School of Computing and Mathematical Sciences

CO7201 Individual Project

Final Report
AN AUTOMATED SYSTEM FOR LOCAL GPS

[SHREEYA DINESH DESAI]

[sdd13@student.le.ac.uk]

[239038636]

Project Supervisor: [DR. YAKUN JU]
Principal Marker: [DR. STANLEY FUNG]

Word Count: [13,410]
[16/05/2025]

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: [Shreeya Dinesh Desai]
Date: [16/05/2025]

Table of Contents

1. Introduction

The National Health Service (NHS) in UK are facing countless problems today, out of which two primary problems are lack of staff and insufficient funds to improve the existing digital infrastructure of the healthcare sector. According to the (Medical staffing in the NHS, 2025) report, a large percentage of General Practitioners / Doctors are resigning from their job due to lack of work life balance and degrading personal health. Therefore, most of workload of the senior staffs is now handed over to beginner level or early age Practitioners which lack experiences.

In the year 2024, approximately 42.19% of the staff experienced workload-stress. While 30.24% of the staff felt burnt out as an outcome of their work. Additionally, the healthcare workers felt unappreciated and unhappy with their salaries (Medical staffing in the NHS, 2025).

Due to these challenges, there was an impact on the services provided to the patients. For instance, “Unfortunately, we don’t have any appointments available for today” is the most common phrase used by the receptionists due to the shortage of availability of GP (MacConnachie, 2024).The General Practitioner and NHS staff are facing numerous challenges such as shortage of staff, limited availability of appointments, adolescents not opting to study in the healthcare industry, lack of skilled Doctors. Due to these issues, patients have been suffering from adequate treatment and care. (Khan, 2023).

1.1 Aim.

This paper mainly aims to develop An Automated System for Local GP’s, which would be less stressful and user friendly for both healthcare staff and Patients. The Web App would help the staff to set their availability smoothly and cancel their scheduled appointment with an ease. Also allowing Patients to manage their appointments, pay for their Prescriptions, all these with a user-friendly interface of the applications for the Admin, Staff and Patients allowing all the users of the web-app to feel supported and less stressed while performing any task on the system. It is essential to understand that this Web App has been developed considering a specific GP.

1.2 Objective.

The objective for this project would be to design a secure and user-friendly web application considering different demography of users, short appointment booking forms to patients with read through articles for minor injuries. Additionally, patients should be able to pay for prescriptions from anywhere in the UK.

On the other hand, the General Practitioner will have a simplified dashboard which would help the doctors to set their availability, view the booked appointment, cancel their availability, view the patient previous medical history and prescribe the medicine.

Lastly, the entire application would be deployed on the Microsoft Azure cloud considering the CAP (Consistency, Availability and Partition Tolerance) theorem.

1.3 Requirements.

1.3.1 Essential Requirements.

Registration – Accomplished

The patient will be able to register themselves to the web-app using secure login credentials.

Registration of Doctors and Nurses will be performed by Admin.

Availability - Accomplished

The Doctor and Nurses will be allowed to set their availability two months prior.

Book appointment - Accomplished

According to the patient's requirement, they can book the appointment with the available Doctor.

Provide prescription - Accomplished

The Doctor will be able to access medical history of patients and provide a digital prescription on the web app.

Admin Dashboard - Accomplished

The Admin Dashboard will allow administrators to add, remove healthcare staff and will also allow administrators to book appointments for aged patients, view the list of patients and the staff.

Staff Dashboard - Accomplished

The Dashboard will help staff to set their availability, view booked appointments, provide prescriptions, view patients medical history and send prescription to pharmacy.

Patient Dashboard - Accomplished

The Dashboard will help patients to book appointment, view prescriptions, upload the prior medical history, previous booked appointments records.

1.3.2 Recommended Requirements.

Deployment on the cloud - Accomplished

Articles for minor injuries & awareness - Accomplished

View prescription - Accomplished

Buy and Pay prescriptions - Accomplished

For the prescribed medicine the patient can buy and pay for the prescription online.

1.3.3 Optional Requirements.

Responsive Web Application - Accomplished

One to one chat - Not Implemented

Due shortage of Doctor/Nurse, if in case there's a follow-up required for a specific patient, or a patient requires immediate attention the chat feature can be leveraged.

Video Consultation - Not Implemented

1.4 Tools and Technology used.

The below mentioned details are the tool and technologies used in this project.

Component	Name	Summary
Database	Azure SQL and Cosmos DB, Blob Storage	Storing and managing the data either in structured or unstructured format.
Backend	Python, Flask, Azure Container Apps	Develop business logic.
Frontend	React JS, HTML, CSS, JS, Azure App service	Design a dynamic and responsive web application.
API	REST	Used to communicate with the database.
Authentication	JWT	Used for Secure authentication (What Is a JWT & How It Works, 2024).
Cloud Deployment	Azure	Hosting the application, considering availability and consistency from CAP principles.
Version control	GitLab	Version controls the project.
IDE	Visual Studio Code	Tool for editing code.
Testing	Manual testing, User Feedback(frontend), Test Cases, Postman (API)	Tests the robustness and security of the application.
Designing	Figma, Sketch (paper & pen), Draw.io	Designed a low-fidelity design, wireframes, ER Diagram, Use case diagram and Class diagram.
Documentation	MS Word	
Operating System	Windows	
Docker Image	Docker	Used to build docker images of the applications.

1.5 Structure Overview.

1. **Introduction**: Provides a brief idea about the project including the aim, objective, requirements and tool and technologies used.

2. **Background Research**: This section tells us about the research that has been conducted during these three months of the dissertation. Moreover, this section is divided into two parts Literature review and existing applications.

3. **System architecture and Design**: This section is divided into two parts: Backend architecture which would have the designs of ER- diagrams, Class diagram and Use-case diagram. Frontend architecture: Display the High-Fidelity designs prototypes and the wireframes plus the core functionalities of the project.

4. **Development process**: This section will give us an idea about the entire development process from the scratch to the integration. The entire system has been tested manually considering different test case scenarios.

5. **Deployment**: This section will speak about the detailed deployment process.

6. **Results**: Learning outcomes, conclusion and Future scope of this project would be discussed in this project.

2. Background Research

This paper involves ample number of background research with reference to problems faced by the General Practitioner and Patients. In this section, questions like how the existing Web-Apps or Mobile applications helped the patients to monitor their health and access GP services, how impactful will the cloud technologies be in the healthcare sector, how the security of user data should be handled with regards to the confidentiality. Research has been conducted with the help of google scholar, IEEE, articles, PubMed, BBC news (for the most updated news about General Practitioner), Oxford academic and Springer.

2.1 Literature Review

Security has been considered as a major part of any Web-App development, primarily for the healthcare sector, where the confidentiality and integrity of the data needs to be maintained. Inorder to maintain the confidentiality and integrity of the data there are various approaches of developing a secure application or software. According to (Shuaibu & Ruqayyat Ahmad Ibrahim, 2017) the three known approaches used even today are Microsoft Security Development Life Cycle (SDLC), Software Security Touchpoint (SST) and Comprehensive Lightweight Application Security Process (CLASP). Since their approaches are different, but the crucial details would always be around developing a secure application. (Refer Appendix Literature 0.A.1: Approaches key point. Image is taken from the literature survey.).

Additional attention should be given to various types of cyber-attacks taking place every second, the goal of these attacks is not just to damage the application servers, but they also intend to steal Users data and sell it on Dark Web. Data stealing is a new norm today, where one can detect or conduct cross-site scripting (XSS) across your web apps and can steal your user data in some hours. The authors (Shuaibu & Ruqayyat Ahmad Ibrahim, 2017) have proposed their own methodology that tries and helps to avoid any sorts of attacks: first was the selection of the framework and designing of the UI for the development, and the second phase was the evaluation. This research paper in specific, exclaims the methodology which the authors seem to rely on. Additionally giving an idea of how important it is to choose appropriate development methodology. (Refer Appendix*Literature 0.C* A.3: Web application Development Model with security Concerns).

In the previous decades, the manual health records system had various cons such as inconsistent data, missing required files, limited storage and misdiagnosed cases. Thanks to the Electronic Healthcare Record system (EHRC) where one could get the patients record fingertip. This technology has helped the doctors to improve the results of any disease accuracy as well as the communication with the patient. But the storage of patient history has always been a concern. Not only due to the size & scale of patient data, but also due to concern around data privacy. Yes, there are various norms and standards around data compliance and protection. But cybercrime still exists and is carried out on dark web even today.

Back in 2015, there was an increase in patients demand for the medical services, where healthcare system might not have been advanced in countries such as Iraq, India, Yemen, Myanmar. A handful of hospitals and private clinics may have computer-based systems that would enable them to maintain data, but there was no mechanism for sharing the information. Inorder to implement an advanced e-healthcare system there are significant components that need to be considered: Personal Health Record (PDR), Medical Health Record (MDR) and Electronic Health Record (EHR).

This research paper (Rasha Talal Hameed, 2015)written by Rasha Talal Hameed informs us about the healthcare system of Iraq and the model that has been designed by them, which is based on the service-orientated architecture and cloud computing. One of the most efficient solutions to tackle the digital problems is with the help of cloud computing. In other words, provision of services over the internet (Mohammad Mehrtak, 2021) Currently, the technologies of the cloud are being used in the healthcare industry in order to maintain the electronic records, which would help the patients to easily access their respective information. However, the cloud's storage, real-time information exchange, infrastructure and operating costs, and security are its most worrisome aspects. On the contrary, perks include scalability, flexibility, speed enhancement, cost reduction, and user cooperation ease (Mohammad Mehrtak, 2021) The proposed system management system makes use of computing with REST (Representational State Transfer), Amazon Relational Database Service (RDS), the cloud, SOA and services for web patients (Refer to Appendix, *Literature 0.F* A.6: Proposed System Architecture..). The use-case diagram for this project has users' doctors, pharmacists, laboratory admins, employees, administration and radiologists, where each of them has various functions (Refer to Appendix, *Literature 0.G* A.7: Use Case Diagram).

The rapid evolvement of technology has supported and will support in solving problems which the modern world faces, however none of this would even be possible without regular feedback and thorough checks of the existing services provided in the healthcare domain. Authors (Nasaruddin & Izzatdin Abdul Aziz, 2018) thought of developing a web-based application where one could provide a feedback based on the service provided by the healthcare provider. The main motive behind this was to gather the feedback from patients and improve the services and facilities provided by the staff and healthcare. For example, an e-commerce website has various type of products where there's a section of feedback that has been provided by the customers which therefore help the other customer to decide on that product. Similarly, if a patient provides feedback for that healthcare staff and facilities provided, this will help the other patients decision making.

The authors (Nasaruddin & Izzatdin Abdul Aziz, 2018) have conducted research on websites and review systems, including Doctor2U E-commerce, Australian digital My Health Record, Rating, review, chat box, and feedback techniques, as well as gaps discovered in related work on the trust factor, which have given them a better understanding of the methodology used to obtain the feedback. Their research paper explains the development of their system considering different Use cases for Patient (Refer Appendix,*Literature 0.E* A.5: Use Case diagram for patient.), Doctor (Refer Appendix, *Literature 0.D* A.4:Use Case Diagram for Doctor Admin.), Admin (Refer Appendix, *Literature 0.B* A.2:Use Case Diagram for Staff Admin.). Furthermore, the testing of their application was done using methodologies like Unit testing and integration testing. Initially, this research paper considers use cases which could be used as reference for this paper of Automated GP System.

2.2 Existing GP Web Applications.

In the United Kingdom, general practitioners have their official website which conveys information to patients regarding the services they provide. The common services provided by most of the websites were registration, articles related to injuries, prescription-related queries, scheduling appointments and many more. During this period of research, multiple GP websites were analysed considering the user interface, user experience and functionalities.

During the process of analysing the website, one of the website's named Queens Road Surgery (http://stpetershealthcentre.com/) system link was not even accessible and displayed an error message (Appendix Existing Websites). This resulted a negative impression towards the GP Service, hence reducing the trust in the services provided by that GP service. On the contrary, most of the websites were accessible, which displayed the landing page. The landing page had both essential and advanced functionalities, but the flaws were around the UI (user interface) and UX (user experience). The UI and UX are an important aspect for the website, as it helps the user to access, navigate and traverse the website with ease.

The website Leicester Holistic GP had an awful user interface, and it was difficult to go through the text (Appendix Existing Websites). This was mainly due to the colour and the typography chosen. Conversely, the website Highfield Surgery displayed a lot of information, which resulted in a packed look (Appendix Existing Websites , **Error! Reference source not found.**).

Furthermore, designing healthcare website must be done considering the key aspects of colour, typography, tag names and not much redirection.

3. System Design & Functionalities.

3.1 Backend

3.1.1 ER-Diagrams.

The entity relationship diagram is a frequently used diagram for structured analysis and conceptual modelling (Song, Mary Evans , & E.K. Park, 1995). Numerous entities, attributes, and relationships between them are used in the ER model (Begum & C. P. Indumathi, 2016). As a foundational step towards the designing, the ER model is the initial step in the project that aids in modelling the database (Begum & C. P. Indumathi, 2016; Daniela Berardi, 2005). The ER model is easy to understand, helps to solve the issue related to the real world, and can be easily translated to a relational database structure (Song, Mary Evans , & E.K. Park, 1995).

To design a self-efficient and automated GP system, it was necessary to develop an ER diagram which not only helps understand essential entities of the system but also will support us during the database modelling phase. Entities like admin, patients, doctors and nurses are the primary tables or objects for the GP system. Furthermore, entities like doctor specialisations, nurse specialisations, doctor availability, nurse availability, medicine, disease, pharmacy, slots and payments are created to support the overall idea of the automated GP system.

The connection / relationship between these entities helps us recognize the foreign key relationships between tables in the database. The relationships also give us a high-level brief idea of how the system would work in a real-world scenario. The ER diagram developed for automated gp system could be referred below.

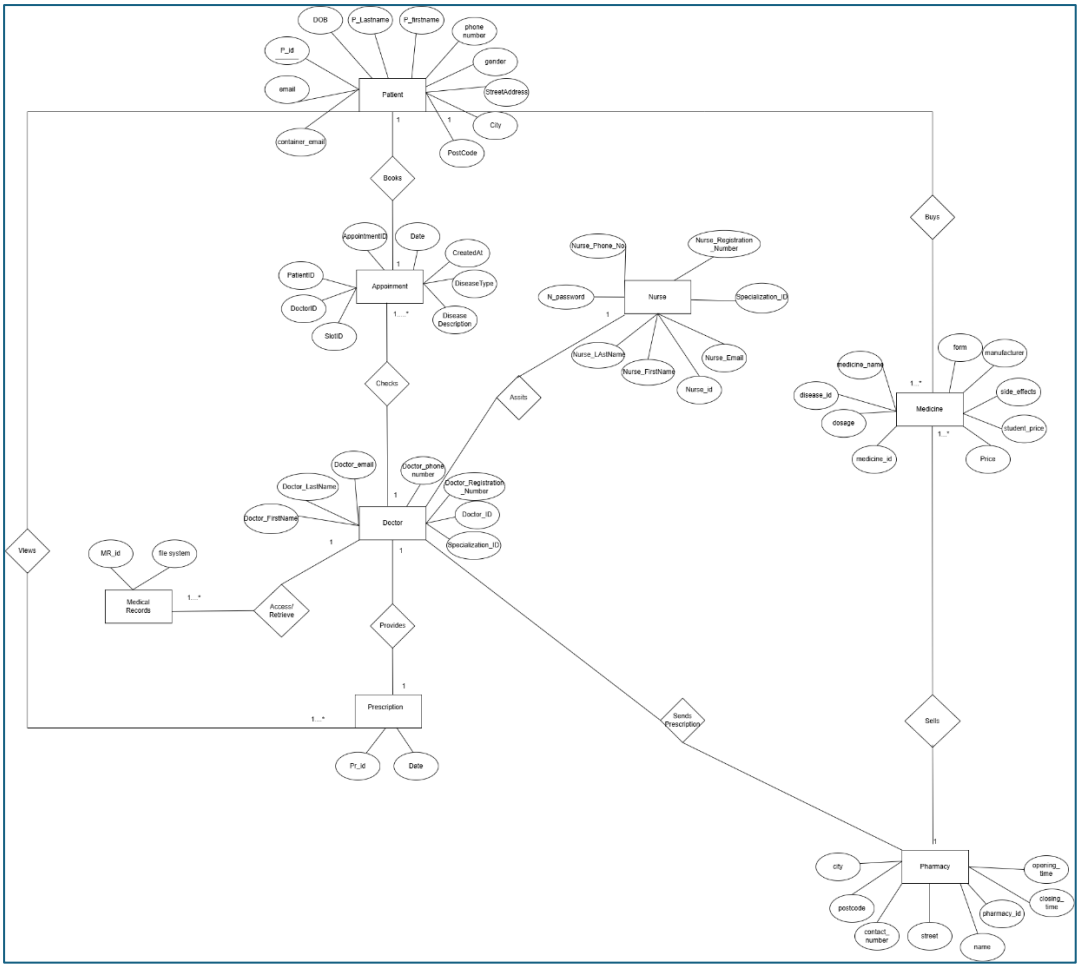


Figure 1: ER Diagram

3.1.2 Class Diagram.

The class diagram basically helps in development process by describes the structure of the system to be designed. A class diagram is a combination of multiple classes that describe the system, and the connections between two classes are defined by relationships((Daniela Berardi, 2005) (Lee, 2004).

Now, considering the automated GP system the most important classes are Patient, Doctor, Nurse and Admin. These classes build the foundation of the entire web app, furthermore attributes and operations allowed by each of these individual classes are depicted in the class diagram provided in the Appendix Diagram N.1: Class Diagram.

For instance, the doctor class is allowed to perform the operations such as cancel appointment, login, profile and delete doctor. For these operations to be carried out successfully, the attributes of the doctor class will be of high importance. A class diagram ideally is the main foundation of the entire GP system, as it was mostly used while database creation and tremendously helped in implementing advanced features like upload medical records and provide prescriptions.

Normalization process for these classes have been performed at the very beginning of the development phase, as the goal was to develop a highly scalable, secure, sophisticated and simple infrastructure for the web-app. Database Normalization is an important aspect while developing any dynamic architecture, this process ensure data integrity and reduces duplicate records to exist in the database.

3.1.3 Use- Case Diagram.

The use case diagram is a diagram which provides high level idea about the system to be designed. Use case diagram are basically leveraged or are developed during the requirement gathering phase, this diagram helps software engineer understand the entire working of the system to be developed (Refer Appendix Diagram N.3: Use Case Diagram., Diagram N.4: Use case Diagram for admin.).

In the context of the paper" The Role of "Roles" in Use Case Diagrams" (Genilloud, 2001), the use case diagram helps to represent the functional requirements of the webapp. The human being like characters is called actors. These actors are basically the actual users of the web-app and anything inside the big rectangle are the features which the web-app will have.

With regards to the Automated GP System, actors for the use-case diagram are admin, doctor, nurse and patient. These actors are basically connected to the features they can use from the automated GP system. For instance, from the designed UI diagram, the actor Doctor can provide prescriptions, set their availability and access medical records.

3.1.4 Service Description.

1. **Users Account**

This section describes essential functionalities related to users, including registration and login. It enables the patient, nurse, doctor and admin to login with their credentials, and according to the user role, which is assigned after login, the user would be navigated to their respective dashboard. Additionally, the admin has the privileges to manage the staff, either to add or delete staff members. This ensures that the users are appropriately managed across the system.

API endpoints

POST/patient/register: Register a patient.
POST/patient/login: Authenticate patient and generates a token.
POST/staff/login: Authenticate staff and generates a token.
POST/staff/registration: Admin Registers staff.
POST/gp-patient/registration: Admin registers the patient.

The associated entity relationship diagram includes tables such as patient, doctor and nurse. The password for each of the users has been encrypted with the help of SHA hashing. The login sessions are being maintained, with the help of JWT tokens. The JWT Tokens are generated in a way that they should expire within an hour, each user is assigned a unique JWT and CSRF token upon login. These tokens are required to access protected routes, if these tokens are missing, the user cannot access any protected route. JWT token has been implemented inorder to reduce the risk of unauthorised access. To maintain integrity and safety, the tokens are generated based on the email of the users, and these tokens are sent to the browser of the user via cookies.

2. **Availability Service**

This service mainly aims to set and cancel the availability set by the staff. With the help of the endpoints mentioned below, the staff (either doctor or nurse) would be able to set their availability. Conversely, due to uncertain circumstances, the staff has the right to cancel the availability. This ensures that the staff can access this service without any difficulties.

API Endpoints

POST/set_doctor_availability : Sets the doctor's availability.
DELETE/cancel_doctor_availability/<int:doctor_id>/<string:date>/<int:slot_id>: Deletes the doctor's availability for a specific date.
POST/set_nurse_availability: Sets the nurse availability.
DELETE /cancel_nurse_availability/<int:nurse_id>/<string:date>/<int:slot_id>: Deletes the nurse availability for a specific date.

The entity relationship diagram associated with this service uses tables named as doctor availability and nurse availability. In this, the availability set by the doctor and nurse would be stored. The doctor's availability will then be displayed to the patient while they are booking the appointment.

3. Appointment Service

This service handles the key functionality of the website, which is appointment booking. The patient books the appointment by selecting the type of disease. According to the disease type selected by the patient, the specialized doctor list would be displayed for the selected date and the slots available will also be displayed. The patient then selects the convenient slot and books the appointment. Additionally, the patient can cancel the appointment if required.

On the doctor’s side, the scheduled appointment could be viewed and cancelled under uncertain circumstances. If the appointment has been cancelled by doctor, an email will be sent to specific patient who has booked that specific appointment. This ensures clear communication and flexibility between the patient and doctor.

API endpoint
GET/ get_diseases: Retrieves the list of disease.
POST/get_doctors_list: Displays the list of doctors.
GET/ get_doctor_availability/<int:doctor_id>/<string:date>: Retrieves the availability of the doctor on a specific date.
POST/book_appointment: Books an appointment with the doctor.
GET/my_appointment: Displays the appointment booked for patient.
DELETE /cancel_appointment/<int:appointment_id>: Delete the booked appointment.
GET/ get_patient_bookings/<string:date>: Retrieves the patient booking on a specific date.
DELETE/cancel_doctor_appointment/<int:appointment_id>: Doctor cancels the appointment.

This service has the most complicated entity relationship diagram that uses tables such as appointment, availability, and slots. These tables are interlinked to each other as they share relationship between them. For instance, the doctor must set their availability with the help of slots. Upon adding the availability, if incase the doctor or nurse wants to cancel the availability, they can cancel their availability only if no patient has booked appointment for the same date. The available slots are then displayed to the patient inorder to book the appointment. This process needs to be handled with utmost care, since the details displayed on the patient and doctor dashboard need to be accurate and up to date, without any misinformation.

4. Medical Records Service

This service aims to mainly upload and view the medical-related documents like X-Rays, Prescriptions and other documents. The patient will be able to upload the medical records, and the same records could be viewed by the doctor as and when needed.

API Endpoints
POST/upload/<patient_id>: This endpoint helps the patient to upload the documents.
GET/patient/files: The patient is able to view uploaded files.
POST/patient/medical records: The doctor only has the right to access medical records of the patient.

5. Prescription Service

This service helps the doctor to provide digital prescription to the patient. The doctor verifies the patient, then prescribes the medicine to the patient which includes fields such as medicine name, dosage and instruction. The doctor then verifies if the patient is student or general inorder to apply any applicable discount on the medicine. Furthermore, the doctor sends the prescription to the pharmacy and selects the type of delivery based on patients' preference.

On the patient dashboard, the system helps the patient to view the prescription provided by the doctor including all the details. If incase, the patients have been provided a prescription, he/she gets an option to buy the prescription, where the payment is done online and the medicine is delivered to their home or could also be collected from the pharmacy they mentioned to doctor while the medicine was prescribed.

API endpoint.
GET/prescription: Receives the prescription provided by the doctor.
GET/medicines: Retrieves the medicine.
GET/pharmacies: Retrieves the pharmacies.
GET/patients/verify: Verifies the patient.
POST/providePrescription: Doctor provides the prescription to the patient.

The associated entities involved in this service are the pharmacy, patient and payments. The backend uses the structed SQL to store the data of the pharmacy, patients and payments. However, the prescription data is being stored in NOSQL, specifically the Azure Cosmos DB. For instance, the doctor prescribes the medicine to the patient and sends it to the pharmacy.

The entities involved in this service are the doctor, patient, payment, medicine, pharmacy making the structing of the database much more complicated.

Additionally, this service assures secure payment using STRIPE and seamless transfer of prescription between the patient and pharmacy, making this as the crucial functionalities of the system.

3.2 Frontend.

3.2.1 UI Design (Wireframes and High-Fidelity Prototype).

Upon successful completion of the necessary diagrams like ER, Class and Use case diagram in previous section. This was the immediate next section which was considered while the development process. This section of UI design mainly exclaims the process involved in designing an interactive user interface.

The process begins with drawing the basic wireframes, where the layouts of the web app are designed with the help of pen and paper. The main objective behind designing the wireframe were to identify and understand the placement of components like Navigation bar, left Navbar panel and the main body. Additionally, there was a need to design essential features of the web app with in-depth knowledge and background research of the user demography, user journey mapping, especially for features like appointment form, upload medical history forms and view prescription table.

Followed by the wireframes were the high-fidelity prototypes, basically high-fidelity prototypes are an iteration over the wireframes. In simple terms, High fidelity means the look and feel of the final version of the website is designed in detail, considering the layout, image, font size and even typography and presented in the form of prototypes (Staff, 2024) (Pernice, 2016). The high-fidelity prototypes are designed with the help of Figma, the designed prototype helped while designing the actual UI of the web-app (Refer Appendix Diagram N.5: High Fidelity Prototype 1.,Diagram N.6: High Fidelity Prototype 2.,Diagram N.7: High Fidelity Prototype 3.).

3.2.2 Core Functionalities of UI.

Before diving into the challenges faced, it is necessary to outline the core functionalities of this web-app. As mentioned prior, this Web App mainly focuses to improve the user experience and provide a user-friendly interface for the staff, admin and patient.

The following are the three key dashboards developed along with their respective function in this project:

Patient dashboard: Designing the dashboard specially for the patient was a challenging task. However, with the help from the high-fidelity prototypes the developed dashboard provides features such as appointment booking, managing medical records and purchasing prescriptions.

Admin Dashboard: This dashboard was another crucial aspect of UI design, as the admin is allowed to add, view and remove the patient and staff. Admins are also allowed to book the appointment for the old-aged patients.

Staff Dashboard: This developed dashboard would be accessed by the doctor as well as the nurse staff of the specific GP, wherein the nurse are only allowed to set their availability. On the contrary, the doctor is allowed to set availability, cancel availability, view the previous medical history of the patients and provide prescriptions to the patients.

The Table 1, Table 2, Table 3 and Table 4 details design description for each of the dashboards s

Screen	Design
Login Screen (Refer Appendix thehealme P.1: Landing/Login page., thehealme P.3: Staff login Page.)	Displays an input field along with icons for email addresses and passwords. The login button is implemented to submit the form and navigate to the dashboard. Further, the staff login button and register button are displayed.
Register Screen (Refer Appendix thehealme P.2: Register Page.)	Displays input fields along with icons for first name, last name, email address, date of birth, gender, address, password and contact number. The register button is implemented to submit the form.
Patient Dashboard (Refer Appendix thehealme P.4: Patient Dashboard.)	Displays a logout button at the top navigation bar. The booking appointment, prescriptions, medical records and Profile are buttons displayed on the left side/navigation panel. By default, the dashboard displays the appointment details that are already scheduled and a cancel button.
Staff Dashboard (ReferAppendixthehealme P.8: Staff Dashboard (Doctor), thehealme P.13: Staff Dashboard (Nurse))	Displays a logout button at the top navigation bar. The set availability, get patient bookings, prescriptions, medical records and Profile are buttons displayed on the left side/navigation panel.
Admin Dashboard (Refer Appendix thehealme P.15: Admin dashboard)	Displays a logout button at the top navigation bar. The Add Patient, Add Staff, Book Appointment, Patient List and Staff List buttons are displayed on the left side/navigation panel.
Profile (ReferAppendixthehealme P.7: Profile(Patient).thehealme P.12: Profile (doctor).thehealme P.14: Profile(Nurse))	Displays the details of specific user.

Table 1: Essential Screens along with brief design description.

Patient Dashboard	
Screen	**Design**
Booking Appointment Screen (Refer Appendix thehealme P.4: Patient Dashboard.)	Form 1: Displays an input field for disease description and selection of disease from a drop-down. Form 2: Displays a calendar to select the date. Form 3: Displays a list of available doctors for the selected date. Form 4: Displays the available time slot of the doctor.
Medical Records Screen	Displays a drop-down where the type of file needs to be selected. The Select File button helps to select the file

(Refer Appendix thehealme P.6: Uplaod Medical Histroy Screen(Patient).)	from the device/system and upload the same with an Upload button.
Prescription Screen (Refer Appendix thehealme P.5: Prescriptions Screen (Patient).)	Displays the prescribed medicine along with details inside the React Card visual. The details displayed include the medicine name, quantity, instructions, pharmacy name, collection method and total cost. The buy now button is displayed along with the total cost.

Table 2: Patient Dashboard screens along with brief design description.

Staff Dashboard	
Screen	**Design**
Set Availability Screen (Refer Appendix thehealme P.8: Staff Dashboard (Doctor).)	Display the Select Date button along with the time slots and the option Select all. The already-available set, along with a cancel button, is displayed.
Get Patient Booking Screen (Refer Appendix thehealme P.9: Get Patient booking (Doctor))	Displays the select date button. Upon selecting, it displays the details of patients booked along with a cancel button.
Prescription Screen (Refer Appendix thehealme P.10: Provide Prescription.)	Form 1: Displays the verify patient form with input fields. Form 2: Displays the details of the patient upon clicking the verify button along with the provide prescription button. Form 3: Displays the header named Provide Prescription along with the patient's essential details. Further, the select medicine, quantity, instruction, patient type and collection method fields are displayed along with the add medicine and pharmacy buttons. Form 4: Display the Add Pharmacy form along with fields such as select pharmacy, street, city, and postcode. Further, the back button and send to pharmacy button are displayed.
Medical records Screen (Refer Appendix thehealme P.11: View Medical Records.)	Form 1: Displays the verify patient form with input fields and a verify button. Form 2: Displays the list of files along with name and link.

Table 3: Staff Dashboard screen along with brief design description

Admin Dashboard	
Screen	**Design**
Add Patient Screen (Refer Appendix thehealme P.16: Add Patient.)	Displays an input field such as first name, last name, date of birth, gender, password, contact number, email address, street address, postcode and city, along with an Add Patient button.
Add Staff Screen (Refer Appendix thehealme P.17: Add Staff.)	Displays input fields such as staff type, first name, last name , date of birth, password, contact number, email address, registration number and specialisation along with the Add Staff Button.

Get Patients List (Refer Appendix thehealme P.19: Patient List.)	Displays a search bar. Displays the list of patients registered along with view details and remove buttons.
Get Staff List (Refer Appendix thehealme P.20: Staff List(Doctor), thehealme P.21: Staff List(Nurse))	Display a search field and select a role field.
Booking Appointment Screen (Refer Appendix thehealme P.18: Book Appointments.)	Displays the same form as one for Patient Dashboard, with only one additional input field in form 1 that's Select the patient id.

Table 4: Admin dashboard screens along with brief design descriptions.

3.3 Design challenges.

During the design phase of this project, there were several challenges encountered that required thoughtful consideration and evaluation. The challenges faced were both technical and non-technical.

Starting with the technical design challenges, Entity relationship diagram was the first step towards designing the database model. It was necessary to design database model first, as database first approach was considered for overall app development. Designing an ER diagram which caters to implement health sector app was a tedious task and implementing the same model in the SQL database was another hurdle to overcome.

Followed by database modelling, the second most time consuming and challenging task faced was to implement and use cookie based JWT and CSRF tokens. In previous projects, I had familiarity of developing secure applications in JWT and storing JWT in local storage. However, for this paper cookie based JWT token implementation was undertaken. After a few hurdles, this security measure was successfully tested and implemented.

Finally, the task which challenged the implementation of the Automated GP system was CORS, CORS stands for cross origin resource sharing. This feature is basically implemented to secure your application from unwanted request from unknown origins. This functionality helps web app to be more secure. After spending more than two weeks, challenges around CORS were resolved, and the developed web app has CORS features built in it.

Non – technical challenges faced were around the designing of the individual pages of admin, staff (doctor and nurse) and the patient. The major point to be considered while designing the application was the look and feel of the application and the designing the User interface such that it should be simple yet professionally appealing as different age group people would be accessing the website. Also choosing the colour combination and typography was also a tricky part. The colours do have their own psychology as calm, emergency, growth, nature, trust and many more.

Since the website was being designed for medical purpose, the colour psychology plays an important role, and the chosen colour were blue, white and off-white. These colours were opted as they resembled calmness and trust.

Additionally, easy navigation through the website with the required information was a key feature to be provided to the users of the web-app.

4. Development Process.

4.1 Database Development

The development process was initiated with database development. The sole reason behind adoption of this methodology was around the database first approach considered for the development of business logic of the web-app. During database development, the majority of informative support was taken from the ER and class diagrams designed in the initial requirements gathering phase.

Initially there was only a vision to use Structured Relational databases. However, after understanding that the developed system would be read-heavy, thoughts were given to indulge Non-Relational database as well. Furthermore, to implement advanced feature of medical records storing BLOB Storage mechanism was also configured to be used.

