CO 328 – SOFTWARE ENGINEERING

PROJECT REPORT

HOSPITAL MANAGEMENT SYSTEM

GROUP: 13

E/15/016 S. Anojan

E/15/171 R. Kapilrajh

E/15/351 S. Thakshajini

Introduction

Hospital Management System is a system for managing all aspects of a hospital's operations such as medical, financial and administrative. It is an effective software specifically designed to fulfil various requirements in managing hospitals. The hospital management system software covers the services that unify and simplify the work of healthcare professionals as well as their interactions with patients. It deals with the collection of patient's information, booking details, etc. Traditionally, it was done manually. The main function of the system is an online booking and store patient details and doctor details and retrieve these details as and when required.

Background

When using a traditional approach to store the information in a filing system leads to many problems such as data redundancy that is the same patient data have to be recorded many times, data dependence which is incompatible with the file format and data inconsistency due to same patient data items which appear in more than one file do not get updated simultaneously in each file. There is only limited data sharing and the retrieval is not easy in a traditional approach. There is no guarantee for the security of the data. The problem arises as it is inefficient to maintain the record of a big hospital having a large number of patients when using a traditional system. Our Hospital Management System provides solutions for all these problems as it enhances information integrity by a reduction in transcription errors and duplication of information entries and improves the communication and interaction of doctors with their patients.

Aim

We aim to create a web application which helps to be aware of patient records and other critical metrics in real-time. It is user-friendly and no error is usually associated with handwriting. Patients can find doctors and book online appointments based on speciality, department and availability. The patients don't have to waste their time in booking for appointments as in the traditional hospital system. Organizing doctor schedules is very efficient and both the patients and doctors can check the schedules from anywhere when using this Hospital Management System.

Structure of the System

The system allows easy access to patient data. In our system, Admin controls the patients and manages the doctors. Admin can control many patients and manage many doctors at a time. Admin, users and doctors all have a unique id and the details such as name, email, address, age and contact no are stored in the database. Further, doctors are classified based on their specialization and department. A doctor can create many schedules based on date and time. A patient can book multiple appointments that are managed by a doctor at a time. A Booking appointment contains the name, id, email and contact no of the relevant patient and also includes a schedule that was created by the relevant doctor.

Benefits

The details of patients can be easily updated and analyzed whenever needed. Our system provides all data in a single platform and hence it improves the quality of patient care. It reduces the expenses of a traditional hospital system because of less paperwork and improved safety and efficiently engenders the distribution of medical services. The system keeps the patients and doctors interconnected. It especially helps to patients suffering from various ailments due to change in climatic conditions, increased workload and stress by providing the best interaction between the doctors. The system is keeping track of all the records day-to-day and provides excellent security of data at every level of user-system interaction and also gives robust and reliable storage and backup facilities. Altogether improved processes, digital medical records and less time consuming are the advantages of this integrated comprehensive and reliable Hospital Management System.

Requirement Analysis

Requirement analysis is essential to define the scope of our system to establish the services that the users require from our system and the constraints under which it operates and is developed. First, we identified the type of users who are going to use our Hospital Management System and the user requirements describing the activities need to be performed by them to interact with our system. Then, we analysed the functional and non-functional requirements of our system.

User Requirements

No	Users	Activities need to be performed
1.	Admin	1. Able to log in as admin in Hospital Management System. 2. Able to update his/her personal details. 3. Able to view, create, update, deactivate doctor accounts. 4. Able to view, update, deactivate patient accounts. 5. Able to view, edit, delete doctor schedules.
2.	Doctors	1. Able to log in as doctor in Hospital Management System. 2. Able to update his/her personal details. 3. Able to view, create, update, delete schedules. 4. Able to view booking details. 5. Able to view his/her patients' details.
3.	Patients	1. Able to create his/her account. 2. Able to log in as a patient in Hospital Management System. 3. Able to update his/her personal details. 4. Able to view, create, delete bookings. 5. Able to view doctor details.

Functional Requirements

Functional requirements specify what our system should do and describe the certain behaviour of a function when certain conditions are met.

No	Functions	Behaviour Description
1.	User login	Need to validate the user input, then need to check for user authentication also user type based on their login.
2.	Signup with e-mail	All the required user input fields in the signup page need to be validated.
3.	Forgot password	The system should allow the user to reset their password if they forgot or unable to login. User needs to provide their e-mail address. Based on the user e-mail, the system should send an email link to reset their password.
4.	Update user details	The system should allow the user to update their details. Also need to validate user input before update the details.
5.	Create a doctor account	The system should allow admin to create, edit doctor accounts. Also, the system should validate user input is necessary.
6.	Create patient account	The system should allow the patient to create, edit his/her accounts. Also, the system should validate user input if necessary.
7.	Create schedules	All doctors can create their schedules. The system should validate user inputs if necessary.
8.	Add, edit and delete bookings	Patients can make bookings and can cancel at any time.

9.	View, edit and delete doctor and patient account	Admin can view, edit and delete doctor and patient accounts.
10.	View booking details	A doctor can view patient booking details which are booked to him.
11.	View patient details	A doctor can view his patients' details.
12.	View Doctor	All the patients should be able to view all doctors' basic details.

Non Functional Requirements

Non-functional requirements specify how our system should behave to fulfil the functional requirements.

1. Security :
    - Unique user id - users of our system need to hold a unique user id to login and the system should not allow duplicate user id.
    - Password - users need to follow the password policy when they create a new account and the password is also encrypted.
    - Unique schedule id - Schedules that are created by the doctors should have its unique id to identify the schedules. The system should not allow duplicate schedule id.
    - Unique email - Every user accounts must be created with a unique email.

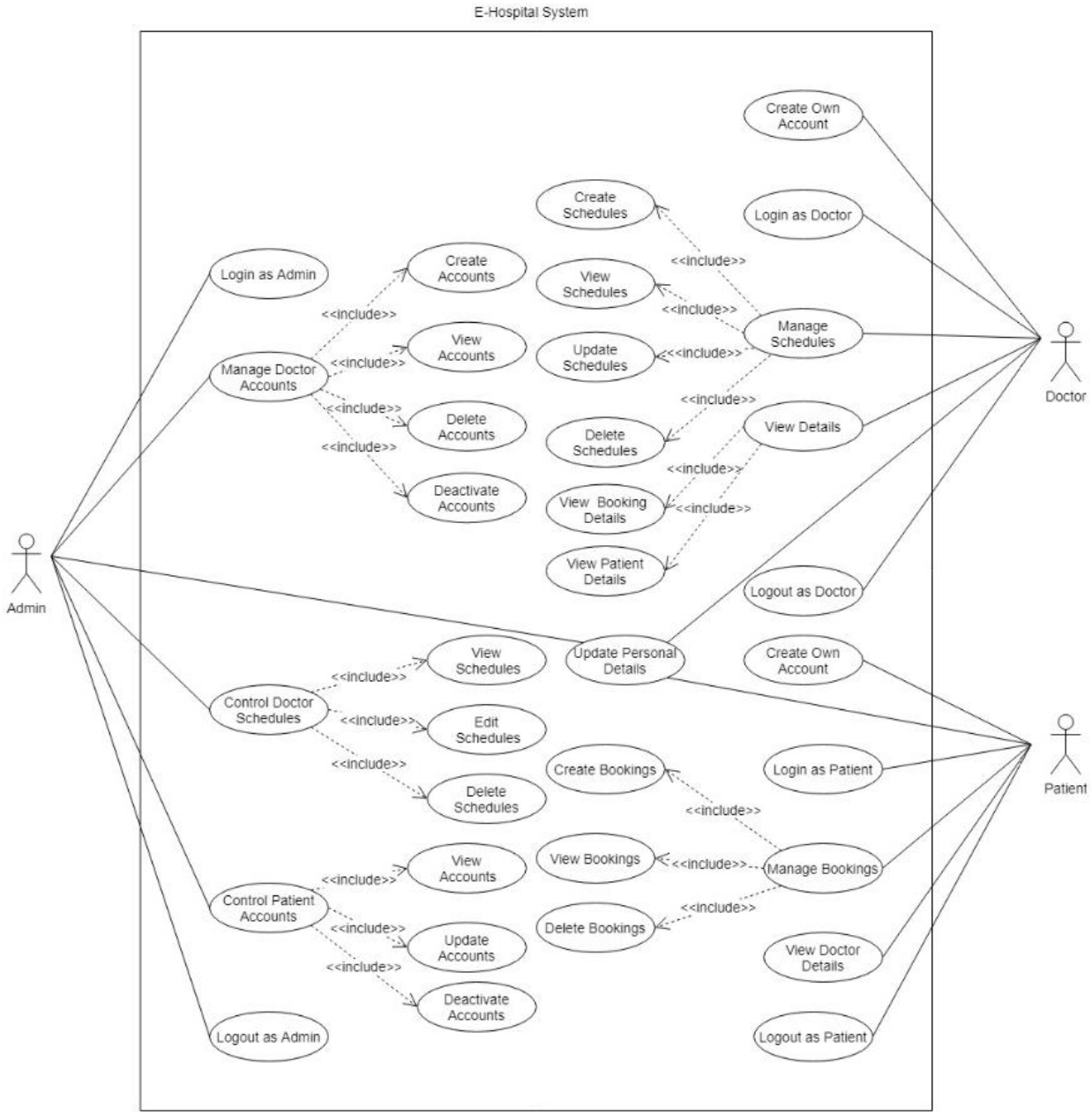
2. Performance:
    - Response time – The system provides acknowledgement in just one second once a user account is created.
    - Capacity – The system needs to support at least 1000 people at once.
    - User interface – The system user interface acknowledges within five seconds.

