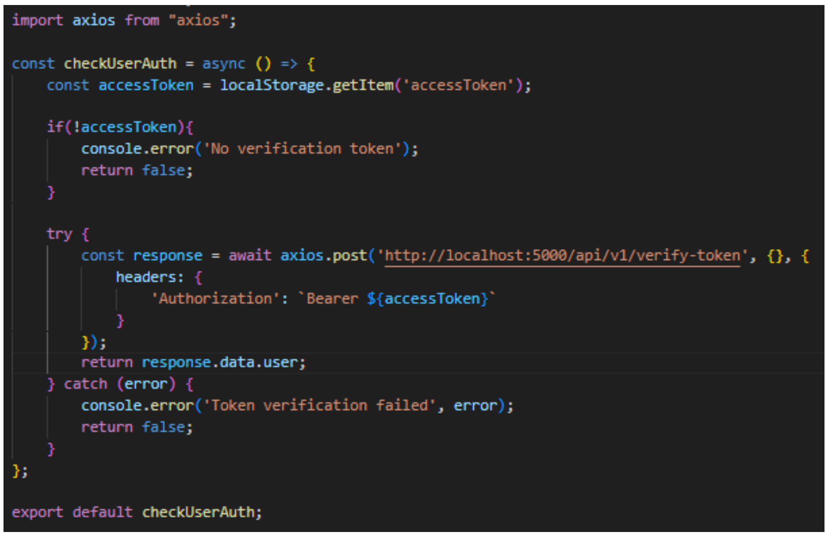
B.1.3 Example of Input Container



```
<div className="input-container">
    <label htmlFor="first_name">First Name</label>

    <input
        id="first_name"
        type="text"
        name="first_name"
        value={formValues.first_name}
        onChange={onChangeHandler}
    />

    <p style={{ color: 'red', fontWeight: 'bold' }}>
        {formErrors.first_name}
    </p>
```

B.2 User Authentication

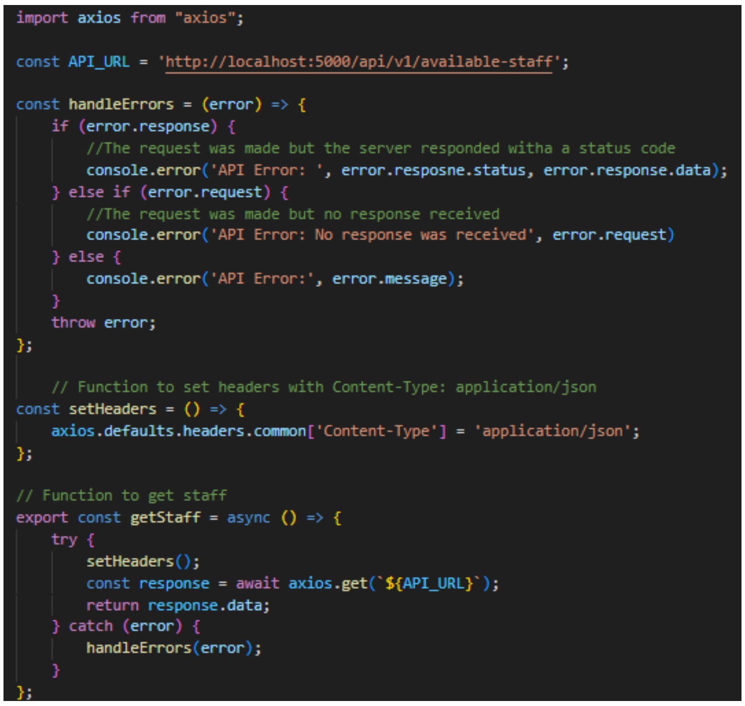
B.2.1 Example of CheckUserAuth component



```javascript
import axios from "axios";

const checkUserAuth = async () => {
    const accessToken = localStorage.getItem('accessToken');

    if(!accessToken){
        console.error('No verification token');
        return false;
    }

    try {
        const response = await axios.post('http://localhost:5000/api/v1/verify-token', {}, {
            headers: {
                'Authorization': `Bearer ${accessToken}`
            }
        });
        return response.data.user;
    } catch (error) {
        console.error('Token verification failed', error);
        return false;
    }
};

export default checkUserAuth;
```