Tool like SQL SERVER a relational database specifically used for Development environments was leveraged to store the data and edited using SSMS (rwestMSFT, 2025). Specifically speaking about the database structure, only on Database 'GP' was configured which furthermore consisted of multiple tables. List of tables within the GP database is as mentioned below:

Followed by Structured Database, the Non-Relation Database was implemented using Azure Cosmos DB Emulator a NoSQL database. Primarily used to store the prescription details provided by the doctors to individual patients. The prescriptions were stored in the Cosmos DB in the form of partitions. The key column for Partitioning the Cosmos DB was the patient Id column. Using the patient Id column, prescriptions assigned to that patient in the past could easily be retrieved, as the prescriptions are stored based on the patient Id. Hence, it was understood during the designing that this feature of the web-app needs to be read-heavy as well write-efficient contributed the idea to opt for Cosmos DB.

Finally, highlighting the last but not the least storage or database implementation the BLOB storage. After going through various articles, research papers and discussion with senior alumni the decision to implement medical record feature seemed feasible. The thorough research through Microsoft's documentation and additional articles, the medical record feature was implemented with the help of Azure Storage account. A in-cloud service to store files, blobs and much more. The configuration was not the hard part, as there was only a need to place the connection string into the env file of Backend API. The challenge faced was across segregating the files as X-rays, Reports and Prescriptions. The business logic to implement a simple but effective file structure for each patient is discussed in the backend development section.

4.2 Backend Development

The backend was developed in Flask API, Python, ensuring maintainability and scalability. The services were designed as a self-standing endpoint, having its associated serviceability and business logic.

The database-first approach was employed where designing the database for each of the entity was done, followed by the endpoints. Each of the endpoints were built according to the features and later were tested in Postman to confirm the desired output.

The folder structure (Figure 2: Folder structure.) of the backend was organized considering the best practices, ensuring scalability, maintainability, understandability, and readability of the application. Under the application folder, sub-folders were created for models, routes, and utils. The model's folder holds the admin.py file which basically is used to register new admin, the routes folder was subdivided into patient routes and staff routes, each of them having their respective endpoints and functionalities, and lastly, the utils had the helper functions.

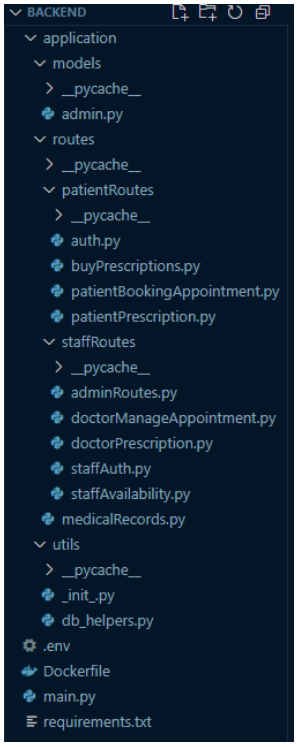


Figure 2: Folder structure.

Additionally, a Docker file, environment file, and main file were also created, which would handle the configuration, development and deployment of the application. In order ensure security and handle cross-origin request, CORS and JWT tokens have been implemented which ensures that data is allowed only from the authorized domain.

4.2.1 Patient Routes/ API endpoints.

This section provides thorough information about the endpoints that are used for the functionalities of the patient, which are authentication, booking appointments, prescriptions, and buying prescriptions.

4.2.1.1 Authentication (auth.py)

For the Authentication module to be implemented, which must only allow access to authorised user. The patient register endpoint (Figure 3) is developed which is a POST method that includes the required parameters. In case any of the fields missing in the post request, a 400 Bad request error message would be sent in response to this API call.

This API endpoint also checks in the database if the patient is already registered based on the email or phone number. Accordingly displays another error message stating that user already exist. In this module itself, when a patient is registered successfully a blob storage is assigned to the respective patient. This blob storage has been used to upload the medical records of patients.

```
# API endpoint to Register patient
@auth_bp.route('/patient/register', methods=['POST'])
def register_patient():
    data = request.get_json()
    p_first_name = data.get('firstName')
    p_last_name = data.get('lastName')
    dob = data.get('dob')
    gender = data.get('gender')
    email = data.get('email')
    phone = data.get('phone')
    password = data.get('password')
    street_address = data.get('streetAddress')
    city = data.get('city')
    postcode = data.get('postcode')
```

Figure 3: Patient Register Endpoint.

The patient login endpoint (Figure 4) is a POST method that validates the entered credentials against SQL database. In case any of the fields are missing or invalid credentials are entered, error messages 400 and 401 are sent in response. Upon successful login, the cookies and refresh cookies are produced, wherein JWT and CSRF tokens are stored in these cookies and are sent to the user's browser.

```
# API endpoint for Patient login
@auth_bp.route('/patient/login', methods=['POST'])
def patient_login():
    data = request.get_json()
    email = data.get('email')
    password = data.get('password')
```

Figure 4: Patient Login endpoint

The patient profile API (Figure 5) is implemented to get the current patients details which is a protected route with a JWT token. This endpoint is a GET method where the patient details are retrieved from the patient email ID and maps the details of the patient from the database to the dictionary. If, in case the patient is not found with the email ID, it responds with an error message.

```
#API Endpoint for Patient profile
@auth_bp.route('/patientProfile', methods=['GET'])
@jwt_required()
def get_patient_profile():
    userEmail = get_jwt_identity()
```

Figure 5: Patient Profile.

Additionally, for all the endpoints, if there's any database connection failure or any exception error 500 error message is displayed.

The utility functions(Function C.1: Container Access.) used in this section are the password hashing function done in the SHA256 algorithm, get database connection from db helpers and the blob storage access rights provision is done using the utility functions, making the code more modular.

4.2.1.2 Book Appointments

This module will basically allow patients to book appointments with an ease. To implement this, feature a set of APIs are used in sequence to allow patient to have a user-friendly experience(Refer Appendix N.2 for detailed flowchart). All the APIs which will be used are protected routes hence they will be secured with a JWT token required flag.

Starting with the get disease endpoint (Refer Appendix Code snippet D.1: Get Diseases.), which is a GET method used to retrieve the list of disease from which patients can select their disease.

The get doctor list endpoint (Refer Appendix Code snippet D.2: Get Doctor List)is the next API call, which is a POST method, where a list of specialized doctors for the selected disease is provided when the post methods is called. In this endpoint the slots table, appointment table, availability table and disease table are accessed using the SQL query, which in response returns with list of doctors.

Followed by the get doctor's availability endpoint (Refer Appendix Code snippet D.3: Get doctors availability) which is a GET method. This endpoint executes the availability of the selected doctor for a particular date. In other words, the endpoint displays the slots available for the doctor for the specific date.

The book appointment (Refer Appendix Code snippet D.4: Book Appointments.) is a post method mainly used to schedule the appointment for the patient. The endpoint fetches the details sent by the patient and checks if the slot selected by the patient is already booked. If booked, it responds with an error message Slot is already booked. If the slot is available, the appointment is booked for that specific patient for the decided datetime and slot with the respective doctor.

Finally, my_appointment endpoint (Refer Appendix Code snippet D.5: My Appointments.)is a GET method that will be used by patient to view the details of the appointment like the appointment ID, doctor's name, specialization ID, date, time, and status.

The delete appointment endpoint (Refer Appendix Code snippet D.6: Delete Appointments.) manages to delete the appointment based on the appointment ID. This endpoint can only be used by patients to cancel their appointment.

4.2.1.3 Prescriptions

The get prescription (Figure 6)is a GET method that fetches the prescription of a specific patient. For this endpoint, the patient ID has been passed as an argument, and the endpoint is secured with JWT authentication, which ensures that only the authorized users can access the prescribed medicine.

The endpoint retrieves the prescription data from the Azure Cosmos DB, which is a NoSQL database. To retrieve the data from Cosmos DB, the setup of the database includes the Cosmos endpoint, Cosmos key, database name, and container name. In the database, the prescriptions are segregated based on the patient id.

```python
@prescription_bp.route('/prescriptions', methods=['GET'])
@jwt_required()
def get_all_prescriptions():
    patient_id = request.args.get('patientId')
    if not patient_id:
        return jsonify({'message': 'Patient ID is required'}), 400

    try:
        query = f'SELECT * FROM c WHERE c.patientid = "{patient_id}"'
        items = list(container.query_items(query=query, enable_cross_partition_query=True))
        return jsonify(items), 200


    except exceptions.CosmosHttpResponseError as e:
        return jsonify({'error': f'Cosmos DB Error: {str(e)}'}), 500
    except Exception as e:
        return jsonify({'error': f'An unexpected error occurred: {str(e)}'}), 500
```

Figure 6: Prescriptions Endpoint.

4.2.1.4 Buy Prescriptions

The buying of prescriptions was one of the crucial endpoints, as it included the Stripe integration. Firstly, the payment sessions (create-payment-session) (Refer Appendix Code snippet E.2: Create Payment Session., Code snippet E.3: Create payment session continuation.) endpoint checked whether the patient had done the payment or not with the help payment table. If the payment was not done, a Stripe checkout session was created, including the total amount. By default, session ID (stripe_payment_intent_id) and payment status as ‘pending’ are stored in the payment table.

Upon successful payment, Stripe provides secure webhooks (Refer Appendix Code snippet E.1: Stripe Webhooks)that confirm the payment status. If the payment is successful, the webhook sends a message that includes the session ID, prescription ID, stripe_payment_intent_id and updates the payment status to ‘succeeded’. The update payment status (Refer Appendix Function C.2: Update Payment Session.) is a helper function that updates the payment table when the webhook sends the response; until that time, the payment status is displayed as pending. This ensures that the transactions made are secure using the JWT authentication and the Stripe signatures.

4.2.2 Staff Routes/ API endpoints.

This section outlines the endpoints developed for the staff functionalities, including registration of staff and patients, prescribing medicine prescriptions, availability setting, and managing the booking appointment.

4.2.2.1 Admin Routes

The admin registers the GP staff which are nurse and doctor with the help of the staff registration endpoint (Refer Appendix Code Snippet F.4: Staff Registration.). In this process, the admin ID is first verified. Upon successful verification of the admin, the helper functions are invoked: register_doctor and register_nurse (Refer Appendix Function C.3:

Register doctor and Register nurse.). The admin then sends the details of the staff to the database and stores it. On the contrary, if the admin ID is missing or invalid, the backend API for staff and patient registrations will not be allowed or executed.

Moreover, the admin has the right to register the patients (Refer Appendix Code Snippet F.5: Patient Registration.). During this process, the register patient API is being called. Like the staff registration endpoint, the admin is verified based on the admin ID. Upon successful validation, the admin enters the data and sends it to the database. However, if the admin ID is not verified or validated, the backend API will not be executed. While registering Patients an additional feature has been implemented with Azure blob storage library. This library is basically used during registering a new patient. This library of Azure mainly creates a unique blob container for every patient. Wherein the patient can upload their medical records in the blob container for doctor's use.

The API endpoints getPatient(Refer Appendix Code Snippet F.3: Get patients.), getDoctor(Refer Appendix Code Snippet F.2: Get Doctors), and getNurse(Refer Appendix Code Snippet F.1: Get Nurses) are developed to retrieve a list of patients, doctors, and staff. These endpoints could only be accessed by the admin when the admin email had been verified. Upon successful verification, the details are being fetched from the respective database: Get patients from the patient table, get doctors from the doctors table, and get nurses from the nurse table.

Furthermore, the admin has the authority to delete the patient and staff. During this stage, the admin is verified based on the email address, and the endpoints involved in this are deletePatient(Code snippet 2), deleteNurse (Code snippet 3), and deleteDoctor(Code snippet 1). The deletion logic is carried out based on their respective email addresses. Additionally, if the email address doesn't exist, the status code 404 which stands for Not found is responded.

```python
@adminRoutes_bp.route('/deleteDoctor', methods=['DELETE'])
@jwt_required()
def delete_doctor():
    admin_email = get_jwt_identity()
```

Code snippet 1 delete doctor API

```python
@adminRoutes_bp.route('/deletePatient', methods=['DELETE'])
@jwt_required()
def delete_patient():
    admin_email = get_jwt_identity()
```

Code snippet 2: Delete patient API

```python
@adminRoutes_bp.route('/deleteNurse', methods=['DELETE'])
@jwt_required()
def delete_nurse():
    admin_email = get_jwt_identity()
```

Code snippet 3: Delete nurse API

4.2.2.2 Doctor Manage Appointments

The endpoint get patient booking (Refer Appendix Code snippet G.1: Get patient Booking.) was developed for the doctor to check the appointment scheduled for a particular

date. In this endpoint, the doctor is verified based on the email address (included in the JWT token). Upon successful verification, the appointments were retrieved from the database based on the doctor's ID with the date.

In addition, if the doctor wants to cancel a specific appointment due to uncertain circumstances, the cancel doctor appointment endpoint (Refer Appendix Code snippet G.2: Cancel Appointment by doctors. would be triggered. During this, the endpoint fetches the appointment details from the database, displays them to the doctor, and then the doctor selects the appointment to be canceled. Upon cancellation of the appointment, send_cancellation_email will be initiated. The send_cancellation_email helper function (Refer Appendix **Error! Reference source not found.**) sends a structured email to the patient's email a ddress. This ensures clear communication between the doctor and patient, where, if required, the patient could reschedule the appointment. For this feature to be implemented, Communication service from azure was leveraged. This service helps implement email communication, for our case when a doctor cancels an appointment it becomes necessary for the system to inform the patient that their appointment has been cancelled. This not only increases the transparency between the two parties using the system but also keeps data consistent and well communicated across the system.

4.2.2.3 Prescription

The doctor provides the prescription to the patient, which includes the medicine details, collection method, prescription date and doctor's email. As an initial step of this process, the doctor must verify the patient by entering the first name, last name and date of birth. To ascertain the patient's identity, the verify patient endpoint would be triggered (Refer Appendix Code snippet H.3: Verify Patient.), which validates the provided data against the patient's database. Upon successful verification, the doctor proceeds to issue the prescription.

During the process of prescribing the medicine, the medicine details along with their prices are extracted from the medicine table. The price of the medicine is mainly based on the type of patient: student/normal. If the patient is a student, each medicine price is calculated with a 10% discount.

The finalised prescription(Refer Appendix Code snippet H.4: Provide Prescription, Code snippet H.5: Provide Prescription Continuation.) is then stored in Azure Cosmos DB which is a NoSQL database. To retrieve the pharmacy and medicine details, the get_pharmacy (Refer Appendix Code snippet H.2: Get Pharmacies.) and get_medicine(Refer Appendix Code snippet H.1: Get medicines.) endpoints are created, which helps us dynamically retrieve all additional information about medicines and pharmacies.

4.2.2.4 Staff Authentication

The staff login endpoint (Refer Appendix Code snippet I.2: Staff Login, Code snippet I.3: Staff Login continuation.) is activated when the staff type is doctor, nurse, or admin. The credentials entered would be the password and email address. The email addresses are created for staffs in such a way that it adds an additional layer of security to the system. For instance, doctor email basically contains '. doctor' in the 'username.doctor@thehealme.com' email needed to login.

During the verification process, the verify_staff helper function (Refer Appendix Function C.5.: Staff Verification.) will be invoked. This function will check the email first and confirm that it does hold the '. doctor' or '. nurse' or '. admin' in the email. Upon verification of the credentials, the cookies are generated which hold the JWT and CSRF token. Conversely, if any email address or password is invalid, the backend will signal a status code 401 which says unauthorized.

The get staff profile endpoint (Refer Appendix Code snippet I.4: Staff Profile , Code snippet I.5: Staff profile Continuation.) is a GET method where the staff is verified based on the email address. After the verification has been completed, the endpoint executes the details of the staff irrespective of their type (nurse/doctor), including the name, email address, registration number, specialization, and phone number.

The get specialization endpoint (Refer Appendix Code snippet I.1: Get Specialization.) extracted the specialization of doctor/nurse. This was retrieved with the help of specialization and the nurse specialization table.

4.2.2.5 Staff Availability

For the doctor to set their availability, the set_doctor_availability endpoint (Refer Appendix Code snippet J.1: Set Doctor Availability.) is called. This endpoint sets the doctors' slots selected for a particular date and inserts them into the doctor availability database table. In case any of the selected slots is already set in the table, a status code 400 will be returned.

Likewise, the nurse can set the availability with the help of the set_nurse_availability endpoint (Refer Appendix Code *snippet* J.3: Set Nurse Availability.). This endpoint initially examines if the selected slot already exists in the nurse availability table, and if so, displays a status code 400. Conversely, the selected slots are sent to the database along with the nurse_id and the date. As a result, the message is returned as "Nurse Availability set successfully."

The staff (either nurse or doctor) has the option to cancel the availability (Refer Appendix Code *snippet* J.2: Cancel Doctor Availability. , Code snippet J.4: Cancel Nurse Availability.) that is previously set in the set_availability endpoint.

To retrieve the availability set by nurse/doctor, the get availability endpoint (Refer Appendix Code snippet J.5: Get the Availability.) is handled. This endpoint utilises the staff type and staff ID to fetch the availability from the doctor availability or nurse availability database table.

4.3 Medical records.

The medical record endpoint allows the authorised patient to upload the medical files (prescriptions, X-rays and reports) and allows the doctor to view the uploaded files. The upload endpoint (Refer Appendix Code snippet K.1: Upload the medical records.) allows patients to upload their documents to the containers assigned to them during the registration. The files are securely uploaded in the Azure Blob Storage, where each of the containers is linked with patient_id. Additionally, the connection string for the Azure Blob Storage has been configured, as it's necessary for secure communication.

The patients' records are protected with the JWT token, where each of the patients must be authenticated via JWT. Each patient is assigned a unique container along with an ID in blob storage. The files can only be accessed once the patient is verified. The upload medical record (/upload/<patient_id>) endpoint is secured with JWT. In this process, the API queries the patient table with patient_id to retrieve the Azure container name assigned to that patient. The patient then uploads the files based on the selected category and returns a success message.

The patient can view the uploaded medical documents with the get/patient/files endpoint (Refer Appendix Code snippet K.2: Get patients file). Like the /upload/<patient_id> endpoint, the API queries the patient_id to fetch the container_name. The response received presents the file name along with the URL published by the blob to access the files.

The doctors are only restricted to accessing the medical records of the patient with the post/patient/medical records endpoint (Refer Appendix Code snippet K.3: Doctors views the medical records., Code snippet k.4: Doctors view the medical record continuation.). During this process, the patient must be validated first with first name, last name and date of birth and If the patient is a valid user, only then the container name is fetched, which retrieves a list of files along with the URL. This helps the doctor to view the records without any complications.

4.4 Frontend Development.

The main aim is to design a user-friendly interface considering the diverse age groups. This section provides a detailed frontend development process. The application is built using React JS and Bootstrap Library, which helps to reuse the components and minimise the code redundancy across the web app (Describing the UI, n.d.).

A modular folder structure (Refer Figure 7) was developed where the main folders were components, pages and app.js. To make the code easy to understand, each of the components was named based on the feature/functionality that was later imported to the respective folders.

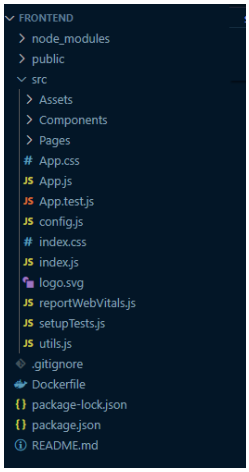


Figure 7: Folder Structure.

4.4.1 Development Process

The table conveys the features that are implemented for each of the users (patient, staff and admin) along with the actual name tags used for the dashboard. The functionalities and user journey of each of the dashboards will be explained module-wise, starting with the appointment module, prescription module, medical documents module, profile module, register staff and patient's module, list of registered patients and staff and lastly the set availability module. Below provided are the feature allowed for different users.

Module Name	Patient	Staff (Doctor/Nurse)	Admin
Appointment Module	Booking appointments	Get Patients Bookings	Booking appointments
Prescription Module	Prescription.	Prescription.	N/A
Medical documents module	Medical Records.	Medical records.	N/A
Profile module	Profile.	Profile.	N/A
Register module.	Register.	N/A	Add staff. Add patients.
List of registered patients and staff module.	N/A	N/A	Staff list. Patients list.
Set availability module.	N/A	Set Availability.	N/A

1. **Appointment module**

Patients:

Upon successful login, the patient is displayed the patient dashboard where the default page viewed is booking appointments. The previously booked appointment details, such as appointment ID, doctor, date , time and status, along with the cancel button, are displayed with the help of the fetchAppointments function(Refer Appendix Code snippet D.7: Fetch Appointment function.). The fetchAppointments function call the my-appointment API that will trigger once the patient is logged in. Due to uncertain circumstances, if the patient wants to cancel the appointment, the cancel button can be clicked where a modal is displayed allowing user to cancel the appointment. Upon successful confirmation, the handleCancelAppointment function (Refer Appendix Code snippet D.8: Cancel Appointment Function.) triggers the cancel_appointment API.

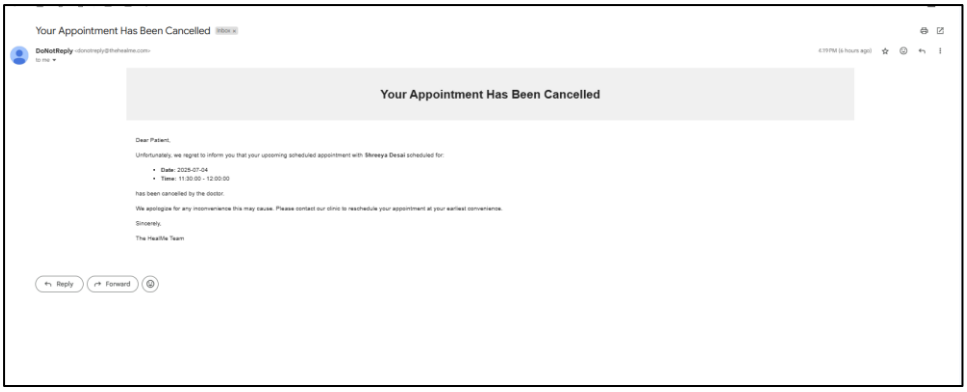
When the Book Appointment button is clicked, the booking form is displayed, whereupon selecting minor injuries displays an article. The patient has the option to read the article and take precautions at the very first stage of the disease. If, in case, the patient still requires a consultation from the GP, the date is selected from the calendar, and the function handleDateChange (Refer Appendix Code snippet D.9: Handle date change function.) is invoked. Further, the specialised doctor for that specific disease is displayed with the get_doctors_list API. Upon selecting the doctor, the available time slots are displayed where the get_doctor_availability endpoint is triggered, and the book button invokes the handleBookDoctor function (Refer Appendix Code snippet D.10: Handle book doctor function.).

Additionally, in this entire process, the components that were imported to the Patient Dashboard page were BookingAppointment, bookappointmentbutton and AppointmentCard. Furthermore, the BookingAppointmentForm, CancelAppointmentButton, CancelAppointmentButtonModal, Abdominal Page, Fever Page and Cough Page were the components utilised during this entire process of booking appointments and viewing the same.

Doctor:

The doctor can view the scheduled appointment with the help of the handleDateChange function (Refer Appendix Code snippet G.3: Handle date change function.). Firstly, the doctor needs to select the date and hit next to get_patient_bookings and then the patient bookings are displayed in the form of a table.

Due to uncertain circumstances, the doctor wants to cancel a specific appointment. The cancel button is clicked where the handleCancelAppointment (Refer Appendix Code snippet G.4: Handle cancel appointment.) function is invoked and the email is sent to the patient. sssss(Please Refer Email: 1 as an example for email cancellation).



Email: 1:Cancellation Email.

Admin:

The admin can also book the appointment on behalf on old aged patients.

The entire process is same as for the patients, but the only difference is that if the staff types is admin, only then the patient ID input field is displayed in the BookingAppointmentForm. The patient ID is retrieved from the patient list component. This ensures that the patient receives the appointment booked on their dashboard.

2. Prescription Module.

The prescription module aims to prescribe the medicine to the patient and send it to the pharmacy and the patients.

The ViewPrescription.jsx component is designed, which includes the fetch prescription function along with the get prescription endpoint. This helps the patient to view the prescription. The prescription form, pharmacy form, provide prescription, payment already done modal, payment success and payment cancel components are used to develop the provide prescription.

Patient:

During the process of viewing the prescription, the patient must click on the prescription option which is on the left sidebar. The system then displays the prescription details, including the patient's name, doctor's email address, and the prescribed medicine details along with the type of delivery. Additionally, the patient can pay for the prescriptions, and this can be achieved by clicking on the Pay now button.

Doctor:

The doctor firstly needs to verify the patient by entering the first name, last name and DOB. Upon successful verification, the doctor can move to the next step, which is the prescription form. The prescription form displays the input fields, such as the selection of the medicine from the drop-down, quantity and instructions. Furthermore, the type of patients and the type of delivery is selected based on the patient preferences.

The next step is to select the pharmacy and send the prescription to the pharmacy and patient.

3. **Register module.**

Patient:

Firstly, the landing page is displayed, which has a Register here button. Upon clicking, the register form is displayed, which is divided into two parts: the first page displays the name and address fields, and the second page displays the contact number, email address and password.

Once all the inputs are filled, the register button is clicked, which then triggers the handleRegister function (Refer Appendix Code snippet L.1: Handle Register.). The function triggers the /patient/register API, which then saves the patient's details to the database.

Admin:

The add staff and add patient component files are developed separately and are imported to the admin dashboard.

Only the admin has the right to add the staff, either doctor or nurse. During this process, the admin is verified based on the email address. Once the admin is verified, the Add Patient tag is displayed on the left side panel/navigation bar.

Upon clicking the button, a form is displayed where the admin enters the data in input fields. Once the details are entered, the add staff button is clicked, which triggers the handle function logic which has the registration endpoint.

Conversely, to add the staff, the Add Staff option from left side bar is clicked. Like the Add patients, the Add staff follows the same process. Once the add staff button is clicked, a form is displayed where the details are entered, and upon entering the details, the register button is clicked. This page triggers two functions: firstly, the fetchSpecializations(Refer Appendix Code snippet L.2: Fetch Specialization function.) and secondly, the handleSubmit.

4. **Medical Records.**

This module aims to upload and view the medical records.

Patient:

The patient firstly clicks on the Medical Records option that's on the left side of the dashboard. Further, the patient selects the type of file from the dropdown. Then the patient selects the file from their device and clicks on the upload now button.

During this process, the handleFinalUpload function (Refer Appendix Code snippet K.5: Handle final upload function. in the UploadMedicalHistory.jsx component handles the main logic of the system where the API call is invoked. The uploaded files are then displayed to the patient, who can view them.

Staff (Doctor):

To view the medical records of the patient, the doctor firstly has o verify the patient by entering their name and date of birth. Upon successful verification, the doctor is able to view the files that were uploaded by the patient. A link is provided to access the same and can also download if required. The component file utilized for designing the logic and frontend is ViewPrescription.jsx

5. Set Availability Module – Staff Specific

This module mainly focuses to set the availability for doctors and nurses.

The SetAvailabilityModal.jsx and SetDoctorAvailability.jsx components are imported to the staff dashboard pages. The functions used for this module were fetchAvailability, handleAddAvailability, and confirmAddAvailability, where the endpoints are triggered for adding the availability, and lastly, handleRemoveAvailability was implemented to remove the availability set.

The process to set availability is as follows: Once the staff is verified, the staff is navigated to the dashboard, which displays a left panel that has a tag named "Set Availability". During the process of setting the availability, the staff firstly needs to click on the set availability button, which further provides the option to select the date to which the slots will be added. The slots are then set by clicking the add button, which displays a confirmation modal. The slots confirmed are then displayed along with the cancel button on the dashboard.

6. List of registered patients and staff module – Admin Specific

The admin only has the authority to view the list of registered patients and staff (doctor/nurse). During this process, the admin is firstly verified based on the email address. Upon successful verification, the admin is redirected to the dashboard, which has a top navbar and left sidebar. The left navigation bar displays the tags "Patients List" or "Staff List".

To view the list of registered patients, the admin clicks on the "Patient List" option from the left sidebar. The system then displays the list of registered patients, which is retrieved from the getPatients API that's in the PatientList function.

On the contrary, to view the list of registered staff, the admin should click on "Staff List" option in the left sidebar. The application then displays the field where the staff role needs to be selected. Upon selecting the staff role, the list of either nurse or doctor is displayed based on the selection. This has been achieved with the help of API getDoctor/getNurse that's in StaffList function.

Additionally, for both the staff and patient, the admin only has the right to remove/delete them.The component files linked with this module are StaffList.jsx and PatientList.jsx, and both are imported in the respective Patient and Admin dashboards.

7. Profile module

The profile section displays the details of the user, who is either a patient, doctor or nurse. The details that are commonly displayed across all the users are first name, last name, email address, and contact number.

The component files that handle the patient profile and staff profile are named Profile.jsx and StaffProfile.jsx.

The patient's profile is displayed when the user clicks on the Patients option from the sidebar, which is on the left navigation/sidebar panel. When clicked, the patientProfile API is triggered and displays the details. The details can only be read and can't be edited. Similarly, the staff (doctor/nurse) can view the details in the Profile section, which, when clicked, triggers the staffProfile API and displays the details.

5. Testing.

5.1 Unit testing

This Section will showcase unit tests conducted around different modules of web app.

5.1.1 Patient Features - Unit testing

1. Book Appointment

Test Case ID	Test Title	Module	Expected Outcome	Actual Outcome	Status
PC_001	Appointment Booked Successfully	Book Appointment	The appointment should be booked successfully by filling the required details.	The appointment is booked.	Pass
PC_002	Failed Appointment Booking	Book Appointment	The incorrect details are entered due to which the patient would not be able to book the appointment.	Due to incorrect selection , the patient is not able to book the appointment.	Pass
PC_003	Slots not available for particular date	Book Appointment	No slots should be available	The slots is not available.	Pass
PC_004	Minor Disease Article	Book Appointment	The article would be visible for minor injuries.	The article is visible when selected (Fever).	Pass

2. Prescription

Test Case ID	Test Title	Module	Expected Outcome	Actual Outcome	Status
PC_006	View Prescription	Prescription	The patient should be able to view the prescription provided by the Doctor.	The patient is able to view the prescription.	Pass
PC_007	Details of Prescribed medicine.	Prescription	The patient should get the details of prescribed medicine.	The patient is able to view the prescribed medicine.	Pass
PC_008	No Prescripitons has been provided	Prescription	If doctor has not provided any prescription to a particular patient he/she should not see any prescriptions.	If doctor has not provided any prescription to a particular patient he/she is not able to see any prescriptions.	Pass
PC_009	Pay for Prescriptions	Prescription	Patient should be able to pay for the presciptions from anywhere across world.	Patient is able to pay for the presciptions from anywhere across world.	Pass

3. Medical Records

Test Case ID	Test Title	Module	Expected Outcome	Actual Outcome	Status
PC_010	Upload medical Records	Medical Records	The patient should be able to upload medical records with appropriate file type and name.	The patient is able to upload files/medical history in the website.	Pass
PC_011	View Medical Records.	Medical Records	The doctor should be able to view medical records with appropriate file type and name.	The doctor is able to retrieve patients medical records based on patient name, dob.	Pass

4. Profile

Test Case ID	Test Title	Module	Expected Outcome	Actual Outcome	Status
PC_005	Patient Profile	Profile	The patient would be able to view the personal details.	The patient is able to view the personal details.	Pass

5.1.2 Staff Features – Unit testing

1. Set Availability

Test Case ID	Test Title	Module	Expected Outcome	Actual Outcome	Status
PC_001	Set Doctor Availabilty	Availability	The doctor should be able to book his availabilty today for two months later.	The doctor is able to book his availabilty today for two months later.	Pass
PC_002	Set Nurse Availabilty	Availability	The nurse should not be able to book his availabilty today for the next day.	The nurse is not be able to book his availabilty today for the next day.	Pass
PC_003	Cancel Doctor Availabilty	Availability	The doctor should able to cancel his availabilty anytime.	The doctor is able to cancel his availabilty anytime.	Pass
PC_004	Cancel Nurse Availabilty	Availability	The nurse should not be able to cancel his availabilty anytime if an appointment is already booked on that day.	The doctor is not able to cancel his availabilty anytime if an appointment is already booked on that day.	Pass

2. Get Patient Booking

Test Case ID	Test Title	Module	Expected Outcome	Actual Outcome	Status
PC_005	getPatientBooking	Patient Booking	The doctor should be able to get all the patients bookings for a particular date.	The doctor is able to get all the patients bookings for a particular date.	Pass

3. Prescription

Test Case ID	Test Title	Module	Expected Outcome	Actual Outcome	Status
PC_006	Provide-Prescription	Prescription	The doctor should be able to provide prescripiton to respective patient.	The doctor is able to provide prescripiton to respective patient.	Pass
PC_007	Provide-Prescription	Prescription	The doctor should be able to provide appropriate medicines and its instructions to respective patient.	The doctor is able to provide appropriate medicines and its instructions to respective patient.	Pass
PC_008	Provide-Prescription	Prescription	The doctor should be able to send the prescriptions of respective patient to appropriate pharmacy.	The doctor is able to send the prescriptions of respective patient to appropriate pharmacy.	Pass

4. Medical Records

Test Case ID	Test Title	Module	Expected Outcome	Actual Outcome	Status
PC_010	View Medical Records.	Medical Records	The doctor should be able to view medical records with appropriate file type and name.	The doctor is able to retrieve patients medical records based on patient name, dob.	Pass

5.1.3 Admin Features - Unit testing

1. Add Patient

Test Case ID	Test Title	Module	Expected Outcome	Actual Outcome	Status
AC_001	Successfully Add Patient	Add Patient	The patient should be added successfully	The patient is been added successfully.	Pass
AC_002	Missing Fields	Add Patient	The patient should not be registered incase of any missing fields(such as email address, DOB).	The patient is not registered incase due to the missing fields that are required	Pass
AC_003	Already Registered	Add Patient	The patient is already registered due to which won't be able to re-register.	Patient Already Exist	Pass

2. Add Staff

Test Case ID	Test Title	Module	Expected Outcome	Actual Outcome	Status
AC_004	Successfully Add Staff	Add Staff	The staff(nurse/doctor) should be registered successfully.	The staff(nurse/ doctor) is been registered.	Pass
AC_005	Missing Staff Role	Add Staff	The staff role would have been missing that is doctor/nurse.	The staff role is not been selected.	Pass
AC_006	Missing Fields	Add Staff	The admin won't be able to register due to the missing fields.	Due to missing field , the admin is not able to register.	Pass

3. Manage Bookings

Test Case ID	Test Title	Module	Expected Outcome	Actual Outcome	Status
AC_007	Appointment Booking	Manage Booking	Appointment Booked Successfully.	Appointment is been booked successfully.	Pass
AC_008	Invaid Patient_Id	Manage Booking	The admin would enter the wrong patient_id and won't be able to book appointment.	The admin is not able to book appointment due to the wrong patient_id.	Pass
AC_009	Patient_Id verified	Manage Booking	The admin should firstly check if the patient is registered, if registered, the patient_id would be available to book the appointment.	The admin cross-verified the patient details, retrieved the patient_id inorder to book the appointment.	Pass

4. Get Staff and Patients list

Test Case ID	Test Title	Module	Expected Outcome	Actual Outcome	Status
AC_010	Get Staff, Staff type is Doctor	Get Staff	Doctors List should be retrieved Successfully.	Doctors List is retrieved Successfully.	Pass
AC_011	Get Staff, Staff type is Nurse	Get Staff	Nurse List should be retrieved Successfully.	Doctors List is retrieved Successfully.	Pass
AC_012	Get Patient List	Get Patient	Patient List should be retrieved successfully	Patient List is retrieved successfully	Pass

6. Deployment.

This section provides an in-depth description of the deployment done on Microsoft Azure leveraging Azure resources.