3. Maintainability:
    - Back up - The system provides the efficiency for data backup.
    - Contact us - The system has a footer with the address and contact details to help users to contact us regarding any issues when using the system.

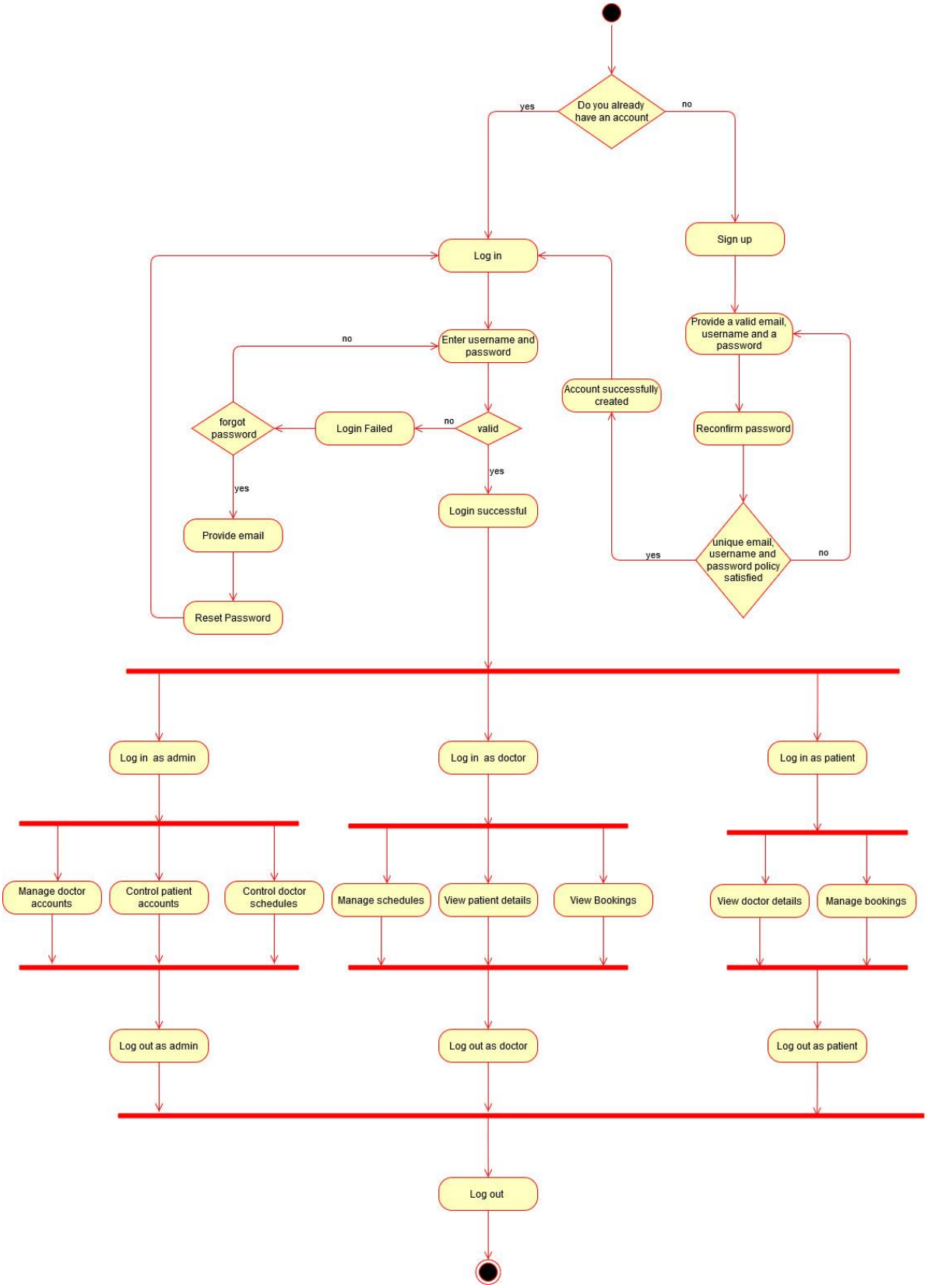
4. Reliability:
    - Availability – The system is available at any time for all the users.

DESIGN

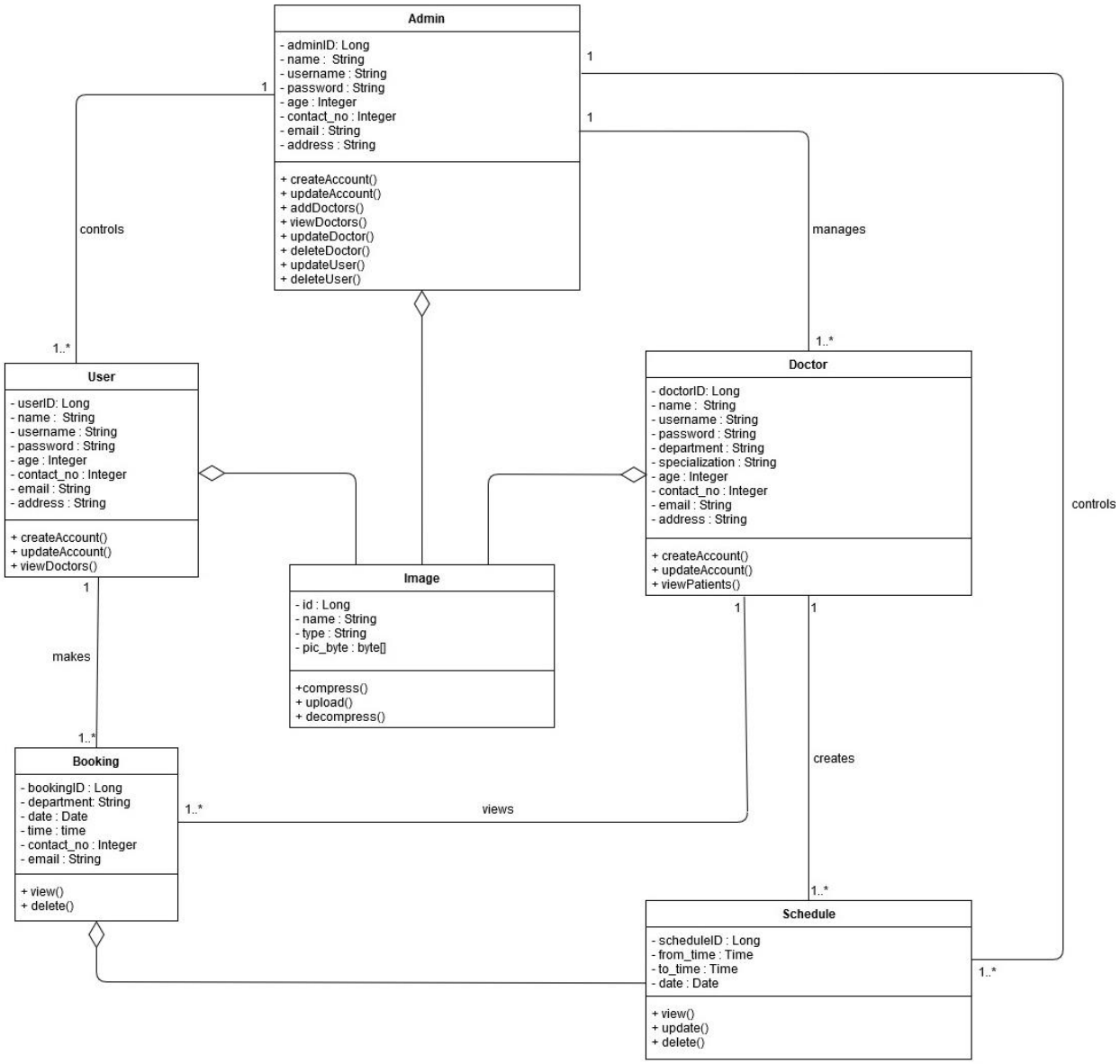
1. Use case Diagram



2. Activity Diagram



3. Class Diagram



TECHNOLOGIES

After creating the system design, we decided what are the technologies that we are going to use to implement our system.

Frontend:

Vue js is an open-source javascript framework which is lightweight and easy to learn compared to other frameworks. It focuses on the view layer and provides a lot of functionalities to develop interactive web interfaces.

Backend:

Springboot is a very convenient java-based framework comparing with others to develop a REST API for the backend. We don't need to waste our time doing a lot of coding for the configuration since spring boot has the feature of autoconfiguration and an embedded inbuilt Tomcat server.

Database:

MySQL is an open-source relational database management system to manage databases efficiently. It is a very flexible DBMS that provides advanced features and reliability.