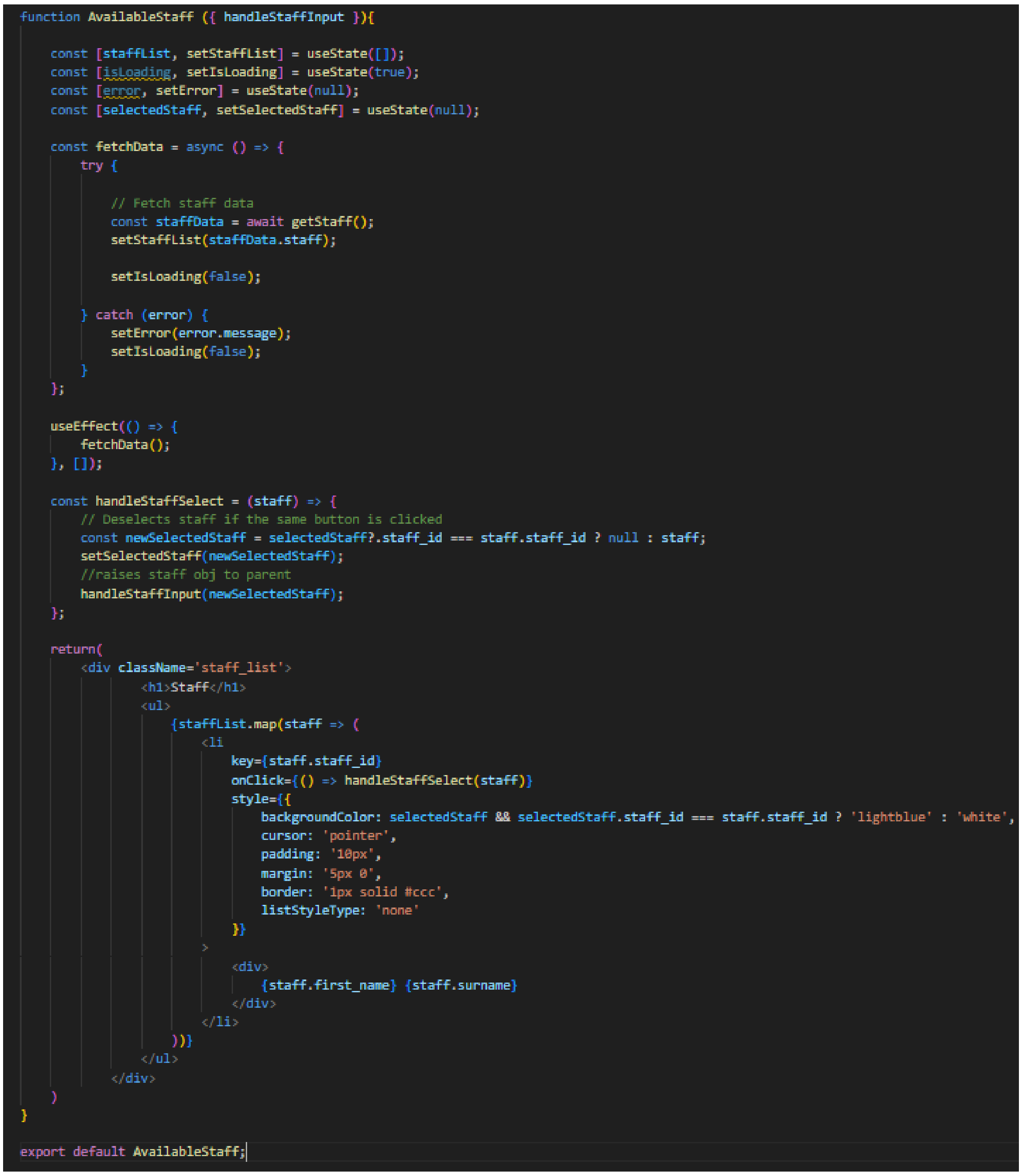
B.3 Available Staff

B.3.1 Example of API file



```javascript
import axios from "axios";

const API_URL = 'http://localhost:5000/api/v1/available-staff';

const handleErrors = (error) => {
    if (error.response) {
        //The request was made but the server responded witha a status code
        console.error('API Error: ', error.resposne.status, error.response.data);
    } else if (error.request) {
        //The request was made but no response received
        console.error('API Error: No response was received', error.request)
    } else {
        console.error('API Error:', error.message);
    }
    throw error;
};

    // Function to set headers with Content-Type: application/json
const setHeaders = () => {
    axios.defaults.headers.common['Content-Type'] = 'application/json';
};

// Function to get staff
export const getStaff = async () => {
    try {
        setHeaders();
        const response = await axios.get(`${API_URL}`);
        return response.data;
    } catch (error) {
        handleErrors(error);
    }
};
```

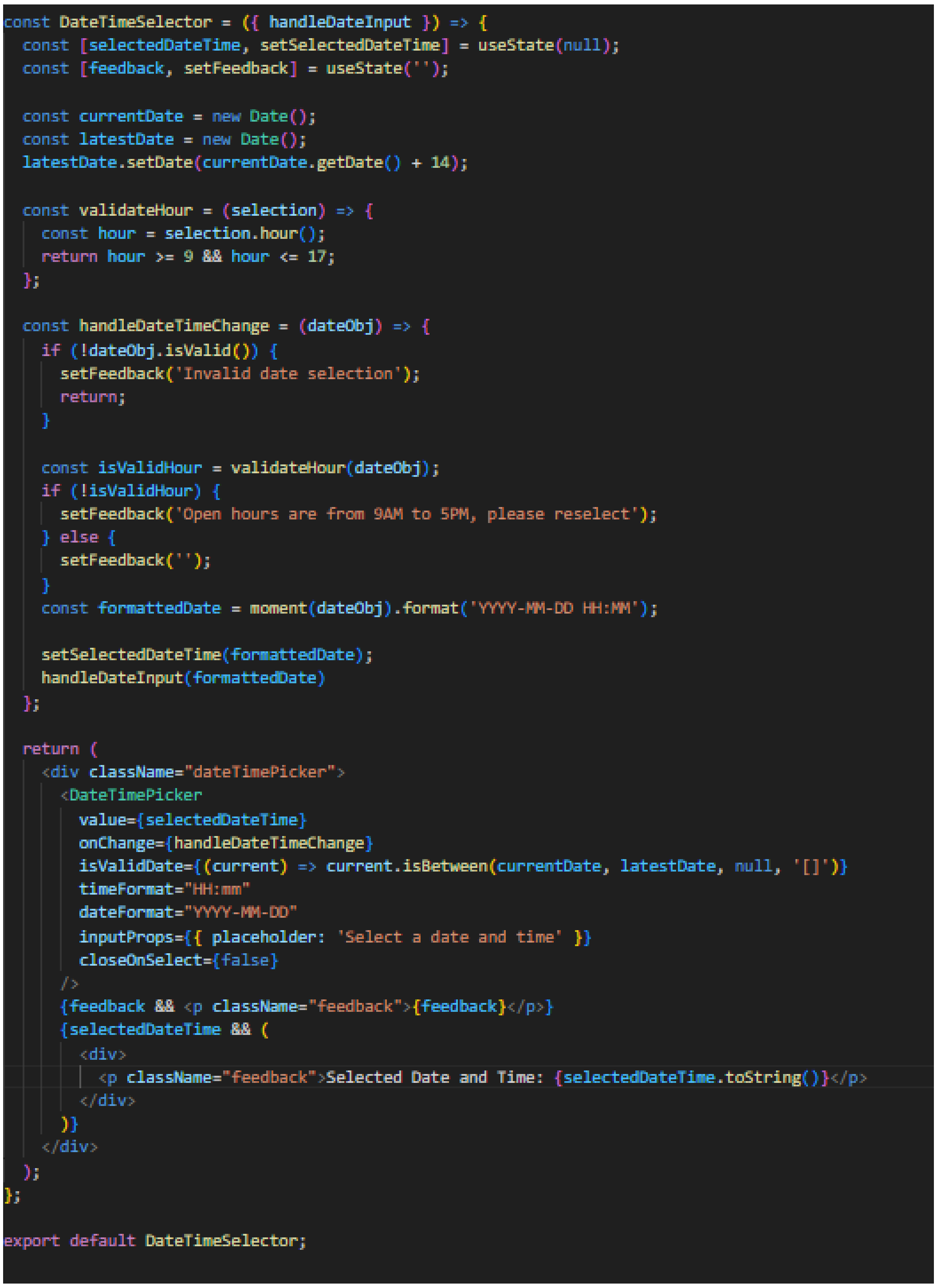
B.3.2 Available Staff Component



```
function AvailableStaff ({ handleStaffInput }){

    const [staffList, setStaffList] = useState([]);
    const [isLoading, setIsLoading] = useState(true);
    const [error, setError] = useState(null);
    const [selectedStaff, setSelectedStaff] = useState(null);

    const fetchData = async () => {
        try {

            // Fetch staff data
            const staffData = await getStaff();
            setStaffList(staffData.staff);

            setIsLoading(false);

        } catch (error) {
            setError(error.message);
            setIsLoading(false);
        }
    };

    useEffect(() => {
        fetchData();
    }, []);

    const handleStaffSelect = (staff) => {
        // Deselects staff if the same button is clicked
        const newSelectedStaff = selectedStaff?.staff_id === staff.staff_id ? null : staff;
        setSelectedStaff(newSelectedStaff);
        //raises staff obj to parent
        handleStaffInput(newSelectedStaff);
    };

    return(
        <div className='staff_list'>
                <h1>Staff</h1>
                <ul>
                    {staffList.map(staff => (
                        <li
                            key={staff.staff_id}
                            onClick={() => handleStaffSelect(staff)}
                            style={{
                                backgroundColor: selectedStaff && selectedStaff.staff_id === staff.staff_id ? 'lightblue' : 'white',
                                cursor: 'pointer',
                                padding: '10px',
                                margin: '5px 0',
                                border: '1px solid #ccc',
                                listStyleType: 'none'
                            }}
                        >
                            <div>
                                {staff.first_name} {staff.surname}
                            </div>
                        </li>
                    ))}
                </ul>
            </div>
    )
}

export default AvailableStaff;
```

