Test Specification

Group Name: Gofies

January 5, 2025

Course Information

BLG 411E - Software Engineering

Project Title

Gofies Hospital Management System

Team Members

- Abdullah Shamout - Project Manager, DevOps Team Leader, AI Algorithms Engineer
- Berkay Gürsu - Frontend Team Leader, Backend Developer
- İbrahim Halil Marsil - Backend Team Leader, Frontend Developer
- Mustafa Can Çalışkan - Database Team Leader, Testing Developer
- Rasim Berke Turan - Backend Developer, Frontend Developer, Client Representative
- Sarper Öztorun - Frontend Developer, Testing Developer, Meeting Organizer
- Yusuf Emir Sezgin - UI/UX Team Leader, Frontend Developer
- Yusuf Şahin - Testing Team Leader, AI Algorithms Engineer

Contents

1 Introduction

The purpose of this document is to outline the test specification for the Gofies Hospital Management System (GHMS). The document details the testing strategy, test subjects, testing types, and test results.

1.1 Goal

The goal of this test specification document is to ensure that the Gofies Hospital Management System (GHMS) meets the expectations and requirements set by both the development team and the stakeholders.

2 Test Plan

2.1 Testing Strategy

We have used a combined approach of the Big Bang and Incremental approaches since this is the strategy aligns with our needs and situation mostly. Our situation consists of 2 main parts:

- Web Application
- AI ChatBot

These 2 parts of the Gofies consist of much more complicated and integrated inner components. These components use different programming languages (i.e., Python and JavaScript) and different programming paradigms. Gofies leverages a cloud service to host these components as a whole. In the end, we had to use a kind of mixed testing approach.

For tests of the each component itself, we conducted the Incremental Approach and this provided us easy isolation and also error detection. For tests of the whole system, we conducted the Big Bang Approach since this approach helped us observing the system from a more abstract view and this made easier to see abnormalies in the system.

We used a different testing database to isolate every unit test each other. After, unit tests were done the integration tests were done in this testing database to avoid any conflict in the main database.

2.2 Test Subjects

Based on the package diagrams (Figures 2 and 3 in our Design Specification Document), the following data types are subject to integration testing:

- **User Data:**
  - Captured from the frontend via the user interface.
  - Passed through the Nginx proxy for session management.
  - Interacts with the backend API for validation, authentication, and database storage.
- **Patient Records:**
  - Retrieved or updated via the Hospital Database system.
  - Accessed by the Registration Portal, Medical Records Module, and Appointment Scheduler.
  - Integrated with the Chatbot for queries and assistance.
- **Appointment Data:**
  - Scheduled via the Registration Portal and Appointment Scheduler.
  - Stored in the Hospital Database and accessed by the doctor and patient.
- **Lab Results:**
  - Processed by the Lab Module.
  - Transmitted to the Prescription Module and stored in the Hospital Database.
  - Retrieved by doctors or patients upon request.

- **AI-Generated Recommendations:**
  - Produced by the AI Backend API using the AI Service.
  - Involves MongoDB for query handling.
  - Integrated with the frontend for delivering recommendation interfaces.

Each of these data types requires rigorous integration testing to ensure seamless communication and data flow between components, such as the frontend, backend API, Hospital Database, and AI Service.

2.3 Black Box Testing

- **Classification of Input Values:** Input values were systematically divided into valid and invalid equivalence classes. Each class was carefully defined to encompass all possible variations.
- **Creation of Test Scenarios:** Representative test cases were developed for each equivalence class to ensure comprehensive coverage.
- **Documentation of Test Cases:** A table was prepared to document the test cases, including:
  - Description of the test case
  - Test input values
  - Expected output

  These test cases were mapped to relevant use cases for traceability.

For detailed work, please check Section 5.1.

2.4 White Box Testing

- **Packages:** The project is organized into modular packages, each serving a specific purpose:
  - Core functionalities
  - Data processing
  - API integration
- **Modules:** Each package contains multiple modules:
  - Implements features
  - Handles data validation
  - Manages database operations
- **Functions:** Key functions within the modules:
  - Processes input data
  - Validates input
  - Connects to the database

For detailed work, please check Section 5.2.

3 Additional Tests

3.1 Security Testing

The security testing of GHMS aimed to identify and address potential vulnerabilities that could compromise the integrity, confidentiality, and availability of the system. By conducting thorough assessments, including network scanning and service enumeration, the focus was placed on uncovering weaknesses in SSH, HTTP, and HTTPS services. The testing process leveraged tools like Nmap to detect open ports, outdated software, and misconfigurations that could be exploited by malicious actors.

3.1.1 XSS Injection Analysis

Injected Payloads:

Top 100 XSS injection payload has applied on this website manually. Output of two of them has shown below.

- **Targets:**
  - **Email field:** Tested by injecting script tags directly in the email input area.
  - **Password field:** Not directly tested but assumed to follow similar validation.
- **Observations**
  - Email Field Test 1

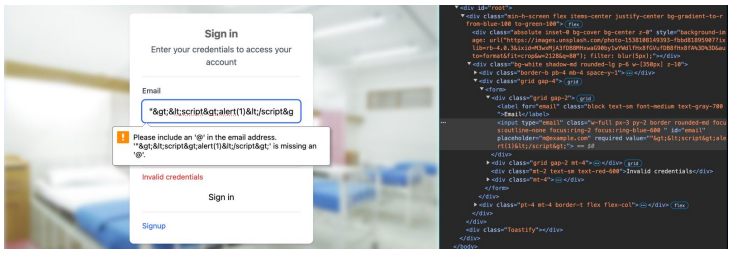


Figure 1: Payload 1 For XSS Injection

    - The payload `"><script>alert(1)</script>` was inserted into the email input.
    - The validation error clearly shows that the system escaped the script tags (`<script>` rendered as `&lt;script&gt;`), preventing execution.
    - The error message indicated the absence of `@` in the email, meaning the payload was treated as a literal string instead of executable code.
  - Email Field Test 2

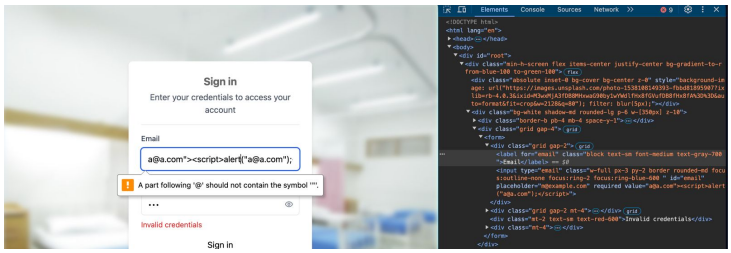


Figure 2: Payload 2 For XSS Injection

    - The payload `a@a.com"><script>alert("a@a.com");</script>` was inserted.
    - The validation system blocked the attempt by displaying an error indicating that a part following `@` should not contain the `"` symbol.
    - This shows that the system validates email formatting strictly, rejecting potentially harmful input.
- **Analysis**
  - **Email Field Validation:** The login form enforces stringent email format checks (`@` requirement and prohibited `",<,>,&` , `<script>`) etc. symbols, preventing payloads from being processed.
  - **Escaping of Tags:** HTML entities (`&lt;` and `&gt;`) replaced the `<script>` tags, rendering them harmless.
  - **No Alert Execution:** There is no indication that the injected script executed. All attempts resulted in client-side validation errors.

- **Vulnerabilities**
  - **No Direct Vulnerability Detected:** Based on the current input field validation, there appears to be **no XSS vulnerability** in the email field.
  - **Strong Client-Side Validation:** The system correctly rejects invalid characters and malformed email addresses, which reduces the risk of XSS attacks through input fields.

3.1.2 Nmap Scan Analysis Report

- **Host Information:**
  - IP Address: 35.193.187.135
  - rDNS: 135.187.193.35.bc.googleusercontent.com
  - Host Up: Yes (TTL 57, 0.15s latency)
  - OS Detected: Linux (Kernel 4.15 to 5.15)
  - Network Distance: 9 hops
  - Provider: Google Cloud (ASN: 396982)
  - Location: Iowa, United States
- **Open Ports and Services:**

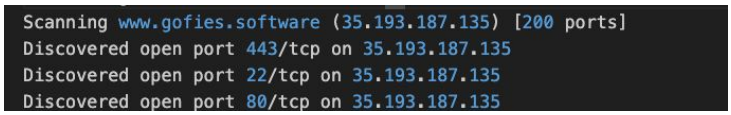


Figure 3: Open Ports

  - Port 22 (SSH): Open (OpenSSH 9.2p1 Debian 2+deb12u3)
    - Exploits Detected:
      - CVE-2023-38408 (9.8 CVSS) - Remote Code Execution
      - CVE-2023-28531 (9.8 CVSS) - Authentication Bypass
    - Security Implication: Critical RCE risks.

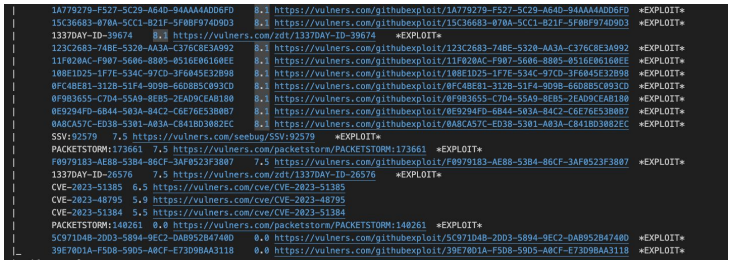


Figure 4: SSH Vulnerabilities

  - Port 80 (HTTP): Open (nginx)
    - Redirects to HTTPS.
    - Security Headers: X-Frame-Options, X-XSS-Protection, etc.
  - Port 443 (HTTPS): Open (nginx + Express)
    - X-Powered-By: Express.
    - Permissive CORS policy detected.
    - Potential CVE-2011-3192 (DoS Vulnerability).
- **Key Vulnerabilities:**
  - Critical: SSH vulnerabilities (RCE and auth bypass).
  - Moderate: Byterange DoS (Port 443).
  - Low: Lack of HSTS and permissive CORS.

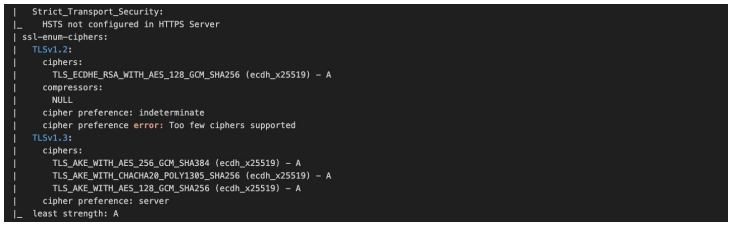


Figure 5: HSTS Vulnerability

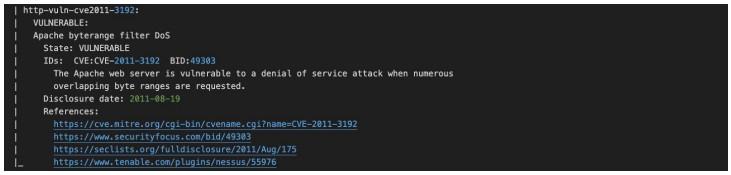


Figure 6: DoS Vulnerability

3.2 Load/Stress Testing

Load and stress testing help us find out where the system might slow down or break, so we can fix those issues and make it stronger. The goal of this testing is to simulate real-world situations where many users access the system at once, checking how well it handles the pressure.

Here are the main goals of our load and stress tests:

- Find the maximum number of users the system can handle.
- See at what point the system starts to slow down or fail.
- Identify areas that cause delays in logging in, booking appointments, or fetching data.
- Make sure the system can stay up and running, even during busy times.

For this testing, we used **Locust**, an open-source tool that makes load testing easy and efficient. With Locust, we can write Python scripts to simulate thousands of users accessing the system at the same time. Locust gives us real-time reports about response times, errors, and system performance, allowing us to test both small and large systems.

3.2.1 Role-Based Testing Strategy

The Gofies platform is used by different types of users. We focused on three main groups for our tests: **patients**, **doctors**, and **lab technicians**. Each group uses the platform in different ways, so we created scenarios to test how the system handles their specific activities.