Unit Testing:

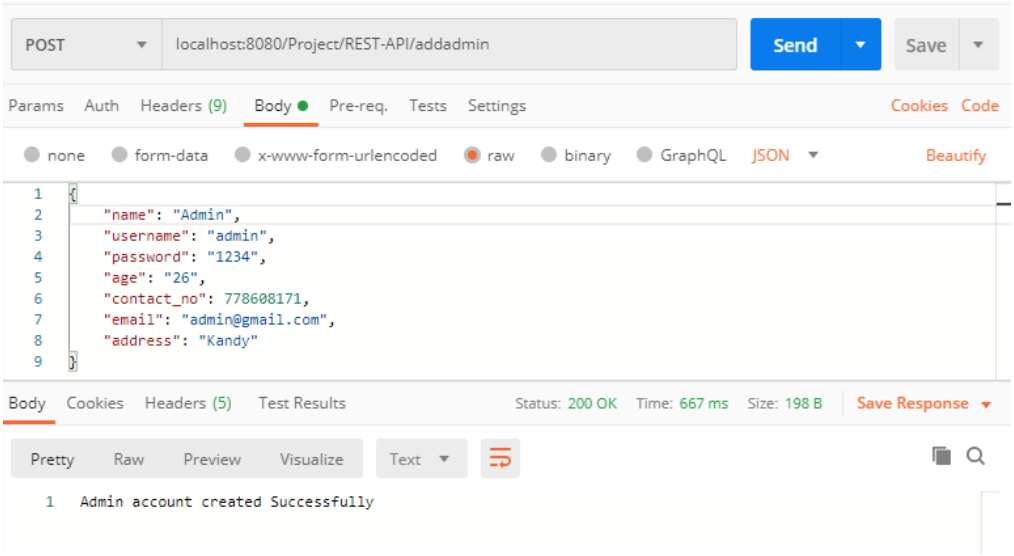
We used Postman to initially test the backend REST API endpoints. Postman provides a sleek user interface with which to make HTML requests, without the difficulty of writing a bunch of code to test an API's functionality.

Junit5 is used for the unit testing of the backend. It is a unit testing framework for java that helps to define the flow of execution of our code using different annotations.

Jest is used for the unit test of the frontend. It is a javascript testing framework that runs unit tests in parallel and helps to unit testing much faster.

IMPLEMENTATION DETAILS

After deciding the technologies, We started to implement our system. First, we developed the backend for our system by creating a REST API for the server-side application using java-based spring boot in IntelliJ IDEA. Then, we configured the backend API to connect to MySQL server @localhost:3306 to access the hospital management database created in the MySQL server which contains the tables for admin, doctor, user, schedule and booking. Then, we initially tested the API using postman by sending data in JSON format through HTTP GET, POST requests using the relevant URL endpoints.



Spring uses a JPA Java specification which is very useful to avoid writing native SQL queries in terms of tables and columns. It uses JPQL(Java Persistence Query Language) which is used to write queries in terms of java entities. Spring JPA helps to map data between a java application and a relational database using ORM(Object Relational Mapping). Hence, the data sent from postman is stored to MySQL database using a POST request and retrieved from
The database using GET request.

After successfully testing the API, we developed the frontend using vue.js in visual studio code. We created the navigator window and designed the structure of our webpage using HTML5, then styled the web page using CSS3 and finally added dynamic and behaviours to our web page using Javascript. We also created a footer at the bottom of our home page.

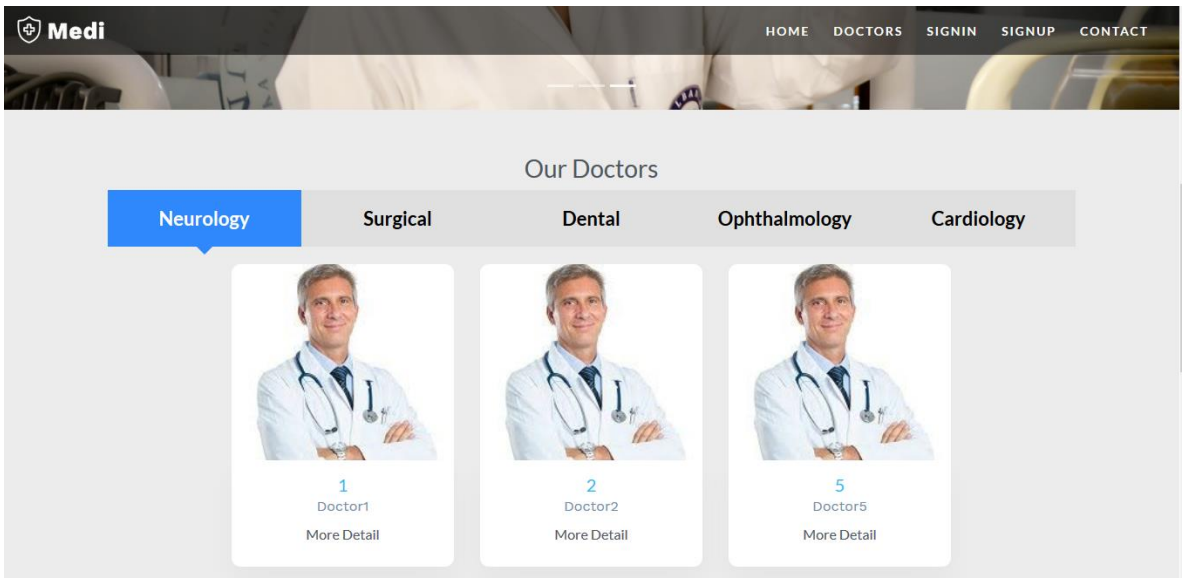


Figure: Part of the home page of Hospital Management System

Thereafter, we integrated the frontend, backend and database to create our hospital management system website. Finally, we tested our web application by performing unit tests using Junit for backend and Jest for the frontend to make sure all the required functionalities of our system are working properly as expected to be made available for the users.

UNIT TESTING

Backend:

01.

Test Case Number	01
Test Case Name	Testing username
Description	Verify that the username already exists when creating a new account
Expected Output	User can't create a new account using an already existing username
Test Case Result	Pass

Function for checking username:

UserController.java | UserControllerTest.java

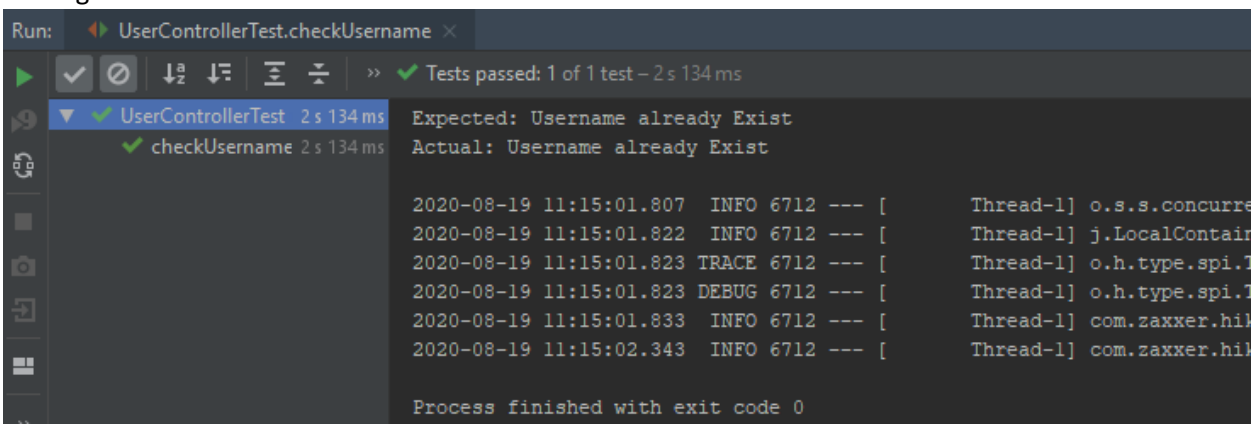
```
43      @Transactional
44      @GetMapping(value="/checkusername", params = { "username"})
45      @ResponseBody
46      public String checkUsername(@RequestParam("username") String username) {
47          if(admin_repository.isAdminExistByUsername(username) == 1
48                  || user_repository.isUserExistByUsername(username) == 1
49                  || doctor_repository.isDoctorExistByUsername(username) == 1) {
50              return "Username already Exist";
51          }
52          return "Username does not Exist";
53      }
```

Unit test for checking username:

UserController.java | UserControllerTest.java

```
19
20      @Autowired
21      private WebApplicationContext webApplicationContext;
22
23      private MockMvc mockMvc;
24
25      @Before
26      public void setUp() {
27          mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
28      }
29
30      @Test
31      public void checkUsername() throws Exception {
32          String uri = "/Project/REST-API/checkusername?username=user1";
33          MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.get(uri)).andReturn();
34
35          int status = mvcResult.getResponse().getStatus();
36          assertEquals( expected: 200, status);
37          String result = mvcResult.getResponse().getContentAsString();
38          String expected = "Username already Exist";
39          System.out.println("Expected: " + expected + "\n");
40          System.out.println("\nActual: " + result);
41          assertEquals(expected, result);
42      }
```