B.4 Booking Date Selector

B.4.1 DateTimeSelector Component

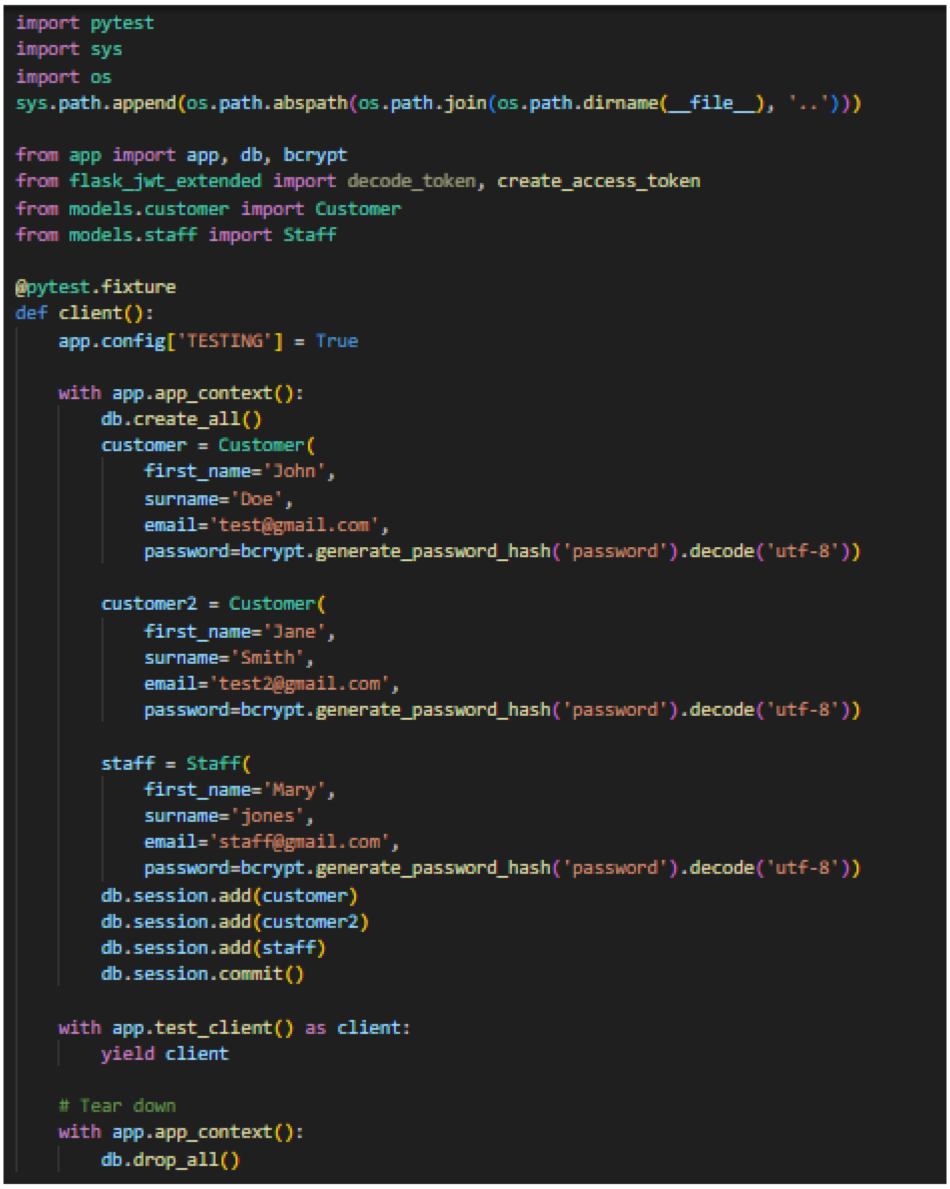


```
const DateTimeSelector = ({ handleDateInput }) => {
  const [selectedDateTime, setSelectedDateTime] = useState(null);
  const [feedback, setFeedback] = useState('');

  const currentDate = new Date();
  const latestDate = new Date();
  latestDate.setDate(currentDate.getDate() + 14);

  const validateHour = (selection) => {
    const hour = selection.hour();
    return hour >= 9 && hour <= 17;
  };

  const handleDateTimeChange = (dateObj) => {
    if (!dateObj.isValid()) {
      setFeedback('Invalid date selection');
      return;
    }

    const isValidHour = validateHour(dateObj);
    if (!isValidHour) {
      setFeedback('Open hours are from 9AM to 5PM, please reselect');
    } else {
      setFeedback('');
    }
    const formattedDate = moment(dateObj).format('YYYY-MM-DD HH:MM');

    setSelectedDateTime(formattedDate);
    handleDateInput(formattedDate)
  };

  return (
    <div className="dateTimePicker">
      <DateTimePicker
        value={selectedDateTime}
        onChange={handleDateTimeChange}
        isValidDate={(current) => current.isBetween(currentDate, latestDate, null, '[]')}
        timeFormat="HH:mm"
        dateFormat="YYYY-MM-DD"
        inputProps={{ placeholder: 'Select a date and time' }}
        closeOnSelect={false}
      />
      {feedback && <p className="feedback">{feedback}</p>}
      {selectedDateTime && (
        <div>
          <p className="feedback">Selected Date and Time: {selectedDateTime.toString()}</p>
        </div>
      )}
    </div>
  );
};

export default DateTimeSelector;
```