6.1 Architecture.

A simple cloud-based architecture is designed to ensure security, maintainability and scalability across the services. The big three cloud providers currently in the market are Microsoft Azure, Amazon Web Services, and Google Cloud Platform, along with IBM Cloud, Oracle Cloud, Salesforce, DigitalOcean and many more. Thus, Microsoft Azure was the selected cloud provider for this project.

The figure illustrates (Figure 8) the system components, such as frontend, backend, and databases (SQL and NoSQL), which are interconnected with the services. In this deployment process, the frontend is deployed within the Azure App Service, as it helps to run the application without the burden of managing the underlying architecture. Similarly, the backend is deployed in Azure Container Apps since it provides reduced infrastructure maintenance and saves cost for containerised applications.

Additionally, the backend communicates with Azure SQL, Azure Cosmos DB, Blob Storage and Azure Communication Services for email, which is achieved with the help of the connection string provided by Azure services that is then set in the environment file of the backend.

In terms of security, cross-origin resource sharing is implemented to ensure that the server accepts and responds to the authorised frontend application. This methodology ensures that only authenticated and authorised requests are accepted, which would prevent malicious cyber-attacks. Additionally, along with CORS, the JWT and CSRF tokens were generated for a secure communication between the backend server and frontend application. The CORS would only accept the request sent by *thehealme.com*; this domain is purchased from GoDaddy.

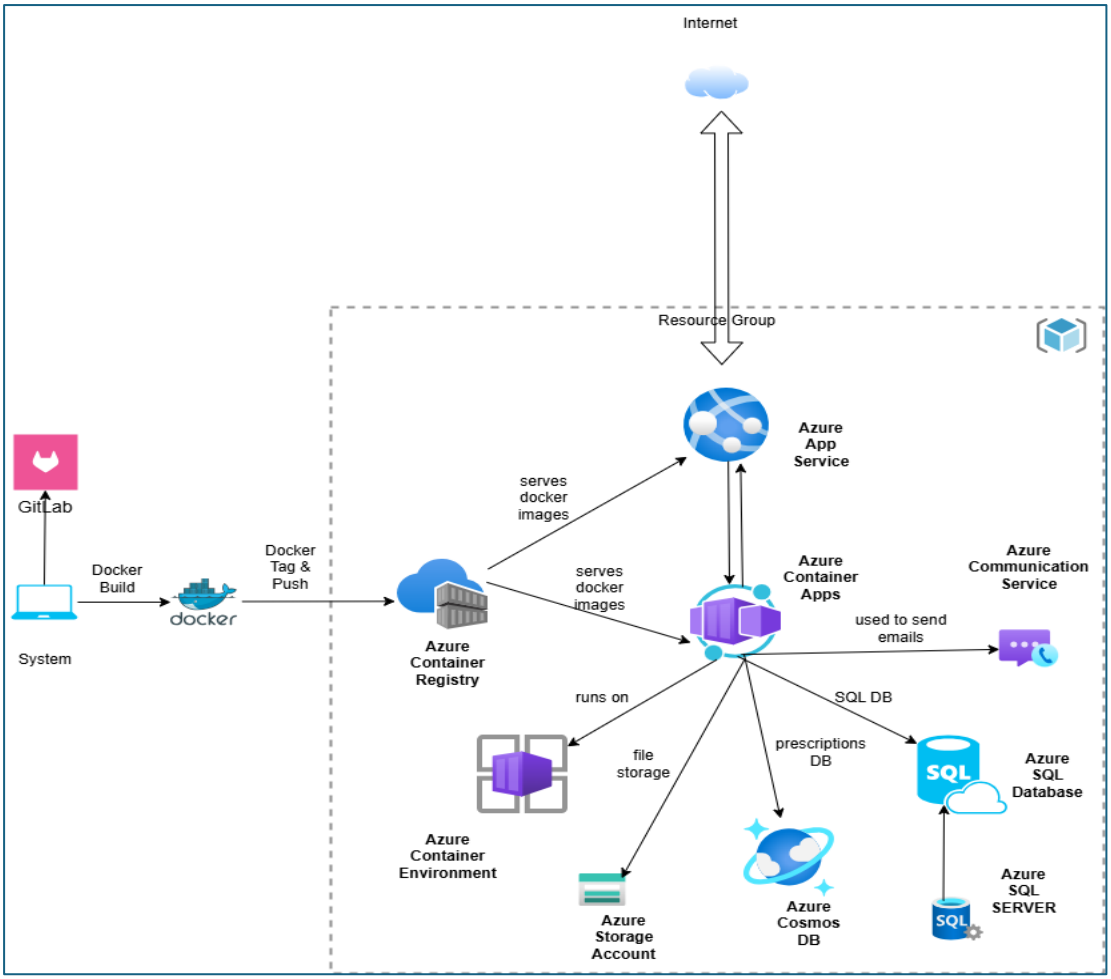


Figure 8: Architecture Diagram.

6.2 Azure Services Used.

The Azure services used throughout this project are as follows:

Azure Container App

The Azure Container App is a serverless platform, mainly used for the deployment of backend (Flask AP) applications. The container app eases the management with auto-scaling, security management, monitoring and flexibility in deployment process. Additionally, the container app needs a container environment which supports the functioning of the container app. (craigshoemaker, 2024).

Azure App Service

The Azure App Service is mainly used for the deployment of React frontend applications. The Azure App Service is a platform that runs the application, abstracting away the infrastructure. Additionally, this service supports various web stacks such as .NET, Java (in Java SE, Tomcat, and JBoss flavours), Node.js, Python, and PHP and runs either on Linux or Windows (msangapu-msft, 2024).

Azure Container Registry

The Azure Container Registry acts as a repository for storing docker images. The images from the Registry are then consumed or deployed by server like Container apps and Azure app service.

Azure SQL

Azure SQL is a fully managed relational database-as-a-service (DBaaS). Moreover, Azure SQL is a managed SQL server database engine which falls under the platform as a service. Moreover, SQL Server is a built-in functionality and feature that requires extensive configuration. Therefore, this service is mainly used for the SQL database queries in this application (WilliamDAssafMSFT, 2024).

Azure Cosmos DB

The Azure Cosmos DB is a NoSQL database and is primarily used to store the prescription details. The Cosmos DB simplifies the development process of the application much faster in a single database (markjbrown, 2024) (AdamGuan13, 2024).

Azure Communication Service.

Azure Communication Service is used for the notification purpose, which will send an email to the patients about the cancellation of the appointment. Additionally, the service supports diverse communication formats (PetoMichalak, 2024) (MelanieHom, 2025).

Azure Storage Accounts

The Azure Storage service is Microsoft's cloud storage solution for modern data storage scenarios. The Azure storage offers durability, high availability, scalability, accessibility, and security for storing the data in the cloud. This service is primarily used to store the medical documents of patients, ensuring easy accessibility from anywhere (martinduefrandsen, 2025) (stevenmatthew, 2022).

6.3 Deployment Process.

After testing the web-app and being satisfied with the test results on local development environment, the next step was to deploy the web-app on Azure cloud. Now, deploying a full stack web application was a challenging experience. After reading through various articles and website blogs (RoseHJM J. , 2024) (Ahamed, 2020) (RoseHJM, 2025) (HeidiSteen, 2025) (JasonFreeberg, 2025) (CamSoper, 2024) (cephalin, 2025), I learned about the deployment methodologies which are been used across industry as of today. Some of the methodologies include CI/CD deployment, deploying application directly to the cloud services using GitHub and lastly deployment with the help of docker images.

Acknowledging that this is the first time, where I will deploy a full stack application on cloud. The methodology I chose for cloud deployment was the docker image deployments, the reason behind opting this methodology was due to its intuitive and simple to implement nature. Although being simple to implement, several official documentation and articles were thoroughly read and understood before even starting the deployment process. A dockerfile is necessary to be provided in the root of the folder structure, which comprises of all the necessary libraries and commands which are needed to run the app independently on the cloud.

This methodology seemed to prove helpful, as it easily aligned to our applications architecture. The developed web app follows N-tier architecture, therefore the frontend and

backend are two separate entities and should be deployed separately. Although the deployment of backend and frontend will be done separately, the initial steps need to be followed for both are similar.

Below are the common steps for both backend and frontend to build docker images and publish them to cloud.
(Refer Appendix O for the snippets of entire development process.)

Step	Action / Use	Docker Command	Backend API	Frontend
1	Create DockerFile in projects root folder, comprising of all the essential libraries and command necessary for the application to run independently on azure cloud.	N/A	Required	Required
2	Build docker image using the docker build command. And provide it a unique tag to the built image for identification and versioning.	docker build -t appname:version .	Required	Required
3	Log into Azure CLI, this needs to be done for backend and frontend in separate consoles.	az login	Required	Required
4	Login into azure portal, and create an Azure Container registry (ACR) with proper region and subscription	N/A	N/A	N/A
5	Using the Azure CLI, log back into the ACR . ACR acts like a repository for container images.	az acrname login	N/A	N/A
6	In this step the image built with docker is tagged.	docker tag appname:version acrname.azurecr.io/appname:version	Required	Required
7	And the same image is pushed into the ACR tagged in previous steps.	docker push acrname.azurecr.io/appname:version	Required	Required
8	In Azure portal, create a new container app service, with minimum storage size and vcores. This is where Backend Api will be deployed	N/A	Required	N/A
9	In Azure portal, create a new Azure Web app service, This is where frontend will be deployed	N/A	N/A	Required
10	Finally, after or while creation of azure resources, we chose the docker image pushed into the acr.	N/A	Required	Required

After successfully deploying the Backend and Frontend to their respective azure services. It was decided to add a custom domain to avoid CORS error. To implement this

additional security layer of CORS, a new domain name for both Backend ‘api.thehealme.com’ and frontend ‘*thehealme.com*’ were purchased from GoDaddy website. Furthermore, after purchasing the DNS, it was added to Azure web app and container app in the custom domain sections of each. Additionally, we configured a azure self-managed SSL certificate for the website or endpoint to be https. During this process, the domain validation was essential. Azure performed the validation since it gave information to fill in the DNS section within GoDaddy. Upon successful verification, the domain was added. Additionally, the certificate is secured with https.

The in-depth explanation of the deployment process is provided below, respective of Frontend and Backend.

Frontend

Initially, we had to create the Azure App service called FrontendGpSystem, assuming we are in the Azure Resource Group (rg_gp_uk). After choosing the app service, the Deployment section appears in a left panel. To view the application deployment details, choose the Deployment Centre inside the deployment option in the left navbar. Here, we select and save the revised picture tag. This procedure assures that the right docker image is being retrieved and ran (Refer Appendix Deployment: O.23: Deployment Center(FrontendGpSystem)).

Backend

Conversely, for the backend application, create the Azure App Container named gpsystemukcont, assuming we are in the Azure Resource Group (rg_gp_uk). After selecting the container app, the Application section appears in the left panel. To view the application deployment details, choose the Containers inside the application section. Here, we should select the revised image tag and save. Therefore, this confirms that the correct image is being retrieved and ran (Refer Appendix Deployment: O.24: Containers (gpsystemukcont).). The Azure Container Apps is configured mainly for Flask API backend applications, which include details of the subscription and resource group. In this process the container app environment was needed to be created, as the environment would secure the container apps running in it

7. Results.

7.1 Achievements.

The outcomes that the project has led to are outlined below:

Successful development and integration of the backend and frontend was accomplished seamlessly. Cross-site request forgery (CSRF) and JWT tokens (cookies) have been implemented to ensure secure communication and authentication between the frontend and backend application layers. Additionally, CORS has been implemented as an additional layer of security. The overall implementation process for the automated GP system, ensures that security has been given the highest priority.

The dashboards for users like admin, doctors/nurses and patients were designed with careful consideration of demographics. Each of these dashboards are given access based on their user roles, ensuring no authorised user can access protected or sensitive information.

Azure Cloud services like Azure Web Apps, Azure Container Apps, Storage Accounts, Azure Cosmos DB, and Azure SQL are used to successfully deploy the web-app. Integration of the business logic with these azure services will help the system to be available 99% of the time, as the chosen services are fully managed by Azure. An instance of using azure service which led to convenient data storage was the medical documents uploaded by the patients are stored securely in the Azure Blob Storage, associating their files with their unique patient_id.

The developed application is fully responsive for all types of screens. Ensuring this web-app can be accessed anywhere and anytime. This functionality has been achieved using React Bootstrap's classes and have been tested across mobile devices. The application's overall usability and navigation are quite straightforward, ensuring simple, user-friendly website navigation.

Stripe payment integration with the backend business logic has been implemented to securely allow patients to purchase the medicines prescribed by the doctor. Additionally, the webhooks feature of Stripe allows us to track the status of every payment made.

The Backend follows a scalable, modular and easy to understand folder structure. This has been done to ensure seamless integration of any features that will be implemented in near future.

7.2 Lesson Learned.

The lessons learnt from this project are mentioned below:

- Importance of literature survey of the existing applications and technology used for developing the application not only supported me during the development phase but also helped build the foundational knowledge necessary in the tech industry today.

- Gained hands-on experience in developing a full-stack application using the React JS, Bootstrap and Flask API (Python) with regards to tech stack. Understood the need to

learn these tools, which is basically to solve real world problems faced in day-to-day life.

- Learnt about the significance of JWT and CSRF tokens in the software engineering space. Implementing of these concepts within the code was a challenge, however both were deeply studied and acknowledged.

- Configured a DNS, which I had never tried before. While configuring the DNS with deployed web-app, came across the concept of CORS (Cross origin resource sharing). After indulging into some articles and documentation, got an understanding of this concept as well.

- Familiarized myself with the deployment process on Microsoft Azure and explored the services provided by Azure. Some of the services were used during the application deployment for better understanding and gaining knowledge.

All the lesson learned from this paper, not only will help in building the foundational knowledge but also allow me to consider and explore these experiences deeper in future.

7.3 Own Piece of Code.

This section will exclaim about my ideas and thoughts used and applied in this paper.

1. **Send Cancellation Email Function** – Book/Cancel Appointment Module

The send_cancellation_email is a helper function that's used to send an email to patient asking them to reschedule their appointment, when an appointment is cancelled by the doctor. To implement this function, initially the Azure Communication service verified the domain 'thehealme.com' purchased from GoDaddy. Upon DNS verification, the communication string which is necessary to use Azure communication Service with our backend API was integrated and stored in the environment variable file.

This function was utilised by the /cancel_doctor_appointment/<int:appointment_id> endpoint. The cancellation email sent to the patient included the doctors' first names, doctors' last names, slots and date on which the appointment was scheduled. The challenging part to implement this utility function was the configuration and integration of the Azure Communication service with the Backend App deployed on Azure Container App. Please Refer Appendix Function C.6: Send cancellation email continuation., Function C.4: Send Cancellation email to view the code implantation and the email sent to patient.

2. **Staff Verification Function** – Login Module

This function is specifically implemented to help users like patient, doctor or admin to navigate to their respective dashboard without much hassle. This function will be invoked when the users are logging into the web-app.

When the user provides credentials and tries to login, the function logic triggers. For example, when the email says ramu.doctor@thehealme.com , the developed function splits the email from the '.' Till the '@' retrieving the doctor string only. This then helps identify staff type and then based on the staff type the users are navigated to their respective dashboard.
Refer Appendix Function C.5.: Staff Verification. for code implementation.

3. **Azure Container name allocation** – Medical Record Module

The main purpose of the container is to store the medical records of each patient. During registration, each of the patients is allocated a container with a unique name. The unique name is a combination of patients first name, last name, date of birth and unique id. To configure this Azure Storage Service, the connection string provided by Azure is added into the environment variable file of Backend APIs. Refer Appendix Code snippet M.1: Container Name

4. **Provide Prescription** – Prescription Module

The prescriptions are stored in the Cosmos DB, which is a NoSQL database. Cosmos DB was primarily chosen to store every patient's prescription separately based on their patient Id. Therefore, the Cosmos DB was configured in such a way that every time a prescription is provided to an existing patient, the prescription will be stored in that specific patient's partition.

To store the prescription data and configure cosmos DB with backend code, a connection string provided by Azure was added in the environment variable list of Backend code.
Refer Appendix Code snippet M.3: Prescription Creation

7.4 Limitations

Most of the features of the Automated system for local GP were implemented, however there are certain limitations which do exist and those will be discussed in this section.

- During the process of registration, the patient is not additionally verified via email verification or SMS, due to which the system lacks an additional security layer.
- The web app does not allow users to reset their password. The password is defined at the time of registration and cannot be changed after the registration.
- The patients can only view the list of medical records uploaded but cannot access the document through the blob storage URL, unlike doctor who can view the document using the blob storage URL. Additionally, once the documents are uploaded the patient can't delete the uploaded records.
- The patient and staff personal details cannot be modified.
- The recommendation of nearby pharmacies using machine learning, and postcode is not implemented. And would be considered to developed in future releases.
- No reminders about the appointment or appointment booking confirmation emails are sent to the patient.
- Usability and User acceptance testing of the application is not performed. This leaves a grey area, where decision on the scalability of the app remains.
- Once the doctor sets the availability for a particular date and instantly a patient books an appointment for the same date and slot. And under certain circumstances if the doctor had to cancel their availability, then the system would not allow the doctor to do so. Hence, this could be one of the major drawbacks of the system.
- Finally, the current cloud infrastructure is not designed to the security standards expected to manage patient data. A lot of improvements like deploying the Web-app in a Virtual network with public and private subnets could have been considered. However, due to shortage of time and lack of industry expertise this improvement will be done in the near future.

8. Conclusion.

To summarise, the development of an Automated GP system acts as a thorough iteration over the existing less user friendly and unavailable GP services. While, the healthcare sector faces tremendous challenges like over-worked staff & lack of digital infrastructure support, the integration of video consulting & machine learning in the existing GP system will significantly empower the healthcare sector in the UK. By strategically implementing the above-mentioned features and strengthening the existing GP system, this system will evolve into a powerful and efficient tool for the healthcare staff and will also pave the way for proactive and patient-centric GP system.

8.1 Future Scope.

Inspecting the current scenario across the world, where issues like global wars, shortage of medical staff, lack of infrastructure to produce pharmaceutical products and provide GP services are at its peak. There are still positive advancements in the healthcare sector like adoption of Artificial intelligence and Machine Learning, IOT and much more. These advancements will support the GP services in-near future to carry out their work more efficiently without being over-stressed.

However, speaking about the Future scope of the developed system. The current system could be further improvised by implementing feature like Video & Text consulting by developing knowledge of Real time communication technologies, early detection of major health issues like cancer and tuberculosis through machine learning and Artificial Intelligence, ensuring data privacy through complying with GDPR policies and focusing on designing cloud infrastructure which is more like a Virtual network wherein traffic from outside world like attackers would not be able to interact with patient data.

All these features mentioned above could be developed one by one in the further releases of An Automated System for Local GP by keeping in mind the time constraints, planning and robust testing of each of these features.

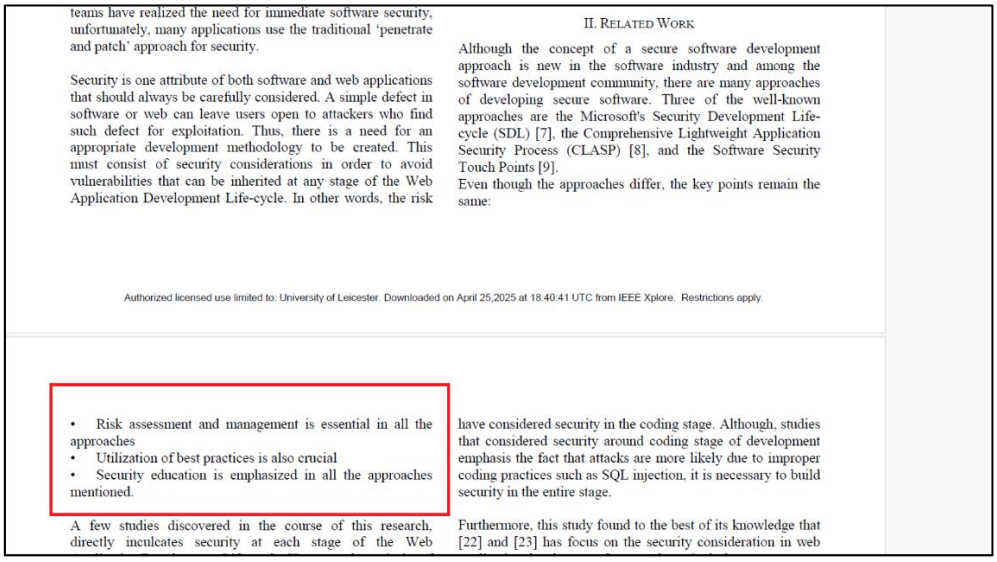
References

1. AdamGuan13. (2024, 09 14). *Understanding distributed NoSQL databases*. Retrieved from Microsoft Leran: https://docs.azure.cn/en-us/cosmos-db/distributed-nosql
2. Ahamed, A. (2020, August 15). *Front End Application Deployment in Azure WebApp*. Retrieved from Medium: https://aashikahamed.medium.com/front-end-application-deployment-in-azure-webapp-519aa747407a
3. Begum, S., & C. P. Indumathi. (2016). *ER diagram based web application testing.* Madurai, India: IEEE. doi:10.1109/ICCIC.2015.7435786
4. CamSoper, D. (2024, 09 15). *Tutorial: Host a RESTful API with CORS in Azure App Service*. Retrieved from Micrsoft Learn: https://learn.microsoft.com/en-us/azure/app-service/app-service-web-tutorial-rest-api
5. cephalin, g. (2025, 03 28). *Authentication types by deployment methods in Azure App Service*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/azure/app-service/deploy-authentication-types
6. craigshoemaker, l. (2024, November 19). *Azure Container Apps overview*. Retrieved from Microsft Learn : https://learn.microsoft.com/en-us/azure/container-apps/overview
7. Daniela Berardi, G. D. (2005). Reasoning on UML class diagrams. *Artificial Intelligence*. doi:https://doi.org/10.1016/j.artint.2005.05.003
8. *Describing the UI*. (n.d.). Retrieved from Learn React: https://react.dev/learn/describing-the-ui
9. Genilloud, A. W. (2001). The Role of “Roles” in Use Case Diagrams. doi:https://doi.org/10.1007/3-540-40011-7_15
10. HeidiSteen, a. (2025, 01 31). *Configure Azure AI services virtual networks*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/azure/ai-services/cognitive-services-virtual-networks?context=%2Fazure%2Fai-services%2Fopenai%2Fcontext%2Fcontext&tabs=portal
11. Igor Kukhtevich, V. G. (2021). *Aspects of Using Intelligent Cloud Technologies in Professional Training of General Practitioners.* IEEE. doi:DOI: 10.1109/TELE52840.2021.9482574
12. JasonFreeberg, y. (2025, 01 24). *Deployment best practices*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/azure/app-service/deploy-best-practices
13. Khan, Z. (2023). The Emerging Challenges and Strengths of the National Health Services: A Physician Perspective. *Cureus*. doi:10.7759/cureus.38617
14. Lee, K. S. (2004). Are use case and class diagrams complementary in requirements analysis? An experimental study on use case and class diagrams in UML. *Requirements Engineering*. doi:https://doi.org/10.1007/s00766-004-0203-7
15. MacConnachie, V. (2024, May 28). *Is it impossible to see a GP?* Retrieved from NHS Confederation: https://www.nhsconfed.org/articles/it-impossible-see-gp
16. markjbrown, W. (2024, 12 03). *Azure Cosmos DB - Database for the AI Era*. Retrieved from Microsft Learn: https://learn.microsoft.com/en-us/azure/cosmos-db/introduction
17. martinduefrandsen, D.-D. (2025, 03 04). *Storage account overview*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/azure/storage/common/storage-account-overview
18. *Medical staffing in the NHS.* (2025, March 28). Retrieved from BMA: https://www.bma.org.uk/advice-and-support/nhs-delivery-and-workforce/workforce/medical-staffing-in-the-nhs#:~:text=The%20high%20rate%20of%20doctors,their%20organisation%20values%20their%20work.

19. *Medical staffing in the NHS.* (2025, April 24). Retrieved from BMA (British Medical Association): https://www.bma.org.uk/advice-and-support/nhs-delivery-and-workforce/workforce/medical-staffing-in-the-nhs
20. MelanieHom. (2025, 01 08). *Azure Communication Services*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/rest/api/communication/
21. Mohammad Mehrtak, S. S. (2021, Jul-Aug). *Security challenges and solutions using healthcare cloud computing.* doi: 10.25122/jml-2021-0100
22. msangapu-msft, P. d.-m. (2024, 04 24). *App Service overview*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/azure/app-service/overview
23. Nasaruddin, N. S., & Izzatdin Abdul Aziz. (2018). *Web-Based Electronic Healthcare Record System (Ehrs) Based on Feedback.* Langkawi, Malaysia: IEEE. doi: 10.1109/AINS.2018.8631427
24. Pernice, K. (2016, December 18). *UX Prototypes: Low Fidelity vs. High Fidelity*. Retrieved from NN/g: https://www.nngroup.com/articles/ux-prototype-hi-lo-fidelity/
25. PetoMichalak, P. (2024, 05 09). *What is Azure Communication Services?* Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/azure/communication-services/overview
26. Rasha Talal Hameed, N. T. (2015). *Design of e-Healthcare Management System Based on Cloud and Service Oriented Architecture.* IEEE.
27. RoseHJM. (2025, 03 07). *Reliability in Azure Deployment Environments*. Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/azure/reliability/reliability-deployment-environments?toc=%2Fazure%2Fdeployment-environments%2Ftoc.json&bc=%2Fazure%2Fdeployment-environments%2Fbreadcrumb%2Ftoc.json
28. RoseHJM, J. (2024, 05 30). *Key concepts for Azure Deployment Environments*. Retrieved from Microsoft Learnc: https://learn.microsoft.com/en-us/azure/deployment-environments/concept-environments-key-concepts
29. rwestMSFT. (2025, 02 13). *What is SQL Server Management Studio (SSMS)?* Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/ssms/sql-server-management-studio-ssms
30. Shuaibu, M. B., & Ruqayyat Ahmad Ibrahim. (2017). *Web application development model with security concern in the entire life-cycle.* Salmabad, Bahrain: IEEE. doi:10.1109/ICETAS.2017.8277849
31. Song, I.-Y., Mary Evans , & E.K. Park. (1995). A Comparative Analysis of Entity-Relationship Diagrams. *Journal of Computer and Software Engineering*, 427 - 459. Retrieved from chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://www.cin.ufpe.br/~in1008/aulas/A%20Comparative%20Analysis%20of%20Entity-Relationship%20Diagrams.pdf
32. Staff, C. (2024, Aug 08). *What Is High Fidelity?* Retrieved from Coursera: https://www.coursera.org/gb/articles/high-fidelity
33. stevenmatthew, t. (2022, 11 21). *What is Azure Blob storage?* Retrieved from Microsoft Learn: https://learn.microsoft.com/en-us/azure/storage/blobs/storage-blobs-overview
34. WilliamDAssafMSFT, J. (2024, 09 27). *What is Azure SQL?* Retrieved from Microsft Learn: https://learn.microsoft.com/en-us/azure/azure-sql/azure-sql-iaas-vs-paas-what-is-overview?view=azuresql

Appendix

A. Background Research



teams have realized the need for immediate software security, unfortunately, many applications use the traditional 'penetrate and patch' approach for security.

Security is one attribute of both software and web applications that should always be carefully considered. A simple defect in software or web can leave users open to attackers who find such defect for exploitation. Thus, there is a need for an appropriate development methodology to be created. This must consist of security considerations in order to avoid vulnerabilities that can be inherited at any stage of the Web Application Development Life-cycle. In other words, the risk

II. Related Work

Although the concept of a secure software development approach is new in the software industry and among the software development community, there are many approaches of developing secure software. Three of the well-known approaches are the Microsoft's Security Development Life-cycle (SDL) [7], the Comprehensive Lightweight Application Security Process (CLASP) [8], and the Software Security Touch Points [9].
Even though the approaches differ, the key points remain the same:

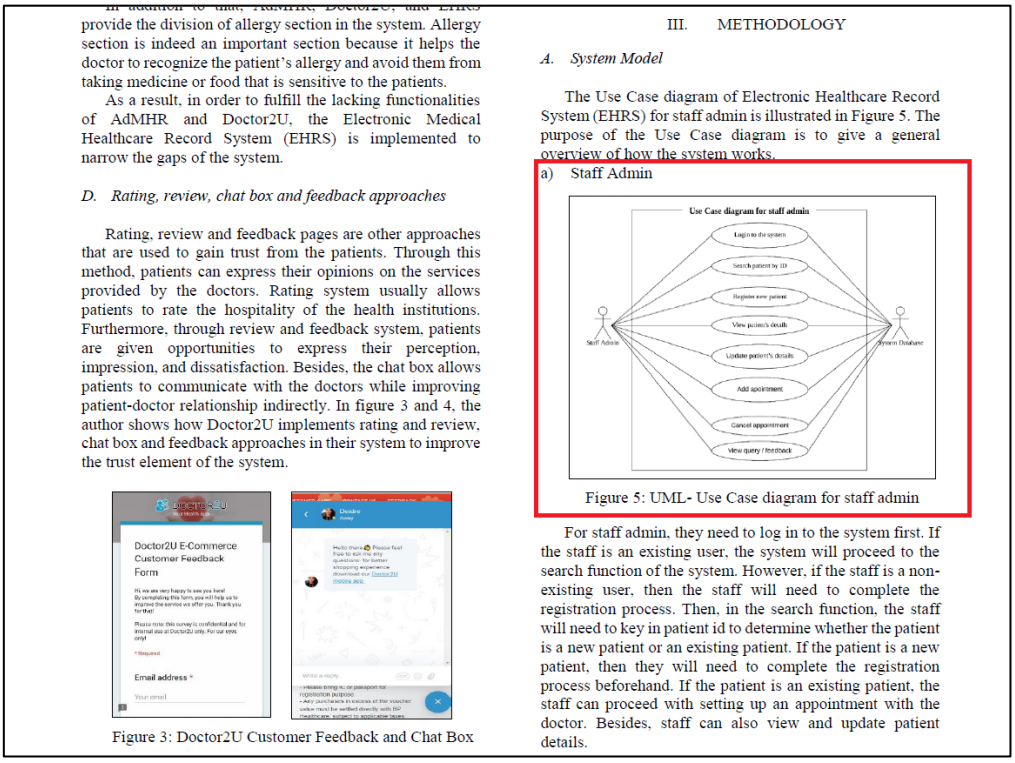
- Risk assessment and management is essential in all the approaches
- Utilization of best practices is also crucial
- Security education is emphasized in all the approaches mentioned.

A few studies discovered in the course of this research, directly inculcates security at each stage of the Web

have considered security in the coding stage. Although, studies that considered security around coding stage of development emphasis the fact that attacks are more likely due to improper coding practices such as SQL injection, it is necessary to build security in the entire stage.

Furthermore, this study found to the best of its knowledge that [22] and [23] has focus on the security consideration in web

Literature 0.A.1: Approaches key point. Image is taken from the literature survey.



provide the division of allergy section in the system. Allergy section is indeed an important section because it helps the doctor to recognize the patient's allergy and avoid them from taking medicine or food that is sensitive to the patients.

As a result, in order to fulfill the lacking functionalities of AdMHR and Doctor2U, the Electronic Medical Healthcare Record System (EHRS) is implemented to narrow the gaps of the system.

D. Rating, review, chat box and feedback approaches

Rating, review and feedback pages are other approaches that are used to gain trust from the patients. Through this method, patients can express their opinions on the services provided by the doctors. Rating system usually allows patients to rate the hospitality of the health institutions. Furthermore, through review and feedback system, patients are given opportunities to express their perception, impression, and dissatisfaction. Besides, the chat box allows patients to communicate with the doctors while improving patient-doctor relationship indirectly. In figure 3 and 4, the author shows how Doctor2U implements rating and review, chat box and feedback approaches in their system to improve the trust element of the system.