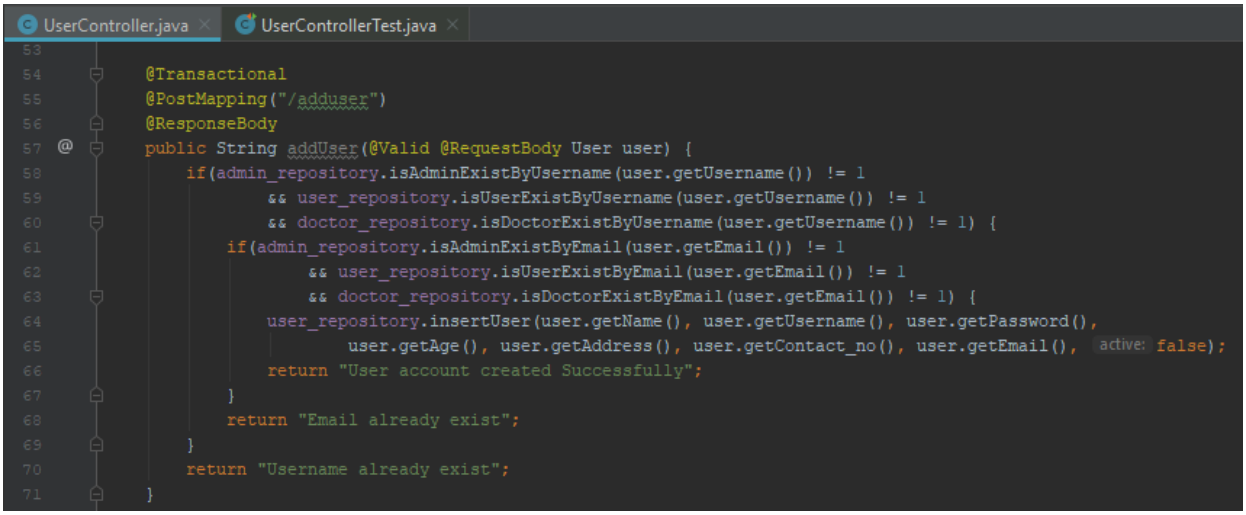
Testing Result:



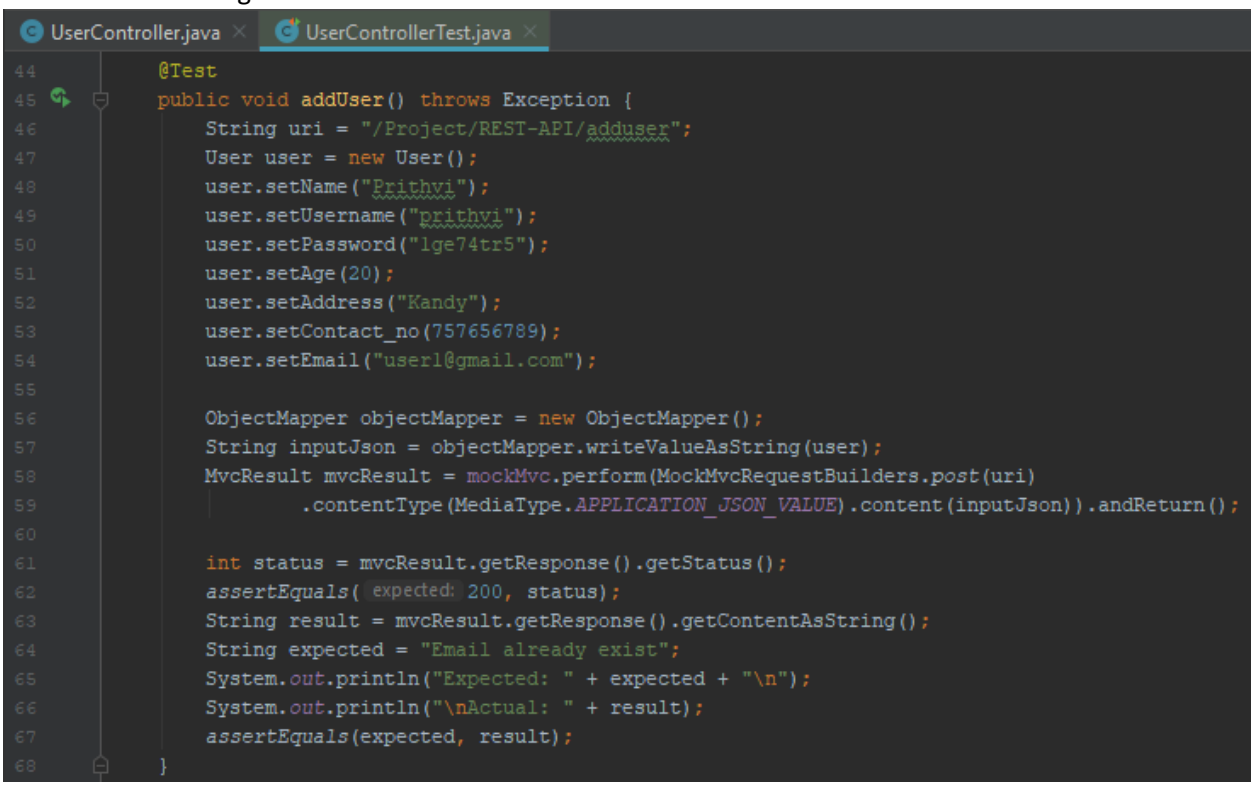
02.

Test Case Number	02
Test Case Name	Testing user signup
Description	Verify that the user can create a new account with an already existing email id
Expected Output	User can't create a new account using already existing email id
Test Case Result	Pass

Function for creating an account:

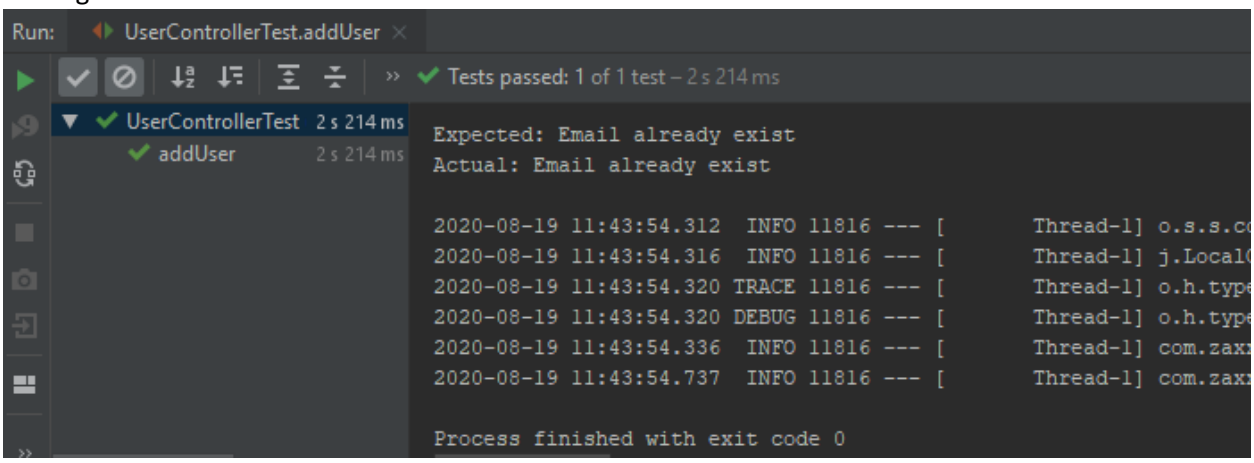


Unit test for creating an account:



```java
@Test
public void addUser() throws Exception {
    String uri = "/Project/REST-API/adduser";
    User user = new User();
    user.setName("Prithvi");
    user.setUsername("prithvi");
    user.setPassword("lge74tr5");
    user.setAge(20);
    user.setAddress("Kandy");
    user.setContact_no(757656789);
    user.setEmail("user1@gmail.com");

    ObjectMapper objectMapper = new ObjectMapper();
    String inputJson = objectMapper.writeValueAsString(user);
    MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.post(uri)
            .contentType(MediaType.APPLICATION_JSON_VALUE).content(inputJson)).andReturn();

    int status = mvcResult.getResponse().getStatus();
    assertEquals( expected: 200, status);
    String result = mvcResult.getResponse().getContentAsString();
    String expected = "Email already exist";
    System.out.println("Expected: " + expected + "\n");
    System.out.println("\nActual: " + result);
    assertEquals(expected, result);
}
```

Testing Result:



```
Run: UserControllerTest.addUser
Tests passed: 1 of 1 test – 2 s 214 ms
UserControllerTest 2 s 214 ms
addUser 2 s 214 ms
Expected: Email already exist
Actual: Email already exist
2020-08-19 11:43:54.312  INFO 11816 --- [       Thread-1] o.s.s.c
2020-08-19 11:43:54.316  INFO 11816 --- [       Thread-1] j.Local
2020-08-19 11:43:54.320 TRACE 11816 --- [       Thread-1] o.h.type
2020-08-19 11:43:54.320 DEBUG 11816 --- [       Thread-1] o.h.type
2020-08-19 11:43:54.336  INFO 11816 --- [       Thread-1] com.zaxx
2020-08-19 11:43:54.737  INFO 11816 --- [       Thread-1] com.zaxx
Process finished with exit code 0
```

03.