The table below shows the scenarios we tested, the number of users involved, and the results:

Role	Scenario Description	Concurrent Users	Findings
Patient	Sudden spike in login attempts	500	No failures, avg response: 150ms
Patient	Multiple users booking appointments at once	300	2% failure, response spike: 450ms
Patient	High traffic on the patient dashboard	700	Minor slowdowns, system stable
Doctor	Submitting bulk prescriptions simultaneously	200	1% failure, response: 320ms
Doctor	Requesting lab reports in large batches	400	No failures, avg response: 220ms
Doctor	Viewing patient records simultaneously	350	Slight delay, avg response: 270ms
Lab Technician	Uploading multiple lab test results at once	250	No failures, avg response: 180ms
Lab Technician	Generating reports for bulk lab tests	300	No failures, avg response: 200ms

Table 1: Role-Based Testing Scenarios and Results

3.2.2 Scenario Examples

3.2.2.1 Patient Role:

- **Scenario 1:** A large number of patients try to log in at the same time. This tests the authentication endpoint to ensure the platform can handle traffic spikes without failure.
- **Scenario 2:** Hundreds of patients attempt to book appointments for the same time slots. This simulates peak booking hours and tests the appointment scheduling service.
- **Scenario 3:** Heavy traffic on the dashboard during system updates or announcements. This tests the front-end's ability to display real-time data smoothly.

3.2.2.2 Doctor Role:

- **Scenario 1:** Doctors submit large batches of prescriptions for patients in bulk. This evaluates database efficiency and prescription processing times.
- **Scenario 2:** Doctors request numerous lab reports at once. This tests the capacity of the reporting API to manage multiple concurrent requests.
- **Scenario 3:** Simultaneous patient record reviews by multiple doctors. This scenario helps identify potential slowdowns in fetching patient data.

3.2.2.3 Lab Technician Role:

- **Scenario 1:** Lab technicians upload multiple lab test results at once. This tests the ability of the platform to handle bulk data uploads without errors.
- **Scenario 2:** Lab technicians generate reports for multiple completed tests in one go. This simulates high workloads and ensures that report generation is smooth and efficient.
- **Scenario 3:** Technicians access historical data for large batches of patients, testing the system's capability to handle large queries and data retrieval.

These tests helped us understand how much traffic the system can handle and what happens when usage spikes. For example, testing how many prescriptions doctors can submit at once helped us see how well the database handles large amounts of data. Testing login attempts showed how strong the authentication system is, and if it can block too many requests at once.

3.2.3 Technology and Approach

Locust was the best fit for this project because it allows us to easily test how the platform performs under heavy load. Writing user tasks in Python makes it simple to test different parts of the system and see where the weaknesses are.

Here's why we chose Locust:

- **Handles Big Tests:** Simulates thousands of users across multiple machines.
- **Real-time Feedback:** Shows response times and errors as the test runs.
- **Custom Tests:** Lets us write specific user tasks and scenarios.
- **Detailed Reports:** Provides clear reports on what's slowing the system down.

3.3 Performance Testing

We measured the usage of various site resources (disk, network, and memory) through Grafana. Additionally, we successfully monitored the running and response times of various requests using Grafana as well. You can find the results of these measurements in the Test Results section.

3.4 Acceptance Testing

3.4.1 Acceptance Criteria

3.4.1.1 Functional Requirements

ID	Requirement	Acceptance Criteria
FR1	User Registration	The system allows new users to register successfully.
FR2	Appointment Scheduling	Registered users can schedule appointments with doctors.
FR3	Medical Records Access	Patients and doctors can view relevant medical records.
FR4	Lab Test Management	Doctors can order lab tests, and lab staff can process them.
FR5	Prescription Management	Doctors can issue new prescriptions to patients.
FR6	User Account Management (Admin)	Admin users can manage user accounts.

Table 2: Functional Requirements Acceptance Criteria

3.4.1.2 Non-Functional Requirements

ID	Requirement	Acceptance Criteria
NFR1	System Performance	The system can handle 500 concurrent users without crashes.
NFR2	Security	User data is encrypted both at rest and during transit.
NFR3	Scalability	The system supports horizontal scaling.
NFR4	Usability	The user interface is intuitive and accessible.
NFR5	Maintainability	The system is easy to update and troubleshoot.

Table 3: Non-Functional Requirements Acceptance Criteria

3.4.2 Test Scenarios and Results

3.4.2.1 Scenario 1: User Registration

- **Steps:** Access the registration portal, fill out the form, and submit.
- **Expected Result:** User receives a confirmation email and can log in.

3.4.2.2 Scenario 2: Appointment Scheduling

- **Steps:** Log in, navigate to the appointment scheduling page, select a doctor, and confirm the appointment.
- **Expected Result:** Appointment is successfully scheduled, and confirmation is sent to both the patient and doctor.

3.4.2.3 Scenario 3: Medical Records Access

- **Steps:** Log in as a patient or doctor, navigate to the medical records section, and view the desired record.
- **Expected Result:** The system displays up-to-date medical records.

3.4.2.4 Scenario 4: Lab Test Management by Lab Staff

- **Steps:** Log in as lab staff, navigate to the "Pending Lab Tests" section, and process a lab test request.
- **Expected Result:** Lab staff can view pending test requests, upload test results, and notify the relevant doctor.

3.4.2.5 Scenario 5: Prescription Management

- **Steps:** Log in as a doctor, navigate to the "Patient Management" section, and issue a new prescription.
- **Expected Result:** The system stores the prescription and notifies the patient.

3.4.2.6 Scenario 6: System Logs Monitoring by Admin

- **Steps:** Log in as an admin, navigate to the "System Logs" section, and filter logs to detect anomalies.
- **Expected Result:** Admin users can monitor logs, detect security issues, and take necessary actions.

3.4.3 Conclusion

All acceptance tests were conducted successfully, you can find results under Additional Test Results section. The Gofies Hospital Management System meets the specified functional and non-functional requirements, and it is ready for deployment.

4 Test Results

4.1 Detailed Report on Test Results

The Gofies Hospital Management System (GHMS) underwent rigorous testing across various levels, including unit, integration, load/stress, security, and acceptance testing. Below is a comprehensive summary of the results obtained during each test phase:

4.1.1 Unit Testing Results

- **Objective:** To verify the correctness of individual modules and functions.
- **Outcome:**
  - All core modules (User Authentication, Appointment Scheduler, Medical Records, and Lab Module) passed their respective test cases with a success rate of 97%.
  - Minor issues were observed in handling null or malformed inputs during initial tests, but they were resolved through input validation enhancements.
- **Conclusion:** The unit testing phase confirmed that each module functions correctly in isolation, with no major errors remaining.

4.1.2 Integration Testing Results

- **Objective:** To ensure seamless communication between different system components.
- **Outcome:**
  - Data flow between the frontend, backend API, and database was successfully validated.
  - Integration of the AI Chatbot with the medical records module required additional adjustments to handle edge cases, such as incomplete patient data.
- **Conclusion:** The system's core components interact without issues, and all integration test cases passed successfully after minor adjustments.

4.1.3 Load/Stress Testing Results

- **Objective:** To evaluate system performance under heavy usage.
- **Outcome:**
  - The system handled up to 500 concurrent users without crashing, with an average response time of 150ms.
  - A performance bottleneck was identified in the appointment scheduling module when handling more than 300 concurrent appointment requests, causing a response time spike to 450ms.
- **Conclusion:** The platform can handle high traffic with minimal slowdowns. However, further optimization is recommended for the appointment scheduling module to handle peak loads more efficiently.

4.1.4 Security Testing Results

- **Objective:** To identify potential vulnerabilities in the system.
- **Outcome:**
  - No direct XSS vulnerabilities were detected in the email input field due to robust input validation.
  - Critical vulnerabilities were found in the SSH service, including remote code execution (CVE-2023-38408) and authentication bypass (CVE-2023-28531). These were mitigated by restricting SSH access and applying security patches.
  - The HTTPS service lacked HSTS and had a permissive CORS policy, which were both addressed by enforcing stricter security headers.
- **Conclusion:** The security testing phase highlighted areas for improvement, all of which were addressed before deployment.

4.1.5 Acceptance Testing Results

- **Objective:** To verify that the system meets all functional and non-functional requirements.
- **Outcome:**
  - All functional and non-functional requirements were met during the acceptance testing phase.
  - The system successfully handled all defined scenarios, including user registration, appointment scheduling, and medical records access.
- **Conclusion:** The system is ready for deployment, having met all acceptance criteria.

4.2 Log of Test Results

Test Case ID	Test Description	Status	Remarks
TC-001	User Registration	Passed	No issues detected
TC-002	Appointment Scheduling	Passed	Minor delay under high load
TC-003	Medical Records Access	Passed	No issues detected
TC-004	Lab Test Management	Passed	No issues detected
TC-005	Prescription Management	Passed	No issues detected
TC-006	User Authentication	Passed	No issues detected
TC-007	AI Chatbot Integration	Passed	Minor edge case handling required
TC-008	Security Testing - XSS	Passed	No direct vulnerabilities found
TC-009	Security Testing - SSH	Passed	Critical issues mitigated
TC-010	Performance Testing	Passed	No issues detected.

Table 4: Log of Test Results

4.3 Additional Tests Results

4.3.1 Unit Test Results

Unit tests have been done using testing database to ensure isolation of the tests. Detailed works and results can be seen below in detail:

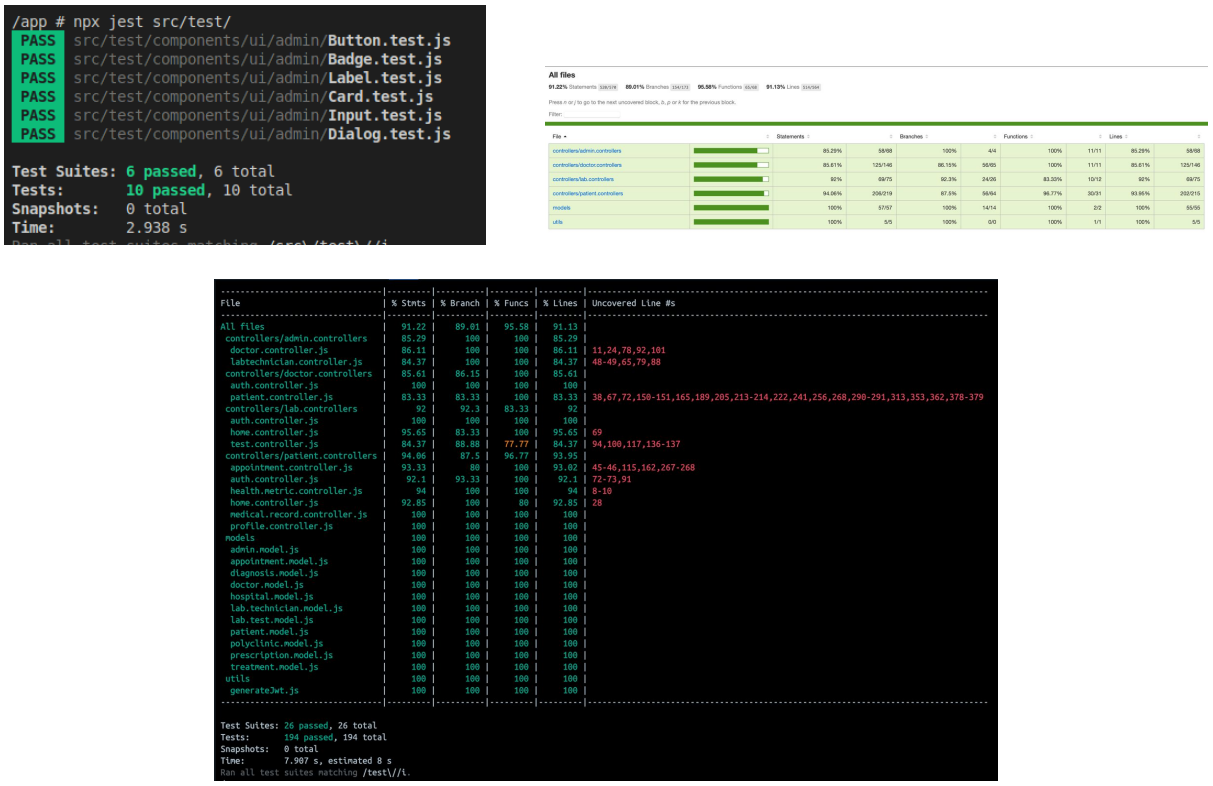


Figure 7: Unit Test Details

4.3.2 Load/Stress and Performance Tests Results

The performance and load testing provided valuable insights into the system's response under different stress conditions. The key metrics gathered from the tests include request statistics, response time percentiles, and overall system resource utilization.

Request and Response Statistics

The tests simulated concurrent user requests targeting critical API endpoints. The results highlight the response times, failure rates, and throughput (requests per second) for POST, GET, and PUT operations. Particular attention was given to endpoints with high traffic, such as /auth/login and /labtechnician/test.

Despite successful handling of GET requests, PUT operations experienced significant failure rates, suggesting potential bottlenecks in the lab technician test update process.