C Testing

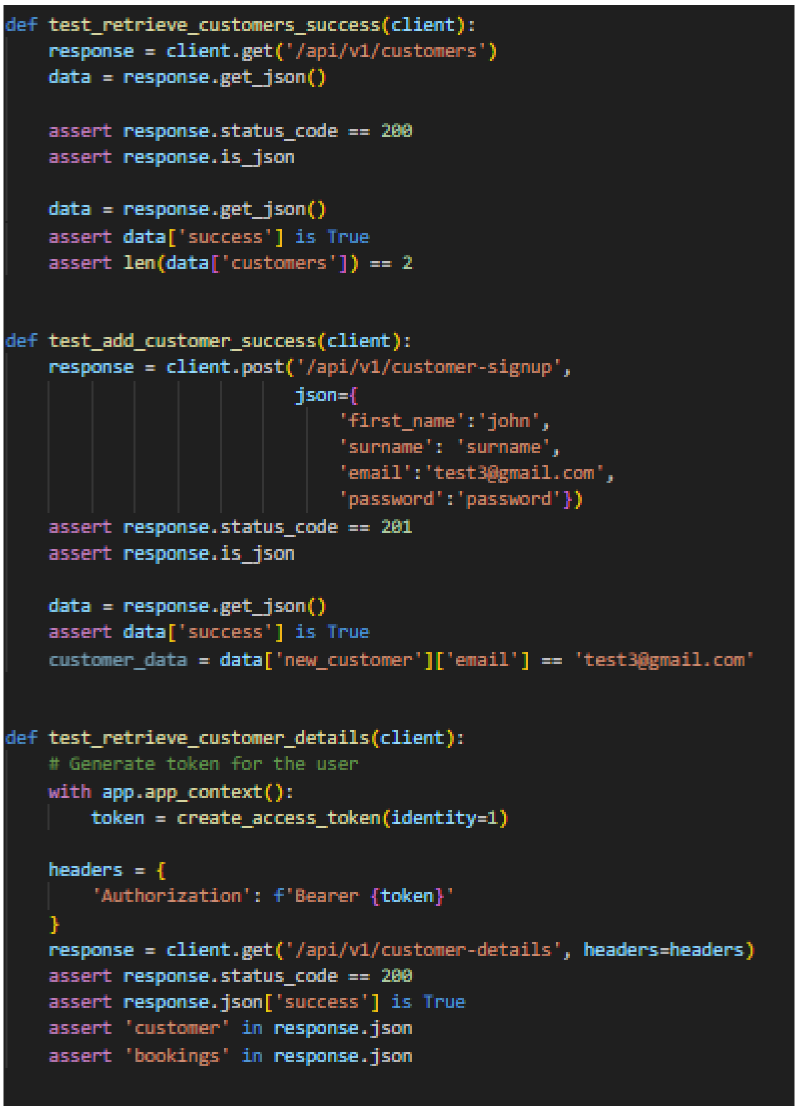
C.1 Unit Testing

C.1.1 Example fixture code



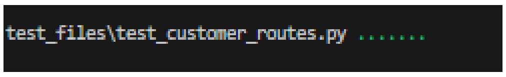
```python
import pytest
import sys
import os
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))

from app import app, db, bcrypt
from flask_jwt_extended import decode_token, create_access_token
from models.customer import Customer
from models.staff import Staff

@pytest.fixture
def client():
    app.config['TESTING'] = True

    with app.app_context():
        db.create_all()
        customer = Customer(
            first_name='John',
            surname='Doe',
            email='test@gmail.com',
            password=bcrypt.generate_password_hash('password').decode('utf-8'))

        customer2 = Customer(
            first_name='Jane',
            surname='Smith',
            email='test2@gmail.com',
            password=bcrypt.generate_password_hash('password').decode('utf-8'))

        staff = Staff(
            first_name='Mary',
            surname='jones',
            email='staff@gmail.com',
            password=bcrypt.generate_password_hash('password').decode('utf-8'))
        db.session.add(customer)
        db.session.add(customer2)
        db.session.add(staff)
        db.session.commit()

    with app.test_client() as client:
        yield client

    # Tear down
    with app.app_context():
        db.drop_all()
```