Test Case Number	03
Test Case Name	Testing the login function
Description	Verify that the user can sign in with wrong username or password
Expected Output	User can't sign in with wrong username or password
Test Case Result	Pass

Function for login:

LoginLogoutController.java | LoginLogoutControllerTest.java

```
@Transactional
@GetMapping(value="/login", params = { "username", "password"})
public String findByUserName(@RequestParam("username") String username, @RequestParam("password") String password) {
    Long id;
    if(user_repository.isUserExistByUsernameAndPassword(username, password) == 1) {
        id = user_repository.findByUsername(username);
        user_repository.updateUserActiveStatus( active: true, id);
        return "/user/" + id + "/home";
    }
    if(doctor_repository.isDoctorExistByUsernameAndPassword(username, password) == 1) {
        id = doctor_repository.findByUsername(username);
        doctor_repository.updateDoctorActiveStatus( active: true, id);
        return "/doctor/" + id + "/home";
    }
    if(admin_repository.isAdminExistByUsernameAndPassword(username, password) == 1) {
        id = admin_repository.findByUsername(username);
        admin_repository.updateAdminActiveStatus( active: true, id);
        return "/admin/" + id + "/home";
    }
    return "Incorrect username or password";
}
```

Unit test for login:

LoginLogoutController.java | LoginLogoutControllerTest.java

```
@Test
public void login() throws Exception {
    String uri = "/Project/REST-API/login?username=user1&password=2e3hr45i";
    MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.get(uri)).andReturn();

    int status = mvcResult.getResponse().getStatus();
    assertEquals( expected: 200, status);
    String result = mvcResult.getResponse().getContentAsString();
    String expected = "Incorrect username or password";
    System.out.println("\nExpected: " + expected);
    System.out.println("Actual: " + result + "\n");
    assertEquals(expected,result);
}
```

Test Result:

Run: LoginLogoutControllerTest.login

Tests passed: 1 of 1 test – 2 s 222 ms

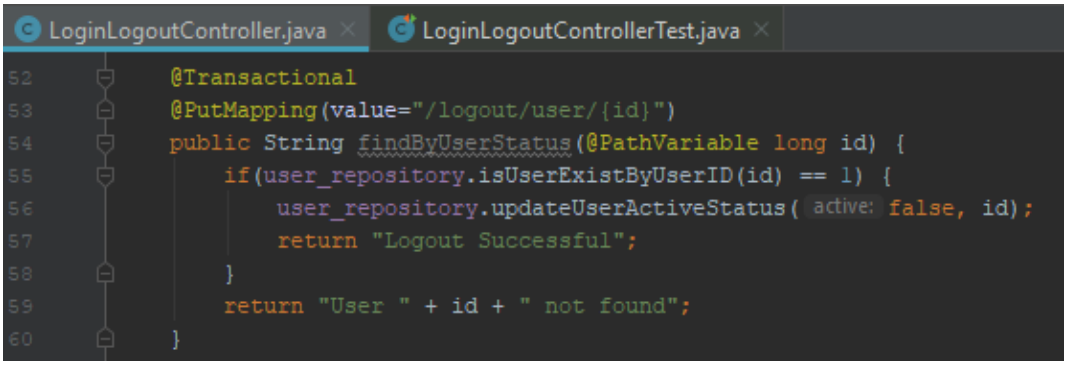
LoginLogoutContr 2 s 222 ms
login 2 s 222 ms

```
Expected: Incorrect username or password
Actual: Incorrect username or password

2020-08-19 12:50:45.477  INFO 12776 --- [       Thread-1] o.s.s.conc
2020-08-19 12:50:45.485  INFO 12776 --- [       Thread-1] j.LocalCon
2020-08-19 12:50:45.485 TRACE 12776 --- [       Thread-1] o.h.type.s
2020-08-19 12:50:45.489 DEBUG 12776 --- [       Thread-1] o.h.type.s
2020-08-19 12:50:45.493  INFO 12776 --- [       Thread-1] com.zaxxer
2020-08-19 12:50:45.577  INFO 12776 --- [       Thread-1] com.zaxxer

Process finished with exit code 0
```

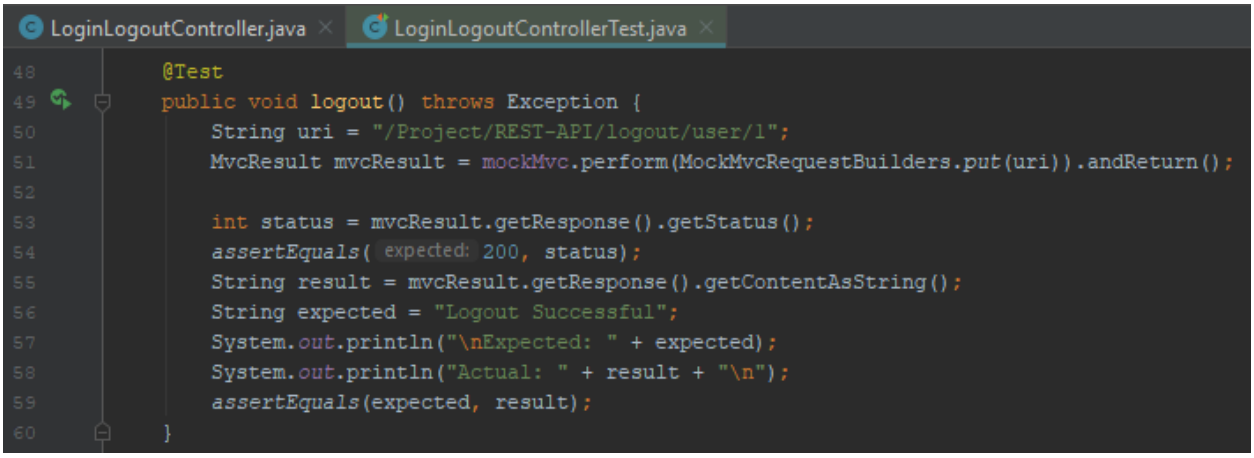
04.

Test Case Number	04
Test Case Name	Testing logout function
Description	Verify that the user can log out successfully
Expected Output	Successful logout
Test Case Result	Pass

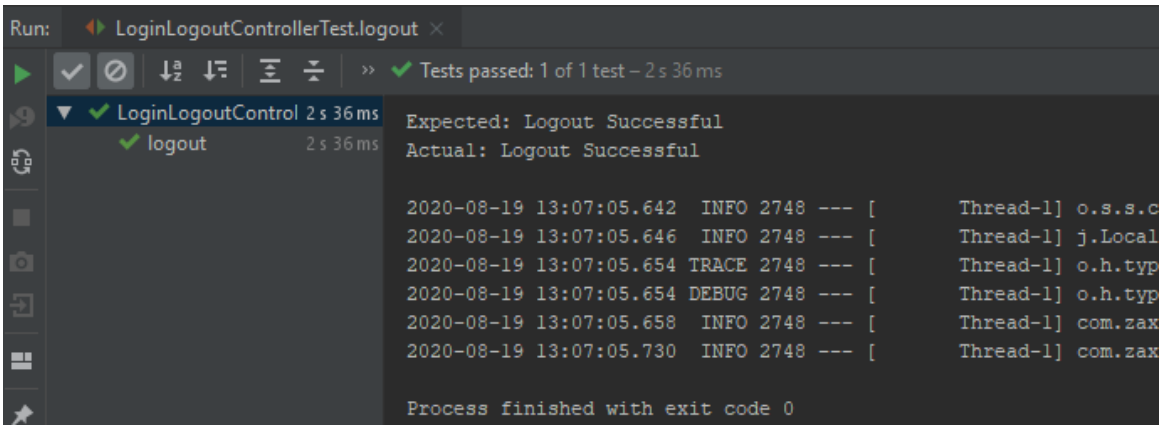
Function for logout:



Unit test for logout:



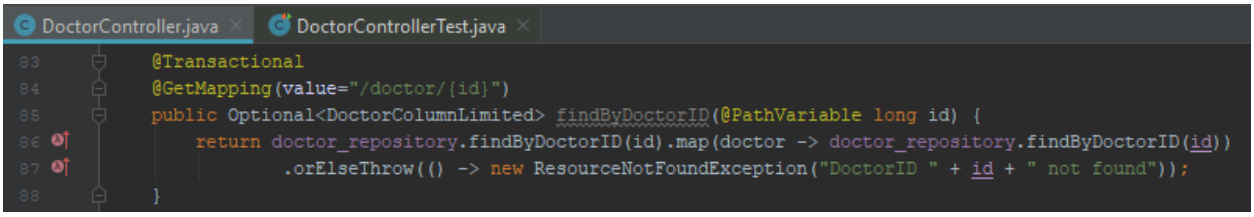
Test Result:



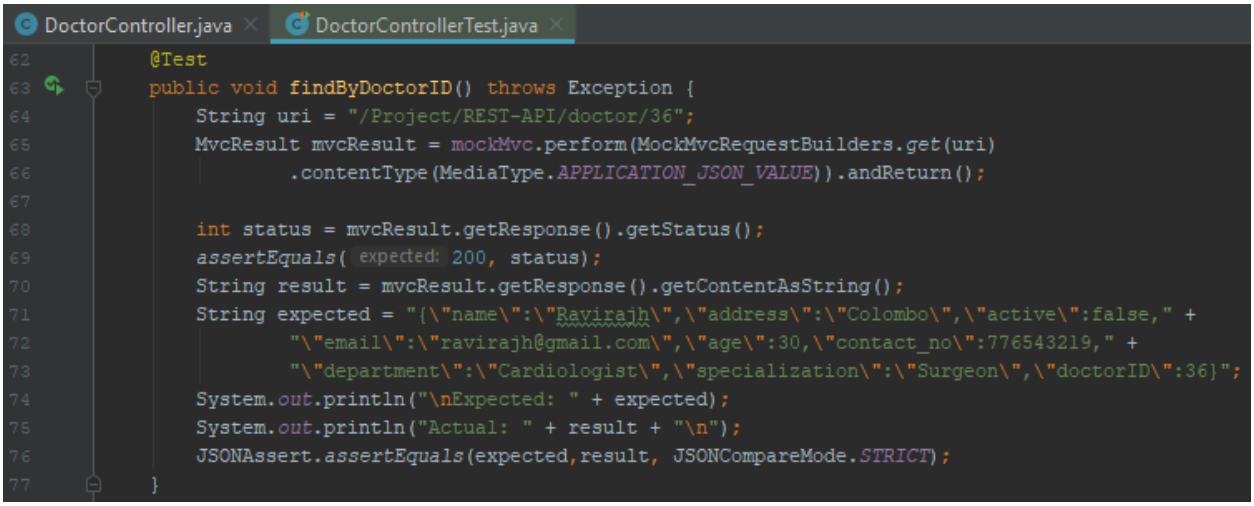
05.