Figure 3: Doctor2U Customer Feedback and Chat Box

III. METHODOLOGY

A. System Model

The Use Case diagram of Electronic Healthcare Record System (EHRS) for staff admin is illustrated in Figure 5. The purpose of the Use Case diagram is to give a general overview of how the system works.

a) Staff Admin

Figure 5: UML- Use Case diagram for staff admin

For staff admin, they need to log in to the system first. If the staff is an existing user, the system will proceed to the search function of the system. However, if the staff is a non-existing user, then the staff will need to complete the registration process. Then, in the search function, the staff will need to key in patient id to determine whether the patient is a new patient or an existing patient. If the patient is a new patient, then they will need to complete the registration process beforehand. If the patient is an existing patient, the staff can proceed with setting up an appointment with the doctor. Besides, staff can also view and update patient details.

Literature 0.B A.2:Use Case Diagram for Staff Admin.

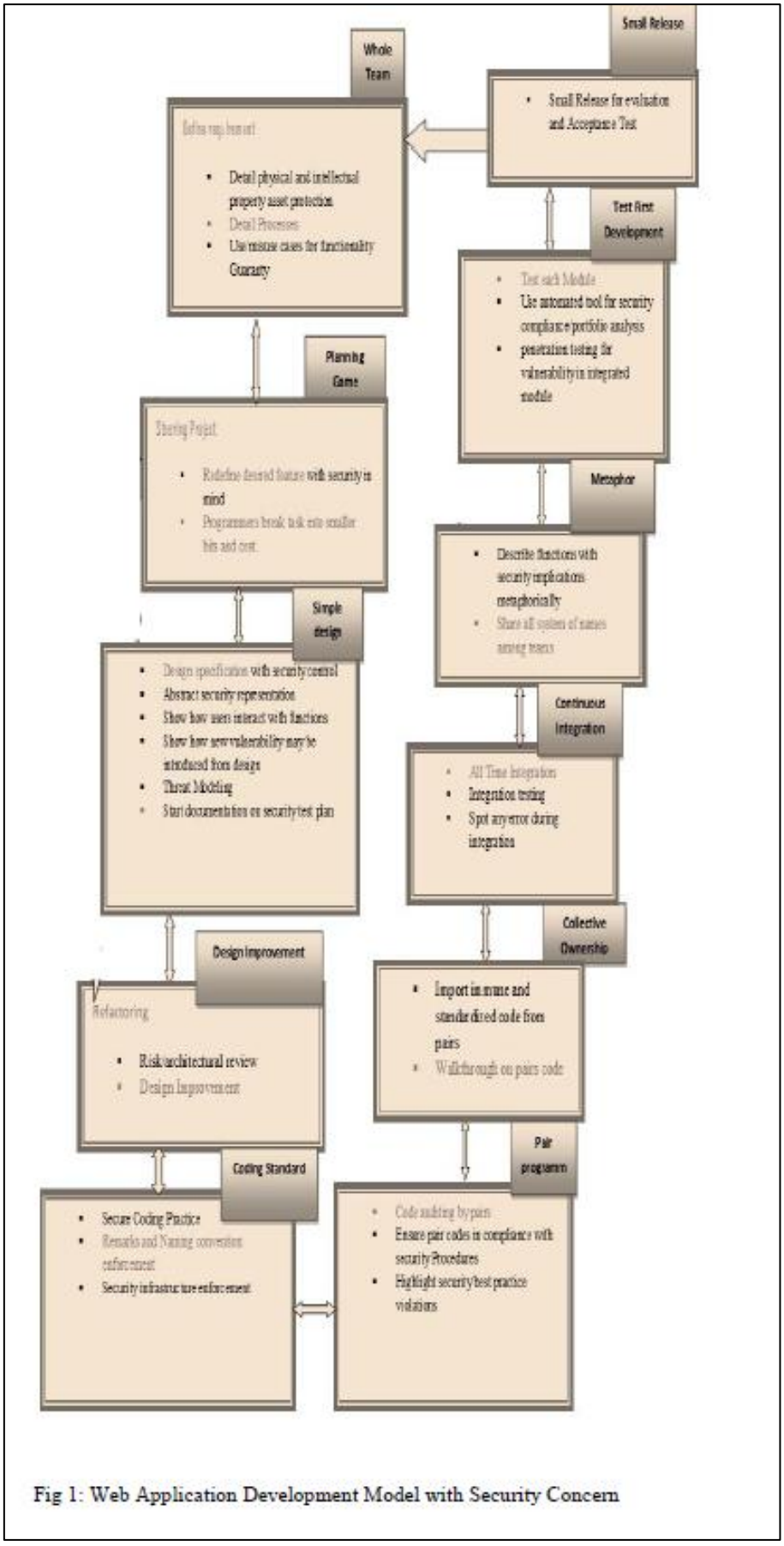


Fig 1: Web Application Development Model with Security Concern

Literature 0.C A.3: Web application Development Model with security Concerns

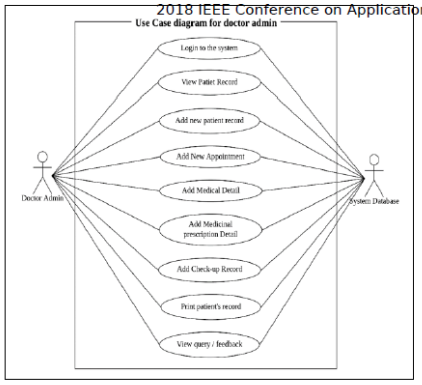


Figure 6: UML- Use Case diagram for doctor admin

The Use Case diagram of Electronic Healthcare Record System (EHRS) for doctor admin is illustrated in Figure 6. For doctor admin, they need log into the system first and register on a new account if they are new users. Doctor admin can have a full access to the patient's record. Thus, doctor

type of treatment provided by the health institution. Furthermore, the registered patient can also request for a documented medical record through the system. After getting treatment from the doctor, the patient can add query or give feedback, comments, and recommendations about the doctors, staffs and the environment of the health institution.

B. System Architecture

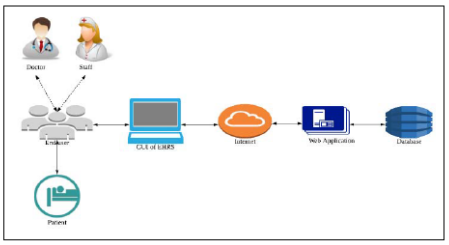


Figure 8: System Architecture of EHRS

Figure 8 shows the architecture design of EHRS where the flow of the system is briefly explained. The system architecture involves the end user, GUI of EHRS, internet,

Literature 0.D A.4:Use Case Diagram for Doctor Admin.

after viewing the patient's record, the page will redirect to add a new patient record page. In this page, the doctor will input the details of the patient's health condition. Besides, doctor admin can also have the access to view and input Medical detail, Medicinal Prescription detail, and Check-up detail.

Upon completing adding a new record for the patient, the page will redirect to the homepage of the system. The doctor will have an option whether to view feedback or review page or not. If the doctor clicked to view feedback page, then the system will display the feedback page. However, if the doctor does not want to view feedback page, the system will log out and redirect to the homepage.

c) Patient

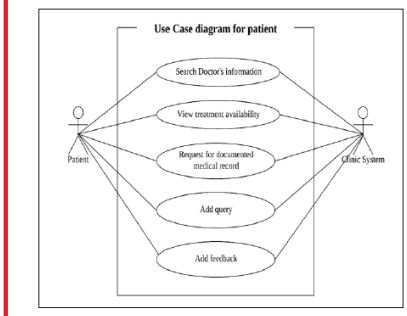


Figure 7: UML- Use Case diagram for patient

Figure 7 shows the Use Case diagram for the patient in EHRS. In this system, the patient can view information of the health institution. Before getting treatment from the doctor, the patient can view doctor's information and the

The end users are the patient, staff admin and doctor admin who will be using the system. Graphical User Interface (GUI) refers to the web browser of EHRS which is in HTML format. The system can only be used if the user has the access to internet connection. Furthermore, to enhance the design of the system, EHRS is supported by a few web applications such as CSS. The data of the system will be stored in a local database which is My SQL. EHRS is a platform where data can be transmitted across many levels of authority. Every medical detail of the patient will be updated in the system. Both patient and medical practitioners have the access to view the medical record.

IV. RESULTS AND DISCUSSIONS

A. Unit Testing

Unit testing is a method in which a module is being tested to figure out any issues occurred during the development of the project. In this project, syntax errors and logical errors are being measured. The table below shows the result of unit testing:

Table 2: The Result of Unit Testing

Unit Function	Flag	Trials	Bugs	Outcome
Login	• Username: must have "@" • Password: password must be matched • Button: print username	5	None	Username is printed
Register	• Username: must have "@" • Password: password must be matched	5	None	Username is printed

Literature 0.E A.5: Use Case diagram for patient.

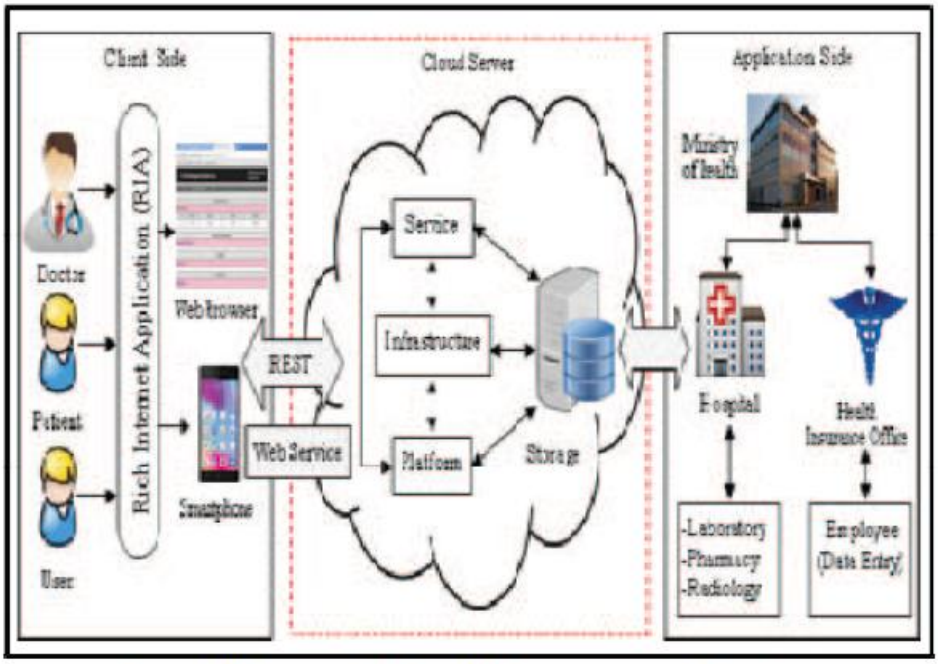


Fig.1. Proposed E-health management system

Literature 0.F A.6: Proposed System Architecture..

Figure 2 is the case diagram of some services provided by the proposed system which also shows interactions between users and system's functionality.

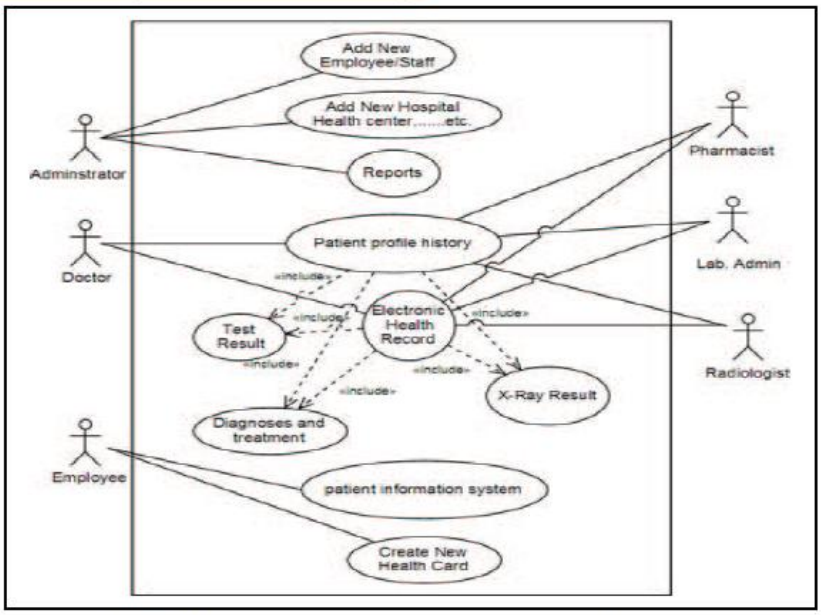
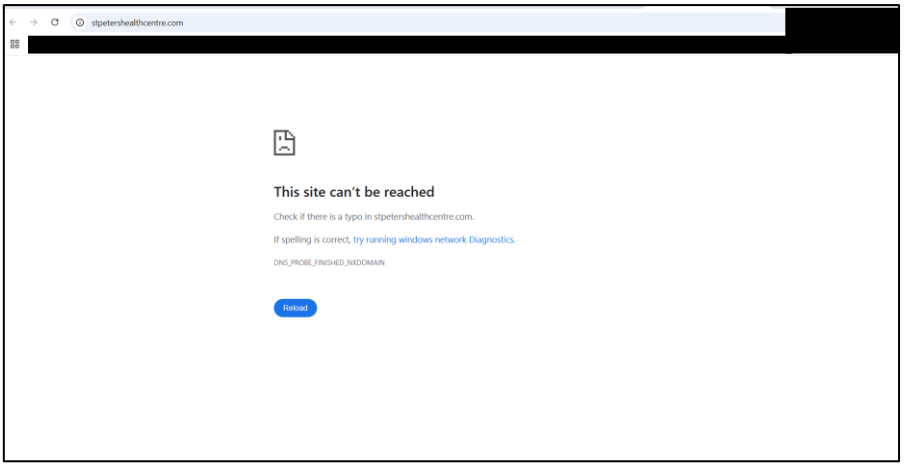


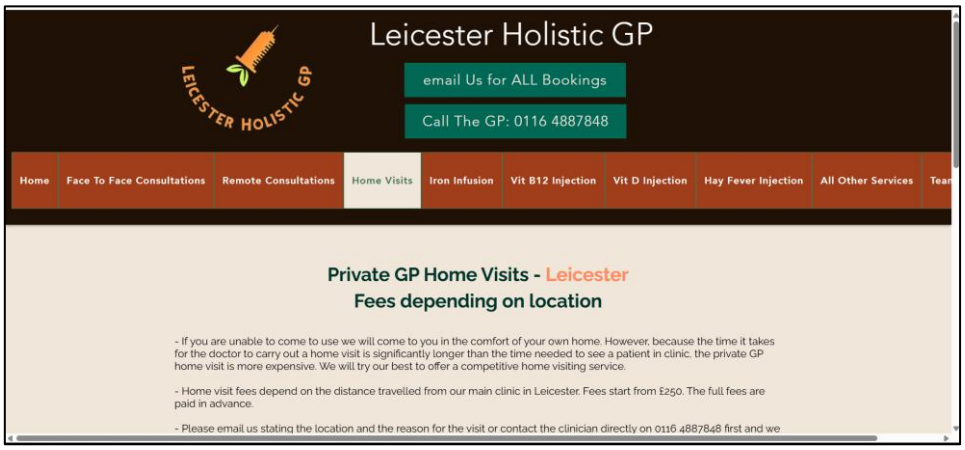
Fig.2. Use case diagrams for proposed system.

Literature 0.G A.7: Use Case Diagram

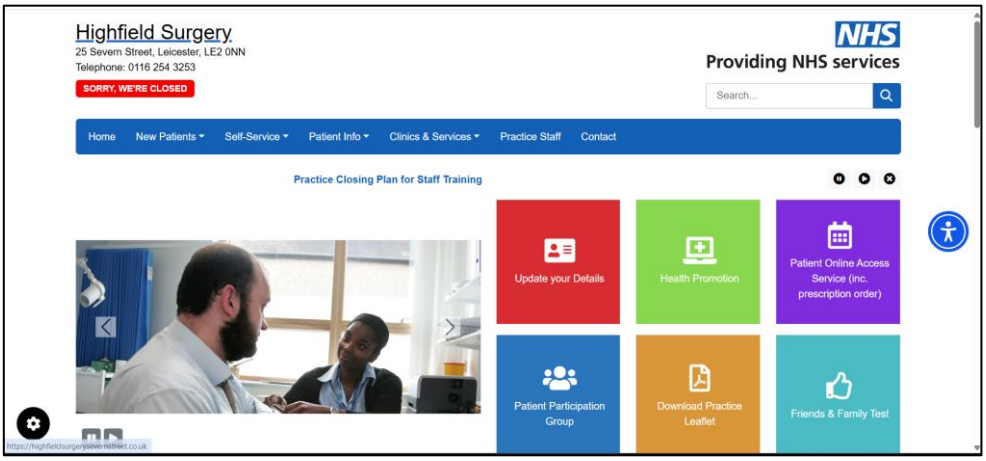
B. Existing GP Websites



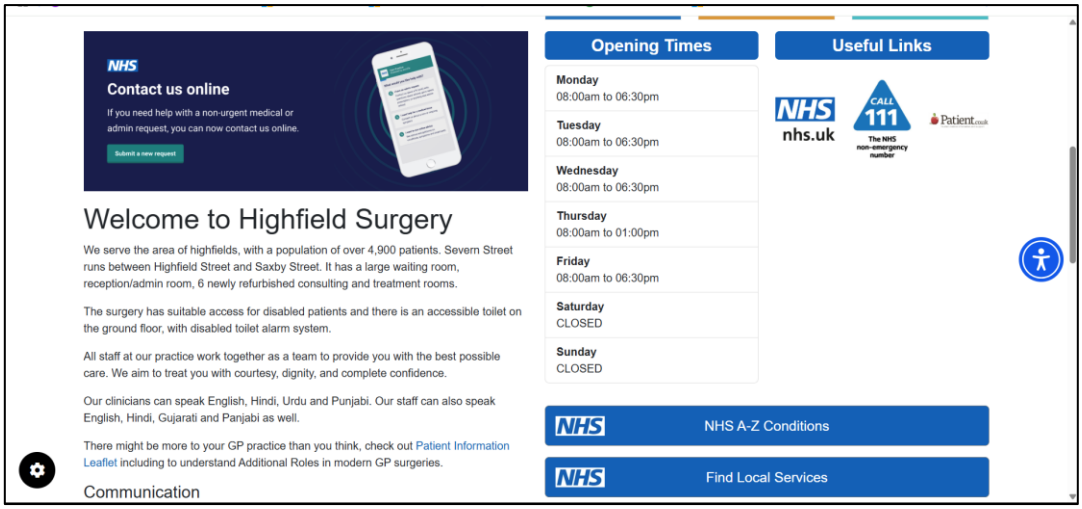
Existing Websites B.1: Queens Road Surgery.



Existing Websites B.2: Leicester Holistic GP



Existing Websites B.3: Highfield Surgery



Existing Websites B.4: Highfield Surgery Website.

C. Utility Functions.

```python
# Utility functions
def hash_password(password):
    return hashlib.sha256(password.encode()).hexdigest()

# Utility function to change access level of containers after patinet is registered
def change_container_access(container_name, public_access_level):
    try:
        container_client = blob_service_client.get_container_client(container_name)
        container_client.set_container_access_policy(public_access=public_access_level,signed_identifiers={})
        print(f"Successfully changed public access for container '{container_name}' to '{public_access_level}'.")
        return True
    except Exception as e:
        print(f"Error changing public access for container '{container_name}': {e}")
        return False


# Connection string for the Azure Blob Storage
CONNECTION_STRING = os.getenv('CONNECTION_STRING')
blob_service_client = BlobServiceClient.from_connection_string(CONNECTION_STRING)
```

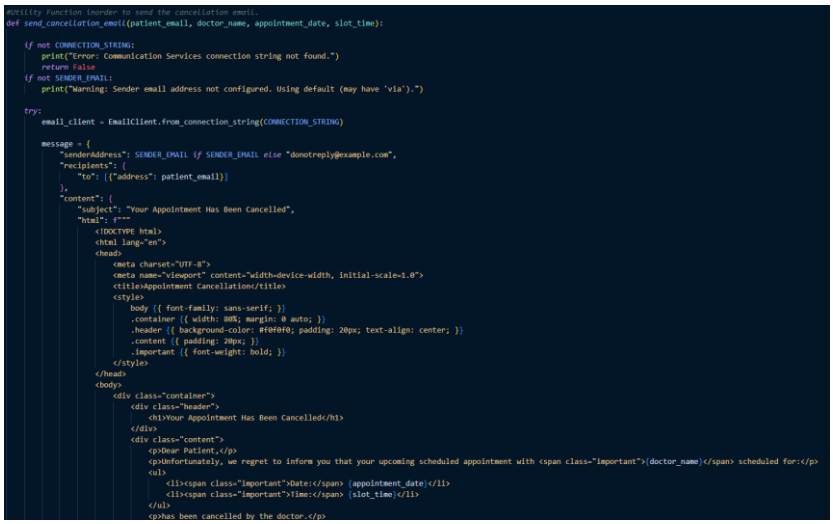
Function C.1: Container Access.

```python
# Utility function to update the payments table
def update_payment_status(stripe_checkout_session_id, status, prescription_id=None):
    conn = get_db_connection()
    if conn:
        try:

            query = """
                UPDATE payments
                SET payment_status = ?, payment_date = ?
                WHERE stripe_payment_intent_id = ?
            """
            params = (status, datetime.datetime.utcnow(), stripe_checkout_session_id)
            execute_query(conn, query, params)

        except pyodbc.Error as ex:
            sqlstate = ex.args[0]
            print(f"Database error during payment status update: {sqlstate}")
            conn.rollback()
        finally:
            conn.close()
```

Function C.2: Update Payment Session.

```
def register_doctor(conn, data, admin_id):
    cursor = conn.cursor()
    if not data['staffEmail'] or not data['staffPhone']:
        return jsonify({'message': 'Invalid email or phone number'}), 400

    if verify_staff(data['staffEmail']) != 'doctor':
        return jsonify({'message': 'Email should have .doctor'}), 403

    cursor.execute("""SELECT 1 FROM Doctor WHERE Doctor_Email = ? OR Doctor_Phone_No = ?""", (data['staffEmail'], data['staffPhone']))
    if cursor.fetchone():
        return jsonify({'message': 'Doctor Already Registered'}), 400

    hashed_password = hash_password(data['staffPassword'])

    cursor.execute("""
        INSERT INTO Doctor (Doctor_FirstName, Doctor_LastName, Doctor_Email, Doctor_Phone_No, Doctor_Registration_Number, Specialization_ID, D_Password, Registered_By_Admin)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?)
    """, (data['staffFirstName'], data['staffLastName'], data['staffEmail'], data['staffPhone'], data['staffRegistrationNumber'], data['staffSpecialization'], hashed_password, admin_id))
    conn.commit()
    return jsonify({'message': 'Doctor registered successfully'}), 201


# Nurse Registration
def register_nurse(conn, data, admin_id):
    cursor = conn.cursor()
    if not data['staffEmail'] or not data['staffPhone']:
        return jsonify({'message': 'Invalid email or phone number'}), 400

    if verify_staff(data['staffEmail']) != 'nurse':
        return jsonify({'message': 'Email should have .nurse'}), 403

    cursor.execute("""SELECT 1 FROM Nurse WHERE Nurse_Email = ? OR Nurse_Phone_No = ?""", (data['staffEmail'], data['staffPhone']))
    if cursor.fetchone():
        return jsonify({'message': 'Nurse Already Registered'}), 400

    hashed_password = hash_password(data['staffPassword'])

    cursor.execute("""
        INSERT INTO Nurse (Nurse_FirstName, Nurse_LastName, Nurse_Email, Nurse_Phone_No, Nurse_Registration_Number, Specialization_ID, N_Password, Registered_By_Admin)
        VALUES (?, ?, ?, ?, ?, ?, ?, ?)
    """, (data['staffFirstName'], data['staffLastName'], data['staffEmail'], data['staffPhone'], data['staffRegistrationNumber'], data['staffSpecialization'], hashed_password, admin_id))
    conn.commit()
    return jsonify({'message': 'Nurse registered successfully'}), 201
```

Function C.3*:* Register doctor and Register nurse.



```
#Utility Function inorder to send the cancellation email.
def send_cancellation_email(patient_email, doctor_name, appointment_date, slot_time):

    if not CONNECTION_STRING:
        print("Error: Communication Services connection string not found.")
        return False
    if not SENDER_EMAIL:
        print("Warning: Sender email address not configured. Using default (may have 'via').")

    try:
        email_client = EmailClient.from_connection_string(CONNECTION_STRING)

        message = {
            "senderAddress": SENDER_EMAIL if SENDER_EMAIL else "donotreply@example.com",
            "recipients": {
                "to": [{"address": patient_email}]
            },
            "content": {
                "subject": "Your Appointment Has Been Cancelled",
                "html": f"""
                    <!DOCTYPE html>
                    <html lang="en">
                    <head>
                        <meta charset="UTF-8">
                        <meta name="viewport" content="width=device-width, initial-scale=1.0">
                        <title>Appointment Cancellation</title>
                        <style>
                            body {{ font-family: sans-serif; }}
                            .container {{ width: 80%; margin: 0 auto; }}
                            .header {{ background-color: #f0f0f0; padding: 20px; text-align: center; }}
                            .content {{ padding: 20px; }}
                            .important {{ font-weight: bold; }}
                        </style>
                    </head>
                    <body>
                        <div class="container">
                            <div class="header">
                                <h1>Your Appointment Has Been Cancelled</h1>
                            </div>
                            <div class="content">
                                <p>Dear Patient,</p>
                                <p>Unfortunately, we regret to inform you that your upcoming scheduled appointment with <span class="important">{doctor_name}</span> scheduled for:</p>
                                <ul>
                                    <li><span class="important">Date:</span> {appointment_date}</li>
                                    <li><span class="important">Time:</span> {slot_time}</li>
                                </ul>
                                <p>has been cancelled by the doctor.</p>
```

Function C.4: Send Cancellation email

```
# Utility Functioncode .

def hash_password(password):
    return hashlib.sha256(password.encode()).hexdigest()

# Staff Verification
def verify_staff(email):
    try:
        dotIndex = email.index('.')
        atIndex = email.index('@')
        if dotIndex < atIndex:
            return email[dotIndex + 1:atIndex]
        else:
            return None
    except ValueError:
        return 'No . or @ found'
```

Function C.5.: Staff Verification.

Function C.6: Send cancellation email continuation.

D. Booking Appointment code snippet

```python
# API endpoint to  get_diseases
@patientBooking_bp.route('/get_diseases', methods=['GET'])
@jwt_required()
def get_diseases():
    try:

        with get_db_connection() as conn:
            cursor = conn.cursor()
            cursor.execute("""
                SELECT Disease_Name, Disease_ID
                FROM Disease
            """, ())

            rows = cursor.fetchall()

            if not rows:
                return jsonify({'message': 'No diseases found'}), 404

            diseases = []
            for row in rows:
                diseases.append({
                    'Disease_ID': row.Disease_ID,
                    'Disease_Name': row.Disease_Name
                })

            return jsonify(diseases), 200

    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Code snippet D.1: Get Diseases.

```python
@patientBooking_bp.route('/get_doctors_list', methods=['POST'])
@jwt_required()
def get_doctors_list():
    try:
        data = request.get_json()
        health_issue = data['health_issue']
        on_date = data['on_date']

        with get_db_connection() as conn:
            cursor = conn.cursor()

            # Get specialization ID based on the disease id
            cursor.execute("""
                SELECT Specialization_id from Disease
                WHERE Disease_id = ?
            """, (health_issue,))
            specialization_row = cursor.fetchone()

            if not specialization_row:
                return jsonify({'error': 'Specialization not found for this disease'}), 404

            specializationId = specialization_row.Specialization_id

            # Retrieve the doctors that match with the specialization id and are avaiable on the date selected by the patient.
            cursor.execute("""
                SELECT DISTINCT d.Doctor_id, d.Doctor_FirstName, d.Doctor_LastName, d.Specialization_id
                FROM Doctor d
                JOIN DoctorAvailability da ON d.Doctor_id = da.DoctorID
                WHERE d.Specialization_id = ?
                AND da.Date = ?
            """, (specializationId, on_date))
            rows = cursor.fetchall()

            doctors = []
            for row in rows:
                doctors.append({
                    'id': row.Doctor_id,
                    'firstname': row.Doctor_FirstName,
                    'lastname': row.Doctor_LastName,
                    'Specialisation': row.Specialization_id,
                    'date': on_date
                })

            return jsonify(doctors), 200

    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Code snippet D.2: Get Doctor List

```python
@patientBooking_bp.route('/get_doctor_availability/<int:doctor_id>/<string:date>', methods=['GET'])
@jwt_required()
def get_doctor_availability(doctor_id, date):
    try:
        with get_db_connection() as conn:
            cursor = conn.cursor()
            cursor.execute("""
                SELECT s.SlotID, s.StartTime, s.EndTime
                FROM Slots s
                LEFT JOIN Appointment a ON s.SlotID = a.SlotID AND a.DoctorID = ? AND a.Date = ?
                WHERE a.AppointmentID IS NULL
                AND s.SlotID IN (SELECT SlotID FROM DoctorAvailability WHERE DoctorID = ? AND Date = ?)
            """, doctor_id, date, doctor_id, date)
            rows = cursor.fetchall()
            available_slots = [{'id': row.SlotID, 'start': row.StartTime.isoformat(), 'end': row.EndTime.isoformat()} for row in rows]
            return jsonify(available_slots), 200

    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Code snippet D.3: Get doctors availability

```python
@patientBooking_bp.route('/book_appointment', methods=['POST'])
@jwt_required()
def book_appointment():
        try:
            data = request.get_json()
            doctor_id = data['doctor_id']
            patient_id = data['patient_id']
            date = data['date']
            slot_id = data['slot_id']
            disease_type = data.get('disease_type')
            disease_description = data.get('disease_description')

            if not disease_type:
                return jsonify({'error': 'Disease type is required'}), 400

            with get_db_connection() as conn:
                cursor = conn.cursor()

                cursor.execute(
                    "SELECT 1 FROM DoctorAvailability WHERE DoctorID = ? AND Date = ? AND SlotID = ?",
                    doctor_id, date, slot_id
                )

                slot_exists = cursor.fetchone()

                if not slot_exists:
                    return jsonify({'error': 'No Slot is available on this date'}), 400

                #Verifies if the Slot is already booked
                cursor.execute(
                    "SELECT 1 FROM Appointment WHERE DoctorID = ? AND Date = ? AND SlotID = ?",
                    doctor_id, date, slot_id
                )
                if cursor.fetchone():
                    return jsonify({'error': 'Slot is already booked'}), 400

                cursor.execute(
                """
                INSERT INTO Appointment (DoctorID, PatientID, Date, SlotID, DiseaseType, DiseaseDescription)
                VALUES (?, ?, ?, ?, ?, ?)
                """,
                doctor_id, patient_id, date, slot_id, disease_type, disease_description
                )
                conn.commit()
                return jsonify({'message': 'Appointment booked successfully'}), 201

                #If incase, two patient try to book the appointment at the same time for the same doctor and
        except pyodbc.IntegrityError:
```

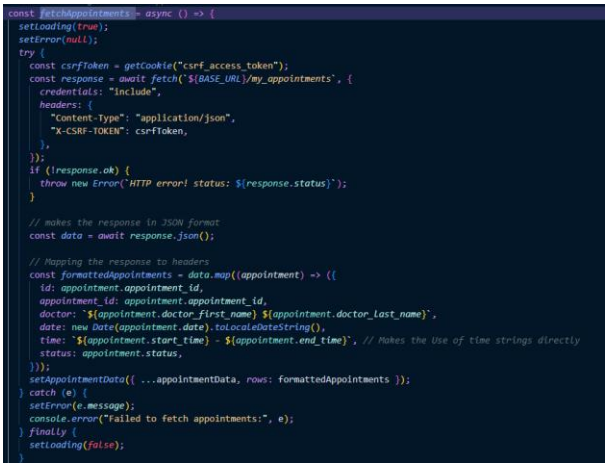
Code snippet D.4: Book Appointments.

```python
@patientBooking_bp.route('/my_appointments', methods=['GET'])
@jwt_required()
def get_patient_appointments():
        try:
            patient_email = get_jwt_identity()
            print(patient_email)
            with get_db_connection() as conn:
                cursor = conn.cursor()
                cursor.execute("""
                    SELECT
                        a.AppointmentID,
                        d.Doctor_FirstName,
                        d.Doctor_LastName,
                        d.Specialization_id,
                        a.Date,
                        s.StartTime,
                        s.EndTime
                    FROM Appointment a
                    JOIN Doctor d ON a.DoctorID = d.Doctor_id
                    JOIN Slots s ON a.SlotID = s.SlotID
                    JOIN Patient p ON a.PatientID = p.P_id
                    WHERE p.Email_Id = ?
                    ORDER BY a.Date, s.StartTime
                """, (patient_email,))
                rows = cursor.fetchall()

                appointments = []
                for row in rows:
                    appointment_date = row.Date  # From the datetime object, only the date will be extracted.
                    status = "Completed" if appointment_date < date.today() else "Scheduled"
                    appointments.append({
                        'appointment_id': row.AppointmentID,
                        'doctor_first_name': row.Doctor_FirstName,
                        'doctor_last_name': row.Doctor_LastName,
                        'doctor_specialization': row.Specialization_id,
                        'date': row.Date.isoformat(),
                        'start_time': row.StartTime.isoformat(),
                        'end_time': row.EndTime.isoformat(),
                        'status':status
                    })

                return jsonify(appointments), 200

        except Exception as e:
            return jsonify({'error':str(e)}),500
```

Code snippet D.5: My Appointments.

```python
@patientBooking_bp.route('/cancel_appointment/<int:appointment_id>', methods=['DELETE'])
@jwt_required()
def cancel_appointment(appointment_id):
        try:
            with get_db_connection() as conn:
                cursor = conn.cursor()


                cursor.execute("SELECT 1 FROM Appointment WHERE AppointmentID = ?", appointment_id)
                if not cursor.fetchone():
                    return jsonify({'error': 'Appointment not found'}), 404

                cursor.execute("DELETE FROM Appointment WHERE AppointmentID = ?", appointment_id)
                conn.commit()

                return jsonify({'message': 'Appointment cancelled successfully'}), 200

        except Exception as e:
            return jsonify({'error': str(e)}), 500
```

Code snippet D.6: Delete Appointments.