C.1.2 Example Test Code



```python
def test_retrieve_customers_success(client):
    response = client.get('/api/v1/customers')
    data = response.get_json()

    assert response.status_code == 200
    assert response.is_json

    data = response.get_json()
    assert data['success'] is True
    assert len(data['customers']) == 2


def test_add_customer_success(client):
    response = client.post('/api/v1/customer-signup',
                           json={
                               'first_name':'john',
                               'surname': 'surname',
                               'email':'test3@gmail.com',
                               'password':'password'})
    assert response.status_code == 201
    assert response.is_json

    data = response.get_json()
    assert data['success'] is True
    customer_data = data['new_customer']['email'] == 'test3@gmail.com'


def test_retrieve_customer_details(client):
    # Generate token for the user
    with app.app_context():
        token = create_access_token(identity=1)

    headers = {
        'Authorization': f'Bearer {token}'
    }
    response = client.get('/api/v1/customer-details', headers=headers)
    assert response.status_code == 200
    assert response.json['success'] is True
    assert 'customer' in response.json
    assert 'bookings' in response.json
```

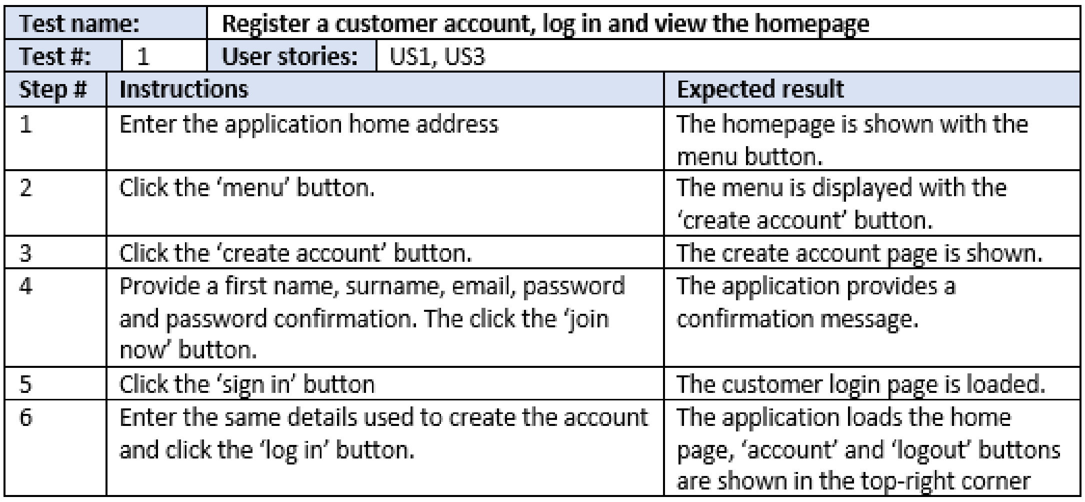
C.1.3 Unit Tests Passing



```
test_files\test_customer_routes.py .......
```

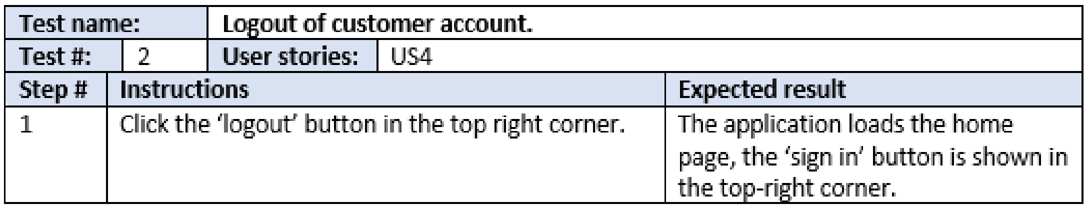
C.2 User Acceptance Testing

C.2.1 UAT 1 – Register, login and view the homepage



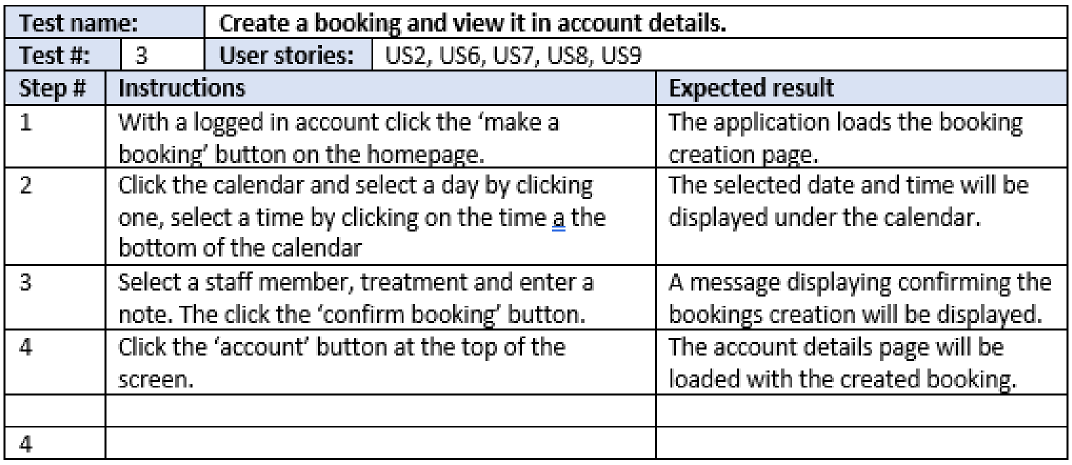
Test name:	**Register a customer account, log in and view the homepage**	
Test #: 1	**User stories:** US1, US3	
Step #	**Instructions**	**Expected result**
1	Enter the application home address	The homepage is shown with the menu button.
2	Click the 'menu' button.	The menu is displayed with the 'create account' button.
3	Click the 'create account' button.	The create account page is shown.
4	Provide a first name, surname, email, password and password confirmation. The click the 'join now' button.	The application provides a confirmation message.
5	Click the 'sign in' button	The customer login page is loaded.
6	Enter the same details used to create the account and click the 'log in' button.	The application loads the home page, 'account' and 'logout' buttons are shown in the top-right corner