Test Case Number	05
Test Case Name	Testing doctor details
Description	Getting doctor details from the database using doctor id
Expected Output	Doctor details in JSON format
Test Case Result	Pass

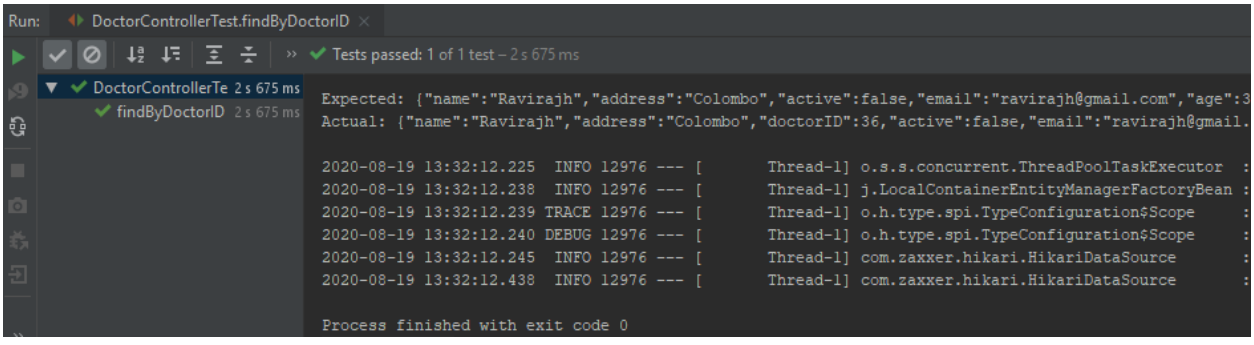
Function for getting doctor details:



Unit test for getting doctor details:



Test Result:



06.

Test Case Number	06
Test Case Name	Testing for remove a doctor account
Description	Delete doctor account from the database using doctor id
Expected Output	Doctor account removed successfully
Test Case Result	Pass

Function for removing doctor account:

DoctorController.java | DoctorControllerTest.java

```
@Transactional
@DeleteMapping(value="/deletedoctor/{id}")
public String deleteDoctor(@PathVariable long id) {
    return doctor_repository.findByDoctorID(id).map(doctor -> {
        booking_repository.deleteAllBookingsByDoctorID(id);
        scheduleRepository.deleteScheduleByDoctorID(id);
        doctorImages_repository.deleteByDoctorID(id);
        doctor_repository.deleteByDoctorID(id);
        return "Doctor Deleted Successfully";
    }).orElseThrow(() -> new ResourceNotFoundException("DoctorID " + id + " not found"));
}
```

Unit test for removing doctor account:

DoctorController.java | DoctorControllerTest.java

```
@Test
public void deleteDoctor() throws Exception {
    String uri = "/Project/REST-API/deletedoctor/36";
    MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.delete(uri)).andReturn();

    int status = mvcResult.getResponse().getStatus();
    assertEquals( expected: 200, status);
    String result = mvcResult.getResponse().getContentAsString();
    String expected = "Doctor Deleted Successfully";
    System.out.println("\nExpected: " + expected);
    System.out.println("Actual: " + result + "\n");
    assertEquals(expected, result);
}
```

Test Result:

Run: DoctorControllerTest.deleteDoctor

Tests passed: 1 of 1 test – 1 s 777 ms

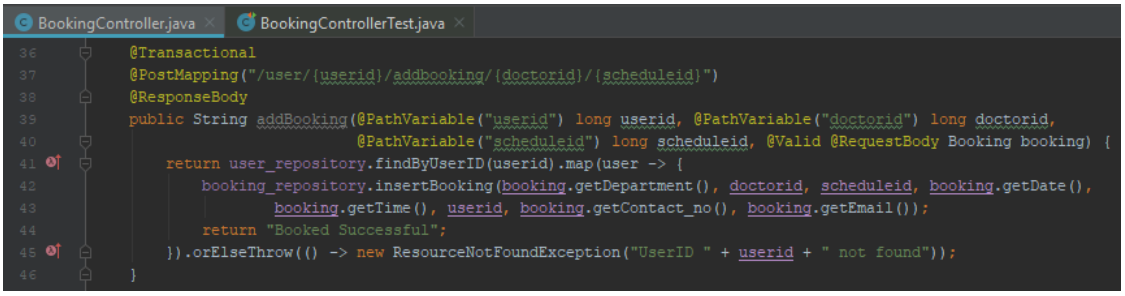
DoctorControllerTe 1 s 777 ms
deleteDoctor 1 s 777 ms

```
Expected: Doctor Deleted Successfully
Actual: Doctor Deleted Successfully

2020-08-19 13:40:04.759  INFO 3836 --- [       Thread-1] o.s.s.con
2020-08-19 13:40:04.765  INFO 3836 --- [       Thread-1] j.LocalCo
2020-08-19 13:40:04.768 TRACE 3836 --- [       Thread-1] o.h.type.
2020-08-19 13:40:04.769 DEBUG 3836 --- [       Thread-1] o.h.type.
2020-08-19 13:40:04.777  INFO 3836 --- [       Thread-1] com.zaxxe
2020-08-19 13:40:05.091  INFO 3836 --- [       Thread-1] com.zaxxe

Process finished with exit code 0
```

07.

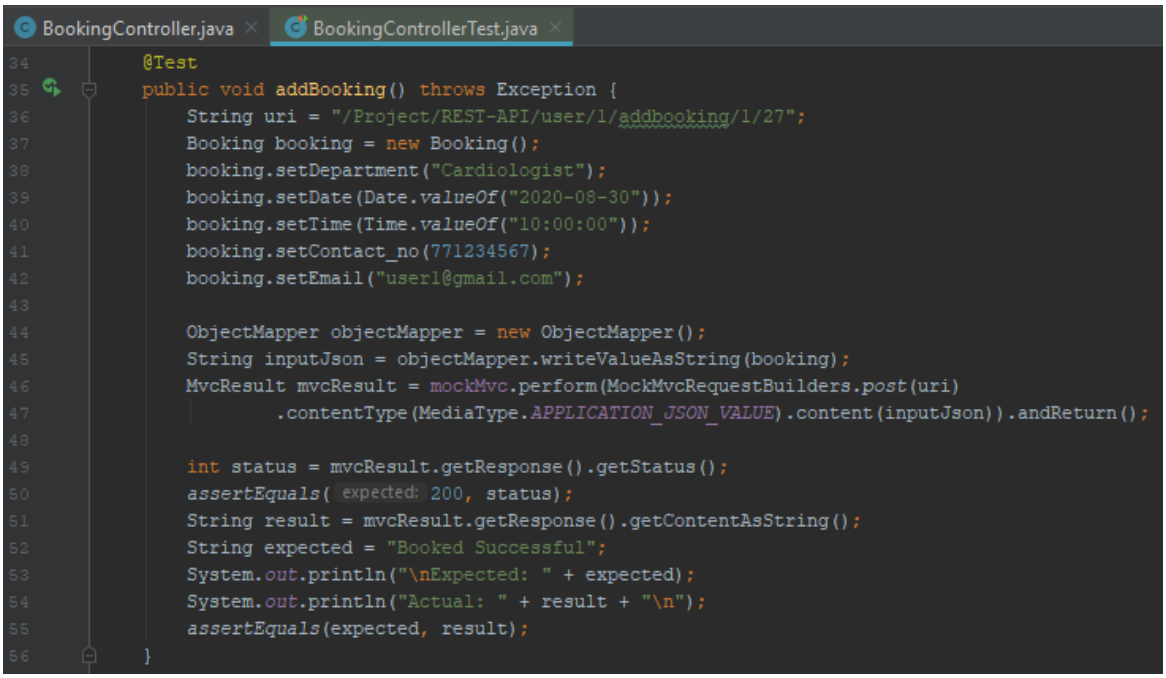
Test Case Number	07
Test Case Name	Testing for a doctor appointment
Description	Verify that a user can make an appointment for a particular doctor
Expected Output	User can make a successful appointment
Test Case Result	Pass

Function for doctor appointment:



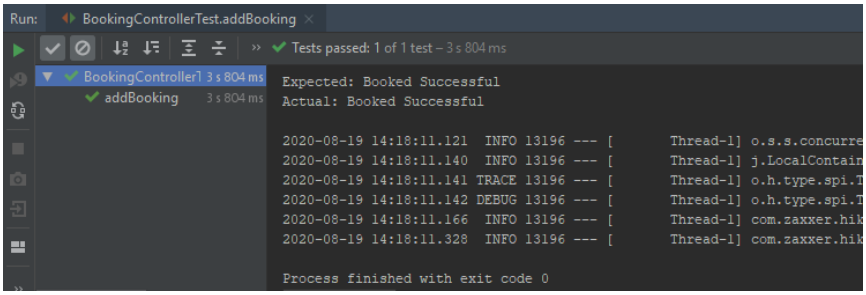
```
@Transactional
@PostMapping("/user/{userid}/addbooking/{doctorid}/{scheduleid}")
@ResponseBody
public String addBooking(@PathVariable("userid") long userid, @PathVariable("doctorid") long doctorid,
                         @PathVariable("scheduleid") long scheduleid, @Valid @RequestBody Booking booking) {
    return user_repository.findByUserID(userid).map(user -> {
        booking_repository.insertBooking(booking.getDepartment(), doctorid, scheduleid, booking.getDate(),
                booking.getTime(), userid, booking.getContact_no(), booking.getEmail());
        return "Booked Successful";
    }).orElseThrow(() -> new ResourceNotFoundException("UserID " + userid + " not found"));
}
```

Unit test for doctor appointment:



```
@Test
public void addBooking() throws Exception {
    String uri = "/Project/REST-API/user/1/addbooking/1/27";
    Booking booking = new Booking();
    booking.setDepartment("Cardiologist");
    booking.setDate(Date.valueOf("2020-08-30"));
    booking.setTime(Time.valueOf("10:00:00"));
    booking.setContact_no(771234567);
    booking.setEmail("user1@gmail.com");

    ObjectMapper objectMapper = new ObjectMapper();
    String inputJson = objectMapper.writeValueAsString(booking);
    MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.post(uri)
            .contentType(MediaType.APPLICATION_JSON_VALUE).content(inputJson)).andReturn();

    int status = mvcResult.getResponse().getStatus();
    assertEquals( expected: 200, status);
    String result = mvcResult.getResponse().getContentAsString();
    String expected = "Booked Successful";
    System.out.println("\nExpected: " + expected);
    System.out.println("Actual: " + result + "\n");
    assertEquals(expected, result);
}
```

Test Result:

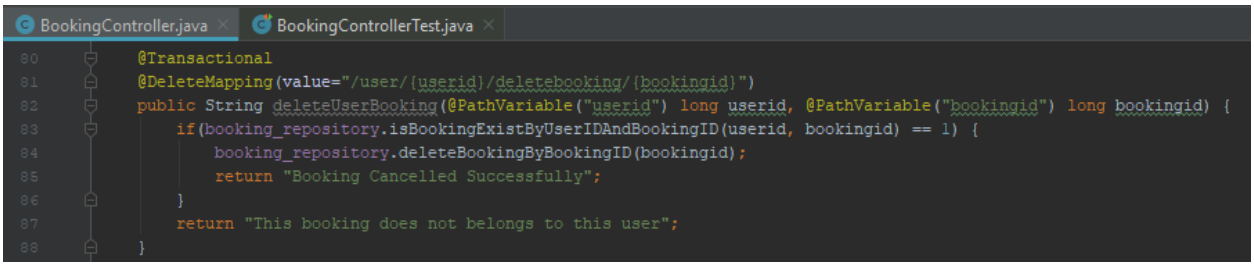


```
Run: BookingControllerTest.addBooking
Tests passed: 1 of 1 test – 3 s 804 ms
BookingController1 3 s 804 ms
addBooking 3 s 804 ms
Expected: Booked Successful
Actual: Booked Successful
2020-08-19 14:18:11.121  INFO 13196 --- [      Thread-1] o.s.s.concurre
2020-08-19 14:18:11.140  INFO 13196 --- [      Thread-1] j.LocalContain
2020-08-19 14:18:11.141 TRACE 13196 --- [      Thread-1] o.h.type.spi.T
2020-08-19 14:18:11.142 DEBUG 13196 --- [      Thread-1] o.h.type.spi.T
2020-08-19 14:18:11.166  INFO 13196 --- [      Thread-1] com.zaxxer.hik
2020-08-19 14:18:11.328  INFO 13196 --- [      Thread-1] com.zaxxer.hik
Process finished with exit code 0
```

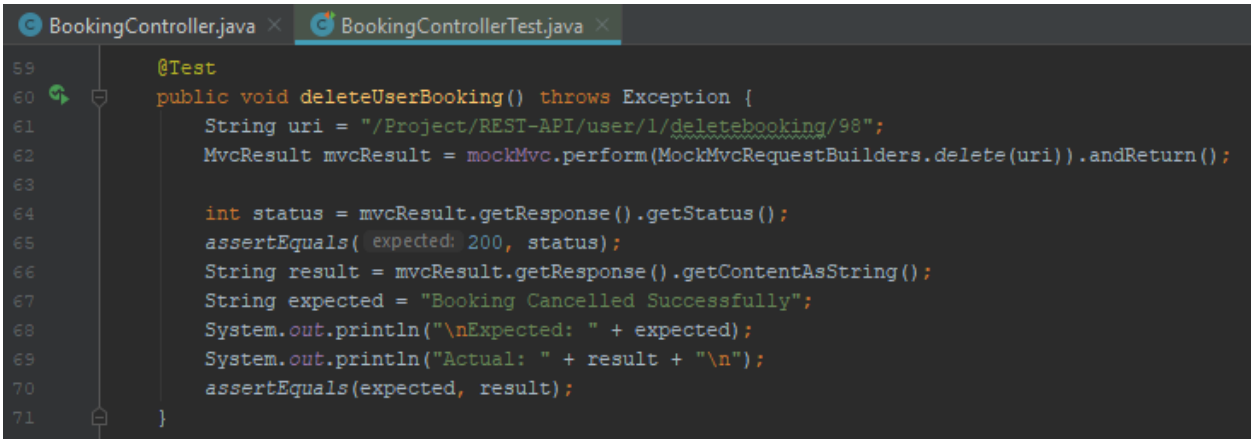
08.

Test Case Number	08
Test Case Name	Testing for cancelling an appointment
Description	Verify that a user can cancel an appointment
Expected Output	A doctor can cancel an appointment successfully
Test Case Result	Pass

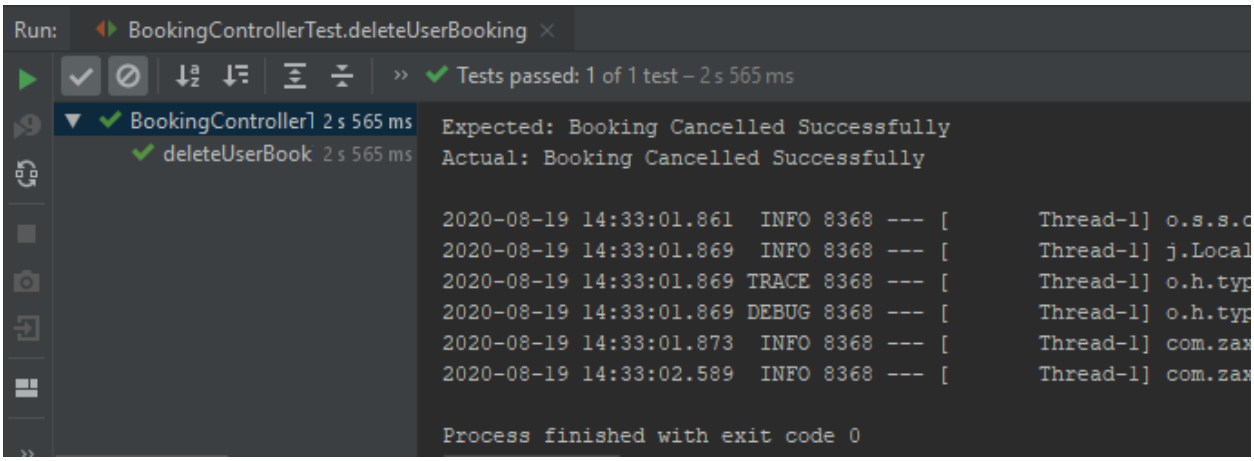
Function for cancelling an appointment:



Unit test for cancelling an appointment:



Test Result:



09.