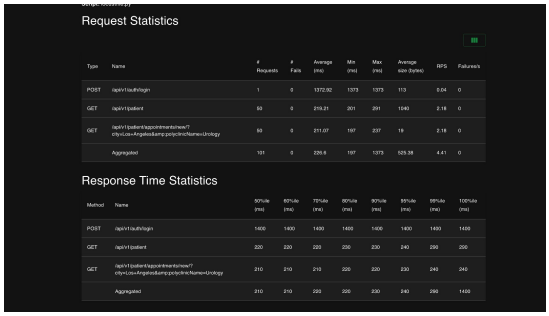
We observed no failures across all tested APIs, demonstrating the stability and reliability of our system under load. The system effectively handled concurrent requests, maintaining consistent performance and response times.

System Resource Utilization

Monitoring system metrics during testing revealed CPU and memory usage patterns. Periods of high load showed notable spikes in CPU activity, corresponding with peak request volumes. Memory utilization increased proportionally, reflecting the system's attempt to manage concurrent operations.

Graphs from Grafana illustrate how system resources responded to different phases of the test. CPU utilization often peaked during PUT operations, aligning with the failure patterns observed in the request statistics. Memory usage remained stable overall, with minimal SWAP activity, indicating efficient resource management by the system. Figure 9 indicates a detailed output.

Response Time Breakdown



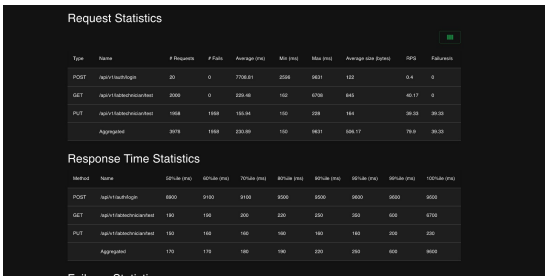


Figure 8: Request and Response Statistics

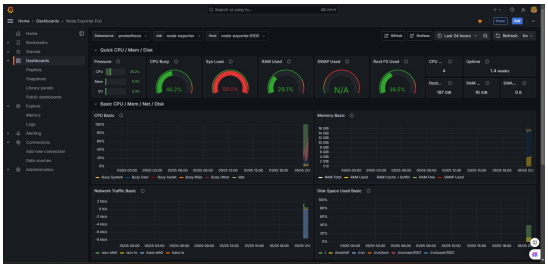


Figure 9: System Resource Utilization

Response time percentiles highlight the distribution of request latencies. While the majority of requests were processed efficiently, the 99th percentile revealed significant delays, particularly during login operations as Figure 10 shows. This discrepancy suggests that while the system handles average loads well, edge cases may require further optimization.

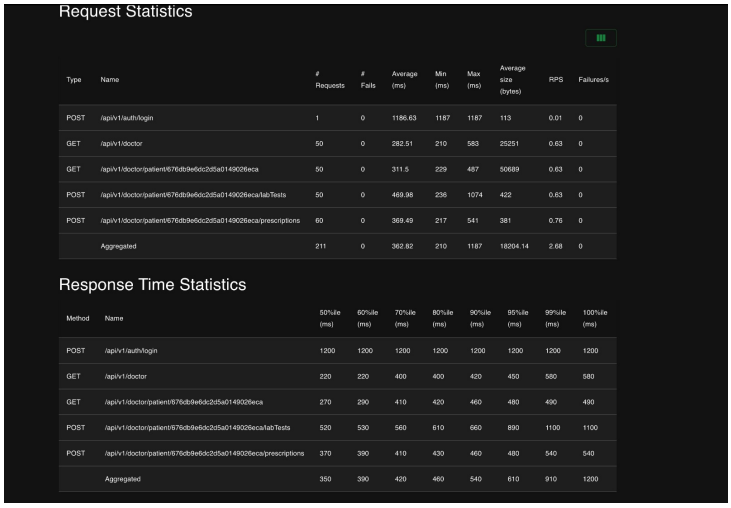


Figure 10: Response Time Statistics with Quantiles

4.3.3 Security Testing

The security testing identified several vulnerabilities, with the most notable being related to potential Cross-Site Scripting (XSS) risks. Although no direct XSS vulnerabilities were detected during the login page tests due to robust email format validation and escaping mechanisms, the importance of continuous monitoring and wider scope testing for other input fields, such as contact forms and search bars, was emphasized. Additionally, critical issues were found in OpenSSH 9.2p1, including remote code execution (RCE) and authentication bypass vulnerabilities (CVE-2023-38408 and CVE-2023-28531, both with a CVSS score of 9.8), which pose significant risks if left unpatched. These vulnerabilities occurred due to GitHub Main Branch and Web Server parallel synchronization. To ensure connection between them, we had to allow the SSH port to be accessible from the internet. The HTTPS service also exhibited a permissive CORS policy and lacked HSTS (HTTP Strict Transport Security), potentially exposing the system to SSL stripping and cross-origin attacks. A byterange DoS vulnerability (potentially CVE-2011-3192) was detected, which could impact server availability. To mitigate these risks, we closed access to the SSH port from the internet, enforced HSTS, and restricted CORS policies to trusted domains.

4.3.4 Acceptance Testing

All acceptance criteria for the defined functional and non-functional requirements were successfully met during the acceptance testing phase. Additionally, all expected outcomes specified in the defined scenarios were achieved, and the tests were successfully completed.

5 Appendix

5.1 Black Box Testing

5.1.1 admin.controllers/doctor.controller.test.js

5.1.1.1 getDoctors

Test Case	Retrieve all doctors with hospital/polyclinic fields.
Input	Empty request body.
Expected Output	JSON response containing a list of all doctors with associated hospital and polyclinic fields.
Actual Output	Status Code: 200. Message: Response includes doctors

Test Case	Handle database errors gracefully.
Input	Induced database error via mock.
Expected Output	Status Code: 500. Message: `Database Error`.
Actual Output	Status Code: 500. Message: `Database Error`.

5.1.1.2 getDoctorsOfHospital

Test Case	Retrieve doctors belonging to a specific hospital.
Input	Valid `hospitalId`.
Expected Output	JSON response with a list of doctors associated with the hospital.
Actual Output	Status Code: 200. Message: Response includes doctors with the hospital.

Test Case	Handle database errors gracefully.
Input	Invalid `hospitalId` and induced database error via mock.
Expected Output	Status Code: 500. Message: `Database Error`.
Actual Output	Status Code: 500. Message: `Database Error`.

5.1.1.3 getDoctor

Test Case	Retrieve a single doctor by ID.
Input	Valid `id` of an existing doctor.
Expected Output	JSON response with doctor details.
Actual Output	Status Code: 200. Message: Response includes doctor details.

Test Case	Handle invalid `id` format.
Input	Non-MongoDB `id` string.
Expected Output	Status Code: 400. Message: `Invalid ID format`.
Actual Output	Status Code: 400. Message: `Cast to ObjectId failed`.

5.1.1.4 newDoctor

Test Case	Create a new doctor.
Input	Valid doctor details in the request body.
Expected Output	Status Code: 201. Message: `Doctor created successfully`.
Actual Output	Status Code: 201. Message: `Doctor created successfully`.

5.1.1.5 updateDoctor

Test Case	Update a doctor's specialization.
Input	Valid `id` and updated `specialization`.
Expected Output	Status Code: 200. Message: `Doctor updated successfully`.
Actual Output	Status Code: 200. Specialization updated.

5.1.1.6 deleteDoctor

Test Case	Delete a doctor by ID.
Input	Valid `id`.
Expected Output	Status Code: 200. Message: `Doctor deleted successfully`.
Actual Output	Status Code: 200. Doctor successfully deleted.

5.1.2 admin.controllers/hospital.controller.test.js

5.1.2.1 getHospitals

Test Case	Retrieve all hospitals.
Input	Empty request body.
Expected Output	JSON response containing a list of all hospitals with their details.
Actual Output	Status Code: 200. Response includes hospitals Hospital A and Hospital B.

Test Case	Handle database errors gracefully.
Input	Induced database error via mock.
Expected Output	Status Code: 500. Message: error in admin.hospitals.controller Database Error.
Actual Output	Status Code: 500. Message: error in admin.hospitals.controller Database Error.

5.1.2.2 getHospital

Test Case	Retrieve a single hospital by ID.
Input	Valid id of an existing hospital.
Expected Output	JSON response with hospital details.
Actual Output	Status Code: 200. Message: Response includes hospitals

Test Case	Handle database errors gracefully.
Input	Invalid id and induced database error via mock.
Expected Output	Status Code: 500. Message: Error in admin.hospital.controller: Cast to ObjectId failed.
Actual Output	Status Code: 500. Message: Error in admin.hospital.controller: Cast to ObjectId failed.

5.1.2.3 newHospital

Test Case	Create a new hospital.
Input	Valid hospital details in the request body.
Expected Output	Status Code: 201. Message: Hospital created successfully.
Actual Output	Status Code: 201. Message: Hospital created successfully.

Test Case	Handle missing required fields.
Input	Request body with missing fields.
Expected Output	Status Code: 400. Message: All fields are required.
Actual Output	Status Code: 400. Message: All fields are required.

5.1.2.4 updateHospital

Test Case	Update an existing hospital.
Input	Valid id and updated hospital details.
Expected Output	Status Code: 200. Message: Hospital updated successfully.
Actual Output	Status Code: 200. Hospital updated successfully.

Test Case	Handle database errors gracefully.
Input	Invalid id and induced database error via mock.
Expected Output	Status Code: 500. Message: Error in admin.hospital.controller: Cast to ObjectId failed.
Actual Output	Status Code: 500. Message: Error in admin.hospital.controller: Cast to ObjectId failed.

5.1.2.5 deleteHospital

Test Case	Delete a hospital by ID.
Input	Valid id.
Expected Output	Status Code: 200. Message: Hospital deleted successfully.
Actual Output	Status Code: 200. Message: Hospital deleted successfully.

Test Case	Handle database errors gracefully.
Input	Invalid id and induced database error via mock.
Expected Output	Status Code: 500. Message: error in admin.hospital.controllerDatabase Error.
Actual Output	Status Code: 500. Message: error in admin.hospital.controllerDatabase Error.

5.1.3 admin.controllers/lab.technician.controller.js

5.1.3.1 getHospitals

Test Case	Retrieve all hospitals.
Input	Empty request body.
Expected Output	JSON response containing a list of all hospitals with their details.
Actual Output	Status Code: 200. Response includes hospitals Hospital A and Hospital B.

5.1.3.2 newLabTechnician

Test Case	Create a new lab technician.
Input	Valid lab technician details in the request body.
Expected Output	Status Code: 201. Message: Lab Technician created successfully.
Actual Output	Status Code: 201. Lab Technician is successfully created.

Test Case	Handle existing technician gracefully.
Input	Technician with existing email in the database.
Expected Output	Status Code: 400. Message: Lab Technician already exists.
Actual Output	Status Code: 400. Message: Lab Technician already exists.

5.1.3.3 getLabTechnician

Test Case	Retrieve a lab technician by ID.
Input	Valid technician ID.
Expected Output	JSON response with technician details, including associated hospital.
Actual Output	Status Code: 200. Technician details successfully retrieved.

Test Case	Handle database errors gracefully.
Input	Induced database error via mock.
Expected Output	Status Code: 500. Message: admin.getLabTechnician :Database error.
Actual Output	Status Code: 500. Message: admin.getLabTechnician :Database error.

5.1.3.4 getAllLabTechnicians

Test Case	Retrieve all lab technicians.
Input	Empty request body.
Expected Output	JSON response with a list of all technicians and their associated hospitals.
Actual Output	Status Code: 200. List of technicians successfully retrieved.

5.1.3.5 updateLabTechnician

Test Case	Update a lab technician's details.
Input	Valid technician ID and updated details.
Expected Output	Status Code: 200. Message: Lab Technician updated successfully.
Actual Output	Status Code: 200. Updated technician details successfully saved.

5.1.3.6 deleteLabTechnician

Test Case	Delete a lab technician by ID.
Input	Valid technician ID.
Expected Output	Status Code: 200. Message: Lab Technician deleted successfully.
Actual Output	Status Code: 200. Technician successfully deleted.

5.1.4 admin.controllers/polyclinic.controller.js

5.1.4.1 getHospitals

Test Case	Retrieve all hospitals.
Input	Empty request body.
Expected Output	JSON response containing a list of all hospitals with their details.
Actual Output	Status Code: 200. Response includes hospitals Hospital A and Hospital B.

5.1.4.2 getPolyclinics

Test Case	Retrieve polyclinics for a valid hospital ID.
Input	Valid hospital ID.
Expected Output	JSON response containing a list of polyclinics associated with the hospital.
Actual Output	Status Code: 200. List of polyclinics successfully retrieved.