```
const fetchAppointments = async () => {
  setLoading(true);
  setError(null);
  try {
    const csrfToken = getCookie("csrf_access_token");
    const response = await fetch(`${BASE_URL}/my_appointments`, {
      credentials: "include",
      headers: {
        "Content-Type": "application/json",
        "X-CSRF-TOKEN": csrfToken,
      },
    });
    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    // makes the response in JSON format
    const data = await response.json();

    // Mapping the response to headers
    const formattedAppointments = data.map((appointment) => ({
      id: appointment.appointment_id,
      appointment_id: appointment.appointment_id,
      doctor: `${appointment.doctor_first_name} ${appointment.doctor_last_name}`,
      date: new Date(appointment.date).toLocaleDateString(),
      time: `${appointment.start_time} - ${appointment.end_time}`, // Makes the Use of time strings directly
      status: appointment.status,
    }));
    setAppointmentData({ ...appointmentData, rows: formattedAppointments });
  } catch (e) {
    setError(e.message);
    console.error("Failed to fetch appointments:", e);
  } finally {
    setLoading(false);
  }
```

Code snippet D.7: Fetch Appointment function.

```
const handleCancelAppointment = async (appointmentId) => {
  // API call for cancel appointment
  try {
    const csrfToken = getCookie("csrf_access_token");
    const response = await fetch(
      `${BASE_URL}/cancel_appointment/${appointmentId}`,
      {
        method: "DELETE",
        credentials: "include",

        headers: {
          "Content-Type": "application/json",
          "X-CSRF-TOKEN": csrfToken,
        },
      }
    );

    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    setAppointmentData((prevData) => ({
      ...prevData,
      rows: prevData.rows.filter(
        (row) => row.appointment_id !== appointmentId
      ),
    }));
    console.log(`Appointment ${appointmentId} canceled successfully`);
  } catch (error) {
    console.error("Failed to cancel appointment:", error);
  }
};
```

Code snippet D.8: Cancel Appointment Function.

```
const handleDateChange = (date) => {
  setSelectedDate(date);
  // API call to get the list of doctors available for specific diseases.
  if (selectedProblem) {
    const csrfToken = getCookie("csrf_access_token");
    fetch(`${BASE_URL}/get_doctors_list`, {
      method: "POST",
      credentials: "include",
      headers: {
        "Content-Type": "application/json",
        "X-CSRF-TOKEN": csrfToken,
      },
      body: JSON.stringify({
        health_issue: selectedProblem,
        on_date: formatDate(date), // Format: YYYY-MM-DD
      }),
    })
      .then((response) => response.json())
      .then((data) => {
        setAvailableDoctors(data);
        setStep(3); // Step 3: Doctor List.
      })
      .catch((error) => {
        console.error("Error fetching doctors:", error);
      });
  }
};
```

Code snippet D.9: Handle date change function.

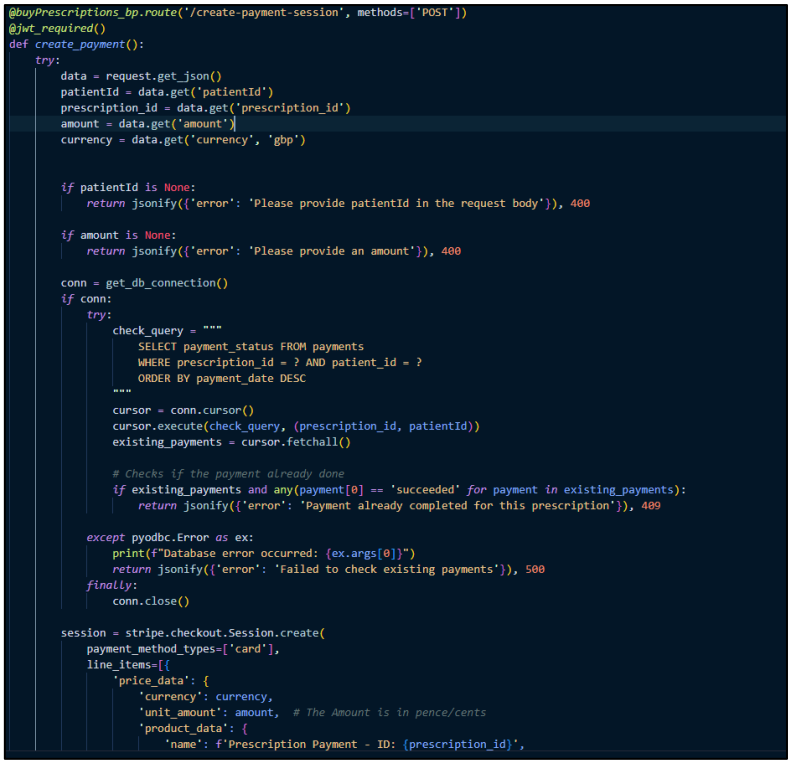
```
const handleBookDoctor = (timeSlot) => {
  // API call to book the appointment for the patient.
  const csrfToken = getCookie("csrf_access_token");
  fetch(`${BASE_URL}/book_appointment`, {
    method: "POST",
    credentials: "include",
    headers: {
      "Content-Type": "application/json",
      "X-CSRF-TOKEN": csrfToken,
    },
    body: JSON.stringify({
      doctor_id: doctor.id,
      patient_id: patientInfo.id || patientId,
      date: formatDate(selectedDate),
      slot_id: timeSlot.id,
      disease_type: selectedProblem,
      disease_description: diseaseDescription,
    }),
  })
    .then((response) => response.json())
    .then((data) => {
      onBookingComplete();
      setStep(1); // Reset to step 1
      onHide(); // Close the book appointment form once completed.
    })
    .catch((error) => {
      console.error("Error booking appointment:", error);
    });
};
```

Code snippet D.10: Handle book doctor function.

E. Buy Prescriptions code snippet

```
@buyPrescriptions_bp.route('/stripe-webhook', methods=['POST'])
def stripe_webhook():
    payload = request.get_data(as_text=True)
    sig_header = request.headers.get('stripe-signature')
    try:
        event = stripe.Webhook.construct_event(
            payload, sig_header, STRIPE_WEBHOOK_SECRET
        )
    except ValueError as e:
        return make_response('Invalid payload', 400)
    except stripe.error.SignatureVerificationError as e:
        return make_response('Invalid signature', 400)

    # Checks the success or fail of stripe event type
    if event['type'] == 'checkout.session.completed':
        session = event['data']['object']
        prescription_id = session.get('client_reference_id')
        stripe_payment_intent_id = session.get('id')

        print(f"Checkout Session for Prescription ID {prescription_id} completed! Session ID: {stripe_payment_intent_id}")
        update_payment_status(stripe_payment_intent_id, 'succeeded', prescription_id)

    elif event['type'] == 'checkout.session.expired':
        session = event['data']['object']
        prescription_id = session.get('client_reference_id')
        stripe_payment_intent_id = session.get('id')
        print(f"Checkout Session for Prescription ID {prescription_id} expired! Session ID: {stripe_payment_intent_id}")
        update_payment_status(stripe_payment_intent_id, 'expired', prescription_id)


    return make_response('Webhook received', 200)
```

Code snippet E.1: Stripe Webhooks



```python
@buyPrescriptions_bp.route('/create-payment-session', methods=['POST'])
@jwt_required()
def create_payment():
    try:
        data = request.get_json()
        patientId = data.get('patientId')
        prescription_id = data.get('prescription_id')
        amount = data.get('amount')
        currency = data.get('currency', 'gbp')


        if patientId is None:
            return jsonify({'error': 'Please provide patientId in the request body'}), 400

        if amount is None:
            return jsonify({'error': 'Please provide an amount'}), 400

        conn = get_db_connection()
        if conn:
            try:
                check_query = """
                    SELECT payment_status FROM payments
                    WHERE prescription_id = ? AND patient_id = ?
                    ORDER BY payment_date DESC
                """
                cursor = conn.cursor()
                cursor.execute(check_query, (prescription_id, patientId))
                existing_payments = cursor.fetchall()

                # Checks if the payment already done
                if existing_payments and any(payment[0] == 'succeeded' for payment in existing_payments):
                    return jsonify({'error': 'Payment already completed for this prescription'}), 409

            except pyodbc.Error as ex:
                print(f"Database error occurred: {ex.args[0]}")
                return jsonify({'error': 'Failed to check existing payments'}), 500
            finally:
                conn.close()

        session = stripe.checkout.Session.create(
            payment_method_types=['card'],
            line_items=[{
                'price_data': {
                    'currency': currency,
                    'unit_amount': amount,  # The Amount is in pence/cents
                    'product_data': {
                        'name': f'Prescription Payment - ID: {prescription_id}',
```

Code snippet E.2: Create Payment Session.

```python
def create_payment():
        session = stripe.checkout.Session.create(
            payment_method_types=['card'],
            line_items=[{
                'price_data': {
                    'currency': currency,
                    'unit_amount': amount,  # The Amount is in pence/cents
                    'product_data': {
                        'name': f'Prescription Payment - ID: {prescription_id}',
                    },
                },
                'quantity': 1,
            }],
            mode='payment',
            success_url=f'{request.headers.get("Origin")}/payment-success?session_id={{CHECKOUT_SESSION_ID}}',
            cancel_url=f'{request.headers.get("Origin")}/payment-cancel',
            client_reference_id=prescription_id, # have provided prescription id as client reference id
        )

        conn = get_db_connection()
        if conn:
            try:
                query = """
                    INSERT INTO payments (patient_id, prescription_id,  stripe_payment_intent_id, amount, currency, payment_date, payment_status)
                    VALUES (?, ?, ?, ?, ?, ?, ?)
                """
                params = (patientId, prescription_id,  session.id, amount, currency, datetime.datetime.utcnow(), 'pending')

                execute_query(conn, query, params)

            except pyodbc.Error as ex:
                sqlstate = ex.args[0]
                print(f"Database error occurred: {sqlstate}")
                conn.rollback()
                return jsonify({'error': 'Failed to record payment intent'}), 500
            finally:
                conn.close()
        else:
            return jsonify({'error': 'Failed to connect to the database'}), 500

        return jsonify({'sessionId': session.id}), 200
    except stripe.error.StripeError as e:

        return jsonify({'error': f'Stripe error: {str(e)}'}), 500
    except Exception as e:

        return jsonify({'error': str(e)}), 500
```

Code snippet E.3: Create payment session continuation.

F. Admin Routes code snippet

```python
@adminRoutes_bp.route('/getNurses', methods=['GET'])
@jwt_required()
def get_nurseslist():
    admin_email = get_jwt_identity()

    if not admin_email:
        return jsonify({'message': 'Missing admin email in token'}), 400

    if verify_staff(admin_email) != 'admin':
        return jsonify({'message': 'Admin level rights required'}), 403

    conn = get_db_connection()
    if not conn:
        return jsonify({'message': 'Database connection failed'}), 500

    try:
        cursor = conn.cursor()
        cursor.execute("""
            SELECT n.*, s.Specialization_Name
            FROM Nurse n
            LEFT JOIN Specialization s ON n.Specialization_ID = s.Specialization_id
        """)
        nurses = cursor.fetchall()

        if not nurses:
            return jsonify({'message': 'No nurses found'}), 404

        columns = [desc[0] for desc in cursor.description]
        nurse_list = []

        for row in nurses:
            nurse = dict(zip(columns, row))
            nurse.pop('N_Password', None)

            nurse_list.append({
                'Name': f"{nurse.get('Nurse_FirstName', '')} {nurse.get('Nurse_LastName', '')}".strip(),
                'Email': nurse.get('Nurse_Email', ''),
                'Phone': nurse.get('Nurse_Phone_No', ''),
                'RegistrationNumber': nurse.get('Nurse_Registration_Number', ''),
                'Specialization': nurse.get('Specialization_Name', 'N/A')
            })

        return jsonify(nurse_list), 200

    except Exception as e:
        return jsonify({'message': f'Error: {str(e)}'}), 500
    finally:
        conn.close()
```

Code Snippet F.1: Get Nurses

```python
@adminRoutes_bp.route('/getDoctors', methods=['GET'])
@jwt_required()
def get_doctorsList():
    admin_email = get_jwt_identity()

    if not admin_email:
        return jsonify({'message': 'Missing admin email in token'}), 400

    if verify_staff(admin_email) != 'admin':
        return jsonify({'message': 'Admin level rights required'}), 403

    conn = get_db_connection()
    if not conn:
        return jsonify({'message': 'Database connection failed'}), 500

    try:
        cursor = conn.cursor()
        cursor.execute("""
            SELECT d.*, s.Specialization_Name
            FROM Doctor d
            LEFT JOIN Specialization s ON d.Specialization_ID = s.Specialization_id
        """)
        doctors = cursor.fetchall()

        if not doctors:
            return jsonify({'message': 'No doctors found'}), 404

        columns = [desc[0] for desc in cursor.description]
        doctor_list = []

        for row in doctors:
            doctor = dict(zip(columns, row))
            doctor.pop('D_Password', None)

            doctor_list.append({
                'Name': f"{doctor.get('Doctor_FirstName', '')} {doctor.get('Doctor_LastName', '')}".strip(),
                'Email': doctor.get('Doctor_Email', ''),
                'Phone': doctor.get('Doctor_Phone_No', ''),
                'RegistrationNumber': doctor.get('Doctor_Registration_Number', ''),
                'Specialization': doctor.get('Specialization_Name', 'N/A')
            })

        return jsonify(doctor_list), 200

    except Exception as e:
        return jsonify({'message': f'Error: {str(e)}'}), 500
    finally:
        conn.close()
```

Code Snippet F.2: Get Doctors

```python
@adminRoutes_bp.route('/getPatients',methods=['GET'])
@jwt_required()
def getPatientsList():
    admin_Email = get_jwt_identity()

    if not all([admin_Email]):
        return jsonify({'message': 'Missing email or password'}), 400

    if verify_staff(admin_Email) != 'admin':
        return jsonify({'message': "Admin level rights required for staff registration"}), 403

    conn = get_db_connection()
    if conn:
        try:
            cursor = conn.cursor()

            cursor.execute("""SELECT P_id,P_FirstName,P_LastName,Gender,DOB,Email_Id,Phone_No,StreetAddress,City,Postcode FROM Patient """, )
            #Returns a list of tuples
            patient_List = cursor.fetchall()
            if not patient_List:
                return jsonify({'message': 'No Patients Found'}), 400

            patients = []
            #conversion of data from tuples to dictionary is been done
            columns = [desc[0] for desc in cursor.description]
            for row in patient_List:
                patient = dict(zip(columns, row))
                patients.append(patient)

            return jsonify(patients), 200
        except Exception as e:
            return jsonify({'message': f'Error: {e}'}), 500
        finally:
            conn.close()
    else:
        return jsonify({'message': 'Database connection failed'}), 500
```

Code Snippet F.3: Get patients.

```python
@adminRoutes_bp.route('/staff/registration',methods=['POST'])
@jwt_required()
def createStaff():
    data = request.get_json()
    admin_Email = get_jwt_identity()
    staffType = data.get('staffType')
    staffFirstName = data.get('staffFirstName')
    staffLastName = data.get('staffLastName')
    staffEmail = data.get('staffEmail')
    staffPhone = data.get('staffPhone')
    staffRegistrationNumber = data.get('staffRegistrationNumber')
    staffSpecialization =  data.get('staffSpecialization')
    staffPassword = data.get('staffPassword')

    if not all([admin_Email,staffType,staffFirstName,staffLastName,staffEmail,staffPhone,staffRegistrationNumber,staffSpecialization,staffPassword]):
        return jsonify({'message': 'Missing email or password'}), 400


    if verify_staff(admin_Email) != 'admin':
        return jsonify({'message': 'Admin level rights required for staff registration'}), 403

    conn = get_db_connection()
    if conn:
        try:
            cursor = conn.cursor()

            cursor.execute("""SELECT Specialization_ID FROM Specialization WHERE Specialization_ID = ?""", (staffSpecialization,))
            specialization_row = cursor.fetchone()
            if not specialization_row:
                return jsonify({'message': 'Invalid specialization ID'}), 400


            cursor.execute("""SELECT Admin_id FROM Admin WHERE Admin_Email = ?""", (admin_Email,))
            admin_row = cursor.fetchone()
            if not admin_row:
                return jsonify({'message': 'Cannot find Admin id'}), 400
            admin_id = admin_row[0]

            if data['staffType'] == 'Doctor':
                return register_doctor(conn, data, admin_id)
            elif data['staffType'] == 'Nurse':
                return register_nurse(conn, data, admin_id)
            else:
                return jsonify({'message': 'Invalid staff type'}), 400
        except Exception as e:
            return jsonify({'message': f'Error: {e}'}), 500
        finally:
            conn.close()
```

Code Snippet F.4: Staff Registration.

```python
@adminRoutes_bp.route('/gp-patient/registration', methods=['POST'])
@jwt_required()
def create_gp_patient():
    data = request.get_json()
    admin_email =  get_jwt_identity()
    patient_first_name = data.get('patientFirstName')
    patient_last_name = data.get('patientLastName')
    patient_email = data.get('patientEmail')
    patient_phone = data.get('patientPhone')
    gender = data.get('gender')
    dob = data.get('dob')
    patient_password = data.get('patientPassword')
    street_address = data.get('streetAddress')
    city = data.get('city')
    postcode = data.get("postcode")

    if not all([ admin_email, patient_first_name, patient_last_name, patient_email, patient_phone, gender, dob, patient_password,street_address,city,postcode]):
        return jsonify({'message': 'Missing required fields'}), 400

    if verify_staff(admin_email) != 'admin':
        return jsonify({'message': 'Admin level rights required for patient registration'}), 403

    conn = get_db_connection()
    if conn:
        try:
            cursor = conn.cursor()

            # Verifies if the Patient already exist
            cursor.execute("""SELECT Admin_id FROM Admin WHERE Admin_Email = ?""", (admin_email,))
            admin_row = cursor.fetchone()
            if not admin_row:
                return jsonify({'message': 'Could not retrieve admin ID from the authenticated user'}), 401
            admin_id = admin_row[0]

            cursor.execute("""
                SELECT 1 FROM Patient WHERE Email_Id = ? OR Phone_No = ?
            """, (patient_email, patient_phone))
            existing_patient = cursor.fetchone()

            if existing_patient:
                return jsonify({'message': 'Patient with this email or phone number already exists'}), 400

            hashed_password = hash_password(patient_password)

            # Conversion of DOB string to Date object
            try:
                dob_date = datetime.datetime.strptime(dob, '%Y-%m-%d').date()
```

Code Snippet F.5: Patient Registration.

G. Doctor Manage Appointments code snippet

```python
@doctorManageAppointments_bp.route('/get_patient_bookings/<string:date>', methods=['GET'])
@jwt_required()
def get_doctor_appointments_by_date(date):
    doctorEmail = get_jwt_identity()

    try:
        with get_db_connection() as conn:
            cursor = conn.cursor()


            cursor.execute("SELECT Doctor_id FROM Doctor WHERE Doctor_Email = ?", (doctorEmail,))
            doctor_row = cursor.fetchone()
            if not doctor_row:
                return jsonify({'error': 'Doctor not found'}), 404

            doctor_id = doctor_row.Doctor_id


            cursor.execute("""
                SELECT d.Disease_Name, a.AppointmentID, a.PatientID,
                       p.P_FirstName, p.P_LastName, s.StartTime, s.EndTime,a.DiseaseDescription
                FROM Appointment a
                JOIN Patient p ON a.PatientID = p.P_id
                JOIN Slots s ON a.SlotID = s.SlotID
                JOIN Disease d ON a.DiseaseType = d.Disease_id
                WHERE a.Date = ? AND a.DoctorID = ?
            """, (date, doctor_id))
            rows = cursor.fetchall()

            appointments = []
            for row in rows:
                appointments.append({
                    'disease_name': row.Disease_Name,
                    'appointment_id': row.AppointmentID,
                    'patient_id': row.PatientID,
                    'patient_firstname': row.P_FirstName,
                    'patient_lastname': row.P_LastName,
                    'start_time': row.StartTime.isoformat(),
                    'end_time': row.EndTime.isoformat(),
                    'disease_description':row.DiseaseDescription
                })

            return jsonify(appointments), 200

    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Code snippet G.1: Get patient Booking.

```python
@doctorManageAppointments_bp.route('/cancel_doctor_appointment/<int:appointment_id>', methods=['DELETE'])
@jwt_required()
def cancel_doctor_appointment(appointment_id):
    try:
        with get_db_connection() as conn:
            cursor = conn.cursor()

            # Get appointment details before deleting
            cursor.execute("""
                SELECT a.PatientID, p.Email_Id, d.Doctor_FirstName, d.Doctor_LastName, a.Date, s.StartTime, s.EndTime
                FROM Appointment a
                JOIN Patient p ON a.PatientID = p.P_id
                JOIN Doctor d ON a.DoctorID = d.Doctor_id
                JOIN Slots s ON a.SlotID = s.SlotID
                WHERE a.AppointmentID = ?
            """, appointment_id)
            appointment_details = cursor.fetchone()

            if not appointment_details:
                return jsonify({'error': 'Appointment not found'}), 404

            patient_id, patient_email, doctor_firstname, doctor_lastname, appointment_date, slot_start, slot_end = appointment_details

            # Delete the appointment
            cursor.execute("DELETE FROM Appointment WHERE AppointmentID = ?", appointment_id)
            conn.commit()

            # Send cancellation email
            slot_time = f"{slot_start.isoformat()} - {slot_end.isoformat()}"
            email_sent = send_cancellation_email(patient_email, f"{doctor_firstname} {doctor_lastname}", appointment_date.isoformat(), slot_time)

            if email_sent:
                return jsonify({'message': 'Appointment cancelled successfully and cancellation email sent'}), 200
            else:
                return jsonify({'message': 'Appointment cancelled successfully, but email sending failed'}), 200

    except Exception as e:
        return jsonify({'error':str(e)}),500
```

Code snippet G.2: Cancel Appointment by doctors.

```
const handleDateChange = (date) => {
  setSelectedDate(date);
  setLoading(true);
  setError(null); // Reset the previous errors

  // The date format 'yyyy-mm-dd' inorder to match the backend API.
  const formattedDate = date.toLocaleDateString("en-CA");

  // API call to get patient booking for specific date
  const csrfToken = getCookie("csrf_access_token");
  fetch(`${BASE_URL}/get_patient_bookings/${formattedDate}`, {
    method: "GET",
    credentials: "include",
    headers: {
      "Content-Type": "application/json",
      "X-CSRF-TOKEN": csrfToken,
    },
  })
    .then((response) => {
      if (!response.ok) {
        throw new Error("Failed to fetch appointments");
      }
      return response.json();
    })
    .then((data) => {
      setAppointments(data);
      setLoading(false);
    })
    .catch((err) => {
      setError(err.message);
      setLoading(false);
    });
};
```

Code snippet G.3: Handle date change function.

```
const handleCancelAppointment = (appointmentId) => {
  setCancelLoading(appointmentId);
  setCancelError(null);

  //API endpoint to cancel the appointmen t
  const csrfToken = getCookie("csrf_access_token");
  fetch(`${BASE_URL}/cancel_doctor_appointment/${appointmentId}`, {
    method: "DELETE",
    credentials: "include",
    headers: {
      "Content-Type": "application/json",
      "X-CSRF-TOKEN": csrfToken,
    },
  })
    .then((response) => {
      if (!response.ok) {
        throw new Error("Failed to cancel appointment");
      }
      return response.json();
    })
    .then((data) => {
      console.log(data.message);

      // Upon cancellation, updates the appointments list
      setAppointments((prevAppointments) =>
        prevAppointments.filter(
          (appointment) => appointment.appointment_id !== appointmentId
        )
      );
      setCancelLoading(null);
    })
    .catch((err) => {
      console.error("Error cancelling appointment:", err);
      setCancelError(err.message);
      setCancelLoading(null);
    });
};
```

Code snippet G.4: Handle cancel appointment.

H. Prescription code snippet

```python
@doctorPrescriptions_bp.route('/medicines', methods=['GET'])
@jwt_required()
def get_medicines():
    try:
        with get_db_connection() as conn:
            cursor = conn.cursor()
            query = """
                SELECT medicine_id, medicine_name, disease_id, dosage, form, manufacturer, side_effects, price
                FROM medicine
            """
            cursor.execute(query,)
            columns = [column[0] for column in cursor.description]
            medicines_list = []
            for row in cursor.fetchall():
                medicines_list.append({
                    'medicine_id': row.medicine_id,
                    'medicine_name': row.medicine_name,
                    'disease_id': row.disease_id,
                    'dosage': row.dosage,
                    'form': row.form,
                    'manufacturer': row.manufacturer,
                    'side_effects': row.side_effects,
                    'price': float(row.price) if row.price is not None else None
                })
            return jsonify(medicines_list)
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Code snippet H.1: Get medicines.

```python
@doctorPrescriptions_bp.route('/pharmacies', methods=['GET'])
@jwt_required()
def get_pharmacies():
    try:
        with get_db_connection() as conn:
            cursor = conn.cursor()
            query = """
                SELECT pharmacy_id, name, street, city, postcode, contact_number, opening_time, closing_time
                FROM pharmacy
            """
            cursor.execute(query)
            pharmacies_list = []
            for row in cursor.fetchall():
                pharmacies_list.append({
                    'pharmacy_id': row.pharmacy_id,
                    'name': row.name,
                    'street': row.street,
                    'city': row.city,
                    'postcode': row.postcode,
                    'contact_number': row.contact_number,
                    'opening_time': str(row.opening_time),
                    'closing_time': str(row.closing_time)
                })
            return jsonify(pharmacies_list)
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Code snippet H.2: Get Pharmacies.

```python
@doctorPrescriptions_bp.route('/patients/verify', methods=['GET'])
@jwt_required()
def search_patient():
    first_name = request.args.get('first_name')
    last_name = request.args.get('last_name')
    dob = request.args.get('dob')

    if not all([first_name, last_name, dob]):
        return jsonify({'error': 'first_name, last_name, and dob are required'}), 400

    try:
        with get_db_connection() as conn:
            cursor = conn.cursor()
            query = """
                SELECT P_id, P_FirstName, P_LastName, DOB, StreetAddress, City, Postcode
                FROM Patient
                WHERE P_FirstName = ? AND P_LastName = ? AND DOB = ?
            """
            cursor.execute(query, (first_name, last_name, dob))
            row = cursor.fetchone()

            if row:
                result = {
                    'p_id' : row.P_id,
                    'first_name': row.P_FirstName,
                    'last_name': row.P_LastName,
                    'dob': row.DOB.strftime('%Y-%m-%d'),
                    'street': row.StreetAddress,
                    'city' : row.City,
                    'postcode' : row.Postcode
                }
                return jsonify(result)
            else:
                return jsonify({'message': 'Patient not found'}), 404
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Code snippet H.3: Verify Patient.

```python
@doctorPrescriptions_bp.route('/providePrescription', methods=['POST'])
@jwt_required()
def providePrescription():
    data = request.get_json()
    doctorEmail = get_jwt_identity()
    patientId = data.get('patientId')
    prescribedMedicineList = data.get('medicines')
    pharmacyName = data.get('pharmacyName')
    collectionMethod = data.get('collectionMethod')
    dateProvided = data.get('dateProvided')
    PatientType = data.get('patientType')

    if not ([PatientType,dateProvided, doctorEmail, patientId, prescribedMedicineList, pharmacyName, collectionMethod]):
        return jsonify({'message': 'Details missing'}), 400

    if not isinstance(prescribedMedicineList, list) or not prescribedMedicineList:
        return jsonify({'message': 'Medicine list must be a non-empty list'}), 400

    try:
        with get_db_connection() as conn:
            cursor = conn.cursor()

            # Get patients first & last name + dob
            cursor.execute("""
                SELECT P_FirstName, P_LastName
                FROM Patient
                WHERE P_id = ?
            """, (patientId,))
            row = cursor.fetchone()
            if not row:
                return jsonify({'message': 'Patient Not Found'}), 400

            # Map medicine names to respective prices either student / normal from the  medicine table
            cursor.execute("SELECT medicine_name, price, student_price FROM Medicine")
            medicine_price_map = {
                name: {
                    'price': float(price),
                    'student_price': float(student_price)
                }
                for name, price, student_price in cursor.fetchall()
            }

        # Calculate Total price
        total_price = 0.0
        enriched_medicine_list = []
```

Code snippet H.4: Provide Prescription

Code snippet H.5: Provide Prescription Continuation.

I. Staff Authentication code snippet

```python
@staffauth_bp.route('/get_specializations', methods=['GET'])
@jwt_required()
def get_specializations():
    try:
        # Retreives the type of staff(Doctor/Nurse) from the query parameter
        staff_type = request.args.get('staff_type')

        if staff_type not in ['Doctor', 'Nurse']:
            return jsonify({'error': 'Invalid staff type. Please choose either "Doctor" or "Nurse".'}), 400

        with get_db_connection() as conn:
            cursor = conn.cursor()

            if staff_type == 'Doctor':
                cursor.execute("""
                    SELECT Specialization_ID, Specialization_Name
                    FROM Specialization
                """)
            elif staff_type == 'Nurse':
                cursor.execute("""
                    SELECT Specialization_ID, Specialization_Name
                    FROM Nurse_Specialization

                """)

            rows = cursor.fetchall()

            specializations = []
            for row in rows:
                specializations.append({
                    'Specialization_ID': row.Specialization_ID,
                    'Specialization_Name': row.Specialization_Name
                })

            return jsonify(specializations), 200

    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Code snippet I.1: Get Specialization.

```python
@staffauth_bp.route('/staff/login', methods=['POST'])
def staff_login():
    data = request.get_json()
    email = data.get('staffEmail')
    password = data.get('staffPassword')

    if not all([email, password]):
        return jsonify({'message': 'Missing email or password'}), 400

    hashed_password = hash_password(password)
    staff_type = verify_staff(email)

    if not staff_type:
        return jsonify({'message': 'Invalid email format or staff type not recognized'}), 400

    conn = get_db_connection()
    if conn:
        try:
            cursor = conn.cursor()
            staff_id = None
            staff_name=None
            if staff_type == "doctor":
                cursor.execute("""
                    SELECT Doctor_id,Doctor_FirstName,Doctor_LastName FROM Doctor
                    WHERE Doctor_Email = ? AND D_Password = ?
                """, (email, hashed_password))
                row = cursor.fetchone()
                if row:
                    staff_id = row[0]
                    staff_name = row[1] +" " +row[2]


            elif staff_type == "nurse":
                cursor.execute("""
                    SELECT Nurse_id,Nurse_FirstName,Nurse_LastName FROM Nurse
                    WHERE Nurse_Email = ? AND N_Password = ?
                """, (email, hashed_password))
                row = cursor.fetchone()
                if row:
                    staff_id = row[0]
                    staff_name = row[1] +" " +row[2]
```

Code snippet I.2: Staff Login

```python
def staff_login():

            elif staff_type == "admin":
                cursor.execute("""
                    SELECT Admin_id FROM Admin
                    WHERE Admin_Email = ? AND Admin_Password = ?
                """, (email, hashed_password))
                row = cursor.fetchone()
                if row:
                    staff_id = row[0]
                    staff_name = ""


            if not row:
                return jsonify({'message': 'Invalid email or password'}), 401

            # creates access and refresh tokens
            access_token = create_access_token(identity=email)
            refresh_token = create_refresh_token(identity=email)

            response = make_response(jsonify({
            'message': 'Staff Login successful',
            'name' : staff_name,
            'email': email,
            'staffType': staff_type,
            'staffId': staff_id
            }))


            # Setting accesss cookie to deal with protected route access
            set_access_cookies(response, access_token)

            # Setting refresh cookie to deal with browser refresh
            set_refresh_cookies(response, refresh_token)


            return response


        except Exception as e:
            return jsonify({'message': f'Error: {e}'}), 500
        finally:
            conn.close()
    else:
        return jsonify({'message': 'Database connection failed'}), 500
```

Code snippet I.3: Staff Login continuation.

```python
@staffauth_bp.route('/staffProfile', methods=['GET'])
@jwt_required()
def getStaffProfile():
    userEmail = get_jwt_identity()

    if not userEmail:
        return jsonify({'message': 'Missing email'}), 400

    userType = verify_staff(userEmail)
    conn = get_db_connection()

    if conn:
        try:
            cursor = conn.cursor()

            if userType == 'doctor':
                cursor.execute("""
                    SELECT D.Doctor_FirstName, D.Doctor_LastName, D.Doctor_Email, D.Doctor_Phone_No,
                           D.Doctor_Registration_Number, S.Specialization_Name
                    FROM Doctor D
                    JOIN Specialization S ON D.Specialization_ID = S.Specialization_ID
                    WHERE D.Doctor_Email = ?
                """, (userEmail,))

                doctorProfile = cursor.fetchone()

                if not doctorProfile:
                    return jsonify({'message': 'No doctor found with this email'}), 400

                profile = {
                    'FirstName': doctorProfile[0],
                    'LastName': doctorProfile[1],
                    'Email': doctorProfile[2],
                    'PhoneNo': doctorProfile[3],
                    'RegistrationNumber': doctorProfile[4],
                    'Specialization': doctorProfile[5]
                }
                return jsonify({'profile': profile}), 200

            elif userType == 'nurse':
                cursor.execute("""
                    SELECT N.Nurse_FirstName, N.Nurse_LastName, N.Nurse_Email, N.Nurse_Phone_No,
                           N.Nurse_Registration_Number, S.Specialization_Name
                    FROM Nurse N
                    JOIN Nurse_Specialization S ON N.Specialization_ID = S.Specialization_ID
                    WHERE N.Nurse_Email = ?
                """, (userEmail,))
```

Code snippet I.4: Staff Profile

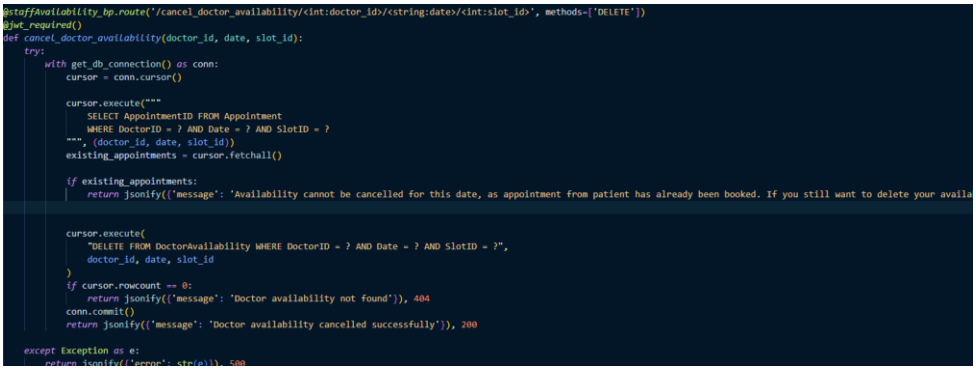
```
def getStaffProfile():
                    'RegistrationNumber': doctorProfile[4],
                    'Specialization': doctorProfile[5]
                }
                return jsonify({'profile': profile}), 200

            elif userType == 'nurse':
                cursor.execute("""
                    SELECT N.Nurse_FirstName, N.Nurse_LastName, N.Nurse_Email, N.Nurse_Phone_No,
                        N.Nurse_Registration_Number, S.Specialization_Name
                    FROM Nurse N
                    JOIN Nurse_Specialization S ON N.Specialization_ID = S.Specialization_ID
                    WHERE N.Nurse_Email = ?
                """, (userEmail,))

                nurseProfile = cursor.fetchone()

                if not nurseProfile:
                    return jsonify({'message': 'No nurse found with this email'}), 400

                profile = {
                    'FirstName': nurseProfile[0],
                    'LastName': nurseProfile[1],
                    'Email': nurseProfile[2],
                    'PhoneNo': nurseProfile[3],
                    'RegistrationNumber': nurseProfile[4],
                    'Specialization': nurseProfile[5]
                }
                return jsonify({'profile': profile}), 200

            else:
                return jsonify({'message': 'User type not recognized'}), 400

        except Exception as e:
            return jsonify({'message': f'Error: {e}'}), 500

        finally:
            conn.close()

    else:
        return jsonify({'message': 'Database connection failed'}), 500
```

Code snippet I.5: Staff profile Continuation.

