CIS 550 – DATABASE DESIGN

SPRING 2024

FINAL PROJECT

HOSPITAL MANAGEMENT SYSTEM

GROUP- 32

Presentation Date- 20th April 2024

Professor:

Name: Dr. Ashok Patel
Office: Dion 302B
Email: apatel38@umassd.edu

Group Members:

R.No	Name	Student ID
71	Jai Saxena	02129249
61	Mayank Raj	02135041

PART-1

INTRODUCTION

The Hospital Management System (HMS) employs a relational schema to streamline operations across various entities such as Patients, Doctors, Appointments, Medical Records, Bills, Staff, and Inventory. Each entity is defined with unique identifiers and specific attributes: Patients are recorded with details such as ID, name, date of birth, address, phone number, and insurance information Doctors are cataloged by ID, name, specialization, and schedule. The system enhances healthcare delivery by linking these entities through well-defined relationships. For instance, the one-to-many relationship between Patients and Appointments allows multiple appointments per patient, facilitating efficient scheduling. Similarly, Doctors and Appointments are linked to optimize time management and patient care continuity.

Further, each patient's multiple Medical Records ensure comprehensive tracking of health history, crucial for ongoing treatment strategies. Bills associated with each patient are managedto streamline financial processes, enhancing transaction efficiency. The relational dynamics extend to staff and doctors, where multiple staff members, like nurses and administrative workers, support various doctors, indicating a versatile and interconnected workforce. Additionally, the system manages Inventory by tracking items, quantities, and reorder levels, which is critical for the operational needs of medical facilities. This structured data organization not only supports day-to-day administrative and medical tasks but also contributes to long-termhealth management and operational planning in hospitals.

APPLICATION REQUIREMENTS/ USE- CASE

The planned Hospital Management System (HMS) utilizes a database-driven application to manage and automate key hospital operations efficiently and effectively. This system is designed to enhance operational efficiency and improve the user experience by integrating core functionalities into a cohesive platform.

1. **Patient Management:** The system enables patient registration, updates to patient profiles, and the management of their medical histories. By using a relational database, the application can quickly retrieve and update patient information, ensuring that medical personnel have access to up-to-date data. This aids in delivering personalized patient care and maintaining accurate health records.

2. **Appointment Scheduling:** This functionality supports seamless booking, rescheduling, and cancellation of appointments. The database stores appointment details, including patient and doctor IDs, which helps in managing and optimizing doctor schedules and reducing patient wait times. The system can also send automated reminders to both doctors and patients, improving the efficiency of the scheduling process.

3. **Medical Records Management:** After each consultation, the system allows for the creation and secure access to detailed medical records. These records are stored in the database with restricted access based on user roles, ensuring confidentiality and compliance with healthcare regulations. Medical staff can access these records as needed, providing continuity in patient care, and facilitating better clinical decisions.

4. **Billing and Insurance Processing:** The HMS automates the generation of bills for medical services and manages insurance claims processing. This module uses the database to track all billing transactions linked to patient and insurance records, streamlining financial operations and ensuring accurate billing and timely insurance processing.

5. **Staff and Schedule Management:** This functionality manages detailed records of hospital staff, including their roles, departments, and schedules. The database system enables HR managers to allocate and manage resources efficiently, ensuring that there are adequate staff members available for different shifts and specialties.

6. **Inventory Management:** The inventory management module tracks medical supplies and medications, monitoring stock levels, reorder points, and expiration dates. This helps in maintaining optimal stock levels, avoiding overstock and outages, and ensuring that the necessary medical supplies are always available for patient care.

By leveraging modern database technologies, the HMS ensures that all these functionalities are integrated into a single, user-friendly platform. This not only enhances the efficiency of hospital operations but also significantly improves the quality of patient care. The database-driven approach allows for scalability and flexibility, making it easier to update the system as healthcare practices evolve and new needs emerge.

RELATIONAL SCHEMA-

ENTITES USED IN THE HOTEL MANGEMENT SYSTEM DATABASE:

1. PATIENTS
2. DOCTORS
3. APPOINTMENTS
4. MEDICAL RECORDS
5. BILLS
6. STAFF
7. INVENTORY

Attributes used for Entities.