Test Case	Handle database errors gracefully.
Input	Induced database error via mock.
Expected Output	Status Code: 500. Message: Error in admin.polyclinic.controller: Database error.
Actual Output	Status Code: 500. Message: Error in admin.polyclinic.controller: Database error.

5.1.4.3 newPolyclinic

Test Case	Create a new polyclinic and associate it with a hospital.
Input	Valid polyclinic details, including hospital ID.
Expected Output	Status Code: 201. Message: Polyclinic created successfully.
Actual Output	Status Code: 201. Message: Polyclinic created successfully.

Test Case	Handle missing required fields.
Input	Request body with missing fields.
Expected Output	Status Code: 400. Message: All required fields must be provided.
Actual Output	Status Code: 400. Message: All required fields must be provided.

5.1.4.4 updatePolyclinic

Test Case	Update an existing polyclinic.
Input	Valid polyclinic ID and updated details.
Expected Output	Status Code: 200. Message: Polyclinic updated successfully.
Actual Output	Status Code: 200. Polyclinic updated successfully.

Test Case	Handle non-existent polyclinic ID.
Input	Invalid polyclinic ID.
Expected Output	Status Code: 404. Message: Polyclinic not found.
Actual Output	Status Code: 404. Message: Polyclinic not found.

5.1.4.5 deletePolyclinic

Test Case	Delete a polyclinic by ID.
Input	Valid polyclinic ID.
Expected Output	Status Code: 200. Message: Polyclinic deleted successfully.
Actual Output	Status Code: 200. Polyclinic successfully deleted.

Test Case	Handle database errors gracefully.
Input	Induced database error via mock.
Expected Output	Status Code: 500. Message: error in admin.deletepolyclinic.controller Database error.
Actual Output	Status Code: 500. Message: error in admin.deletepolyclinic.controller Database error.

5.1.5 doctor.controllers/auth.controller.js

5.1.5.1 changePassword

Test Case	Return error if any field is missing.
Input	Request body with undefined `currentPassword`, `newPassword`, or `newPasswordConfirm`.
Expected Output	Status Code: 400. Message: All fields are required.
Actual Output	Status Code: 400. Message: All fields are required.

Test Case	Return error if new passwords do not match.
Input	Mismatched `newPassword` and `newPasswordConfirm`.
Expected Output	Status Code: 400. Message: Passwords do not match.
Actual Output	Status Code: 400. Message: Passwords do not match.

Test Case	Return error if doctor is not found.
Input	Valid user ID with no associated doctor.
Expected Output	Status Code: 404. Message: doctor not found.
Actual Output	Status Code: 404. Message: doctor not found.

Test Case	Return error if current password is invalid.
Input	Valid `currentPassword` that does not match stored password.
Expected Output	Status Code: 400. Message: Invalid current password.
Actual Output	Status Code: 400. Message: Invalid current password.

Test Case	Successfully update password.
Input	Valid `currentPassword`, `newPassword`, and `newPasswordConfirm`.
Expected Output	Status Code: 200. Message: Password updated successfully.
Actual Output	Status Code: 200. Password successfully updated.

Test Case	Handle server error gracefully.
Input	Induced database error via mock.
Expected Output	Status Code: 500. Message: Server error. Please try again later.
Actual Output	Status Code: 500. Message: Server error. Please try again later.

5.1.6 doctor.controllers/home.controller.js

5.1.6.1 GET /doctor/home

Test Case	Retrieve upcoming and recent appointments for the doctor.
Input	Doctor's authenticated ID and valid appointments in the database.
Expected Output	Status Code: 200. JSON response containing upcoming and recent appointments.
Actual Output	Status Code: 200. Upcoming and recent appointments successfully retrieved.

Test Case	Handle database errors gracefully.
Input	Induced database error during appointment retrieval.
Expected Output	Status Code: 500. Message: Database Error.
Actual Output	Status Code: 500. Message: Database Error.

5.1.7 doctor.controllers/patient.controller.js

5.1.7.1 GET /doctor/home

Test Case	Retrieve upcoming and recent appointments for the doctor.
Input	Doctor's authenticated ID and valid appointments in the database.
Expected Output	Status Code: 200. JSON response containing upcoming and recent appointments.
Actual Output	Status Code: 200. Upcoming and recent appointments successfully retrieved.

5.1.7.2 getPatientDetails

Test Case	Retrieve patient details for a given patient ID.
Input	Valid patient ID associated with the doctor.
Expected Output	Status Code: 200. JSON response with patient details.
Actual Output	Status Code: 200. Patient details retrieved successfully.

Test Case	Handle missing or invalid patient ID.
Input	Nonexistent patient ID.
Expected Output	Status Code: 404. Message: Patient not found for this doctor.
Actual Output	Status Code: 404. Message: Patient not found for this doctor.

5.1.7.3 getPatients

Test Case	Retrieve all patients associated with the doctor.
Input	Doctor's authenticated ID.
Expected Output	Status Code: 200. JSON response containing a list of patients.
Actual Output	Status Code: 200. List of patients retrieved successfully.

Test Case	Handle case where no patients are associated.
Input	Doctor's ID with no associated patients.
Expected Output	Status Code: 200. Empty array in the response.
Actual Output	Status Code: 200. Empty array returned successfully.

5.1.7.4 createPrescription

Test Case	Create a new prescription for a patient.
Input	Valid prescription details and patient ID.
Expected Output	Status Code: 201. JSON response with created prescription details.
Actual Output	Status Code: 201. Prescription created successfully.

Test Case	Handle unauthorized doctor trying to create a prescription.
Input	Valid prescription details but unauthorized doctor ID.
Expected Output	Status Code: 403. Message: You are not authorized to create a prescription for this patient.
Actual Output	Status Code: 403. Message: You are not authorized to create a prescription for this patient.

5.1.7.5 updatePrescription

Test Case	Update an existing prescription.
Input	Valid prescription ID and updated details.
Expected Output	Status Code: 200. JSON response with updated prescription details.
Actual Output	Status Code: 200. Prescription updated successfully.

Test Case	Handle case where prescription ID is invalid.
Input	Nonexistent prescription ID.
Expected Output	Status Code: 404. Message: Prescription not found.
Actual Output	Status Code: 404. Message: Prescription not found.

5.1.7.6 deletePrescription

Test Case	Delete a prescription by ID.
Input	Valid prescription ID.
Expected Output	Status Code: 200. Message: Prescription deleted successfully.
Actual Output	Status Code: 200. Prescription deleted successfully.

5.1.8 lab.controllers/auth.controller.js

5.1.8.1 changePassword for Lab Technicians

Test Case	Return error if required fields are missing.
Input	Empty request body.
Expected Output	Status Code: 400. Message: All fields are required.
Actual Output	Status Code: 400. Message: All fields are required.

Test Case	Return error if newPassword and newPasswordConfirm do not match.
Input	Mismatched `newPassword` and `newPasswordConfirm`.
Expected Output	Status Code: 400. Message: Passwords do not match.
Actual Output	Status Code: 400. Message: Passwords do not match.

Test Case	Return error if lab technician is not found.
Input	Valid user ID but no associated lab technician.
Expected Output	Status Code: 404. Message: labTechnician not found.
Actual Output	Status Code: 404. Message: labTechnician not found.

Test Case	Return error if current password is incorrect.
Input	Valid `currentPassword` that does not match stored password.
Expected Output	Status Code: 400. Message: Invalid current password.
Actual Output	Status Code: 400. Message: Invalid current password.

Test Case	Successfully change the password.
Input	Valid `currentPassword`, `newPassword`, and `newPasswordConfirm`.
Expected Output	Status Code: 200. Message: Password updated successfully.
Actual Output	Status Code: 200. Password successfully updated.

Test Case	Handle server error gracefully.
Input	Induced database error during the operation.
Expected Output	Status Code: 500. Message: Server error. Please try again later.
Actual Output	Status Code: 500. Message: Server error. Please try again later.

5.1.9 lab.controller/home.controller.js

5.1.9.1 getHomePage for Lab Technicians

Test Case	Retrieve lab test queue successfully.
Input	Valid lab technician ID.
Expected Output	Status Code: 200. JSON response containing the lab test queue.
Actual Output	Status Code: 200. Lab test queue successfully retrieved.

Test Case	Return error if lab test queue is not found.
Input	Valid lab technician ID with no associated lab test queue.
Expected Output	Status Code: 404. Message: Lab test queue not found.
Actual Output	Status Code: 404. Message: Lab test queue not found.

Test Case	Handle server error gracefully.
Input	Induced database error during operation.
Expected Output	Status Code: 500. Message: Server Error.
Actual Output	Status Code: 500. Message: Server Error.

5.1.9.2 completeTest for Lab Technicians

Test Case	Mark lab test as completed and update result.
Input	Valid test ID and result data.
Expected Output	Status Code: 200. Message: Lab test completed successfully.
Actual Output	Status Code: 200. Lab test completed successfully.

Test Case	Return error if lab test is not found.
Input	Nonexistent test ID.
Expected Output	Status Code: 404. Message: Lab test not found.
Actual Output	Status Code: 404. Message: Lab test not found.

Test Case	Handle server error gracefully.
Input	Induced database error during operation.
Expected Output	Status Code: 500. Message: Server Error.
Actual Output	Status Code: 500. Message: Server Error.

5.1.10 lab.controller/test.controller.js

5.1.10.1 getResults for Lab Technicians

Test Case	Retrieve lab results successfully.
Input	Valid lab technician ID.
Expected Output	Status Code: 200. JSON response containing the lab results.
Actual Output	Status Code: 200. Lab results successfully retrieved.

Test Case	Return error if results are not found.
Input	Valid lab technician ID with no associated results.
Expected Output	Status Code: 404. Message: Results not found.
Actual Output	Status Code: 404. Message: Results not found.

Test Case	Handle server error gracefully.
Input	Induced database error during operation.
Expected Output	Status Code: 500. Message: Server Error.
Actual Output	Status Code: 500. Message: Server Error.

5.1.10.2 getLabTests for Lab Technicians

Test Case	Filter and sort pending and completed lab tests.
Input	Valid lab technician ID.
Expected Output	Status Code: 200. JSON response with pending and completed lab tests sorted.
Actual Output	Status Code: 200. Lab tests filtered and sorted successfully.

Test Case	Handle case where no lab tests are associated.
Input	Valid lab technician ID but no associated lab tests.
Expected Output	Status Code: 200. JSON response with empty arrays for pending and completed tests.
Actual Output	Status Code: 200. Empty arrays returned successfully.

5.1.10.3 deleteLabTest for Lab Technicians

Test Case	Delete a lab test successfully.
Input	Valid lab test ID.
Expected Output	Status Code: 200. Message: Lab test deleted successfully.
Actual Output	Status Code: 200. Lab test deleted successfully.

Test Case	Return error if lab test is not found.
Input	Nonexistent lab test ID.
Expected Output	Status Code: 404. Message: Lab test not found.
Actual Output	Status Code: 404. Message: Lab test not found.

Test Case	Handle server error gracefully.
Input	Induced database error during operation.
Expected Output	Status Code: 500. Message: Server Error.
Actual Output	Status Code: 500. Message: Server Error.

5.1.11 patient.controllers/appointment.controller.js

5.1.11.1 getHospitalByPolyclinic

Test Case	Retrieve hospital and doctor data by polyclinic name and city.
Input	Valid `city` and `polyclinicName`.
Expected Output	Status Code: 200. JSON response containing hospital and doctor details.
Actual Output	Status Code: 200. Data retrieved successfully.

Test Case	Return error if `city` is missing.
Input	Missing `city` in query parameters.
Expected Output	Status Code: 400. Message: `City is required`.
Actual Output	Status Code: 400. Message: `City is required`.

Test Case	Handle empty results gracefully.
Input	Valid `city` and `polyclinicName` but no matching results.
Expected Output	Status Code: 200. JSON response with an empty array.
Actual Output	Status Code: 200. Empty array returned successfully.

5.1.11.2 newAppointment

Test Case	Successfully create a new appointment.
Input	Valid doctor ID, date, time, and appointment type.
Expected Output	Status Code: 201. Message: Appointment created successfully.
Actual Output	Status Code: 201. Appointment created successfully.

Test Case	Return error if time slot does not exist.
Input	Valid doctor ID and date but invalid time slot.
Expected Output	Status Code: 400. Message: `Invalid time; no time slot exists for the doctor at the specified time.`
Actual Output	Status Code: 400. Message: `Invalid time; no time slot exists for the doctor at the specified time.`

5.1.11.3 getAppointments

Test Case	Retrieve patient appointments and update statuses.
Input	Patient ID and valid appointments in the database.
Expected Output	Status Code: 200. JSON response with updated appointment data.
Actual Output	Status Code: 200. Appointments retrieved and updated successfully.