C.2.2 UAT2 – Logout



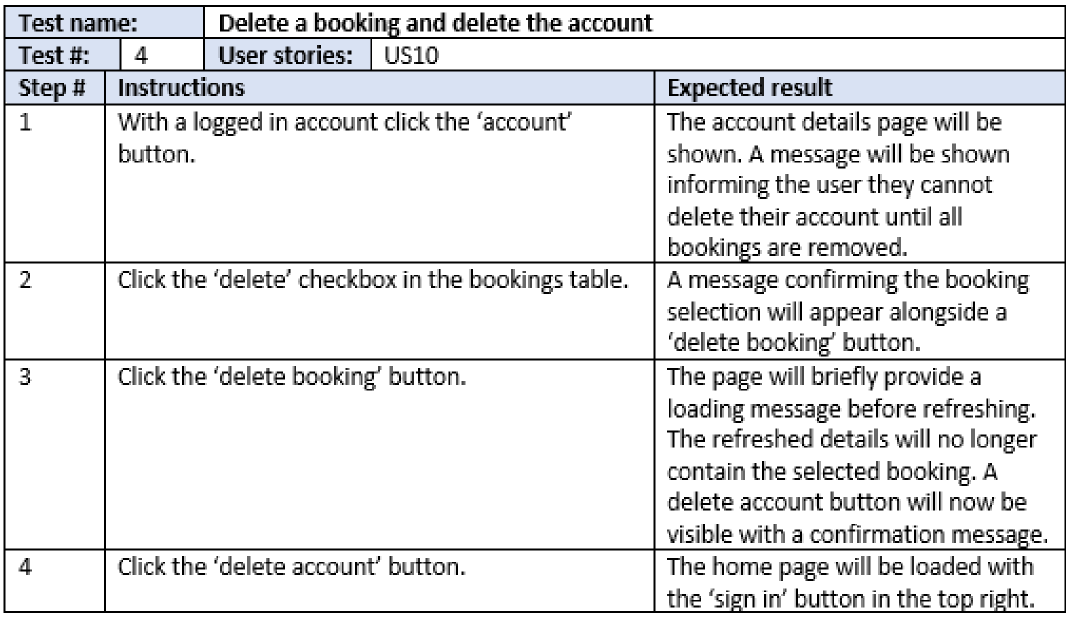
Test name:	**Logout of customer account.**	
Test #: 2	**User stories:** US4	
Step #	**Instructions**	**Expected result**
1	Click the 'logout' button in the top right corner.	The application loads the home page, the 'sign in' button is shown in the top-right corner.

C.2.3 UAT3 – Create a booking and view account details



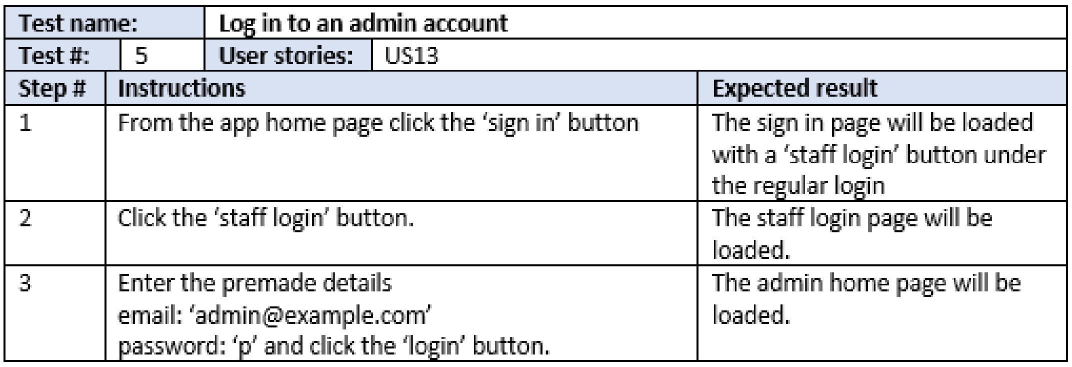
Test name:	**Create a booking and view it in account details.**	
Test #: 3	**User stories:** US2, US6, US7, US8, US9	
Step #	**Instructions**	**Expected result**
1	With a logged in account click the 'make a booking' button on the homepage.	The application loads the booking creation page.
2	Click the calendar and select a day by clicking one, select a time by clicking on the time a the bottom of the calendar	The selected date and time will be displayed under the calendar.
3	Select a staff member, treatment and enter a note. The click the 'confirm booking' button.	A message displaying confirming the bookings creation will be displayed.
4	Click the 'account' button at the top of the screen.	The account details page will be loaded with the created booking.
4		

C.2.4 UAT4 – Delete a booking and delete the account



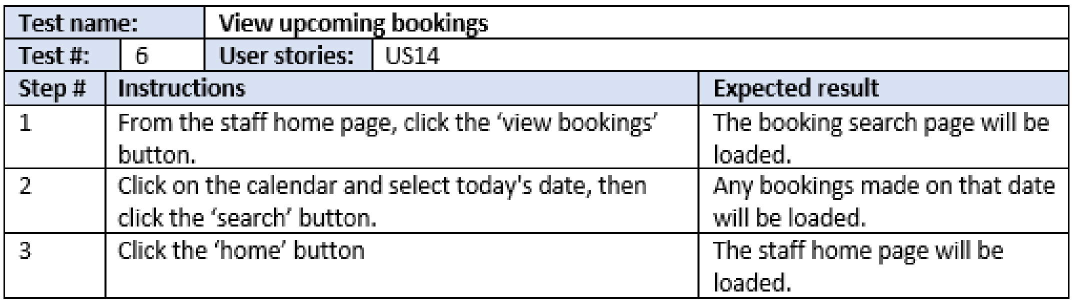
Test name:	**Delete a booking and delete the account**		
Test #:	4	**User stories:**	US10
Step #	**Instructions**	**Expected result**	
1	With a logged in account click the ‘account’ button.	The account details page will be shown. A message will be shown informing the user they cannot delete their account until all bookings are removed.	
2	Click the ‘delete’ checkbox in the bookings table.	A message confirming the booking selection will appear alongside a ‘delete booking’ button.	
3	Click the ‘delete booking’ button.	The page will briefly provide a loading message before refreshing. The refreshed details will no longer contain the selected booking. A delete account button will now be visible with a confirmation message.	
4	Click the ‘delete account’ button.	The home page will be loaded with the ‘sign in’ button in the top right.	