J. Staff Availability code snippet

```
@staffAvailability_bp.route('/set_doctor_availability', methods=['POST'])
@jwt_required()
def set_doctor_availability():

    try:
        data = request.get_json()
        doctor_id = data['doctor_id']
        date = data['date']
        slot_ids = data['slot_ids']  # array of slot IDs

        with get_db_connection() as conn:
            cursor = conn.cursor()
            for slot_id in slot_ids:

                cursor.execute("""
                    SELECT 1
                    FROM DoctorAvailability
                    WHERE DoctorID = ? AND Date = ? AND SlotID = ?
                """, doctor_id, date, slot_id)

                existing_availability = cursor.fetchone()

                if existing_availability:
                    return jsonify({'message': f'Doctor availability already set for slot ID: {slot_id}'}), 400
                else:
                    cursor.execute(
                        "INSERT INTO DoctorAvailability (DoctorID, Date, SlotID) VALUES (?, ?, ?)",
                        doctor_id, date, slot_id
                    )

            conn.commit()

            return jsonify({'message': 'Doctor availability set successfully'}), 201
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Code snippet J.1: Set Doctor Availability.



```
@staffAvailability_bp.route('/cancel_doctor_availability/<int:doctor_id>/<string:date>/<int:slot_id>', methods=['DELETE'])
@jwt_required()
def cancel_doctor_availability(doctor_id, date, slot_id):
    try:
        with get_db_connection() as conn:
            cursor = conn.cursor()

            cursor.execute("""
                SELECT AppointmentID FROM Appointment
                WHERE DoctorID = ? AND Date = ? AND SlotID = ?
            """, (doctor_id, date, slot_id))
            existing_appointments = cursor.fetchall()

            if existing_appointments:
                return jsonify({'message': 'Availability cannot be cancelled for this date, as appointment from patient has already been booked. If you still want to delete your availa

            cursor.execute(
                "DELETE FROM DoctorAvailability WHERE DoctorID = ? AND Date = ? AND SlotID = ?",
                doctor_id, date, slot_id
            )
            if cursor.rowcount == 0:
                return jsonify({'message': 'Doctor availability not found'}), 404
            conn.commit()
            return jsonify({'message': 'Doctor availability cancelled successfully'}), 200

    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Code snippet J.2: Cancel Doctor Availability.

```
@staffAvailability_bp.route('/set_nurse_availability', methods=['POST'])
@jwt_required()
def set_nurse_availability():
    try:
        data = request.get_json()
        nurse_id = data['nurse_id']
        date = data['date']
        slot_ids = data['slot_ids']

        with get_db_connection() as conn:
            cursor = conn.cursor()
            for slot_id in slot_ids:

                cursor.execute("""
                    SELECT 1
                    FROM NurseAvailability
                    WHERE NurseID = ? AND Date = ? AND SlotID = ?
                """, nurse_id, date, slot_id)

                existing_availability = cursor.fetchone()

                if existing_availability:
                    return jsonify({'message': f'Nurse availability already set for slot ID: {slot_id}'}), 400
                else:
                    cursor.execute(
                        "INSERT INTO NurseAvailability (NurseID, Date, SlotID) VALUES (?, ?, ?)",
                        nurse_id, date, slot_id
                    )
            conn.commit()
            return jsonify({'message': 'Nurse availability set successfully'}), 201
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Code snippet J.3: Set Nurse Availability.

```
@staffAvailability_bp.route('/cancel_nurse_availability/<int:nurse_id>/<string:date>/<int:slot_id>', methods=['DELETE']
@jwt_required()
def cancel_nurse_availability(nurse_id, date, slot_id):
    try:
        with get_db_connection() as conn:
            cursor = conn.cursor()
            cursor.execute(
                "DELETE FROM NurseAvailability WHERE NurseID = ? AND Date = ? AND SlotID = ?",
                nurse_id, date, slot_id
            )
            if cursor.rowcount == 0:
                return jsonify({'message': 'Nurse availability not found'}), 404
            conn.commit()
            return jsonify({'message': 'Nurse availability cancelled successfully'}), 200

    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Code snippet J.4: Cancel Nurse Availability.

```python
@staffAvailability_bp.route('/get_availability/<string:staff_type>/<int:staff_id>', methods=['GET'])
@jwt_required()
def get_availability(staff_type, staff_id):
    try:
        with get_db_connection() as conn:
            cursor = conn.cursor()

            if staff_type.lower() == "doctor":
                cursor.execute("""
                    SELECT Date, SlotID
                    FROM DoctorAvailability
                    WHERE DoctorID = ?
                    ORDER BY Date, SlotID
                """, staff_id)

            elif staff_type.lower() == "nurse":
                cursor.execute("""
                    SELECT Date, SlotID
                    FROM NurseAvailability
                    WHERE NurseID = ?
                    ORDER BY Date, SlotID
                """, staff_id)

            else:
                return jsonify({'message': 'Invalid staff type'}), 400

            rows = cursor.fetchall()

            if not rows:
                return jsonify([]), 200


            # Basically the below provided code is list comprehension which traverses through each row and extracts date and slot from it
            availability = [
                {
                    'date': row.Date.strftime('%Y-%m-%d'),
                    'slot_id': row.SlotID
                }
                for row in rows
            ]

            return jsonify(availability), 200

    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

Code snippet J.5: Get the Availability.

K. Medical Records code snippets.

```python
@medicalRecords_bp.route('/upload/<patient_id>', methods=['POST'])
@jwt_required()
def upload_multiple_medical_records(patient_id):
    if 'files' not in request.files:
        return jsonify({'message': 'No files part in the request'}), 400

    files = request.files.getlist('files')
    categories = request.form.getlist('categories')  # gets a list of categories along with its file

    # This checks the length of the list above, to ensure that the number of categories matches the number of files
    if len(files) != len(categories):
        return jsonify({'message': 'Mismatch between number of files and categories'}), 400

    # checks categores from list of allowed records.
    for category in categories:
        if category not in ALLOWED_RECORD_TYPES:
            return jsonify({'message': f'Invalid category: {category}'}), 400

    conn = get_db_connection()
    if conn:
        try:
            cursor = conn.cursor()
            cursor.execute("SELECT container_name FROM Patient WHERE P_id = ?", (patient_id,))
            patient_data = cursor.fetchone()

            if not patient_data:
                return jsonify({'message': f'Patient with ID {patient_id} not found'}), 404

            container_name = patient_data[0]
            uploaded_files = []

            # Upload files under their categorie names
            for file, category in zip(files, categories):
                if file.filename != '':
                    subfolder_name = category  #category is used for providing the subfolder name
                    blob_client = blob_service_client.get_blob_client(
                        container=container_name,
                        blob=f"{subfolder_name}/{file.filename}"
                    )
                    blob_client.upload_blob(
                        file.stream.read(),
                        overwrite=True,  # we are overwriting if file already exists
                        content_settings=ContentSettings(content_type=file.content_type)
                    )
                    uploaded_files.append(f"{subfolder_name}/{file.filename}")
```

Code snippet K.1: Upload the medical records.

```python
@medicalRecords_bp.route('/patient/files', methods=['GET'])
@jwt_required()
def get_patient_files():
    patient_id = request.args.get('patient_id')


    if not patient_id:
        return jsonify({'message': 'Missing required parameters: firstName, lastName, dob'}), 400

    conn = get_db_connection()
    if conn:
        cursor = conn.cursor()
        cursor.execute("""
            SELECT container_name
            FROM Patient
            WHERE P_id = ?
        """, (patient_id))
        patient_data = cursor.fetchone()

        conn.close()

        if not patient_data:
            return jsonify({'message': 'Patient not found with the provided details'}), 404

        container_name = patient_data[0]
        file_list = []

        try:
            container_client = blob_service_client.get_container_client(container=container_name)
            blobs = container_client.list_blobs()
            for blob in blobs:
                file_list.append({'name': blob.name, 'url': container_client.get_blob_client(blob=blob.name).url})
            return jsonify({'patient_files': file_list}), 200
        except ResourceNotFoundError:
            return jsonify({'message': f'Container "{container_name}" not found for the patient'}), 404
        except Exception as e:
            return jsonify({'message': f'Error retrieving files: {e}'}), 500
    else:
        return jsonify({'message': 'Database connection failed'}), 500
```

Code snippet K.2: Get patients file

```python
@medicalRecords_bp.route('/patient/medicalRecords', methods=['POST'])
@jwt_required()
def get_patient_files_for_doctor():
    docEmail = get_jwt_identity()
    data = request.get_json()
    first_name = data.get('firstName')
    last_name = data.get('lastName')
    dob = data.get('dob')

    if not docEmail:
        return jsonify({'message': 'Not Authorised'}), 401

    if not first_name or not last_name or not dob:
        return jsonify({'message': 'Missing required parameters: firstName, lastName, dob'}), 400


    conn = None
    try:
        conn = get_db_connection()
        if not conn:
            return jsonify({'message': 'Database connection failed'}), 500

        cursor = conn.cursor()
        cursor.execute("""
            SELECT container_name
            FROM Patient
            WHERE P_FirstName=? AND P_LastName=? AND DOB=?
        """, (first_name, last_name, dob))
        patient_data = cursor.fetchone()

        if not patient_data:
            return jsonify({'message': 'Patient not found with the provided details'}), 404

        container_name = patient_data[0]
        container_client = blob_service_client.get_container_client(container=container_name)
        folders = set()
        blob_list = container_client.list_blobs()

        for blob in blob_list:
            parts = blob.name.split('/')
            if len(parts) > 1:
                folders.add(parts[0])

        folder_structure = {}
        for folder in folders:
            folder_structure[folder] = []
            blobs_in_folder = container_client.list_blobs(name_starts_with=f"{folder}/")
            for blob in blobs_in_folder:
```

Code snippet K.3: Doctors views the medical records.

```python
def get_patient_files_for_doctor():
        """, (first_name, last_name, dob))
        patient_data = cursor.fetchone()

        if not patient_data:
            return jsonify({'message': 'Patient not found with the provided details'}), 404

        container_name = patient_data[0]
        container_client = blob_service_client.get_container_client(container=container_name)
        folders = set()
        blob_list = container_client.list_blobs()

        for blob in blob_list:
            parts = blob.name.split('/')
            if len(parts) > 1:
                folders.add(parts[0])

        folder_structure = {}
        for folder in folders:
            folder_structure[folder] = []
            blobs_in_folder = container_client.list_blobs(name_starts_with=f"{folder}/")
            for blob in blobs_in_folder:
                folder_structure[folder].append({
                    'name': blob.name.split('/')[-1],
                    'url': container_client.get_blob_client(blob=blob.name).url
                })

        return jsonify({'folders': folder_structure}), 200

    except ResourceNotFoundError:
        return jsonify({'message': f'Container "{container_name}" not found for the patient'}), 404
    except Exception as e:
        return jsonify({'message': f'Error fetching medical records: {str(e)}'}), 500
    finally:
        if conn:
            conn.close()
```

Code snippet k.4: Doctors view the medical record continuation.

```javascript
const handleFinalUpload = async () => {
  if (!patient_id) {
    setUploadError("Patient ID is not available.");
    return;
  }

  const formData = new FormData();
  allSelectedFiles.forEach((item) => {
    formData.append("files", item.file);
    formData.append("categories", item.category);
    formData.append("record_type", item.category);
  });

  // API endpoint to upload the docs
  try {
    const csrfToken = getCookie("csrf_access_token");

    const response = await fetch(`${BASE_URL}/upload/${patient_id}`, {
      method: "POST",
      credentials: "include",
      headers: {
        "X-CSRF-TOKEN": csrfToken,
      },
      body: formData,
    });

    if (response.ok) {
      const data = await response.json();
      setUploadSuccess(data.message);
      setUploadError(null);
      setAllSelectedFiles([]);
    } else {
      const errorData = await response.json();
      setUploadError(errorData.message || "Upload failed.");
      setUploadSuccess(null);
    }
  } catch (error) {
    setUploadError(error.message);
    setUploadSuccess(null);
```

Code snippet K.5: Handle final upload function.

L. Register code snippets

```
const handleRegister = async (registrationData) => {
  //API call for patient regsitration
  try {
    const csrfToken = getCookie("csrf_access_token");
    const response = await fetch(`${BASE_URL}/patient/register`, {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
        "X-CSRF-TOKEN": csrfToken,
      },
      body: JSON.stringify(registrationData),
    });

    if (response.ok) {
      navigate("/");
    } else {
      const errorData = await response.json();
      setRegisterError(errorData.message || "Registration failed");
    }
  } catch (error) {
    setRegisterError("An error occurred during registration.");
    console.error("Registration error:", error);
  }
};
```

Code snippet L.1: Handle Register.

```
const fetchSpecializations = async (staffType) => {
  //API Call to get the specializations for the staff based on their type i.e Nurse/Doctor
  try {
    const csrfToken = getCookie("csrf_access_token");
    const response = await fetch(
      `${BASE_URL}/get_specializations?staff_type=${staffType}`,
      {
        method: "GET",
        headers: {
          "Content-Type": "application/json",
          "X-CSRF-TOKEN": csrfToken,
        },
        credentials: "include",
      }
    );
    const data = await response.json();

    if (response.ok) {
      setSpecializations(data);
    } else {
      console.error("Error fetching specializations:", data.error);
    }
  } catch (error) {
    console.error("Network error:", error);
  }
};
```

Code snippet L.2: Fetch Specialization function.

M. Own Piece of Code snippets

```
unique_id = uuid.uuid4().hex[:8] # here a unique ID is been created of first 8 digits only
container_name = f"patient-{p_first_name.lower()}{p_last_name.lower()}{dob.replace('-', '')}-{unique_id}"

cursor.execute("""
    INSERT INTO Patient (P_FirstName, P_LastName, Gender, DOB, Email_Id, Phone_No, PatientPassword,StreetAddress,City,Postcode,container_name)
    VALUES (?, ?, ?, ?, ?, ?, ?,?,?,?,?)
""", (p_first_name, p_last_name, gender, dob, email, phone, hashed_password,street_address,city,postcode,container_name))
conn.commit()
```

Code snippet M.1: Container Name

```python
# Creates prescription
prescription_id = str(uuid.uuid4())
prescription = {
    'id': str(uuid.uuid4()),
    'patientid': patientId,
    'patientFirstName': row.P_FirstName,
    'patientLastName': row.P_LastName,
    'prescriptionid': prescription_id,
    'doctorEmail': doctorEmail,
    'medicines': enriched_medicine_list,
    'pharmacyName': pharmacyName,
    'collectionMethod': collectionMethod,
    'dateProvided': dateProvided,
    'totalCost': total_price
}

container.create_item(body=prescription)