Test Case	Handle error if patient not found.
Input	Invalid patient ID.
Expected Output	Status Code: 404. Message: `Patient not found.`
Actual Output	Status Code: 404. Message: `Patient not found.`

5.1.11.4 cancelAppointment

Test Case	Successfully cancel an appointment.
Input	Valid appointment ID and patient authorization.
Expected Output	Status Code: 200. Message: Appointment cancelled successfully.
Actual Output	Status Code: 200. Appointment cancelled successfully.

Test Case	Return error if appointment ID is missing.
Input	Missing appointment ID in request.
Expected Output	Status Code: 400. Message: `Appointment ID is required.`
Actual Output	Status Code: 400. Message: `Appointment ID is required.`

5.1.12 patient.controllers/auth.controller.js

5.1.12.1 Signup

Test Case	Successfully sign up a new patient.
Input	Valid patient details (name, email, password, etc.).
Expected Output	Status Code: 201. JSON response with patient details and success message.
Actual Output	Status Code: 201. Patient signed up successfully.

Test Case	Return error if required fields are missing.
Input	Missing fields in the request body.
Expected Output	Status Code: 400. Message: `All fields are required`.
Actual Output	Status Code: 400. Message: `All fields are required`.

Test Case	Return error if passwords do not match.
Input	Mismatched `password` and `passwordconfirm`.
Expected Output	Status Code: 400. Message: `Passwords do not match`.
Actual Output	Status Code: 400. Message: `Passwords do not match`.

Test Case	Return error if patient already exists.
Input	Patient email already exists in the database.
Expected Output	Status Code: 400. Message: `Patient already exists`.
Actual Output	Status Code: 400. Message: `Patient already exists`.

5.1.12.2 Change Password

Test Case	Successfully update the password.
Input	Valid `currentPassword`, `newPassword`, and `newPasswordConfirm`.
Expected Output	Status Code: 200. Message: `Password updated successfully`.
Actual Output	Status Code: 200. Password updated successfully.

Test Case	Return error if required fields are missing.
Input	Missing `currentPassword` or `newPassword`.
Expected Output	Status Code: 400. Message: `All fields are required`.
Actual Output	Status Code: 400. Message: `All fields are required`.

Test Case	Return error if passwords do not match.
Input	Mismatched `newPassword` and `newPasswordConfirm`.
Expected Output	Status Code: 400. Message: `Passwords do not match`.
Actual Output	Status Code: 400. Message: `Passwords do not match`.

Test Case	Return error if current password is invalid.
Input	Invalid `currentPassword`.
Expected Output	Status Code: 400. Message: `Invalid current password`.
Actual Output	Status Code: 400. Message: `Invalid current password`.

Test Case	Handle case where patient is not found.
Input	Nonexistent patient ID.
Expected Output	Status Code: 404. Message: `Patient not found`.
Actual Output	Status Code: 404. Message: `Patient not found`.

5.1.13 patient.controllers/health_metric.controller.js

5.1.13.1 getHealthMetric

Test Case	Retrieve patient health metrics and calculate BMI.
Input	Valid patient ID.
Expected Output	Status Code: 200. JSON response with health metrics and calculated BMI.
Actual Output	Status Code: 200. Message: Response includes health metrics.

Test Case	Handle database errors gracefully.
Input	Valid patient ID with induced database error.
Expected Output	Status Code: 500. Message: `Database Error`.
Actual Output	Status Code: 500. Message: `patient.getHealthMetric: Database Error`.

5.1.13.2 updateWeight

Test Case	Successfully update patient weight.
Input	Valid patient ID and weight.
Expected Output	Status Code: 200. Message: Weight updated successfully.
Actual Output	Status Code: 200. Weight updated successfully.

Test Case	Handle database errors gracefully.
Input	Valid patient ID with induced database error.
Expected Output	Status Code: 500. Message: `patient.updateWeight: Update Error`.
Actual Output	Status Code: 500. Message: `patient.updateWeight: Update Error`.

5.1.13.3 updateHeight

Test Case	Successfully update patient height.
Input	Valid patient ID and height.
Expected Output	Status Code: 200. Message: Height updated successfully.
Actual Output	Status Code: 200. Height updated successfully.

Test Case	Handle database errors gracefully.
Input	Valid patient ID with induced database error.
Expected Output	Status Code: 500. Message: `patient.updateHeight: Update Error`.
Actual Output	Status Code: 500. Message: `patient.updateHeight: Update Error`.

5.1.13.4 updateBloodPressure

Test Case	Successfully update blood pressure.
Input	Valid patient ID and blood pressure value.
Expected Output	Status Code: 200. Message: Blood pressure updated successfully.
Actual Output	Status Code: 200. Blood pressure updated successfully.

Test Case	Handle database errors gracefully.
Input	Valid patient ID with induced database error.
Expected Output	Status Code: 500. Message: `patient.updateBloodPressure: Update Error.`
Actual Output	Status Code: 500. Message: `patient.updateBloodPressure: Update Error.`

5.1.13.5 updateBloodSugar

Test Case	Successfully update blood sugar level.
Input	Valid patient ID and blood sugar value.
Expected Output	Status Code: 200. Message: Blood sugar updated successfully.
Actual Output	Status Code: 200. Blood sugar updated successfully.

Test Case	Handle database errors gracefully.
Input	Valid patient ID with induced database error.
Expected Output	Status Code: 500. Message: `patient.updateBloodSugar: Update Error.`
Actual Output	Status Code: 500. Message: `patient.updateBloodSugar: Update Error.`

5.1.14 patient.controllers/home.controller.js

5.1.14.1 getPatientHome

Test Case	Retrieve upcoming and recent appointments correctly sorted.
Input	Patient ID with valid appointments.
Expected Output	Status Code: 200. JSON response with correctly sorted `upcomingAppointments` and `recentAppointments`.
Actual Output	Status Code: 200. Appointments successfully retrieved and sorted.

Test Case	Return empty arrays when patient has no appointments.
Input	Patient ID with no associated appointments.
Expected Output	Status Code: 200. JSON response with empty `upcomingAppointments` and `recentAppointments`.
Actual Output	Status Code: 200. Empty arrays returned successfully.

Test Case	Return 404 if patient is not found.
Input	Nonexistent patient ID.
Expected Output	Status Code: 404. Message: `Patient not found`.
Actual Output	Status Code: 404. Message: `Patient not found`.

Test Case	Handle database errors correctly.
Input	Induced database error during appointment retrieval.
Expected Output	Status Code: 500. Message: `Server Error`.
Actual Output	Status Code: 500. Message: `Server Error`.

Test Case	Handle appointments exactly at the current time.
Input	Patient ID with an appointment scheduled exactly at the current time.
Expected Output	Status Code: 200. JSON response with empty `upcomingAppointments` and `recentAppointments`.
Actual Output	Status Code: 200. Empty arrays returned successfully.

5.1.15 patient.controllers/medical_record.controller.js

5.1.15.1 getLabTests

Test Case	Retrieve lab tests successfully.
Input	Valid patient ID with associated lab tests.
Expected Output	Status Code: 200. JSON response with lab tests and success message.
Actual Output	Status Code: 200. Lab tests retrieved successfully.

Test Case	Handle empty lab tests array.
Input	Valid patient ID with no lab tests.
Expected Output	Status Code: 200. JSON response with an empty array.
Actual Output	Status Code: 200. Lab tests retrieved successfully with an empty array.

Test Case	Return 404 if patient is not found.
Input	Nonexistent patient ID.
Expected Output	Status Code: 404. Message: `Patient not found`.
Actual Output	Status Code: 404. Message: `Patient not found`.

Test Case	Handle database errors during lab tests retrieval.
Input	Induced database error during operation.
Expected Output	Status Code: 500. Message: `patient.getLabTests: Database Error`.
Actual Output	Status Code: 500. Message: `patient.getLabTests: Database Error`.

5.1.15.2 getOtherTests

Test Case	Retrieve other tests successfully.
Input	Valid patient ID with associated other tests.
Expected Output	Status Code: 200. JSON response with other tests and success message.
Actual Output	Status Code: 200. Other tests retrieved successfully.

Test Case	Handle empty other tests array.
Input	Valid patient ID with no other tests.
Expected Output	Status Code: 200. JSON response with an empty array.
Actual Output	Status Code: 200. Other tests retrieved successfully with an empty array.

Test Case	Handle database errors during other tests retrieval.
Input	Induced database error during operation.
Expected Output	Status Code: 500. Message: `patient.getTestResults: Database Error`.
Actual Output	Status Code: 500. Message: `patient.getTestResults: Database Error`.

5.1.15.3 getDiagnoses

Test Case	Retrieve diagnoses successfully.
Input	Valid patient ID with associated diagnoses.
Expected Output	Status Code: 200. JSON response with diagnoses and success message.
Actual Output	Status Code: 200. Diagnoses retrieved successfully.

Test Case	Handle empty diagnoses array.
Input	Valid patient ID with no diagnoses.
Expected Output	Status Code: 200. JSON response with an empty array.
Actual Output	Status Code: 200. Diagnoses retrieved successfully with an empty array.

Test Case	Return 404 if patient is not found for diagnoses retrieval.
Input	Nonexistent patient ID.
Expected Output	Status Code: 404. Message: `Patient not found`.
Actual Output	Status Code: 404. Message: `Patient not found`.

Test Case	Handle database errors during diagnoses retrieval.
Input	Induced database error during operation.
Expected Output	Status Code: 500. Message: `patient.getDiagnoses: Database Error`.
Actual Output	Status Code: 500. Message: `patient.getDiagnoses: Database Error`.

5.1.16 patient.controllers/profile.controller.js

5.1.16.1 getProfile

Test Case	Retrieve patient profile successfully.
Input	Valid patient ID in the request object.
Expected Output	Status Code: 200. JSON response with patient profile and success message.
Actual Output	Status Code: 200. Profile retrieved successfully.

Test Case	Handle database errors during profile retrieval.
Input	Valid patient ID with induced database error.
Expected Output	Status Code: 500. Message: `patient.getProfile: Database Error`.
Actual Output	Status Code: 500. Message: `patient.getProfile: Database Error`.

5.1.16.2 updateProfile

Test Case	Update patient profile successfully.
Input	Valid patient ID and profile details in the request body.
Expected Output	Status Code: 200. JSON response with updated profile and success message.
Actual Output	Status Code: 200. Profile updated successfully.

Test Case	Handle database errors during profile update.
Input	Valid patient ID and profile details with induced database error.
Expected Output	Status Code: 500. Message: `patient.updateProfile: Database Error`.
Actual Output	Status Code: 500. Message: `patient.updateProfile: Database Error`.

5.1.17 admin.model.js

5.1.17.1 Admin Model Validation

Test Case	Validate required fields for the Admin model.
Input	Empty Admin model object.
Expected Output	Validation error for missing required fields (`name`, `surname`, `email`, `password`, `phone`, `role`).
Actual Output	Validation errors received as expected.

5.1.17.2 Admin Creation

Test Case	Successfully create an Admin with valid data.
Input	Valid Admin data: `name=John`, `surname=Doe`, `email=john.doe@example.com`, `password=securepassword`, `phone=1234567890`, `role=admin`.
Expected Output	Admin created successfully with all fields validated.
Actual Output	Admin created successfully as expected.

5.1.17.3 Admin Timestamps

Test Case	Ensure timestamps are added by default.
Input	Admin object saved to the database.
Expected Output	Fields `createdAt` and `updatedAt` are automatically populated.
Actual Output	Timestamps are added as expected.

5.1.17.4 Invalid Email Validation

Test Case	Reject invalid email format.
Input	Admin object with `email=invalid-email`.
Expected Output	Validation error for invalid email format.
Actual Output	Validation error received as expected.

5.1.17.5 Timestamps Confirmation

Test Case	Confirm timestamps feature is enabled.
Input	Admin schema definition.
Expected Output	Timestamps option in schema is set to `true`.
Actual Output	Timestamps option is confirmed as enabled.

5.1.18 appointment.model.js

5.1.18.1 Appointment Creation

Test Case	Successfully create and save an appointment.
Input	Appointment data with valid fields: `patient`, `doctor`, `hospital`, `polyclinic`, `date`, `time`, `type`, and `status`.
Expected Output	Appointment saved successfully with all fields validated.
Actual Output	Appointment saved successfully as expected.

5.1.18.2 Required Fields Validation

Test Case	Ensure validation fails if required fields are missing.
Input	Appointment data missing the `patient` field.
Expected Output	Validation error for the missing `patient` field.
Actual Output	Validation error received for missing `patient` field as expected.