C.2.5 UAT5 – Log into an admin account



Test name:	**Log in to an admin account**		
Test #:	5	**User stories:**	US13
Step #	**Instructions**	**Expected result**	
1	From the app home page click the ‘sign in’ button	The sign in page will be loaded with a ‘staff login’ button under the regular login	
2	Click the ‘staff login’ button.	The staff login page will be loaded.	
3	Enter the premade details email: ‘admin@example.com’ password: ‘p’ and click the ‘login’ button.	The admin home page will be loaded.	

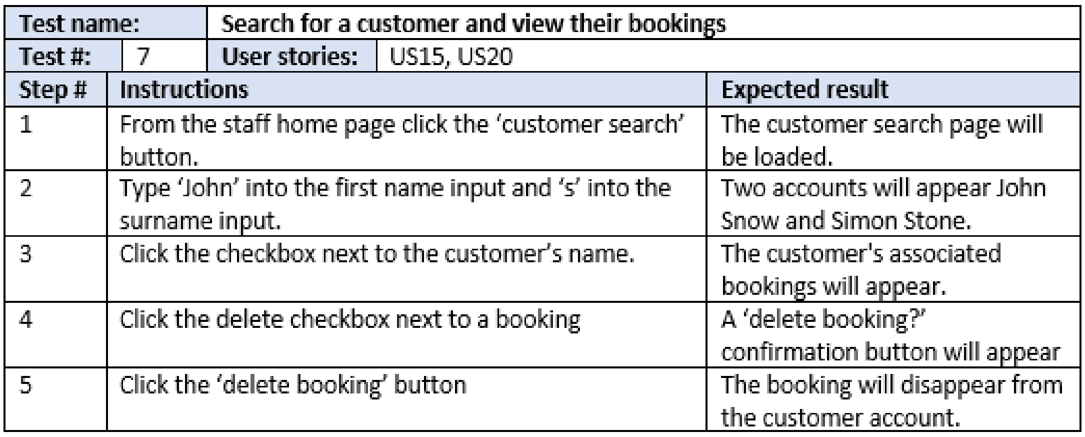
C.2.6 UAT6 – View upcomming bookings



Test name:	View upcoming bookings		
Test #:	6	User stories:	US14

Step #	Instructions	Expected result
1	From the staff home page, click the ‘view bookings’ button.	The booking search page will be loaded.
2	Click on the calendar and select today's date, then click the ‘search’ button.	Any bookings made on that date will be loaded.
3	Click the ‘home’ button	The staff home page will be loaded.

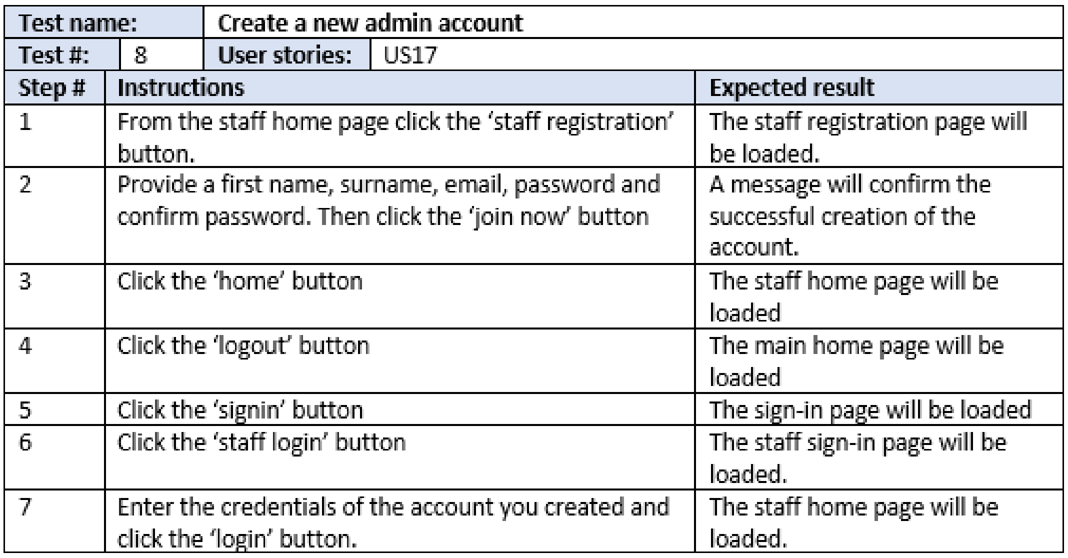
C.2.7 UAT7 Search for a customer and veiw their bookings



Test name:	Search for a customer and view their bookings		
Test #:	7	User stories:	US15, US20

Step #	Instructions	Expected result
1	From the staff home page click the ‘customer search’ button.	The customer search page will be loaded.
2	Type ‘John’ into the first name input and ‘s’ into the surname input.	Two accounts will appear John Snow and Simon Stone.
3	Click the checkbox next to the customer’s name.	The customer's associated bookings will appear.
4	Click the delete checkbox next to a booking	A ‘delete booking?’ confirmation button will appear
5	Click the ‘delete booking’ button	The booking will disappear from the customer account.