1. **Patients**
- PatientID (PK)
- Name
- DOB (Date of Birth)
- Address
- Phone
- InsuranceInfo

2. **Doctors**
- DoctorID (PK)
- Name
- Specialization
- Schedule

3. **Appointments**
- AppointmentID (PK)
- PatientID (FK to Patients)
- DoctorID (FK to Doctors)
- Date
- Time
- Purpose

4. **Medical Records**
- RecordID (PK)
- PatientID (FK to Patients)
- Visit Date
- Diagnosis
- Treatment

5. **Bills**
- BillID (PK)
- PatientID (FK to Patients)
- Date
- Amount
- Status (e.g., Paid, Unpaid, Pending Insurance)

6. **Staff**
- StaffID (PK)
- Name
- Role
- Department

7. Inventory
- temID (PK)
- Name
- Quantity
- ReorderLevel

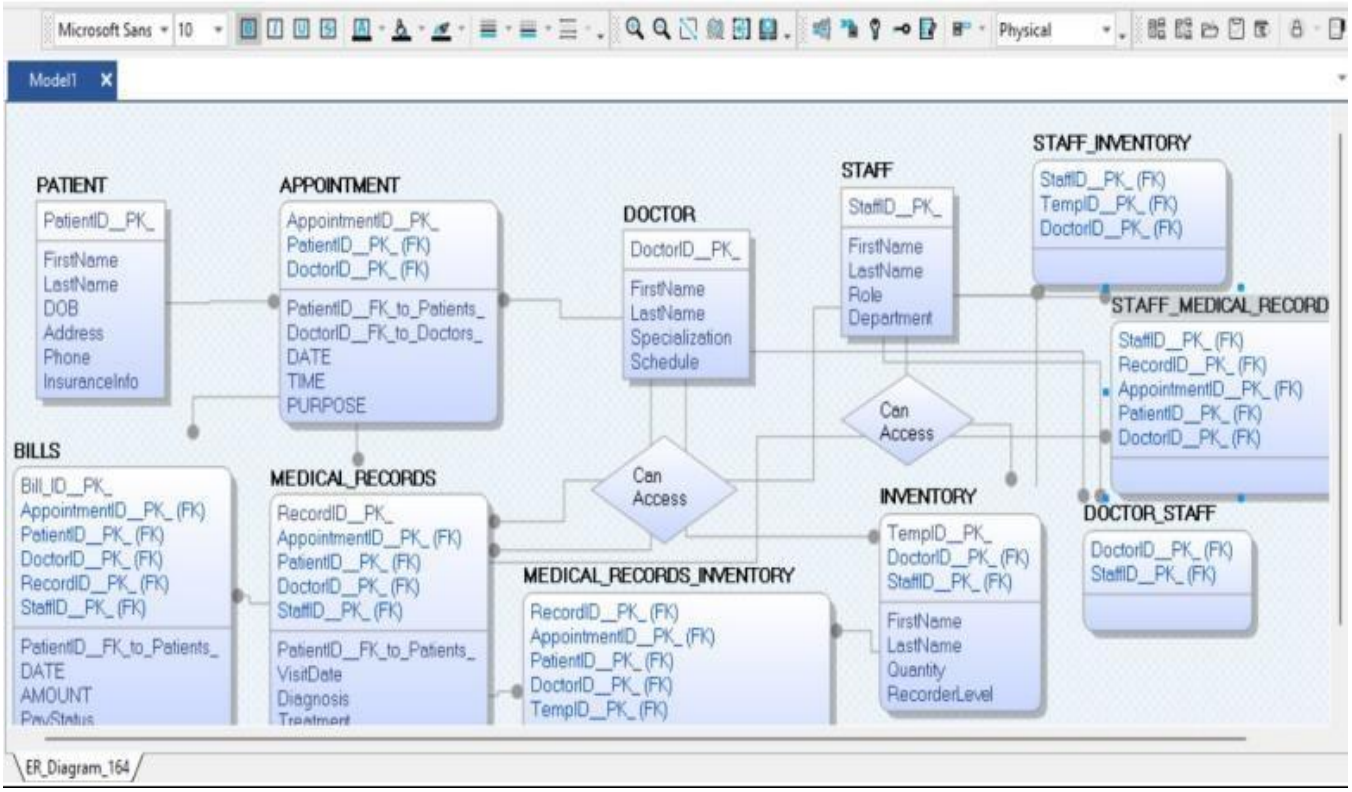
RELATIONSHIP BETWEEN ENTITIES IN THIS HOTEL MANAGEMENT SYSTEM DATABASE

In the Hospital Management System, the relationships between entities ensure efficient dataorganization and functionality:

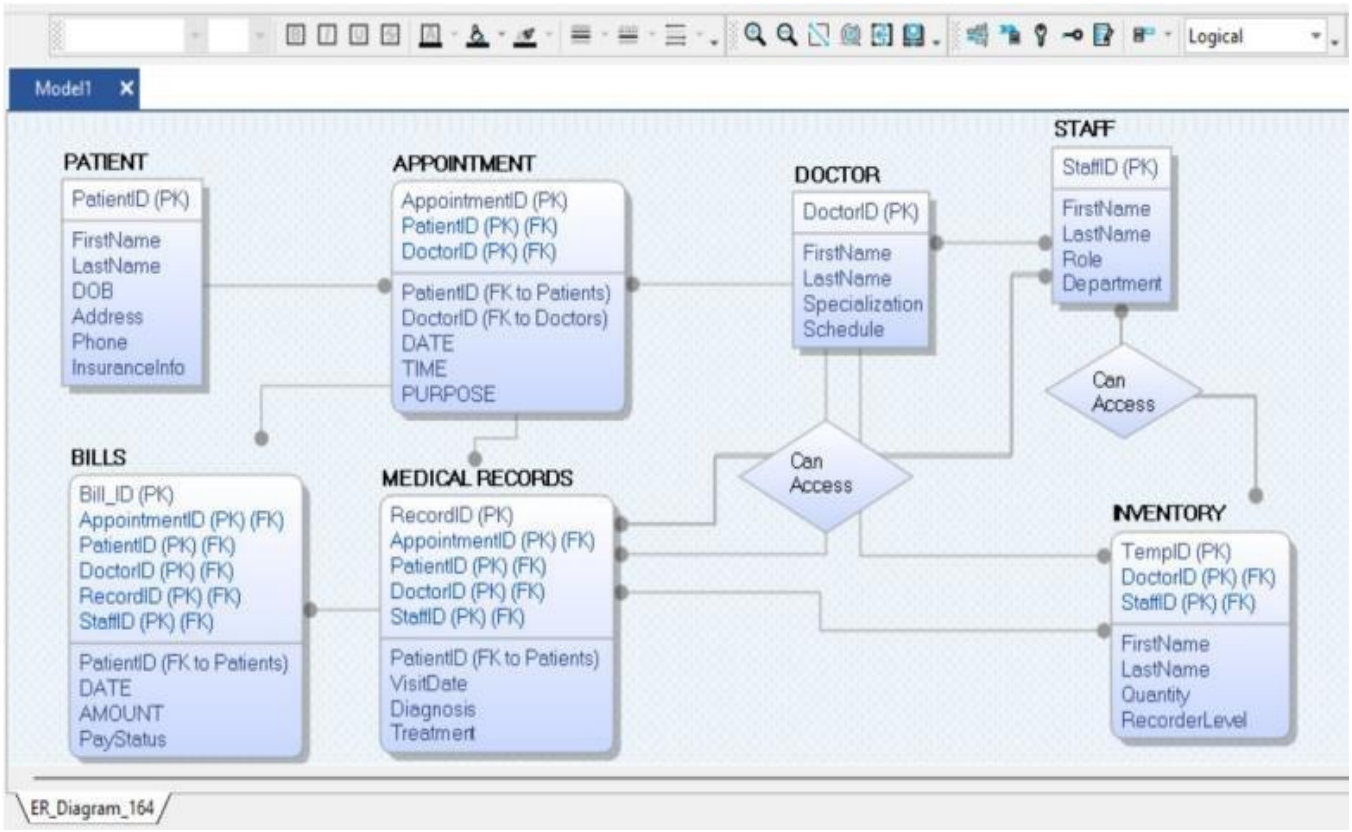
1. Patients and Appointments (One-to-Many): A patient can have multiple appointments,organize schedules, and facilitate healthcare management for individuals.
2. Doctors and Appointments (One-to-Many): Each doctor is linked to many appointments,streamlining doctor schedules and patient care.
3. Patients and Medical Records (One-to-Many): Multiple medical records are maintained foreach patient, centralizing health history for better care continuity.
4. Patients and Bills (One-to-Many): Patients generate multiple bills over time, simplifyingfinancial transactions and tracking.
5. 'M' Staff could consist of 'N' many Doctors.
6. 'M' Doctors could have 'N' many staff. (e.g. compounder, ward boy, etc.)
7. '1' Doctor (based on specialization) could access many 'N' Medical Records.
8. 'M' Staff could access many 'N' Medical Records.
9. '1'' Medical Record could have many 'N' Bills.
10. 'M' Inventory could access many 'N' Medical Records.
11. 'M' Medical Records could access many 'N' Inventory.