result = {
    'patientid': patientId,
    'prescriptionid': prescription_id,
    'totalCost': total_price,
    'message': 'Prescription created successfully'
}
return jsonify(result), 201
```

Code snippet M.3: Prescription Creation

N. Diagrams.

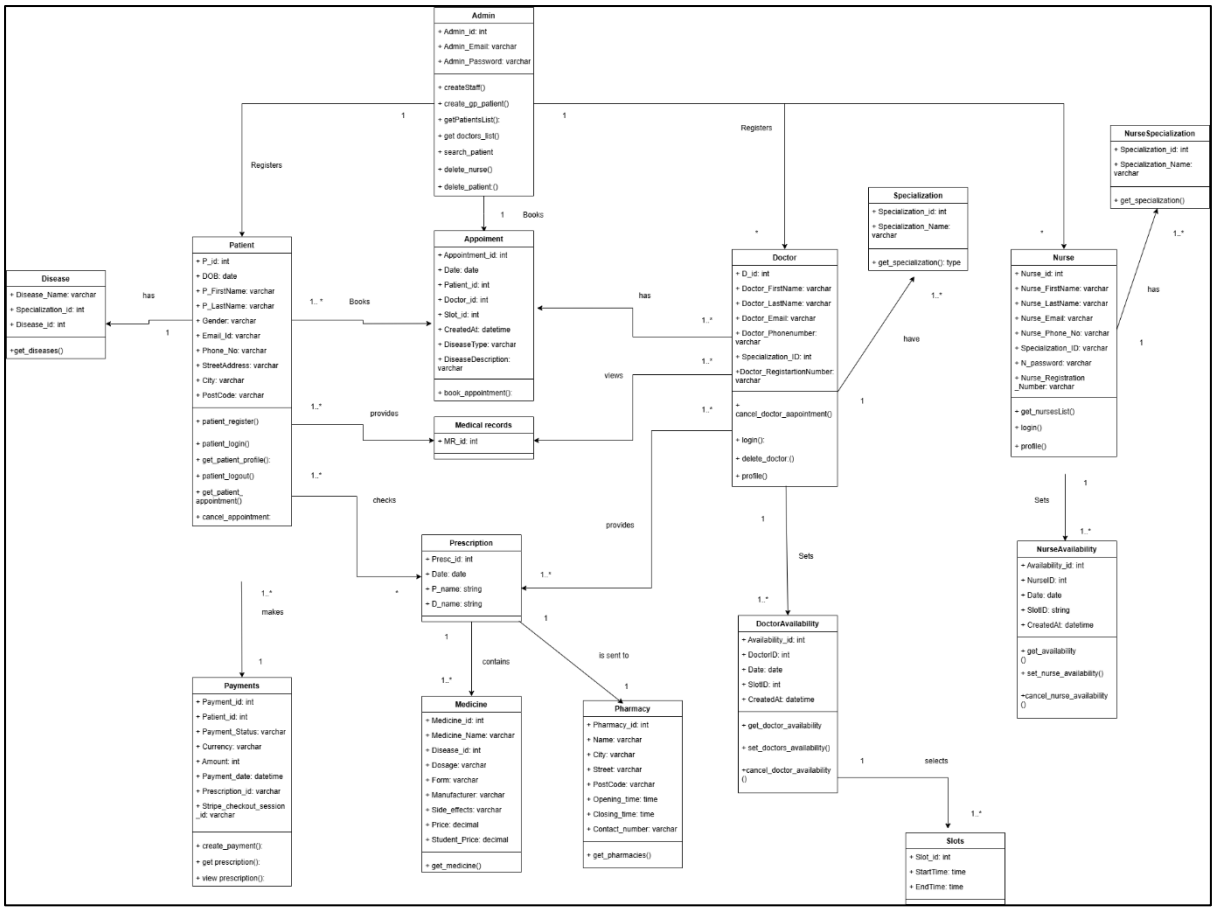


Diagram N.1: Class Diagram .Done by Author

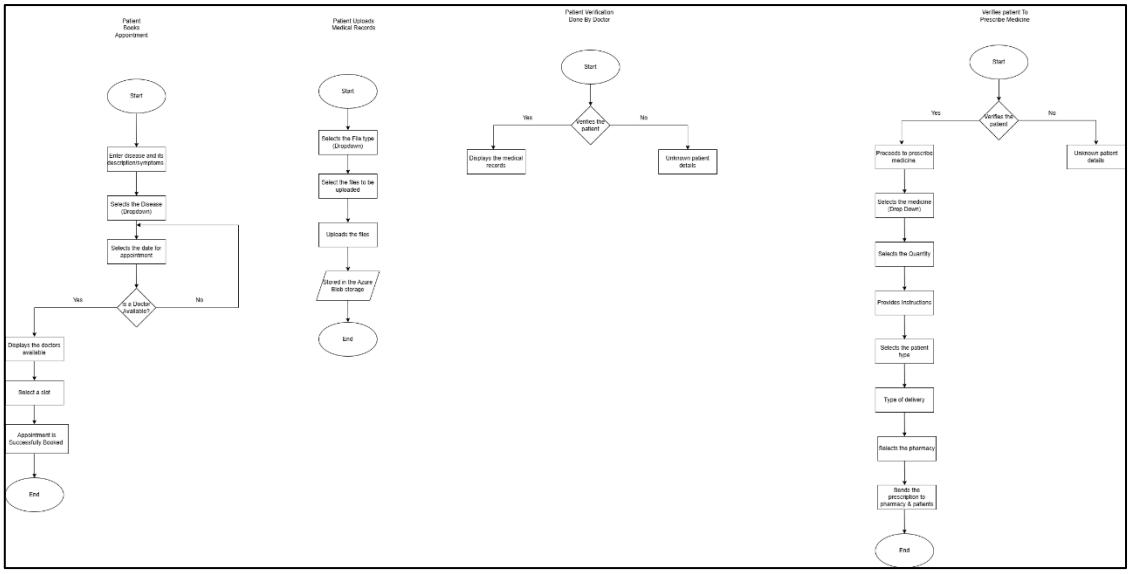


Diagram N.2: Flow Diagram .Done by Author

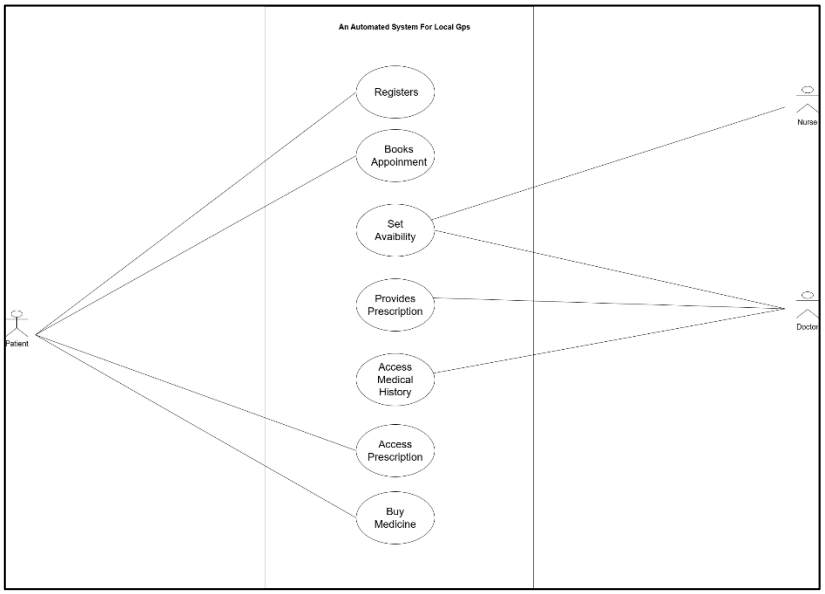


Diagram N.3: Use Case Diagram. .Done by Author

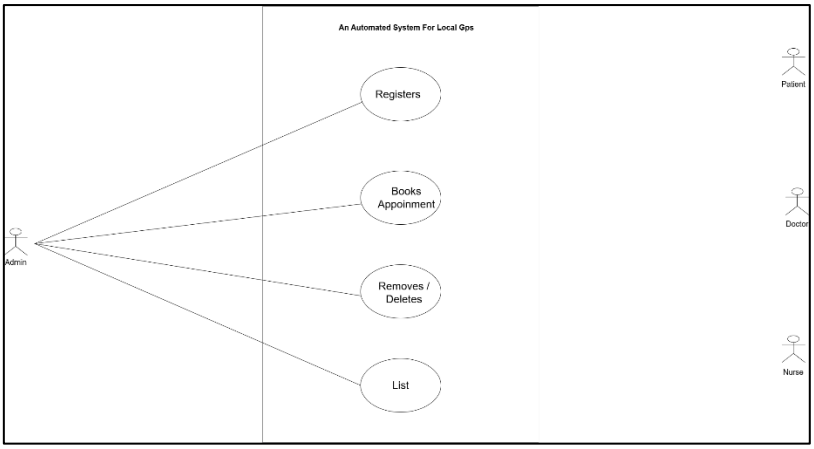


Diagram N.4: Use case Diagram for admin. .Done by Author

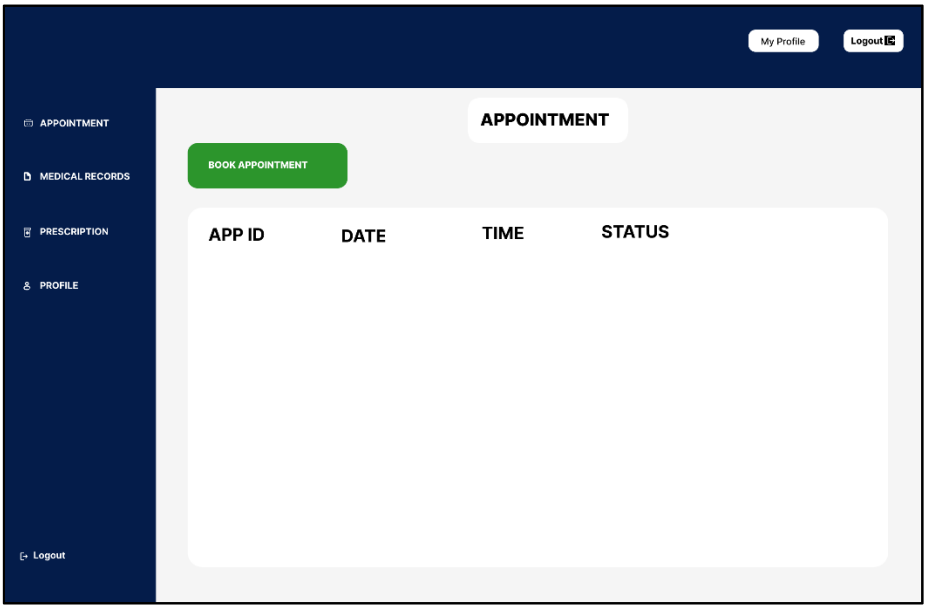


Diagram N.5: High Fidelity Prototype 1. .Done by Author

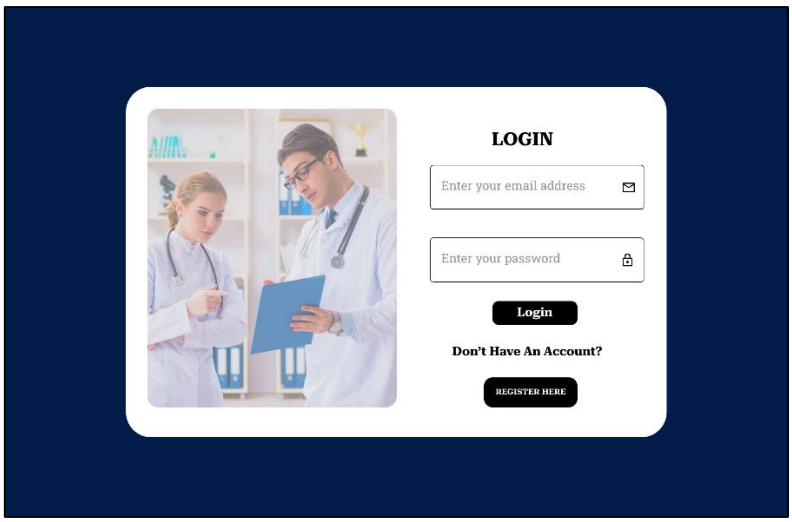


Diagram N.6: High Fidelity Prototype 2. .Done by Author

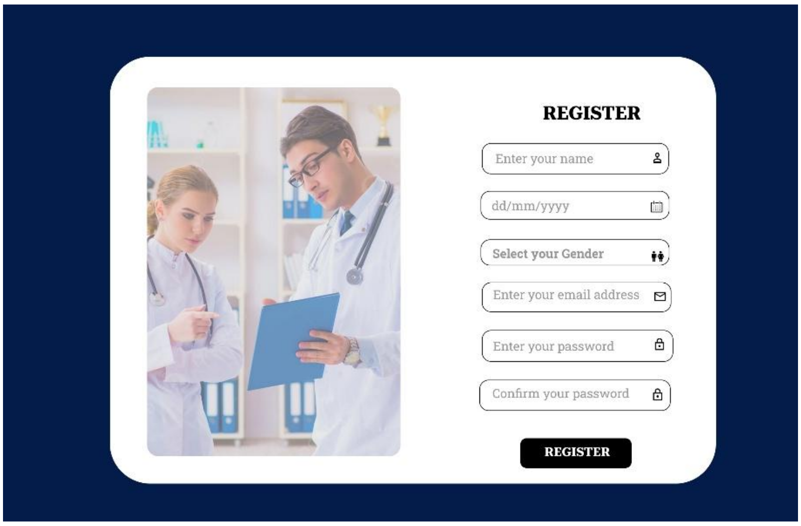
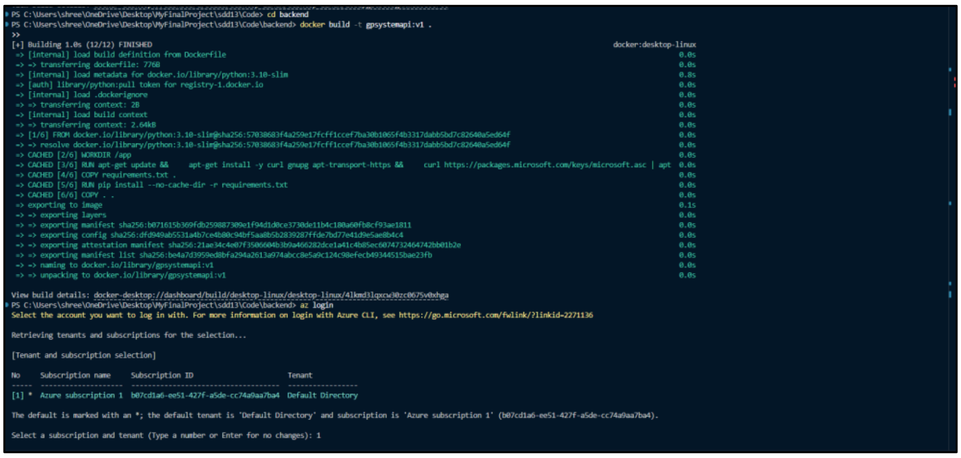
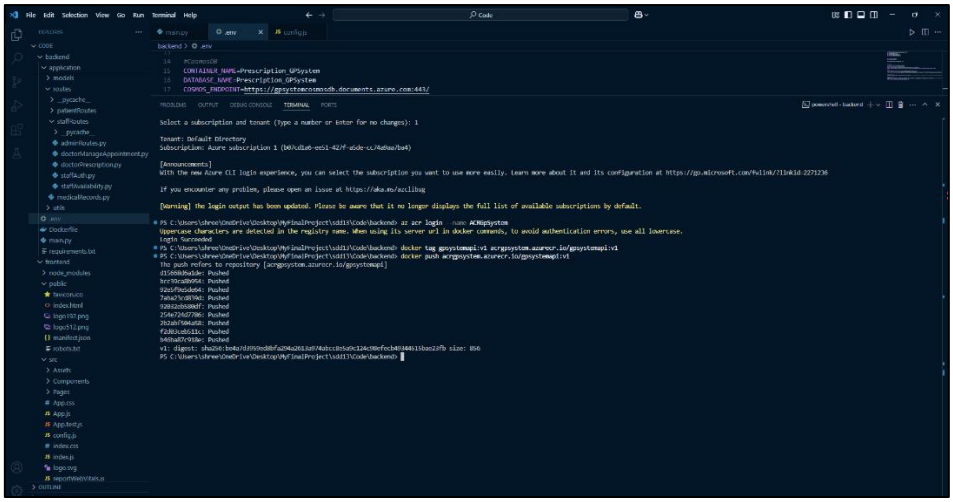


Diagram N.7: High Fidelity Prototype 3.Done by Author

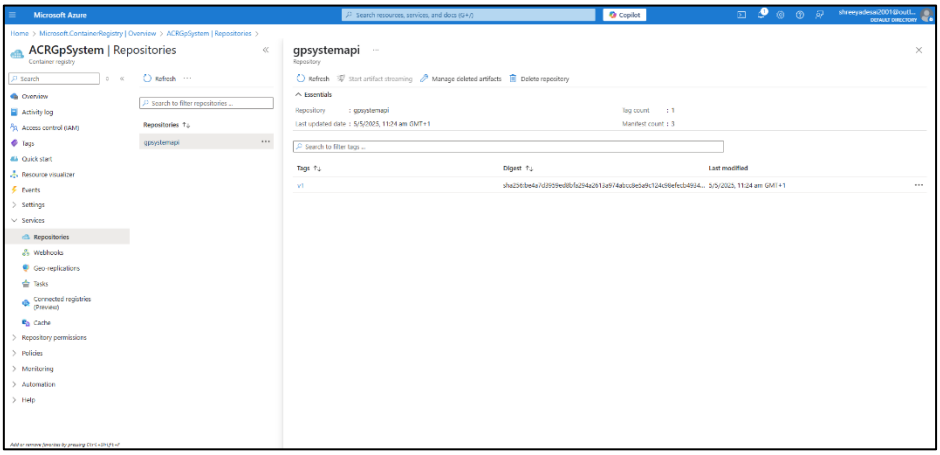
O. Deployment Snippets



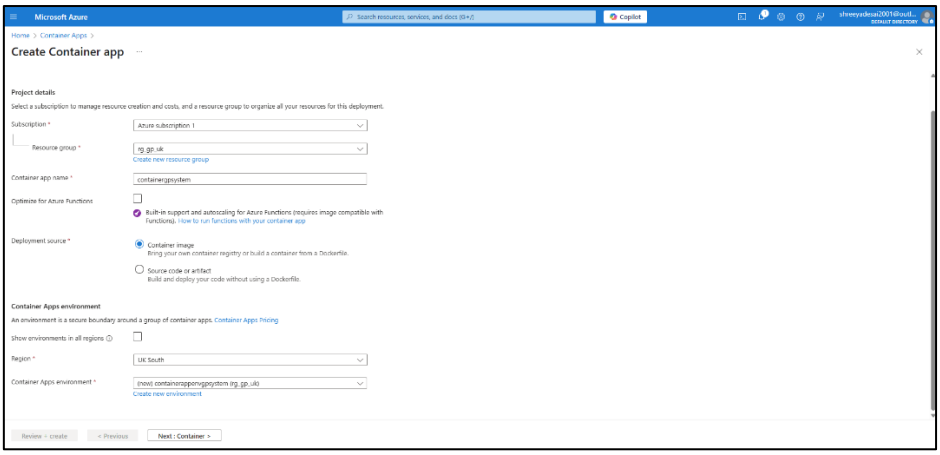
Deployment: O.1: Docker build(Backend)



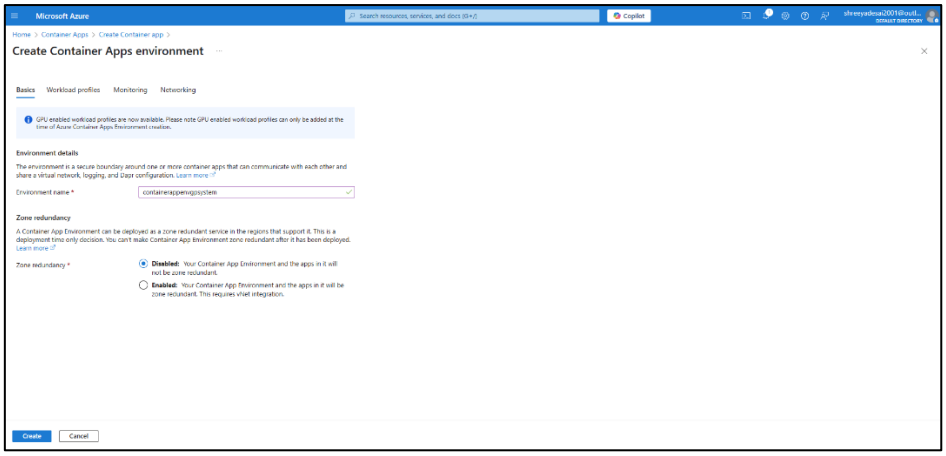
Deployment: O.2: Docker tag and push (backend).



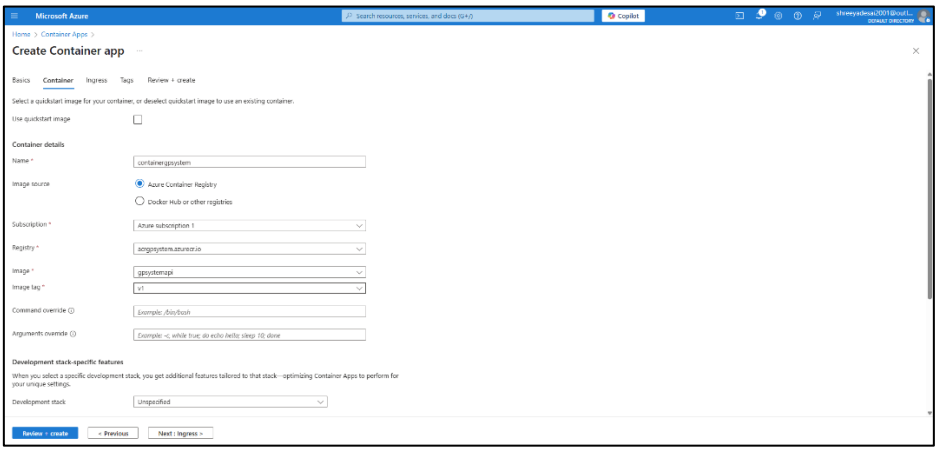
Deployment: O.3: Azure Dashboard (Container registry)



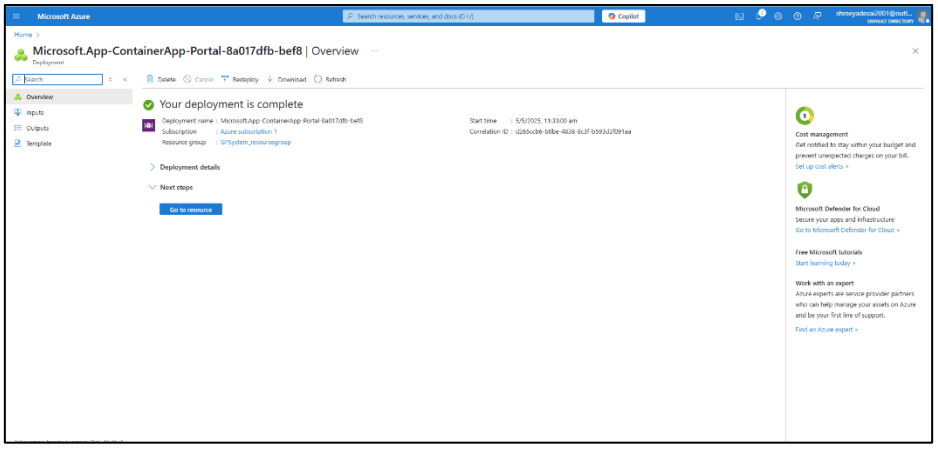
Deployment: O.4: Create Container App Configuration.



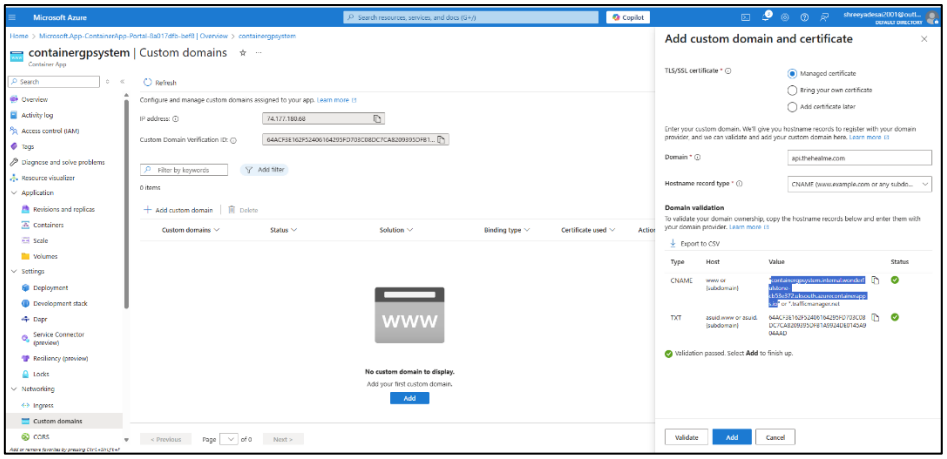
Deployment: O.5: Create Container App Environment.



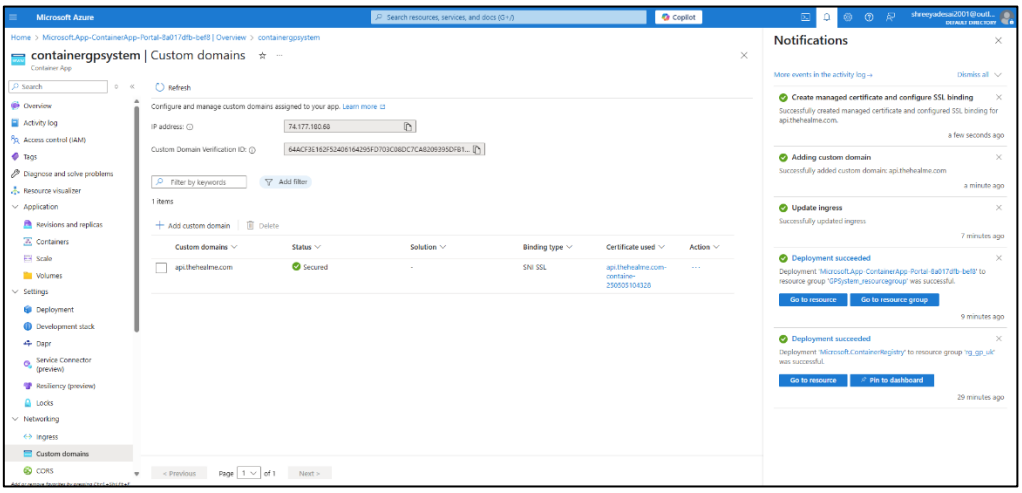
Deployment: O.6: Container App (Container Configuration).



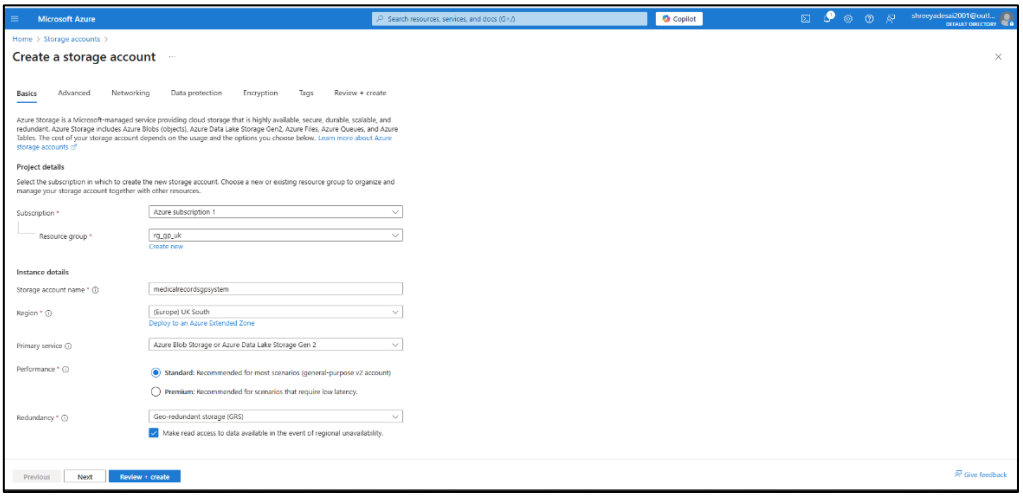
Deployment: O.7: Deployment complete (Azure Container App)



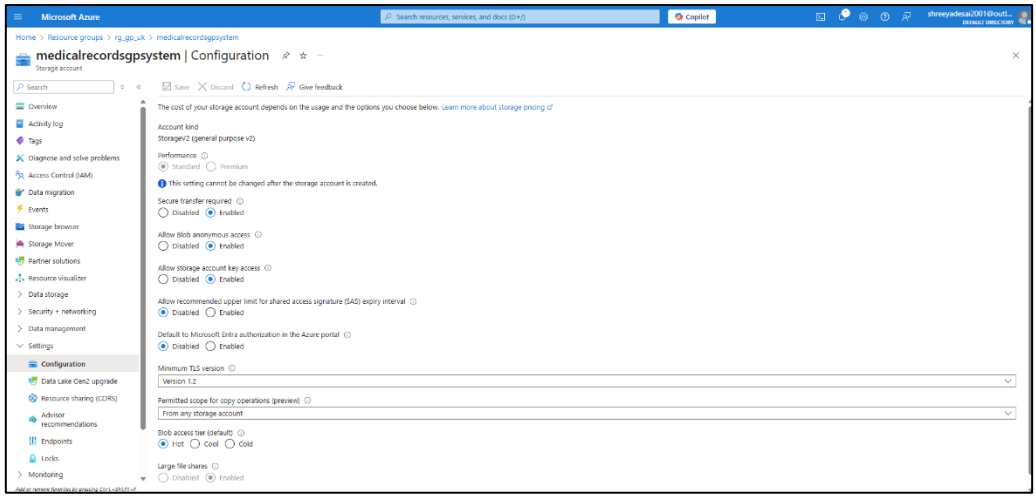
Deployment: O.8: Adding Custom Domain(Container App)



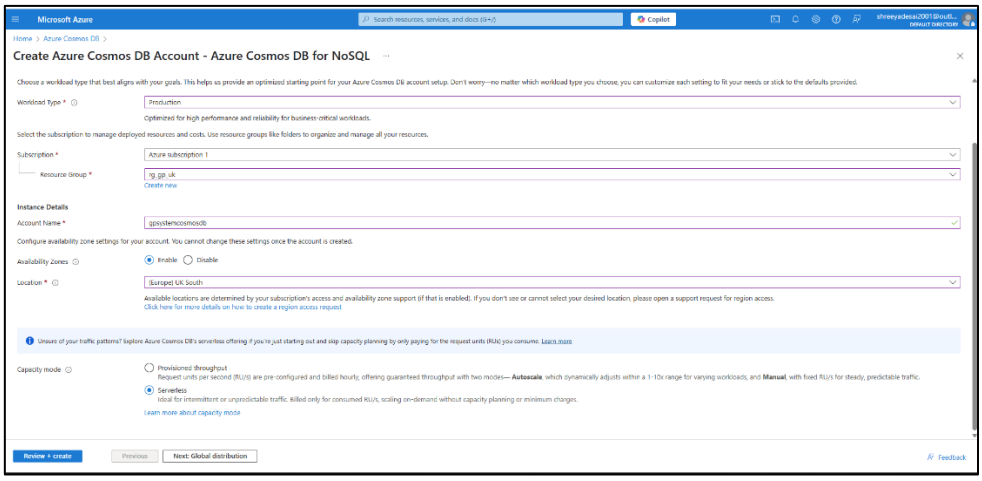
Deployment: O.9: Added Successfully.



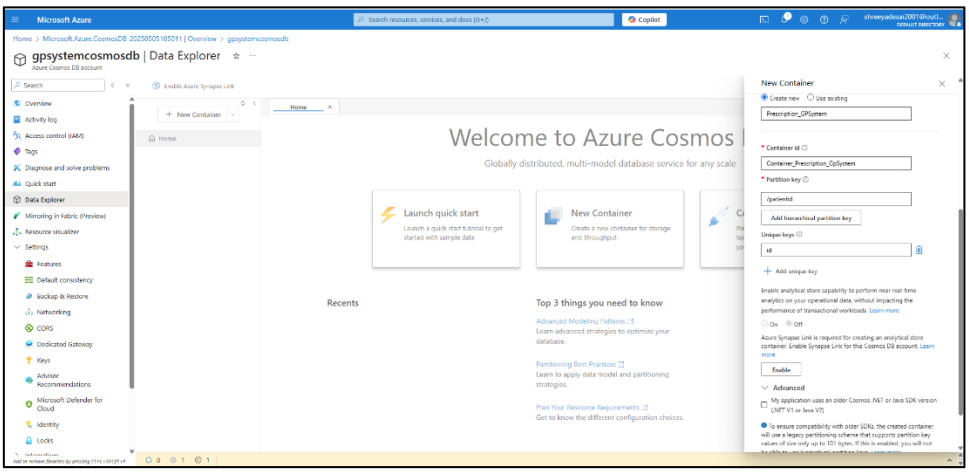
Deployment: O.10: Storage Account Configurations.



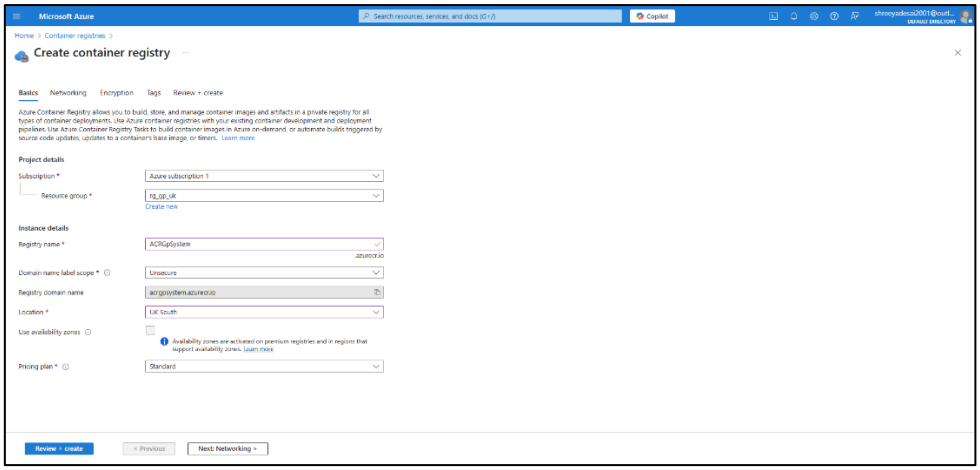
Deployment: O.11: Storage Account Configurations.



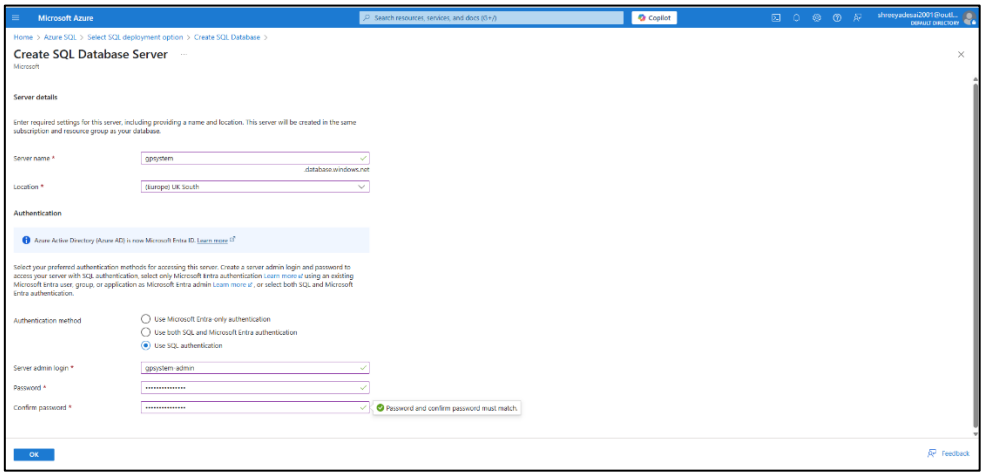
Deployment: O.12: Cosmos DB Creation.



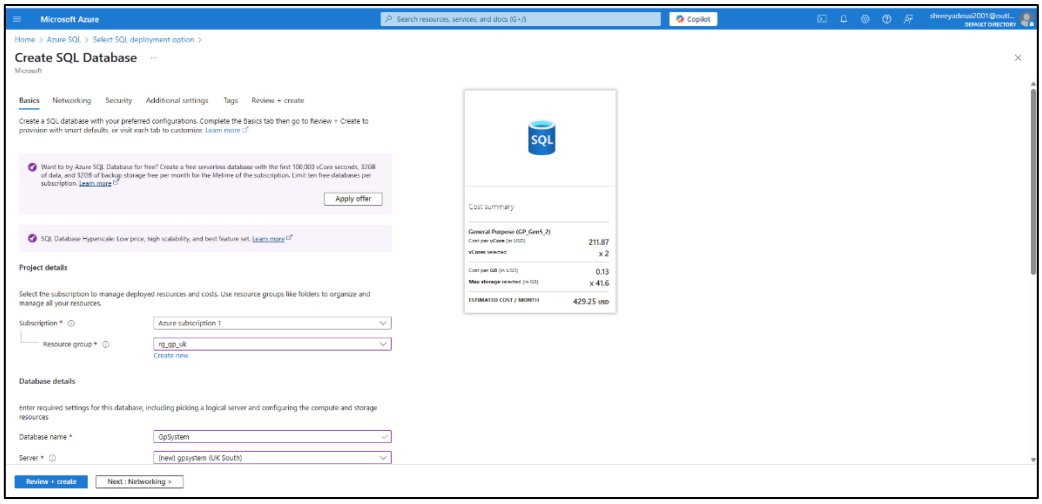
Deployment: O.13: Adding the New Container



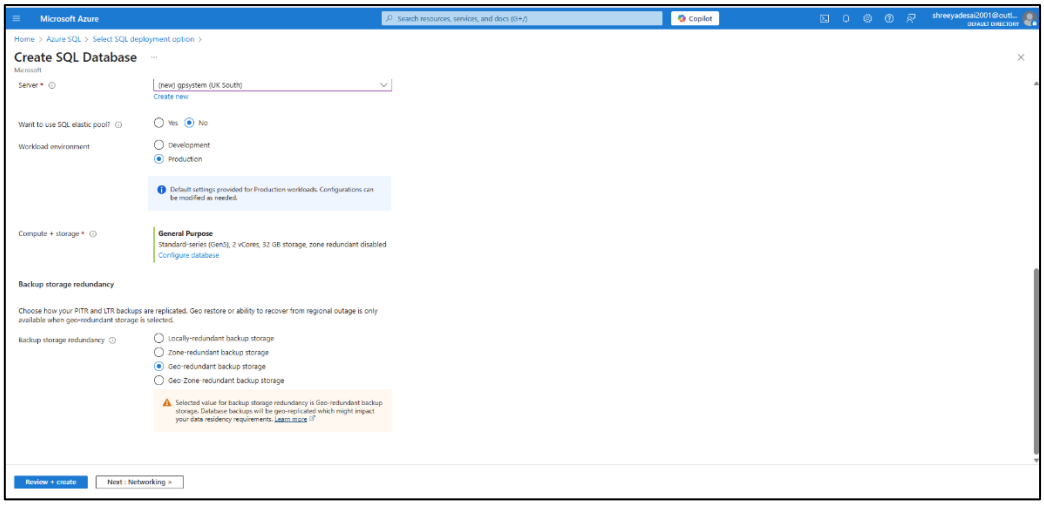
Deployment: O.14: Creation of Container Registry.



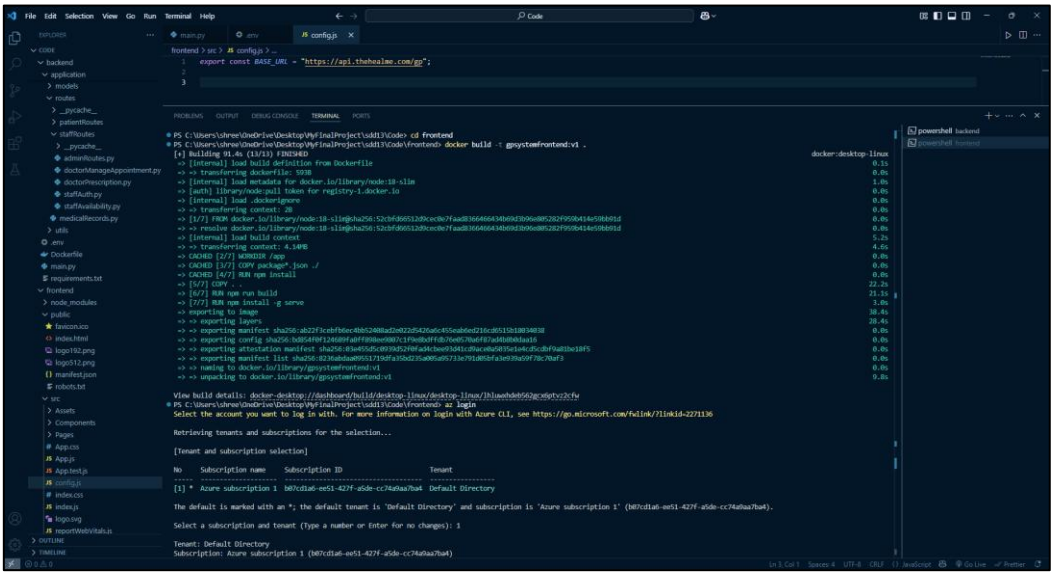