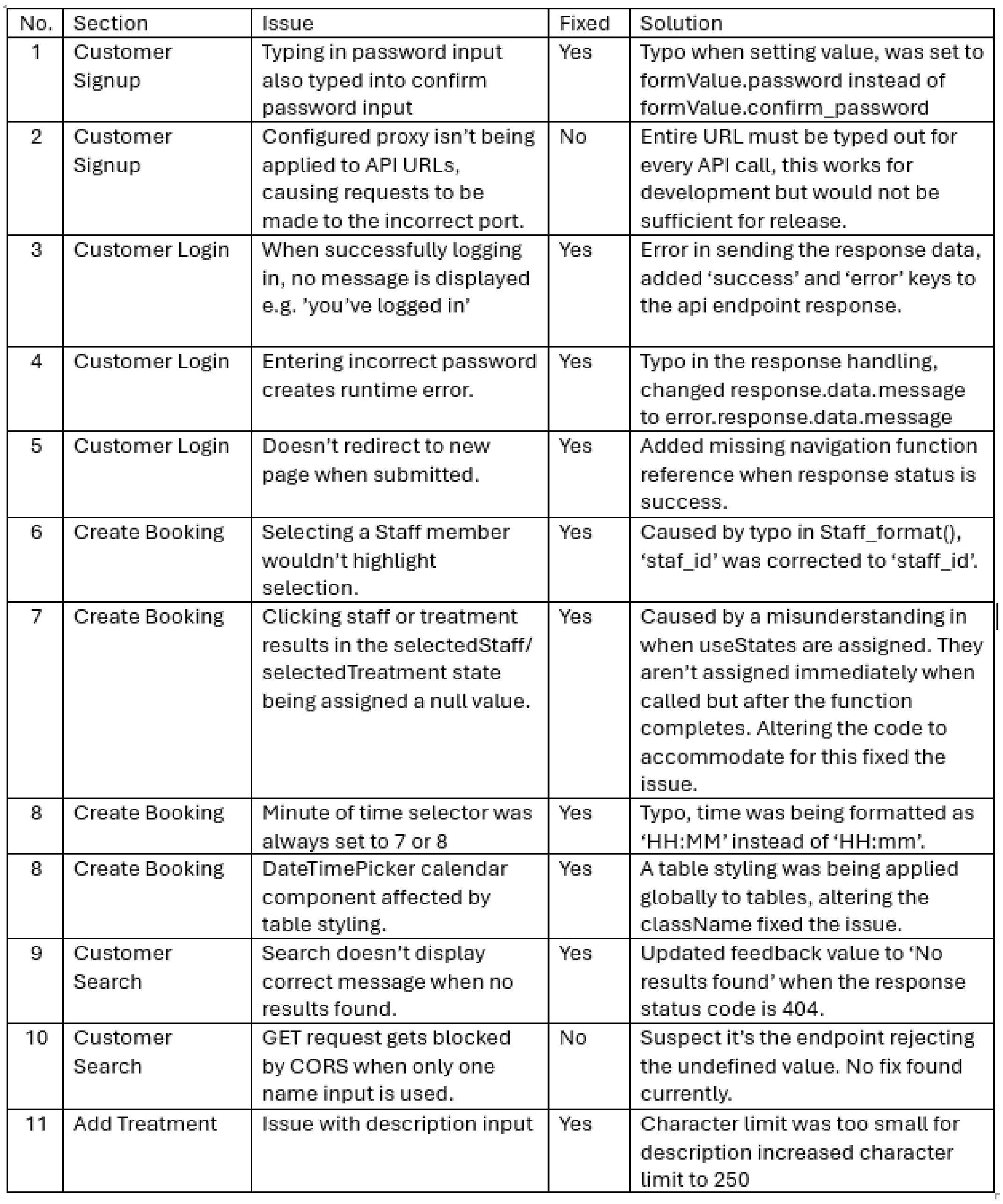
C.2.8 Create a new admin account



Test name:	Create a new admin account		
Test #:	8	User stories:	US17

Step #	Instructions	Expected result
1	From the staff home page click the ‘staff registration’ button.	The staff registration page will be loaded.
2	Provide a first name, surname, email, password and confirm password. Then click the ‘join now’ button	A message will confirm the successful creation of the account.
3	Click the ‘home’ button	The staff home page will be loaded
4	Click the ‘logout’ button	The main home page will be loaded
5	Click the ‘signin’ button	The sign-in page will be loaded
6	Click the ‘staff login’ button	The staff sign-in page will be loaded.
7	Enter the credentials of the account you created and click the ‘login’ button.	The staff home page will be loaded.

C.3 Manual Testing



No.	Section	Issue	Fixed	Solution
1	Customer Signup	Typing in password input also typed into confirm password input	Yes	Typo when setting value, was set to formValue.password instead of formValue.confirm_password
2	Customer Signup	Configured proxy isn't being applied to API URLs, causing requests to be made to the incorrect port.	No	Entire URL must be typed out for every API call, this works for development but would not be sufficient for release.
3	Customer Login	When successfully logging in, no message is displayed e.g. 'you've logged in'	Yes	Error in sending the response data, added 'success' and 'error' keys to the api endpoint response.
4	Customer Login	Entering incorrect password creates runtime error.	Yes	Typo in the response handling, changed response.data.message to error.response.data.message
5	Customer Login	Doesn't redirect to new page when submitted.	Yes	Added missing navigation function reference when response status is success.
6	Create Booking	Selecting a Staff member wouldn't highlight selection.	Yes	Caused by typo in Staff_format(), 'staf_id' was corrected to 'staff_id'.
7	Create Booking	Clicking staff or treatment results in the selectedStaff/ selectedTreatment state being assigned a null value.	Yes	Caused by a misunderstanding in when useStates are assigned. They aren't assigned immediately when called but after the function completes. Altering the code to accommodate for this fixed the issue.
8	Create Booking	Minute of time selector was always set to 7 or 8	Yes	Typo, time was being formatted as 'HH:MM' instead of 'HH:mm'.
8	Create Booking	DateTimePicker calendar component affected by table styling.	Yes	A table styling was being applied globally to tables, altering the className fixed the issue.
9	Customer Search	Search doesn't display correct message when no results found.	Yes	Updated feedback value to 'No results found' when the response status code is 404.
10	Customer Search	GET request gets blocked by CORS when only one name input is used.	No	Suspect it's the endpoint rejecting the undefined value. No fix found currently.
11	Add Treatment	Issue with description input	Yes	Character limit was too small for description increased character limit to 250