Test Case Number	09
Test Case Name	Testing for doctors creating a schedule
Description	Verify that a doctor can create schedules, so the patients can make appointments based on doctor schedules
Expected Output	A doctor can create a schedule successfully
Test Case Result	Pass

Function for creating a schedule:

ScheduleController.java | ScheduleControllerTest.java

```
@Transactional
@PostMapping("/doctor/{doctorid}/addschedule")
@ResponseBody
public String addSchedule(@PathVariable("doctorid") long doctorid, @Valid @RequestBody Schedule schedule) {
    return doctorRepository.findByDoctorID(doctorid).map(doctor -> {
        if(scheduleRepository.isScheduleExistByDoctorIDAndDate(doctorid, schedule.getDate()) == 1) {
            return "Schedule Already Added";
        }
        scheduleRepository.insertSchedule(schedule.getFrom_time(), schedule.getTo_time(),
                schedule.getDate(), doctorid);
        return "Schedule Added Successful";
    }).orElseThrow(() -> new ResourceNotFoundException("DoctorID " + doctorid + " not found"));
}
```

Unit test for creating a schedule:

ScheduleController.java | ScheduleControllerTest.java

```
@Test
public void addSchedule() throws Exception {
    String uri = "/Project/REST-API/doctor/1/addschedule";
    Schedule schedule = new Schedule();
    schedule.setFrom_time(Time.valueOf("08:00:00"));
    schedule.setTo_time(Time.valueOf("12:00:00"));
    schedule.setDate(Date.valueOf("2020-08-29"));

    ObjectMapper objectMapper = new ObjectMapper();
    String inputJson = objectMapper.writeValueAsString(schedule);
    MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.post(uri)
            .contentType(MediaType.APPLICATION_JSON_VALUE).content(inputJson)).andReturn();

    int status = mvcResult.getResponse().getStatus();
    assertEquals( expected: 200, status);
    String result = mvcResult.getResponse().getContentAsString();
    String expected = "Schedule Added Successful";
    System.out.println("\nExpected: " + expected);
    System.out.println("Actual: " + result + "\n");
    assertEquals(expected, result);
}
```

Test Result:

Run: ScheduleControllerTest.addSchedule

Tests passed: 1 of 1 test – 4 s 215 ms

ScheduleController 4 s 215 ms
addSchedule 4 s 215 ms

```
Expected: Schedule Added Successful
Actual: Schedule Added Successful

2020-08-19 15:10:28.662  INFO 1132 --- [       Thread-1] o.s.s.
2020-08-19 15:10:28.673  INFO 1132 --- [       Thread-1] j.Loca
2020-08-19 15:10:28.674 TRACE 1132 --- [       Thread-1] o.h.ty
2020-08-19 15:10:28.676 DEBUG 1132 --- [       Thread-1] o.h.ty
2020-08-19 15:10:28.682  INFO 1132 --- [       Thread-1] com.za
2020-08-19 15:10:28.759  INFO 1132 --- [       Thread-1] com.za

Process finished with exit code 0
```

10.

Test Case Number	10
Test Case Name	Testing for doctors deleting their schedule
Description	Verify that a doctor can delete their schedules using schedule id
Expected Output	A doctor can delete schedule successfully
Test Case Result	Pass

Function for deleting schedule:

ScheduleController.java | ScheduleControllerTest.java

```
82      @Transactional
83      @DeleteMapping(value="/doctors/{id}/deleteschedules/{scheduleid}")
84      public String deleteSchedule(@PathVariable (value = "id") long doctorid,
85                                   @PathVariable (value = "scheduleid") long scheduleid) {
86          if(scheduleRepository.isScheduleExistByDoctorIDAndScheduleID(doctorid, scheduleid) == 1) {
87              booking_repository.deleteBookingByScheduleIDAndDoctorID(doctorid, scheduleid);
88              scheduleRepository.deleteScheduleByScheduleID(scheduleid);
89              return "Schedule Deleted Successfully";
90          }
91          return "This schedule does not belongs to this doctor";
92      }
```

Unit test for deleting schedule:

ScheduleController.java | ScheduleControllerTest.java

```
56      @Test
57      public void deleteSchedule() throws Exception {
58          String uri = "/Project/REST-API/doctors/1/deleteschedules/29";
59          MvcResult mvcResult = mockMvc.perform(MockMvcRequestBuilders.delete(uri)).andReturn();
60
61          int status = mvcResult.getResponse().getStatus();
62          assertEquals( expected: 200, status);
63          String result = mvcResult.getResponse().getContentAsString();
64          String expected = "Schedule Deleted Successfully";
65          System.out.println("\nExpected: " + expected);
66          System.out.println("Actual: " + result + "\n");
67          assertEquals(expected, result);
68      }
```

Test Result:

Run: ScheduleControllerTest.deleteSchedule

Tests passed: 1 of 1 test – 1 s 698 ms

ScheduleController 1 s 698 ms
deleteSchedule 1 s 698 ms

```
Expected: Schedule Deleted Successfully
Actual: Schedule Deleted Successfully

2020-08-19 15:26:39.654  INFO 5916 --- [       Thread-1] o.s.s
2020-08-19 15:26:39.667  INFO 5916 --- [       Thread-1] j.Loc
2020-08-19 15:26:39.668 TRACE 5916 --- [       Thread-1] o.h.t
2020-08-19 15:26:39.669 DEBUG 5916 --- [       Thread-1] o.h.t
2020-08-19 15:26:39.675  INFO 5916 --- [       Thread-1] com.z
2020-08-19 15:26:39.739  INFO 5916 --- [       Thread-1] com.z

Process finished with exit code 0
```

Frontend:

11.

Test Case Number	11
Test Case Name	Testing for enabling a button
Description	Verify that the signup button is enabled when all the required fields are filled with a strong password
Expected Output	If any required field is not filled or the password strength is low then the button should not be enabled
Test Case Result	Pass

Unit test:

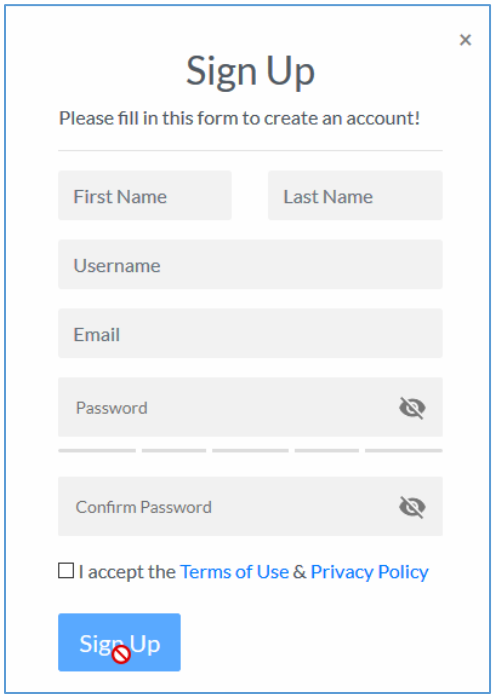
Signup.spec.js × Signup.vue

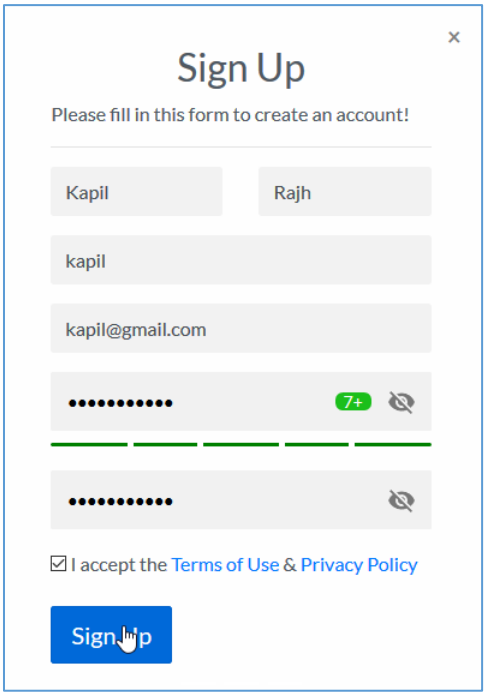
tests > unit > Signup.spec.js > ...

```
/* eslint-disable no-undef */
import { mount } from '@vue/test-utils'
import Signup from '@/components/Signup'

describe('Signup', () => {
  test('if any required field is not filled, signup button should be disabled', () => {
    const wrapper = mount(Signup, {
      data () {
        return {
          user: {
            fname: '',
            lname: '',
            username: '',
            email: '',
            password1: '',
            password2: '',
            selected: false
          },
          password_score: 0
        }
      }
    })
    expect(wrapper.find('#signup_button').element.hasAttribute('disabled')).toBe(true)
  })

  test('if all required field are filled,signup button should be enabled', () => {
    const wrapper = mount(Signup, {
      data () {
        return {
          user: {
            fname: 'Kapil',
            lname: 'Rajh',
            username: 'kapil',
            email: 'kapil@gmail.com',
            password1: 'engpdn2020@',
            password2: 'engpdn2020@',
            selected: true
          },
          password_score: 4
        }
      }
    })
    expect(wrapper.find('#signup_button').element.hasAttribute('disabled')).toBe(false)
  })
})
```

Test Result:

```
SQL CONSOLE  PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PASS tests/unit/Signup.spec.js
  Signup
    √ if any required field is not filled, signup button should be disabled (55ms)
    √ if all required field are filled,signup button should be enabled (27ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        1.356s, estimated 7s
Ran all test suites matching /sign/i.

Watch Usage: Press w to show more.
```





Summary

Taking into account all the mentioned details, we can conclude that the hospital management system is the inevitable part of the lifecycle of the modern medical institution. It automates numerous daily operations and enables smooth interactions of the users. Developing the hospital system software is a great opportunity to create the distinct, efficient and fast delivering healthcare model. Implementation of this hospital management system helps to store all the kinds of records, provide coordination and user communication. Overall, It will be a very userfriendly system controlled by admin to manage the interactions between doctors and patients efficiently and effectively in a hospital.