Deployment: O.15: Creation of SQL Database Server.



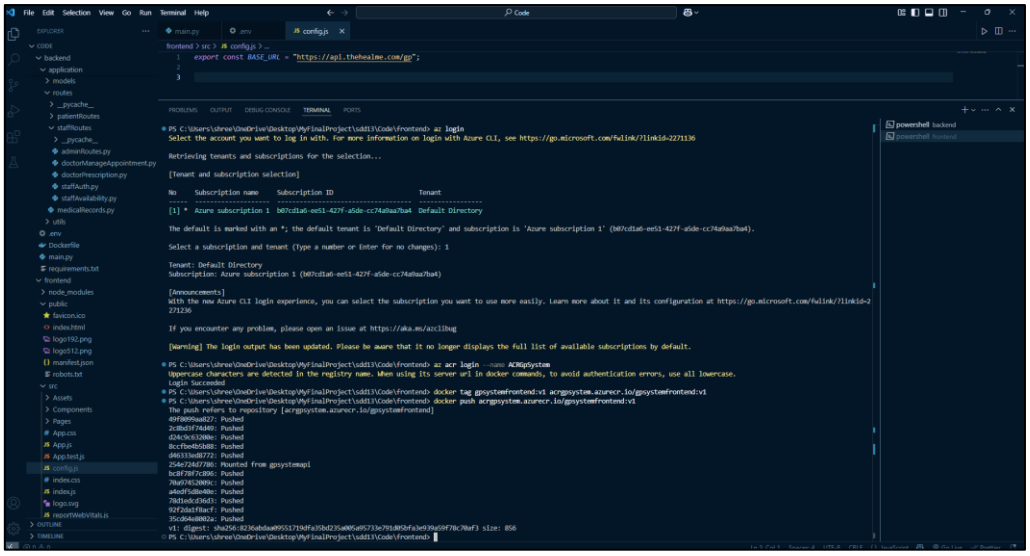
Deployment: O.16: Creation of SQL database.



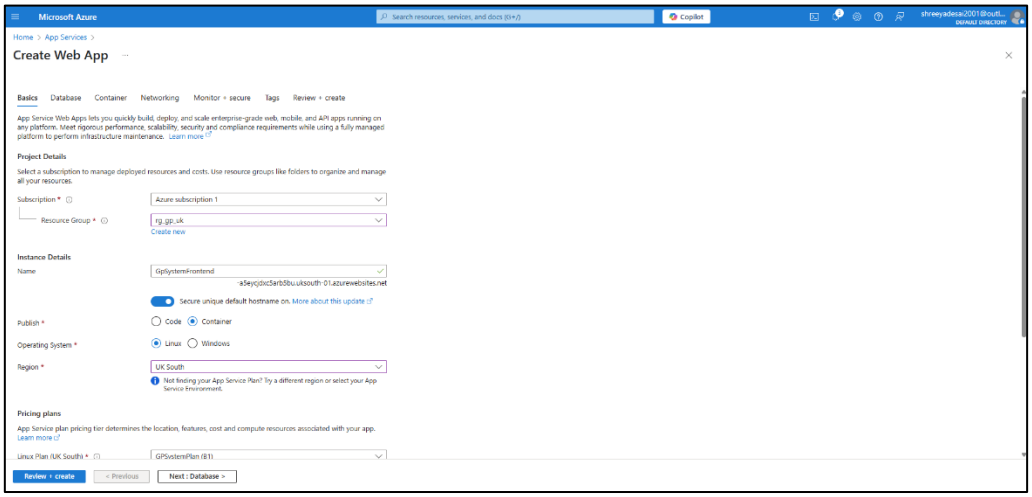
Deployment: O.17*:* Configuration of SQL database.



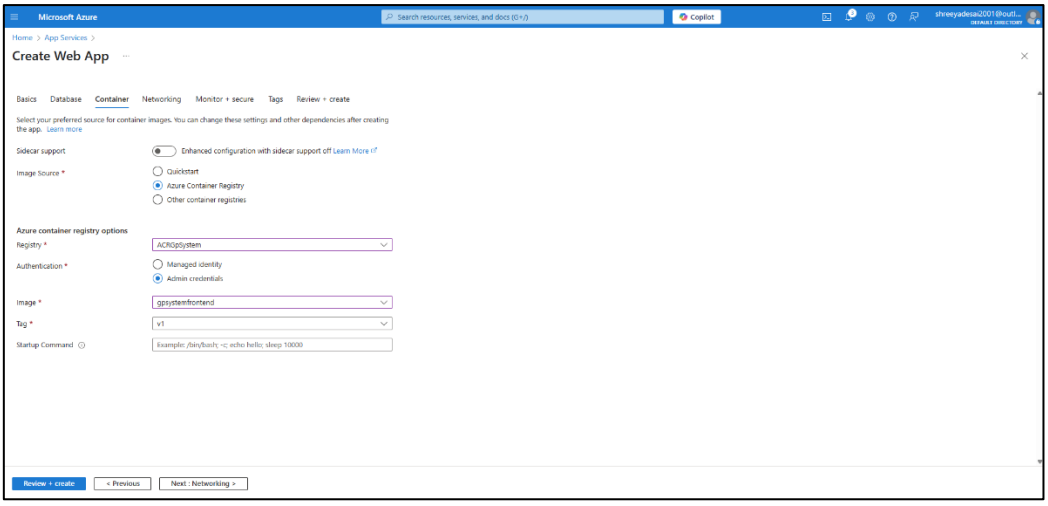
Deployment: O.18: Docker build(frontend).



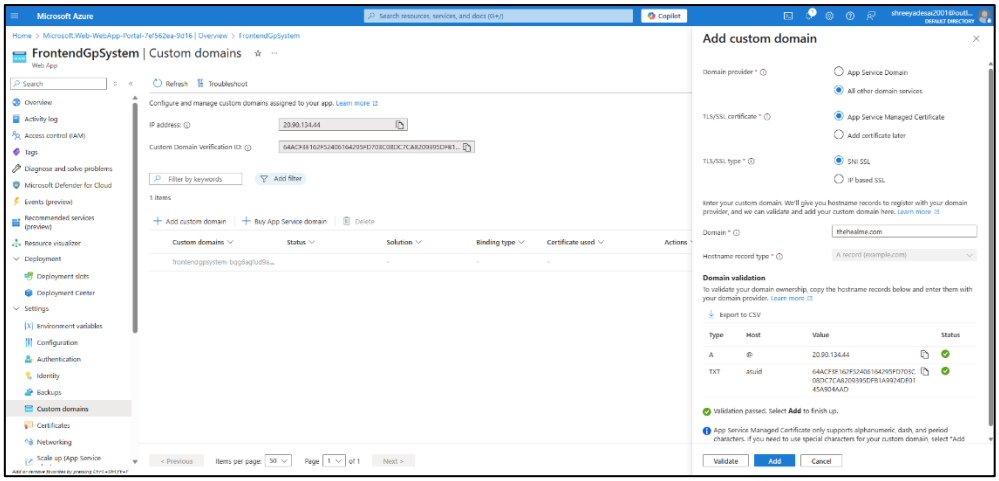
Deployment: O.18: Docker tag and push(Frontend).



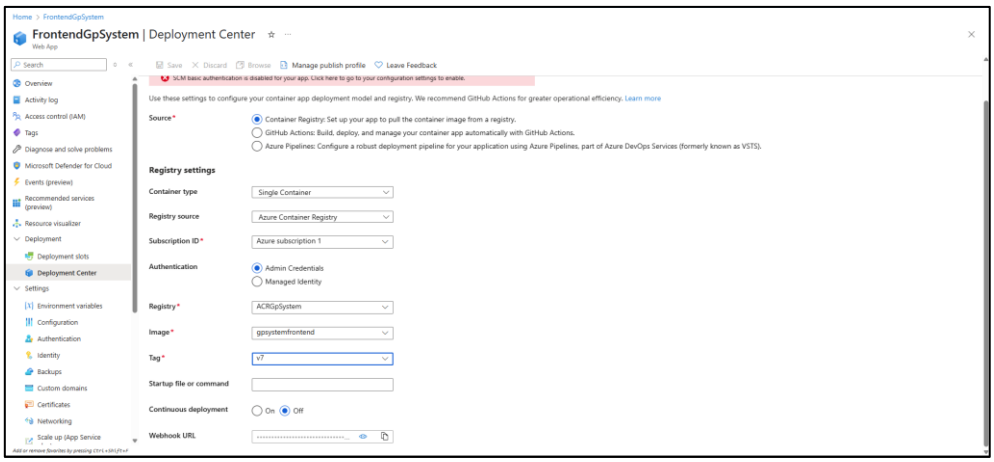
Deployment: O.20: Create Web App



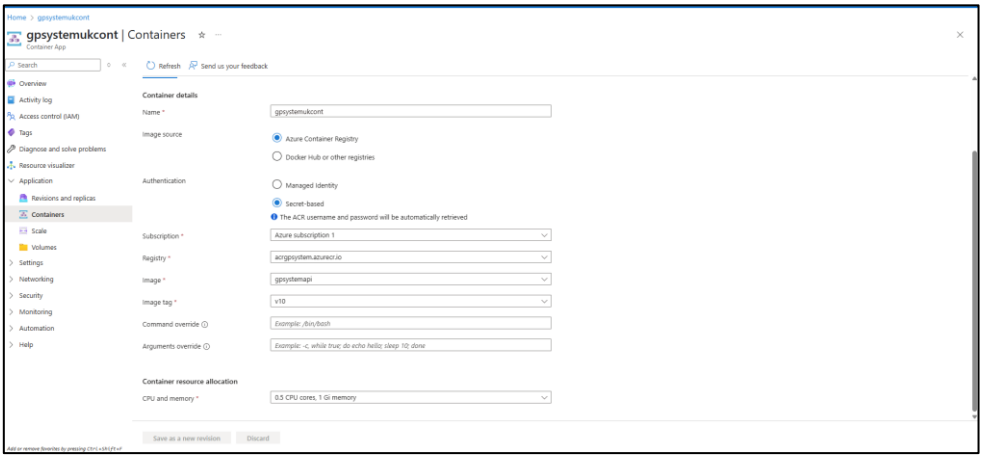
*Deployment:O.*21: Container configurations(Web app).



Deployment: O.22: Adding custom Domain.

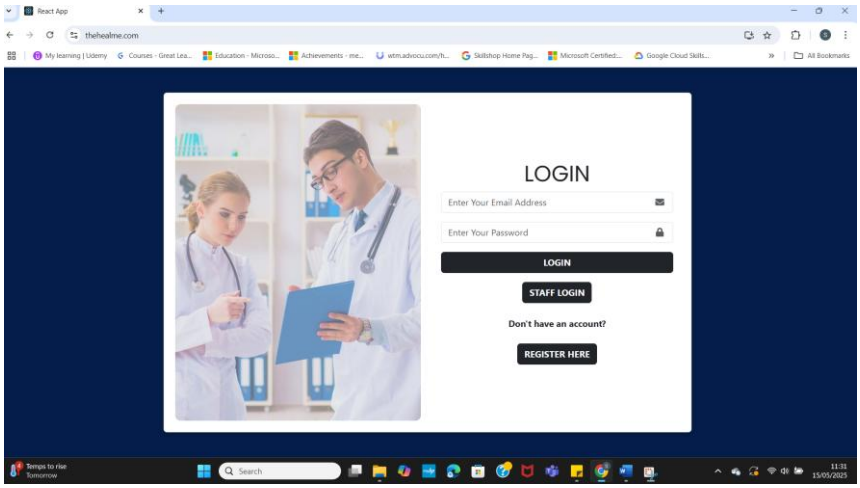


Deployment: O.23: Deployment Center(FrontendGpSystem)

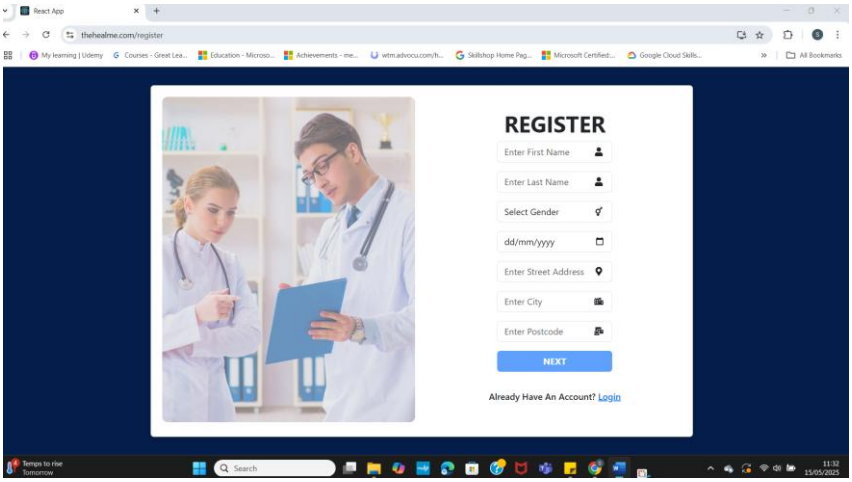


Deployment: O.24: Containers (gpsystemukcont).

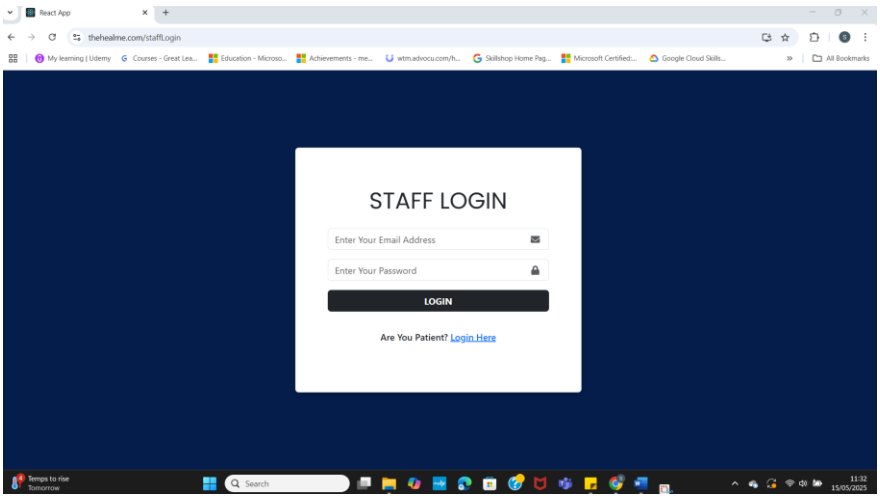
P. Application Snippets.



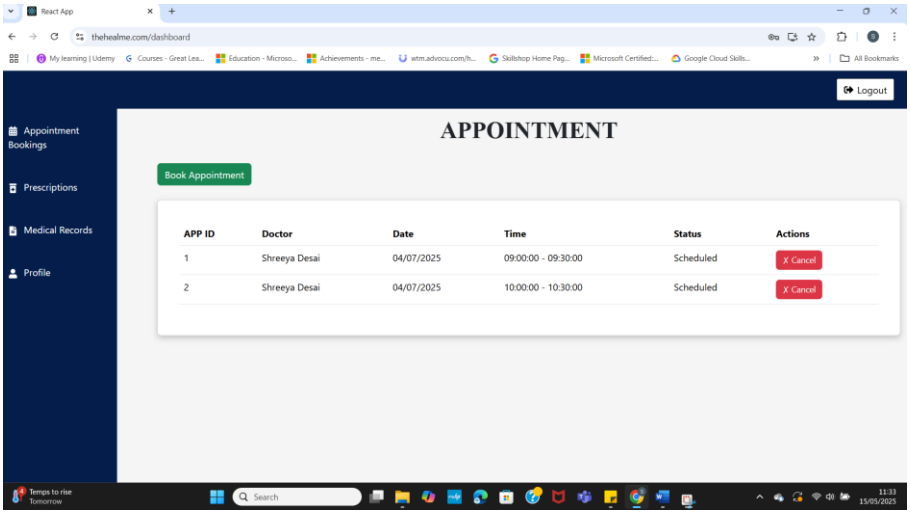
thehealme P.1: Landing/Login page.



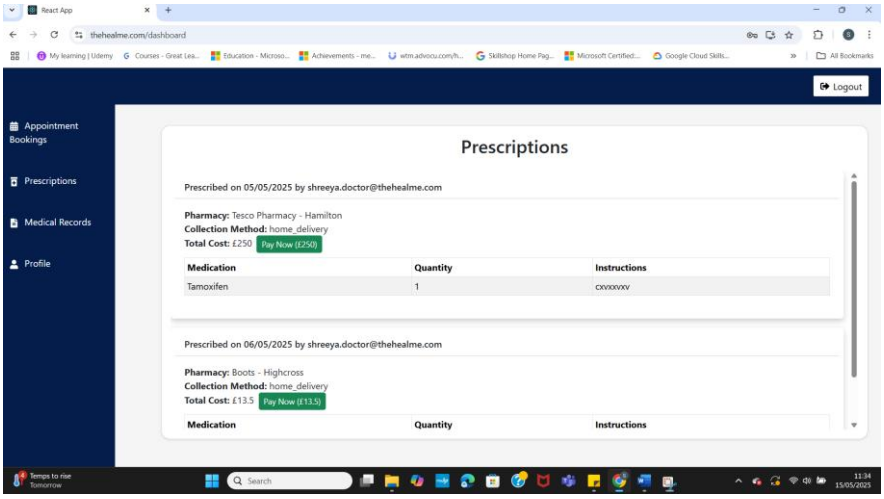
thehealme P.2: Register Page.



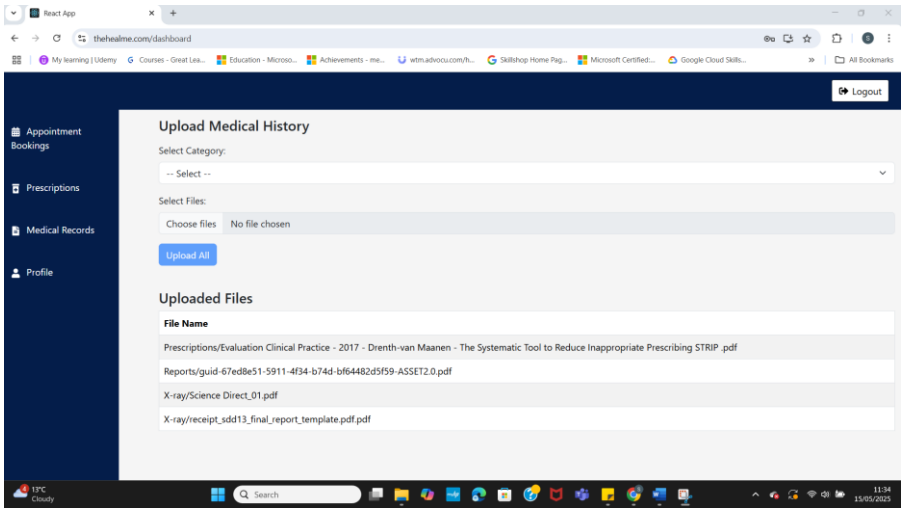
thehealme P.3: Staff login Page.



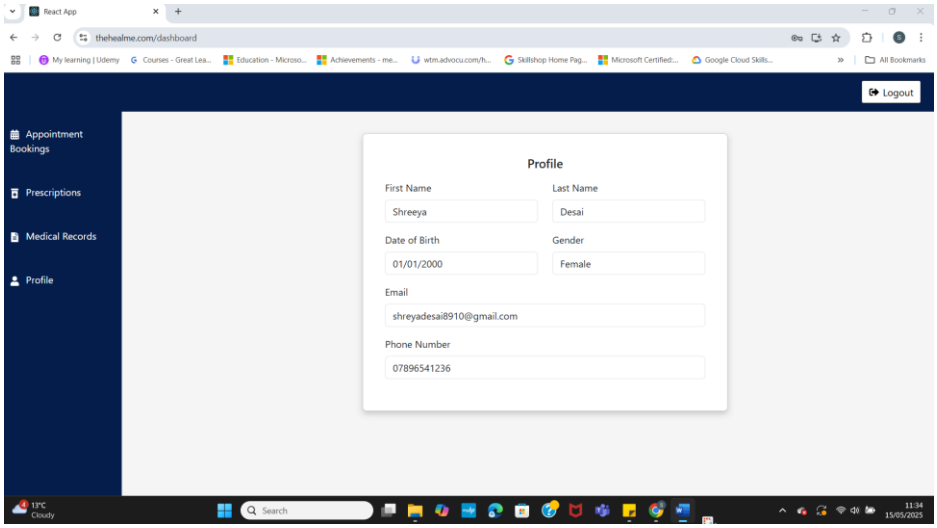
thehealme P.4: Patient Dashboard.



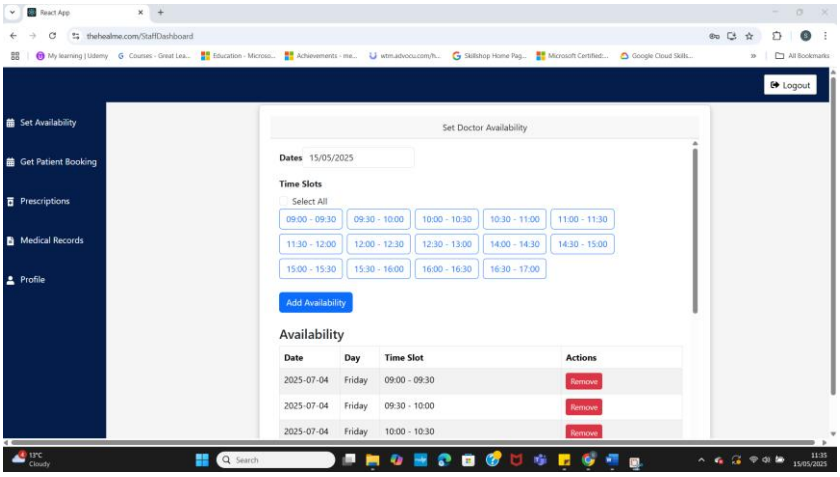
thehealme P.5: Prescriptions Screen (Patient).



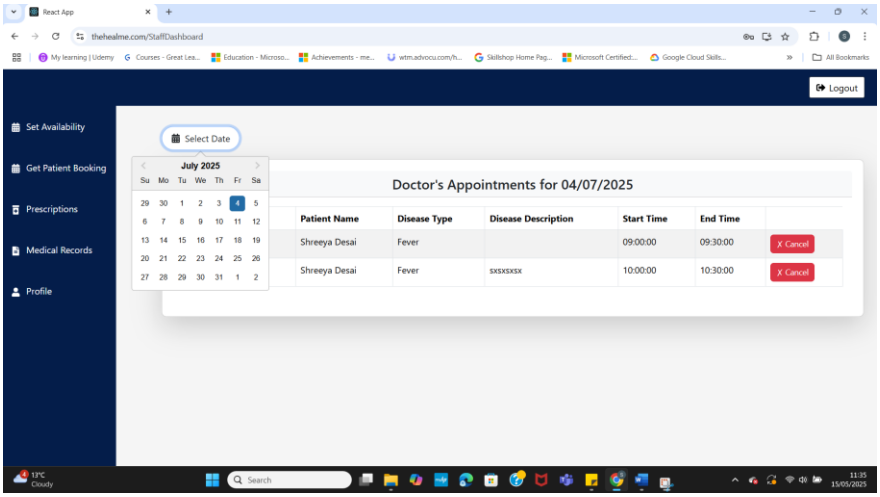
thehealme P.6: Uplaod Medical Histroy Screen(Patient).



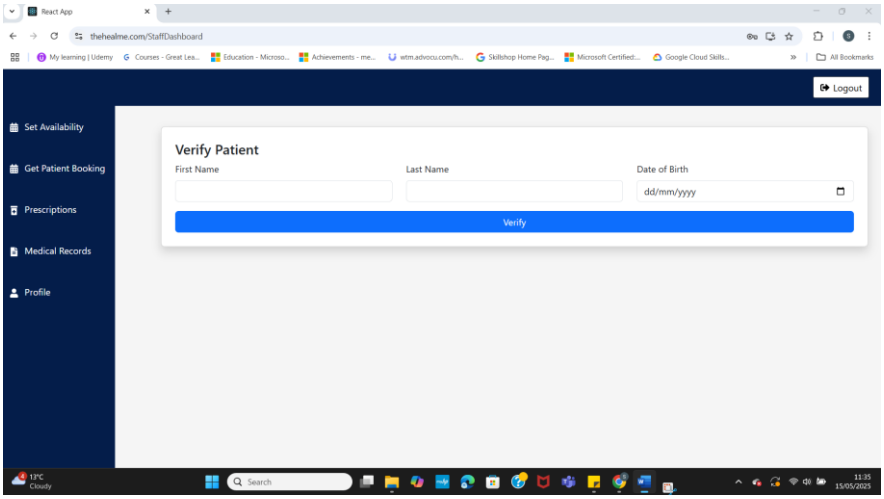
thehealme P.7: Profile(Patient).



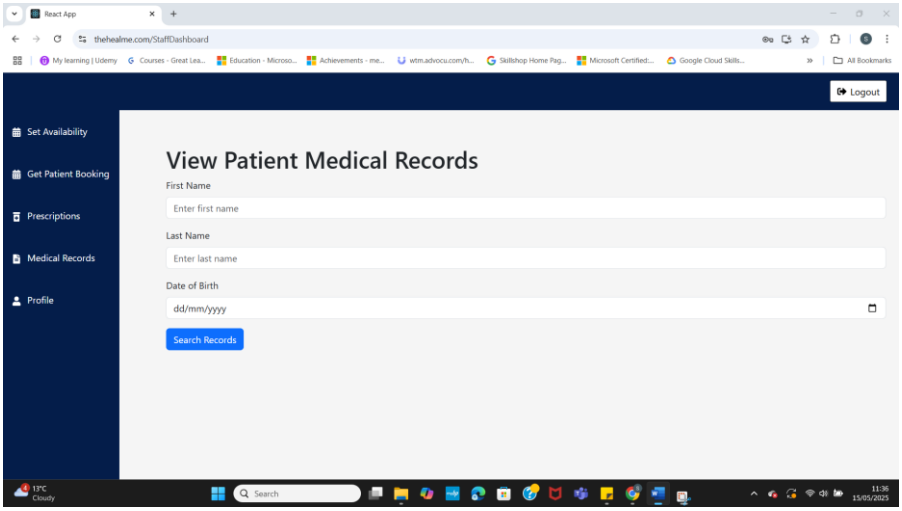
thehealme P.8: Staff Dashboard (Doctor)



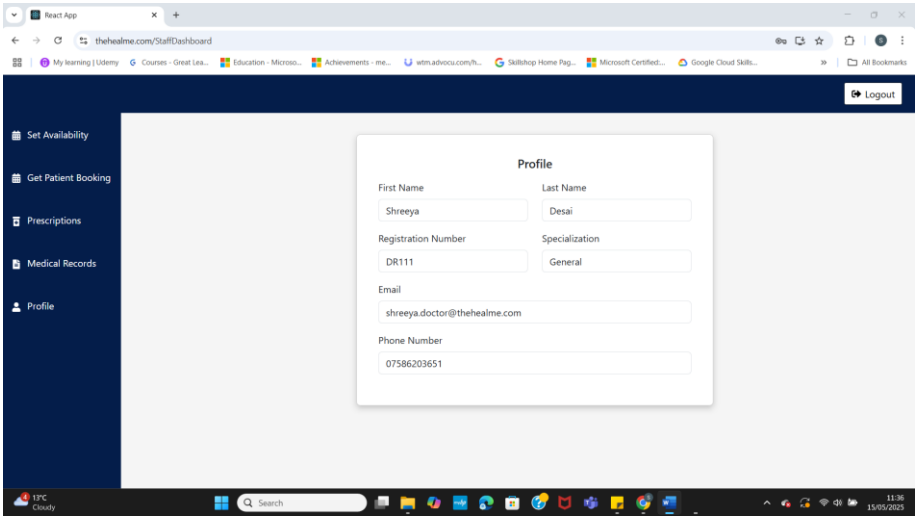
thehealme P.9: Get Patient booking (Doctor)



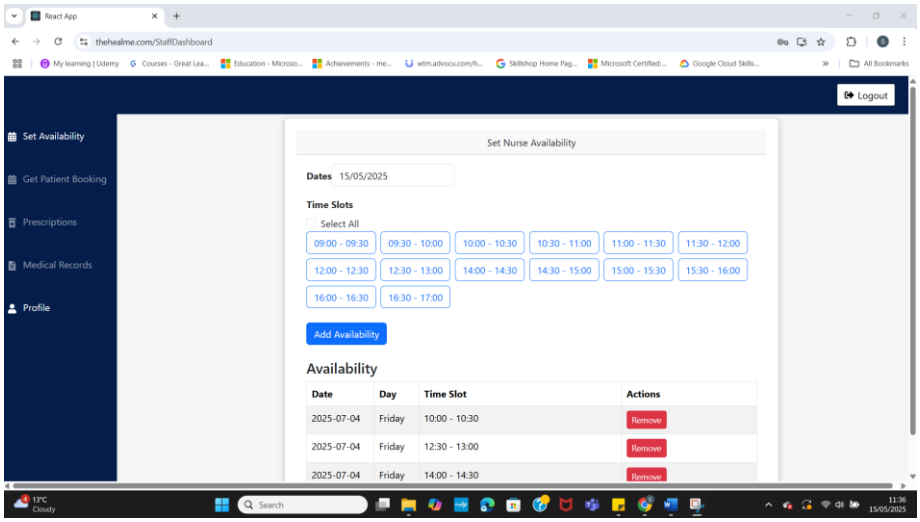
thehealme P.10: Provide Prescription.



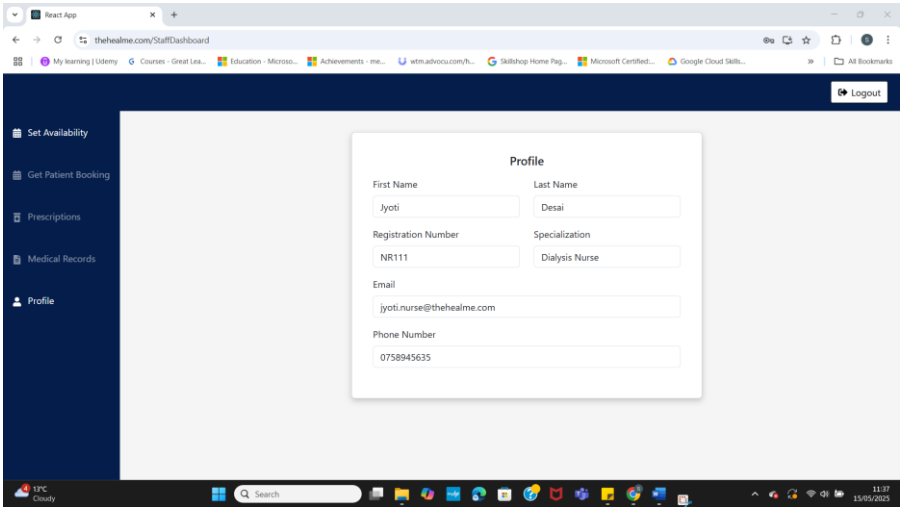
thehealme P.11: View Medical Records.



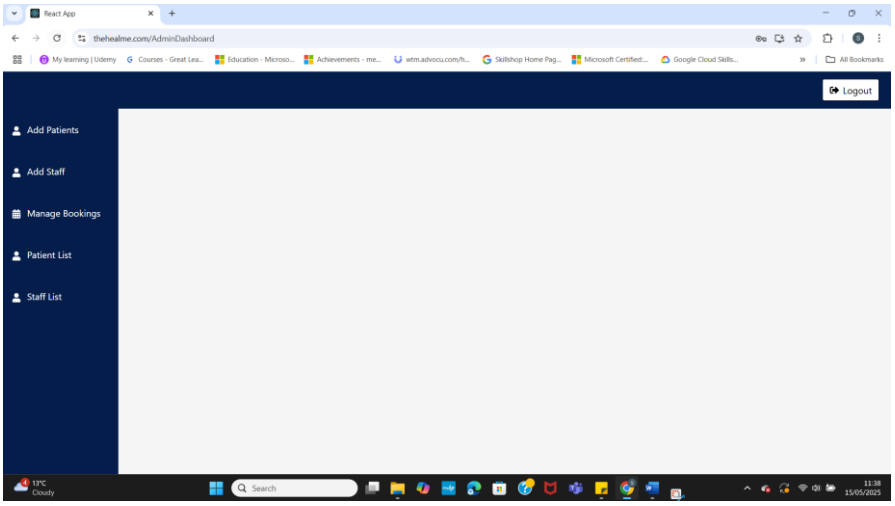
thehealme P.12: Profile (doctor).



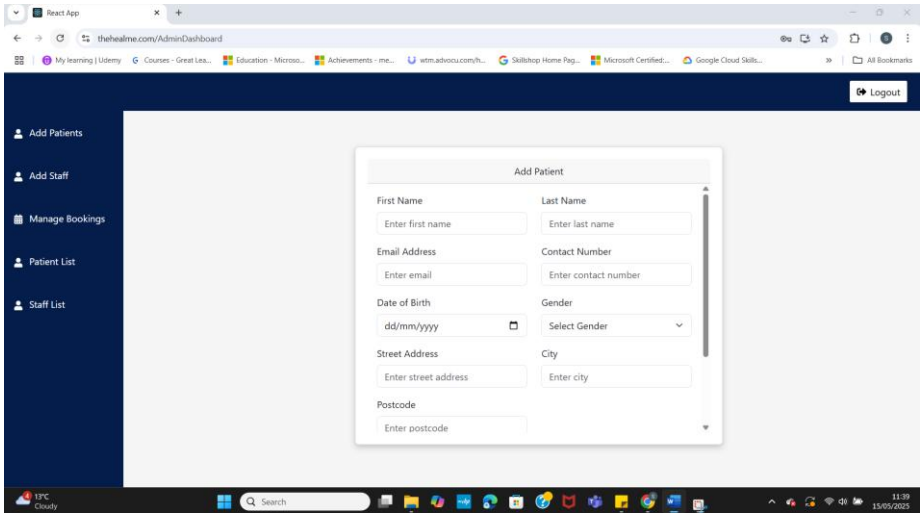
thehealme P.13: Staff Dashboard (Nurse)



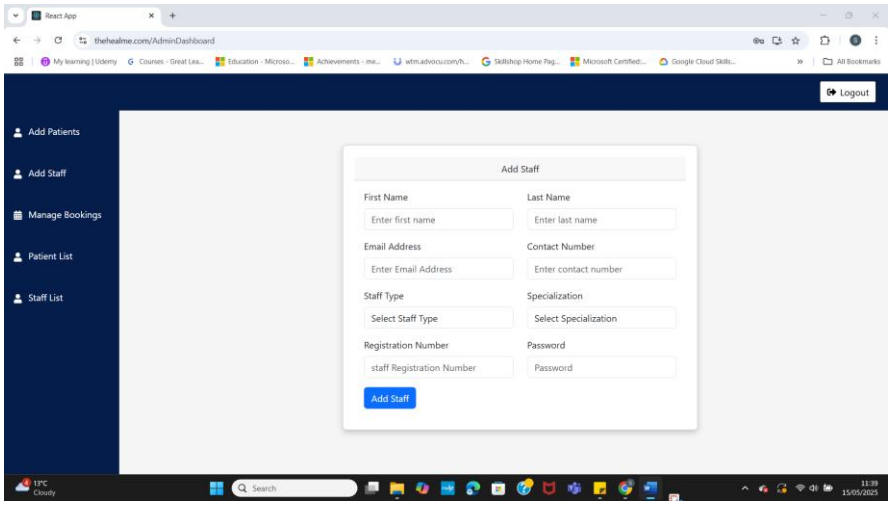
thehealme P.14: Profile(Nurse)



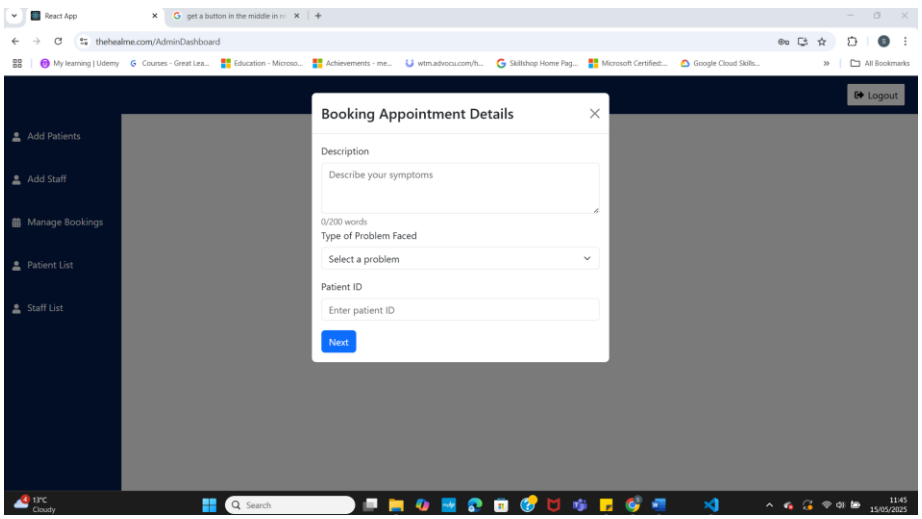
thehealme P.15: Admin dashboard



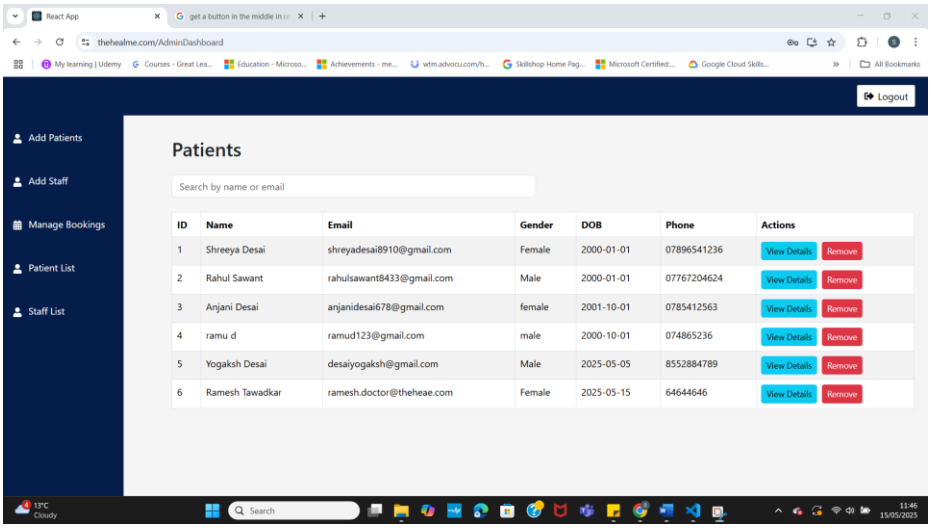
thehealme P.16: Add Patient.



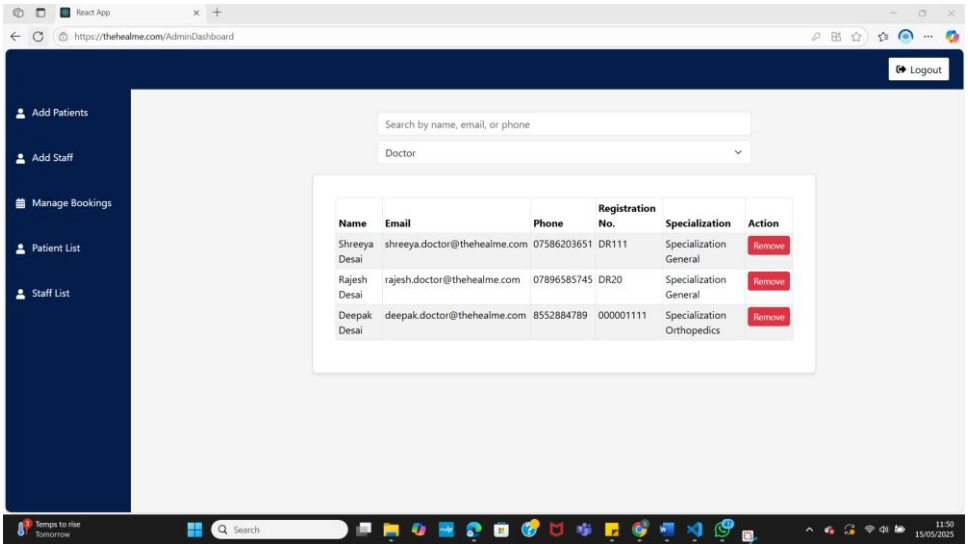
thehealme P.17: Add Staff.



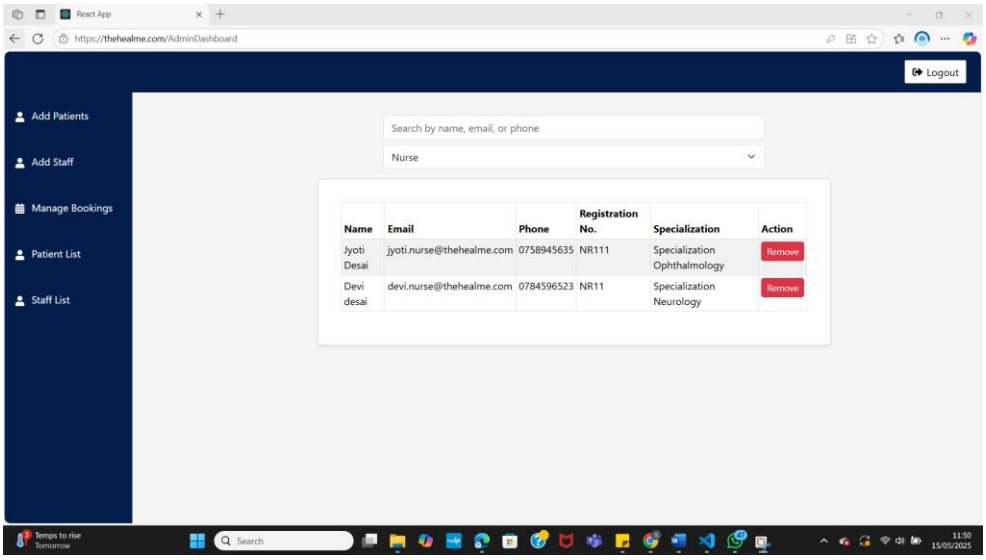
thehealme P.18: Book Appointments.



thehealme P.19: Patient List.



thehealme P.20: Staff List(Doctor)



thehealme P.21: Staff List(Nurse)