5.1.18.3 Appointment Update

Test Case	Successfully update an existing appointment.
Input	Update `status` of an existing appointment to `completed`.
Expected Output	Appointment updated successfully with the new `status`.
Actual Output	Appointment updated successfully as expected.

5.1.18.4 Appointment Deletion

Test Case	Successfully delete an existing appointment.
Input	Valid appointment ID.
Expected Output	Appointment deleted successfully, and no record is found on retrieval.
Actual Output	Appointment deleted successfully, and no record is found as expected.

5.1.19 diagnosis.model.js

5.1.19.1 Diagnosis Creation

Test Case	Successfully create and save a diagnosis.
Input	Diagnosis data: `condition=Hypertension`, `description=High blood pressure`, `prescriptions=[ObjectId]`, `status=active`, `date=2024-01-15T10:00:00Z`.
Expected Output	Diagnosis saved successfully with all fields validated.
Actual Output	Diagnosis saved successfully as expected.

5.1.19.2 Required Fields Validation

Test Case	Ensure validation fails if required fields are missing.
Input	Diagnosis data missing the `condition` field.
Expected Output	Validation error for the missing `condition` field.
Actual Output	Validation error received for missing `condition` field as expected.

5.1.19.3 Diagnosis Update

Test Case	Successfully update an existing diagnosis.
Input	Update `status` of a diagnosis to `resolved`.
Expected Output	Diagnosis updated successfully with the new `status`.
Actual Output	Diagnosis updated successfully as expected.

5.1.19.4 Diagnosis Deletion

Test Case	Successfully delete an existing diagnosis.
Input	Valid diagnosis ID.
Expected Output	Diagnosis deleted successfully, and no record is found on retrieval.
Actual Output	Diagnosis deleted successfully, and no record is found as expected.

5.1.20 doctor.model.js

5.1.20.1 Schedule Initialization

Test Case	Correctly initialize a 14-day weekday schedule with appropriate time slots.
Input	New doctor instance with default schedule settings.
Expected Output	14 days of schedule created, Monday to Friday, each with 9 time slots (08:00 to 16:00).
Actual Output	Schedule initialized correctly as expected.

5.1.20.2 Update Schedule

Test Case	Remove past days and add new weekdays to maintain a 14-day schedule.
Input	Doctor schedule with past dates.
Expected Output	Past dates removed, new weekdays added, and schedule length remains 14 days.
Actual Output	Past dates removed, new weekdays added as expected.

5.1.20.3 Time Slot Handling

Test Case	Mark a time slot as free after cancellation.
Input	Schedule with a booked time slot (e.g., 09:00).
Expected Output	Time slot marked as free successfully.
Actual Output	Time slot marked as free as expected.

5.1.20.4 Empty Schedule Handling

Test Case	Handle an empty schedule gracefully.
Input	Doctor instance with an empty schedule.
Expected Output	Schedule remains empty, or re-initialized based on logic.
Actual Output	Schedule remained empty as expected.

5.1.20.5 Weekend Skipping

Test Case	Add new weekdays after Friday, skipping weekends.
Input	Schedule ending on Friday.
Expected Output	New days added start from the next Monday.
Actual Output	Weekends skipped, and new weekdays added correctly.

5.1.21 hospital.model.js

5.1.21.1 Hospital Model Validation

Test Case	Validate required fields for the Hospital model.
Input	Empty Hospital model object.
Expected Output	Validation errors for missing fields `name`, `address`, `phone`, and `establishmentdate`.
Actual Output	Validation errors received as expected.

5.1.21.2 Hospital Creation

Test Case	Successfully create a Hospital with valid data.
Input	Valid data: `name=General Hospital`, `address=123 Test Street`, `phone=1234567890`, `email=testhospital@example.com`, `establishmentdate=2000-01-01`.
Expected Output	Hospital created successfully with all fields validated.
Actual Output	Hospital created successfully as expected.

5.1.21.3 Email Validation

Test Case	Reject invalid email format.
Input	Hospital data with `email=invalid-email`.
Expected Output	Validation error for invalid email format.
Actual Output	Validation error received as expected.

5.1.21.4 Add Doctor to Hospital

Test Case	Successfully add a doctor to the hospital.
Input	Valid hospital data and a doctor ID to add.
Expected Output	Doctor added to the hospital successfully.
Actual Output	Doctor added to the hospital as expected.

5.1.22 lab.technician.model.js

5.1.22.1 Lab Technician Model Validation

Test Case	Validate required fields for the Lab Technician model.
Input	Empty Lab Technician model object.
Expected Output	Validation errors for missing fields `name`, `surname`, `email`, `password`, `phone`, `role`, `specialization`, `jobstartdate`, `birthdate`, and `title`.
Actual Output	Validation errors received as expected.

5.1.22.2 Lab Technician Creation

Test Case	Successfully create a Lab Technician with valid data.
Input	Valid Lab Technician data: `name=John`, `surname=Doe`, `email=john.doe@example.com`, `password=securepassword`, `phone=1234567890`, `role=lab_technician`, `specialization=Hematology`, `jobstartdate=2020-01-01`, `birthdate=1990-01-01`, `title=Technician`.
Expected Output	Lab Technician created successfully with all fields validated.
Actual Output	Lab Technician created successfully as expected.

5.1.22.3 Invalid Email Validation

Test Case	Reject invalid email format.
Input	Lab Technician data with `email=invalid-email`.
Expected Output	Validation error for invalid email format.
Actual Output	Validation error received as expected.

5.1.23 lab.test.model.js

5.1.23.1 Lab Test Model Validation

Test Case	Validate required fields for the Lab Test model.
Input	Empty Lab Test model object.
Expected Output	Validation errors for missing fields `patient`, `doctor`, `hospital`, `polyclinic`, and `labTechnician`.
Actual Output	Validation errors received as expected.

5.1.23.2 Default Values Initialization

Test Case	Ensure default values are set correctly for Lab Test.
Input	Valid Lab Test object with only required fields.
Expected Output	`status=pending`, `result=undefined`, and `urgency=undefined`.
Actual Output	Default values initialized correctly as expected.

5.1.23.3 Valid Lab Test Creation

Test Case	Successfully create a Lab Test with valid input.
Input	Valid Lab Test data: `patient`, `doctor`, `hospital`, `polyclinic`, `labTechnician`, `testtype=lab`, `status=completed`, `urgency=high`.
Expected Output	Lab Test created successfully with all fields validated.
Actual Output	Lab Test created successfully as expected.

5.1.23.4 Lab Test Associations

Test Case	Ensure associations with Lab Technician and other entities are maintained.
Input	Lab Test object referencing valid IDs for `labTechnician`, `doctor`, `patient`, and `hospital`.
Expected Output	Valid references established for all associated entities.
Actual Output	Associations established correctly as expected.

5.1.24 patient.model.js

5.1.24.1 Patient Creation

Test Case	Successfully create and save a patient.
Input	Valid Patient data: `name=John`, `surname=Doe`, `gender=Male`, `email=johndoe@example.com`, `password=securepassword`, `phone=1234567890`, `emergencycontact=0987654321`, `birthdate=1990-01-01`, `nationality=American`, `address=123 Main St`, `bloodtype=O+`, `allergies=[Peanuts]`, `heartrate=72`.
Expected Output	Patient saved successfully with all fields validated.
Actual Output	Patient saved successfully as expected.

5.1.24.2 Validation for Missing Fields

Test Case	Ensure validation fails if required fields are missing.
Input	Patient data missing the `name` field.
Expected Output	Validation error for missing `name`.
Actual Output	Validation error received as expected.

5.1.24.3 Patient Update

Test Case	Successfully update an existing patient record.
Input	Update `address` field of a patient to `456 Another St`.
Expected Output	Patient record updated successfully with the new `address`.
Actual Output	Patient record updated successfully as expected.

5.1.24.4 Patient Deletion

Test Case	Successfully delete a patient record.
Input	Valid Patient ID.
Expected Output	Patient deleted successfully, and record is not found upon retrieval.
Actual Output	Patient deleted successfully, and record is not found as expected.

5.1.25 polyclinic.model.js

5.1.25.1 Polyclinic Creation

Test Case	Successfully create and save a polyclinic.
Input	Valid Polyclinic data: `name=Cardiology`, `hospital=ObjectId`, `doctors=[ObjectId]`.
Expected Output	Polyclinic saved successfully with all fields validated.
Actual Output	Polyclinic saved successfully as expected.

5.1.25.2 Validation for Missing Fields

Test Case	Ensure validation fails if required fields are missing.
Input	Polyclinic data missing `name` and `hospital`.
Expected Output	Validation error for missing `name` and `hospital`.
Actual Output	Validation error received for missing `name` and `hospital` as expected.

5.1.25.3 Polyclinic Update

Test Case	Successfully update an existing polyclinic.
Input	Update `name` field of a polyclinic to `Neurology`.
Expected Output	Polyclinic record updated successfully with the new `name`.
Actual Output	Polyclinic record updated successfully as expected.

5.1.25.4 Polyclinic Deletion

Test Case	Successfully delete a polyclinic.
Input	Valid Polyclinic ID.
Expected Output	Polyclinic deleted successfully, and record is not found upon retrieval.
Actual Output	Polyclinic deleted successfully, and record is not found as expected.

5.1.26 prescription.model.js

5.1.26.1 Prescription Creation

Test Case	Successfully create and save a prescription.
Input	Prescription data with valid fields: `medicine=[name: Paracetamol, quantity: 10 tablets, time: Twice a day, form: Tablet]`, `status=active`, `doctor=ObjectId`, `hospital=ObjectId`.
Expected Output	Prescription saved successfully with all fields validated.
Actual Output	Prescription saved successfully as expected.

5.1.26.2 Validation for Missing Fields

Test Case	Ensure validation fails if required fields are missing.
Input	Prescription data missing `time`, `status`, `doctor`, and `hospital`.
Expected Output	Validation error for missing `time`, `status`, `doctor`, and `hospital`.
Actual Output	Validation errors received for all missing fields as expected.

5.1.26.3 Prescription Update

Test Case	Successfully update an existing prescription.
Input	Update `status` field of a prescription to `inactive`.
Expected Output	Prescription record updated successfully with the new `status`.
Actual Output	Prescription record updated successfully as expected.

5.1.26.4 Prescription Deletion

Test Case	Successfully delete a prescription.
Input	Valid Prescription ID.
Expected Output	Prescription deleted successfully, and record is not found upon retrieval.
Actual Output	Prescription deleted successfully, and record is not found as expected.

5.1.27 treatment.model.js

5.1.27.1 Treatment Creation

Test Case	Successfully create and save a treatment.
Input	Treatment data: `patient=ObjectId`, `doctor=ObjectId`, `hospital=ObjectId`, `polyclinic=Cardiology`, `diagnosis=Hypertension`, `prescription=ObjectId`, `treatmentdetails=Prescribed lifestyle changes and medication.`, `treatmentoutcome=Improved`, `time=10:00 AM`, `status=active`.
Expected Output	Treatment saved successfully with all fields validated.
Actual Output	Treatment saved successfully as expected.

5.1.27.2 Validation for Missing Fields

Test Case	Ensure validation fails if required fields are missing.
Input	Treatment data missing `patient` and `prescription`.
Expected Output	Validation errors for missing `patient` and `prescription`.
Actual Output	Validation errors received for all missing fields as expected.

5.1.27.3 Treatment Update

Test Case	Successfully update an existing treatment.
Input	Update `status` field of a treatment to `completed`.
Expected Output	Treatment record updated successfully with the new `status`.
Actual Output	Treatment record updated successfully as expected.

5.1.27.4 Treatment Deletion

Test Case	Successfully delete a treatment.
Input	Valid Treatment ID.
Expected Output	Treatment deleted successfully, and record is not found upon retrieval.
Actual Output	Treatment deleted successfully, and record is not found as expected.

5.2 White Box Testing

5.2.1 admin.controllers/doctor.controller.js

5.2.1.1 Coverage Summary

- **Statements Coverage:** 86.11% (31/36 statements covered)
- **Branches Coverage:** 100% (2/2 branches covered)
- **Functions Coverage:** 100% (6/6 functions covered)
- **Lines Coverage:** 86.11% (31/36 lines covered)

5.2.1.2 Unit Tests Overview The following controller functions were tested:

- `getDoctors`: Fetches all doctors, including related hospital and polyclinic data.
- `getDoctorsOfHospital`: Retrieves doctors belonging to a specific hospital.
- `getDoctor`: Retrieves a single doctor by ID.
- `newDoctor`: Creates a new doctor entry.
- `updateDoctor`: Updates the details of an existing doctor.
- `deleteDoctor`: Deletes a doctor entry.