These relationships enable the HMS to coordinate care, manage schedules, track health histories,and handle billing efficiently, ensuring comprehensive and integrated healthcare services. This schema covers the basic structure and relationships needed for a hospital management system,focusing on patient care, appointments, medical records, billing, staff management, andinventory tracking. It lays the groundwork for developing a database that supports the system's key functionalities.

PHYSICAL MODEL

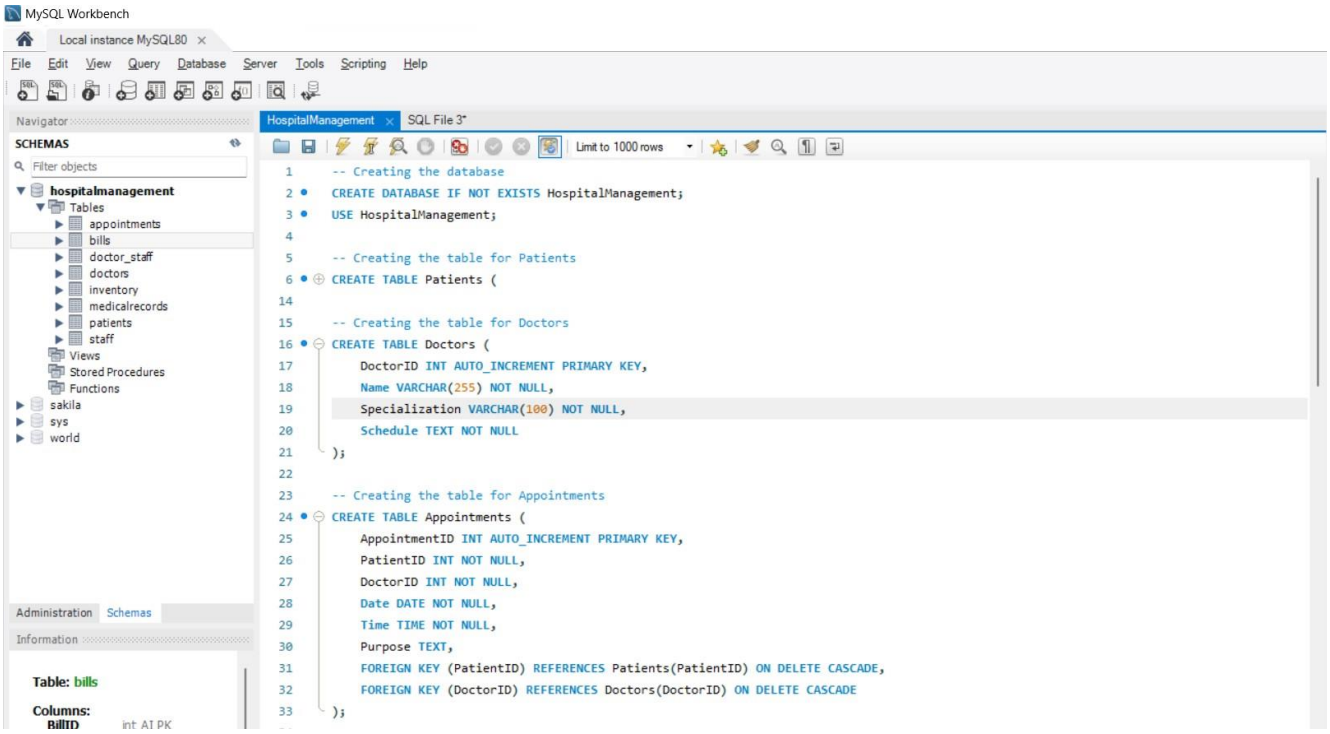


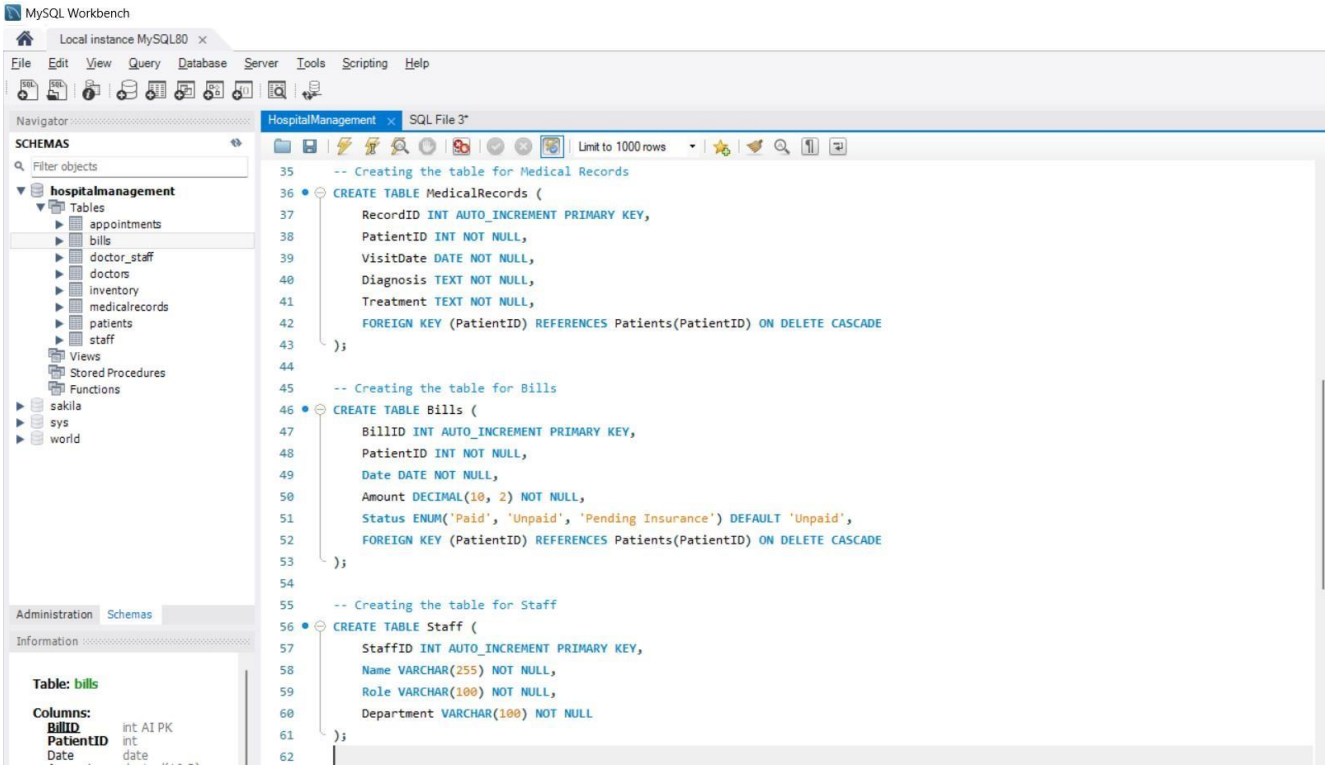