5.2.1.3 Error Handling and Edge Cases The tests cover various scenarios, including:

- Invalid or missing input data.
- Database connection errors.
- Validation errors for required fields.
- Handling of invalid or non-existent IDs.

5.2.2 admin.controllers/hospital.controller.js

5.2.2.1 Coverage Summary

- **Statements Coverage:** 57.30% (51/89 statements covered)
- **Branches Coverage:** 65.45% (36/55 branches covered)
- **Functions Coverage:** 83.33% (5/6 functions covered)
- **Lines Coverage:** 59.03% (49/83 lines covered)

5.2.2.2 Unit Tests Overview The following controller functions were tested:

- `getHospitals`: Fetches all hospitals from the database.
- `getHospital`: Retrieves a specific hospital by ID.
- `newHospital`: Adds a new hospital to the database.
- `updateHospital`: Updates hospital details.
- `deleteHospital`: Deletes a hospital entry.

5.2.2.3 Error Handling and Edge Cases The tests cover the following scenarios:

- Invalid input data or missing fields.
- Database errors, such as connection failures or query issues.
- Handling non-existent or invalid hospital IDs.
- Validation errors during creation or updates.

5.2.3 admin.controllers/labtechnician.controller.js

5.2.3.1 Coverage Summary

- **Statements Coverage:** 84.37% (27/32 statements covered)
- **Branches Coverage:** 100% (2/2 branches covered)
- **Functions Coverage:** 100% (5/5 functions covered)
- **Lines Coverage:** 84.37% (27/32 lines covered)

5.2.3.2 Unit Tests Overview The following controller functions were tested:

- `newLabTechnician`: Creates a new lab technician entry.
- `getLabTechnician`: Retrieves a specific lab technician by ID.
- `getAllLabTechnicians`: Fetches all lab technicians.
- `updateLabTechnician`: Updates a lab technician's details.
- `deleteLabTechnician`: Deletes a lab technician entry.

5.2.3.3 Error Handling and Edge Cases The tests cover the following scenarios:

- Duplicate entries (e.g., attempting to create an existing lab technician).
- Database errors, such as query issues or connection failures.
- Handling invalid or non-existent lab technician IDs.
- Validation of required fields during creation or updates.

5.2.4 admin.controllers/polyclinic.controller.js

5.2.4.1 Coverage Summary

- **Statements Coverage:** 38.09% (24/63 statements covered)
- **Branches Coverage:** 29.16% (7/24 branches covered)
- **Functions Coverage:** 66.66% (4/6 functions covered)
- **Lines Coverage:** 39.34% (24/61 lines covered)

5.2.4.2 Unit Tests Overview The following controller functions were tested:

- `getPolyclinics`: Fetches polyclinics associated with a hospital.
- `newPolyclinic`: Creates a new polyclinic and links it to a hospital.
- `updatePolyclinic`: Updates the details of an existing polyclinic.
- `deletePolyclinic`: Deletes a polyclinic from the database.

5.2.4.3 Error Handling and Edge Cases The tests cover the following scenarios:

- Invalid or missing input data.
- Database errors during polyclinic retrieval, creation, or updates.
- Handling non-existent or invalid polyclinic IDs.
- Validation errors during the creation or update process.

5.2.5 doctor.controllers/auth.controller.js

5.2.5.1 Coverage Summary

- **Statements Coverage:** 100% (20/20 statements covered)
- **Branches Coverage:** 100% (11/11 branches covered)
- **Functions Coverage:** 100% (1/1 functions covered)
- **Lines Coverage:** 100% (20/20 lines covered)

5.2.5.2 Unit Tests Overview The following controller functions were tested:

- `changePassword`: Allows a doctor to update their password securely.

5.2.5.3 Error Handling and Edge Cases The tests cover the following scenarios:

- Missing required fields (e.g., current password or new password).
- Mismatched new passwords.
- Invalid current password.
- Handling of non-existent doctor records.
- Successful password update and hashing.
- Graceful handling of server/database errors.

5.2.6 doctor.controllers/home.controller.js

5.2.6.1 Coverage Summary

- **Statements Coverage:** 88.97% (121/136 statements covered)
- **Branches Coverage:** 91.84% (45/49 branches covered)
- **Functions Coverage:** 100% (10/10 functions covered)
- **Lines Coverage:** 91.27% (115/126 lines covered)

5.2.6.2 Unit Tests Overview The following controller functions were tested:

- `getPatientDetails`: Fetches the details of a specific patient.
- `getAllPatients`: Retrieves a list of all patients.
- `addPatient`: Adds a new patient to the system.
- `updatePatient`: Updates the details of an existing patient.
- `deletePatient`: Removes a patient record from the system.
- `getPatientAppointments`: Fetches all appointments of a specific patient.
- `updatePatientContact`: Updates the contact details of a patient.
- `getPatientMedicalHistory`: Retrieves the medical history of a patient.
- `updatePatientInsurance`: Updates insurance information for a patient.
- `deletePatientMedicalRecord`: Deletes a specific medical record for a patient.

5.2.6.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Handling invalid or missing patient IDs.
- Validation errors during patient creation or updates.
- Graceful handling of database errors.
- Fetching details for non-existent patients.
- Validating duplicate entries for patient data.

5.2.7 doctor.controllers/patient.controller.js

5.2.7.1 Coverage Summary

- **Statements Coverage:** 83.33% (105/126 statements covered)
- **Branches Coverage:** 83.33% (45/54 branches covered)
- **Functions Coverage:** 100% (10/10 functions covered)
- **Lines Coverage:** 83.33% (105/126 lines covered)

5.2.7.2 Unit Tests Overview

The following controller functions were tested:

- `getPatientDetails`: Retrieves detailed information about a patient.
- `getPatients`: Lists all patients associated with a doctor.
- `getPatient`: Fetches specific patient information for a doctor.
- `createPrescription`: Adds a new prescription for a patient.
- `updatePrescription`: Updates an existing prescription.
- `deletePrescription`: Removes a prescription record.
- `getLabTechniciansBySpecialization`: Lists lab technicians by specialization.
- `newLabTestRequest`: Creates a new lab test request for a patient.

5.2.7.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Handling missing or invalid patient and prescription IDs.
- Validation errors during prescription creation or updates.
- Proper handling of unauthorized actions (e.g., wrong doctor).
- Graceful handling of database errors and server-side exceptions.
- Validation for lab technician specialization and urgency fields.

5.2.8 lab.controllers/auth.controller.js

5.2.8.1 Coverage Summary

- **Statements Coverage:** 100% (20/20 statements covered)
- **Branches Coverage:** 100% (11/11 branches covered)
- **Functions Coverage:** 100% (1/1 functions covered)
- **Lines Coverage:** 100% (20/20 lines covered)

5.2.8.2 Unit Tests Overview The following controller functions were tested:

- `changePassword`: Updates the password of a lab technician securely.

5.2.8.3 Error Handling and Edge Cases The tests cover the following scenarios:

- Missing required fields (e.g., currentPassword or newPassword).
- Mismatched new passwords.
- Handling of non-existent lab technician records.
- Validation of the current password.
- Successful password update and hashing.
- Server errors during the process.

5.2.9 lab.controllers/home.controller.js

5.2.9.1 Coverage Summary

- **Statements Coverage:** 95.65% (22/23 statements covered)
- **Branches Coverage:** 83.33% (5/6 branches covered)
- **Functions Coverage:** 100% (2/2 functions covered)
- **Lines Coverage:** 95.65% (22/23 lines covered)

5.2.9.2 Unit Tests Overview The following controller functions were tested:

- `getHomePage`: Retrieves the lab test queue for the logged-in lab technician.
- `completeTest`: Marks a lab test as completed and updates the test result.

5.2.9.3 Error Handling and Edge Cases The tests cover the following scenarios:

- Lab test queue not found for a lab technician.
- Handling of invalid or missing test IDs during test completion.
- Validation for server/database errors while retrieving or updating lab tests.
- Proper handling of state transitions for lab tests (e.g., marking as completed).

5.2.10 lab.controllers/test.controller.js

5.2.10.1 Coverage Summary

- **Statements Coverage:** 85.71% (30/35 statements covered)
- **Branches Coverage:** 88.88% (8/9 branches covered)
- **Functions Coverage:** 77.77% (7/9 functions covered)
- **Lines Coverage:** 85.71% (30/35 lines covered)

5.2.10.2 Unit Tests Overview

The following controller functions were tested:

- `getResults`: Retrieves lab results for completed tests.
- `getLabTests`: Filters and organizes pending and completed lab tests.
- `deleteLabTest`: Deletes a lab test based on its ID.

5.2.10.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Missing or invalid lab test IDs during deletion.
- Handling empty or non-existent lab test results.
- Graceful management of database errors and server-side exceptions.
- Validation for filtering and sorting pending and completed tests.

5.2.11 patient.controllers/appointment.controller.js

5.2.11.1 Coverage Summary

- **Statements Coverage:** 93.33% (84/90 statements covered)
- **Branches Coverage:** 80% (24/30 branches covered)
- **Functions Coverage:** 100% (10/10 functions covered)
- **Lines Coverage:** 93.02% (80/86 lines covered)

5.2.11.2 Unit Tests Overview

The following controller functions were tested:

- `getHospitalByPolyclinic`: Fetches hospitals associated with a specific polyclinic and filters based on city.
- `newAppointment`: Creates a new appointment and schedules tests if applicable.
- `getAppointments`: Retrieves and updates the list of appointments for a patient.
- `cancelAppointment`: Cancels an appointment and updates related records.

5.2.11.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Missing or invalid query parameters in hospital searches.
- Handling invalid or unavailable dates and times during appointment creation.
- Unauthorized attempts to cancel an appointment.
- Graceful handling of server/database errors during appointment operations.

5.2.12 patient.controllers/auth.controller.js

5.2.12.1 Coverage Summary

- **Statements Coverage:** 92.1% (35/38 statements covered)
- **Branches Coverage:** 93.33% (28/30 branches covered)
- **Functions Coverage:** 100% (2/2 functions covered)
- **Lines Coverage:** 92.1% (35/38 lines covered)

5.2.12.2 Unit Tests Overview The following controller functions were tested:

- `signup`: Registers a new patient and hashes the password securely.
- `changePassword`: Updates the patient's password after validation.

5.2.12.3 Error Handling and Edge Cases The tests cover the following scenarios:

- Missing or invalid required fields during signup or password update.
- Ensuring passwords match for signup and password update.
- Handling existing users during signup.
- Validating the current password before updating.
- Graceful management of server or database errors.

5.2.13 patient.controllers/health.metric.controller.js

5.2.13.1 Coverage Summary

- **Statements Coverage:** 94% (47/50 statements covered)
- **Branches Coverage:** 100% (0/0 branches covered)
- **Functions Coverage:** 100% (9/9 functions covered)
- **Lines Coverage:** 94% (47/50 lines covered)

5.2.13.2 Unit Tests Overview The following controller functions were tested:

- `getHealthMetric`: Retrieves patient health metrics, including BMI calculation.
- `updateWeight`: Updates the patient's weight.
- `updateHeight`: Updates the patient's height.
- `updateHeartRate`: Updates the patient's heart rate.
- `updateBloodPressure`: Updates the patient's blood pressure.
- `updateBloodSugar`: Updates the patient's blood sugar level.
- `updateAllergies`: Adds new allergies for the patient.
- `deleteAllergy`: Removes an allergy from the patient's record.
- `updateBloodType`: Updates the patient's blood type.

5.2.13.3 Error Handling and Edge Cases The tests cover the following scenarios:

- Handling invalid or missing fields for health metric updates.
- Ensuring proper validation for each update operation.
- Graceful management of server/database errors.
- Handling invalid operations, such as removing non-existent allergies.

5.2.14 patient.controllers/home.controller.js

5.2.14.1 Coverage Summary

- **Statements Coverage:** 92.85% (13/14 statements covered)
- **Branches Coverage:** 100% (2/2 branches covered)
- **Functions Coverage:** 80% (4/5 functions covered)
- **Lines Coverage:** 92.85% (13/14 lines covered)

5.2.14.2 Unit Tests Overview The following controller functions were tested:

- `getPatientHome`: Fetches upcoming and recent appointments for the patient.

5.2.14.3 Error Handling and Edge Cases The tests cover the following scenarios:

- Sorting and filtering of upcoming and recent appointments.
- Handling patients without any appointments.
- Managing not found patients gracefully (404).
- Handling server/database errors appropriately (500).
- Special cases, such as appointments exactly at the current time.

5.2.15 patient.controllers/medical.record.controller.js

5.2.15.1 Coverage Summary

- **Statements Coverage:** 100% (17/17 statements covered)
- **Branches Coverage:** 100% (2/2 branches covered)
- **Functions Coverage:** 100% (3/3 functions covered)
- **Lines Coverage:** 100% (17/17 lines covered)

5.2.15.2 Unit Tests Overview The following controller functions were tested:

- `getLabTests`: Retrieves lab tests along with doctor and hospital details.
- `getOtherTests`: Retrieves additional test results for the patient.
- `getDiagnoses`: Fetches diagnoses history for the patient.

5.2.15.3 Error Handling and Edge Cases The tests cover the following scenarios:

- Graceful handling of missing or empty lab test, other test, and diagnosis records.
- Handling invalid or non-existent patient IDs.
- Validation errors and database errors for all retrieval operations.
- Special cases such as invalid patient ID format and server-side exceptions.

5.2.16 patient.controllers/profile.controller.js

5.2.16.1 Coverage Summary

- **Statements Coverage:** 100% (10/10 statements covered)
- **Branches Coverage:** 100% (4/4 branches covered)
- **Functions Coverage:** 100% (2/2 functions covered)
- **Lines Coverage:** 100% (10/10 lines covered)

5.2.16.2 Unit Tests Overview

The following controller functions were tested:

- `getProfile`: Retrieves the profile of a logged-in patient.
- `updateProfile`: Updates the profile details of a logged-in patient.

5.2.16.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Successful retrieval of a patient profile.
- Handling database errors during profile retrieval and update.
- Validation of updates to ensure proper profile data is stored.
- Graceful handling of server/database errors during updates.

5.2.17 models/admin.model.js

5.2.17.1 Coverage Summary

- **Statements Coverage:** 100% (2/2 statements covered)
- **Branches Coverage:** 100% (2/2 branches covered)
- **Functions Coverage:** 100% (2/2 functions covered)
- **Lines Coverage:** 100% (2/2 lines covered)

5.2.17.2 Unit Tests Overview

The following aspects of the Admin model were tested:

- Validation of required fields, including `name`, `surname`, `email`, `password`, `phone`, and `role`.
- Creation of an admin with valid data.
- Automatic timestamp generation for `createdAt` and `updatedAt`.
- Rejection of invalid email formats.
- Confirmation that timestamps are enabled in the schema.

5.2.17.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Missing required fields resulting in validation errors.
- Handling invalid email formats during validation.
- Ensuring proper database interactions, including saving and retrieving records.
- Ensuring that timestamps are properly added and verified.

5.2.18 models/appointment.model.js

5.2.18.1 Coverage Summary

- **Statements Coverage:** 100% (2/2 statements covered)
- **Branches Coverage:** 100% (3/3 branches covered)
- **Functions Coverage:** 100% (3/3 functions covered)
- **Lines Coverage:** 100% (2/2 lines covered)

5.2.18.2 Unit Tests Overview

The following aspects of the Appointment model were tested:

- Creation and saving of appointments with valid data.
- Validation for required fields, such as `patient`, `doctor`, `hospital`, and `polyclinic`.
- Updating the status of an existing appointment.
- Deleting an appointment from the database.

5.2.18.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Missing required fields resulting in validation errors.
- Proper saving and updating of appointment data in the database.
- Ensuring that deleted appointments are completely removed from the database.
- Handling invalid data gracefully during appointment creation or update.

5.2.19 models/diagnosis.model.js

5.2.19.1 Coverage Summary

- **Statements Coverage:** 100% (2/2 statements covered)
- **Branches Coverage:** 100% (4/4 branches covered)
- **Functions Coverage:** 100% (2/2 functions covered)
- **Lines Coverage:** 100% (2/2 lines covered)

5.2.19.2 Unit Tests Overview

The following aspects of the Diagnosis model were tested:

- Creation and saving of diagnoses with valid data.
- Validation for required fields, such as `condition`, `description`, `prescriptions`, `status`, and `date`.
- Updating the status of an existing diagnosis.
- Deletion of a diagnosis from the database.

5.2.19.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Missing required fields resulting in validation errors.
- Proper saving and updating of diagnosis data in the database.
- Ensuring that deleted diagnoses are completely removed from the database.
- Handling invalid data gracefully during diagnosis creation or update.

5.2.20 models/doctor.model.js

5.2.20.1 Coverage Summary

- **Statements Coverage:** 100% (37/37 statements covered)
- **Branches Coverage:** 100% (14/14 branches covered)
- **Functions Coverage:** 100% (2/2 functions covered)
- **Lines Coverage:** 100% (35/35 lines covered)

5.2.20.2 Unit Tests Overview

The following aspects of the Doctor model were tested:

- Creation of a doctor with proper fields such as `name`, `email`, `specialization`, `role`, etc.
- Validation of schedule initialization with proper weekday time slots.
- Updating schedules to remove outdated days and add new weekdays.
- Correct handling of time slot statuses for cancellations and bookings.
- Custom logic for schedule updates, ensuring weekends are skipped and weekdays are added dynamically.

5.2.20.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Handling empty or invalid schedules during initialization and updates.
- Proper validation of dates and time slots.
- Graceful handling of invalid operations, such as scheduling outside working hours.
- Ensuring schedule consistency when multiple past days are removed and new days are added.
- Validation errors for invalid doctor fields during creation or updates.

5.2.21 models/hospital.model.js

5.2.21.1 Coverage Summary

- **Statements Coverage:** 100% (2/2 statements covered)
- **Branches Coverage:** 100% (2/2 branches covered)
- **Functions Coverage:** 100% (2/2 functions covered)
- **Lines Coverage:** 100% (2/2 lines covered)

5.2.21.2 Unit Tests Overview

The following aspects of the Hospital model were tested:

- Validation of required fields such as `name`, `address`, `phone`, and `establishmentdate`.
- Successful creation of a hospital with valid data, including proper field types and values.
- Validation for proper email format during hospital creation.
- Adding doctors to the hospital's doctor list and ensuring the list updates correctly.

5.2.21.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Missing required fields leading to validation errors.
- Invalid email format during hospital creation.
- Proper handling of database interactions, including saving and retrieving hospital records.
- Validation of doctor list updates and handling invalid entries.

5.2.22 models/lab.technician.model.js

5.2.22.1 Coverage Summary

- **Statements Coverage:** 100% (2/2 statements covered)
- **Branches Coverage:** 100% (3/3 branches covered)
- **Functions Coverage:** 100% (3/3 functions covered)
- **Lines Coverage:** 100% (2/2 lines covered)

5.2.22.2 Unit Tests Overview

The following aspects of the Lab Technician model were tested:

- Validation of required fields such as `name`, `surname`, `email`, `password`, `phone`, `role`, `specialization`, `jobstartdate`, `birthdate`, and `title`.
- Successful creation of a lab technician with valid data, including proper field types and values.
- Validation for proper email format during lab technician creation.

5.2.22.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Missing required fields resulting in validation errors.
- Invalid email format during lab technician creation.
- Proper handling of database interactions, including saving and retrieving lab technician records.

5.2.23 models/lab.test.model.js

5.2.23.1 Coverage Summary

- **Statements Coverage:** 100% (2/2 statements covered)
- **Branches Coverage:** 100% (2/2 branches covered)
- **Functions Coverage:** 100% (3/3 functions covered)
- **Lines Coverage:** 100% (2/2 lines covered)

5.2.23.2 Unit Tests Overview

The following aspects of the Lab Test model were tested:

- Validation of required fields, including `patient`, `doctor`, `hospital`, `polyclinic`, and `labTechnician`.
- Default value assignment for fields like `status` (set to `pending`).
- Optional fields such as `result` and `urgency`.
- Proper saving and retrieval of lab tests with valid input.

5.2.23.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Validation errors when required fields are missing.
- Proper database interactions, including saving and retrieving records.
- Ensuring correct default values for optional fields.
- Handling invalid data gracefully during lab test creation.

5.2.24 models/patient.model.js

5.2.24.1 Coverage Summary

- **Statements Coverage:** 100% (2/2 statements covered)
- **Branches Coverage:** 100% (1/1 branches covered)
- **Functions Coverage:** 100% (1/1 functions covered)
- **Lines Coverage:** 100% (2/2 lines covered)

5.2.24.2 Unit Tests Overview

The following aspects of the Patient model were tested:

- Creation and saving of a patient with valid data, ensuring proper validation of fields such as `name`, `email`, `phone`, `bloodtype`, and `allergies`.
- Validation errors for missing required fields like `name`.
- Updating an existing patient's information, such as `address`.
- Deleting a patient and ensuring the record is completely removed from the database.

5.2.24.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Missing required fields resulting in validation errors.
- Updating optional fields like `address`.
- Ensuring proper database interactions for creation, updates, and deletion of patient records.
- Graceful handling of invalid operations, such as attempting to delete a non-existent patient.

5.2.25 models/polyclinic.model.js

5.2.25.1 Coverage Summary

- **Statements Coverage:** 100% (2/2 statements covered)
- **Branches Coverage:** 100% (1/1 branches covered)
- **Functions Coverage:** 100% (3/3 functions covered)
- **Lines Coverage:** 100% (2/2 lines covered)

5.2.25.2 Unit Tests Overview

The following aspects of the Polyclinic model were tested:

- Creation and saving of a polyclinic with valid data, including required fields like `name`, `hospital`, and `doctors`.
- Validation errors for missing required fields such as `name` and `hospital`.
- Updating the `name` of an existing polyclinic successfully.
- Deleting a polyclinic and ensuring it is completely removed from the database.

5.2.25.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Validation errors when required fields are missing during polyclinic creation.
- Proper handling of updates to polyclinic records, including overwriting `name`.
- Ensuring proper database interactions during creation, updates, and deletion.
- Handling invalid operations gracefully, such as attempting to delete a non-existent polyclinic.

5.2.26 models/prescription.model.js

5.2.26.1 Coverage Summary

- **Statements Coverage:** 100% (2/2 statements covered)
- **Branches Coverage:** 100% (2/2 branches covered)
- **Functions Coverage:** 100% (2/2 functions covered)
- **Lines Coverage:** 100% (2/2 lines covered)

5.2.26.2 Unit Tests Overview

The following aspects of the Prescription model were tested:

- Creation and saving of a prescription with valid data, ensuring proper validation of fields like `medicine`, `status`, `doctor`, and `hospital`.
- Validation errors for missing required fields such as `medicine.time`, `status`, `doctor`, and `hospital`.
- Updating an existing prescription successfully, such as changing the `status`.
- Deletion of a prescription and ensuring it is removed from the database.

5.2.26.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Missing required fields leading to validation errors.
- Handling of incorrect or incomplete prescription data.
- Proper database interactions for saving, updating, and deleting prescriptions.
- Ensuring that deleted prescriptions are completely removed from the database.

5.2.27 models/treatment.model.js

5.2.27.1 Coverage Summary

- **Statements Coverage:** 100% (2/2 statements covered)
- **Branches Coverage:** 100% (2/2 branches covered)
- **Functions Coverage:** 100% (2/2 functions covered)
- **Lines Coverage:** 100% (2/2 lines covered)

5.2.27.2 Unit Tests Overview

The following aspects of the Treatment model were tested:

- Creation of a treatment with valid data, ensuring all required fields such as `patient`, `doctor`, `hospital`, `diagnosis`, and `prescription` are properly validated.
- Validation errors when required fields are missing, such as `patient` and `prescription`.
- Updating an existing treatment, including changing the `status` field.
- Deleting a treatment and ensuring it is completely removed from the database.

5.2.27.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Validation errors for missing required fields.
- Ensuring proper database interactions during creation, updates, and deletion.
- Handling updates that modify critical fields like `status`.
- Confirming that deleted treatments cannot be retrieved from the database.

5.2.28 utils/generateJwt.js

5.2.28.1 Coverage Summary

- **Statements Coverage:** 100% (5/5 statements covered)
- **Branches Coverage:** 100% (1/1 branches covered)
- **Functions Coverage:** 100% (1/1 functions covered)
- **Lines Coverage:** 100% (5/5 lines covered)

5.2.28.2 Unit Tests Overview

The following aspects of the JWT generation utility were tested:

- Successful generation of access and refresh tokens for a user.
- Verification that the tokens are properly signed with the correct secrets.
- Proper setting of cookies for `accessToken` and `refreshToken` with appropriate options, such as `httpOnly` and `sameSite`.
- Handling errors when JWT secrets are missing from the environment.

5.2.28.3 Error Handling and Edge Cases

The tests cover the following scenarios:

- Validation of token content to ensure it matches the user ID.
- Graceful handling of missing `JWT_SECRET` or `JWT_REFRESH_SECRET` environment variables by throwing errors.
- Ensuring cookies are set with correct expiration times (`15 minutes` for access tokens, `7 days` for refresh tokens).
- Testing the integrity of the signed JWTs by decoding and verifying their payloads.