LOGICAL MODEL

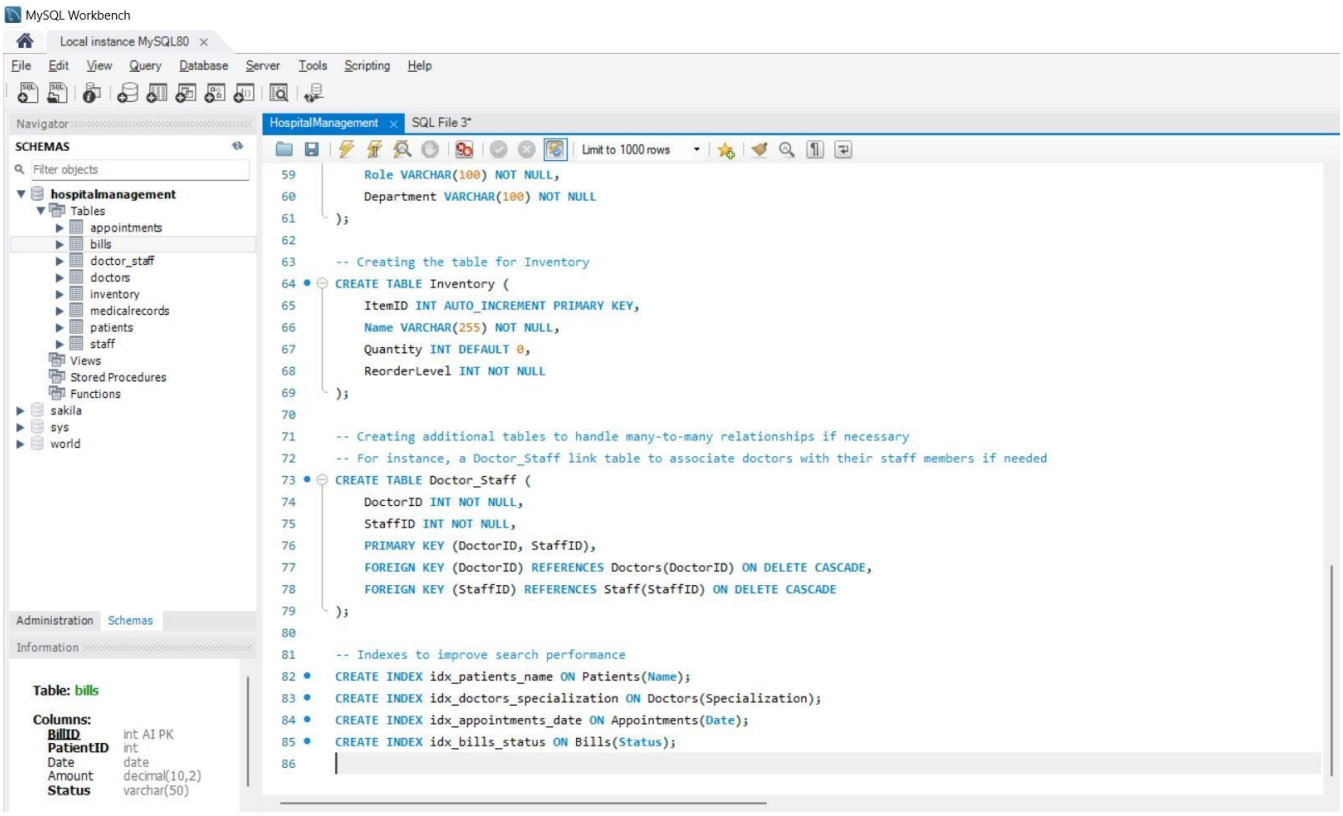


PART-2: DATABASE CODES

Below is the database code we used to create all the entities and their structures in the database.I explicitly created the database and its entities to reduce the amount of code at the backend.

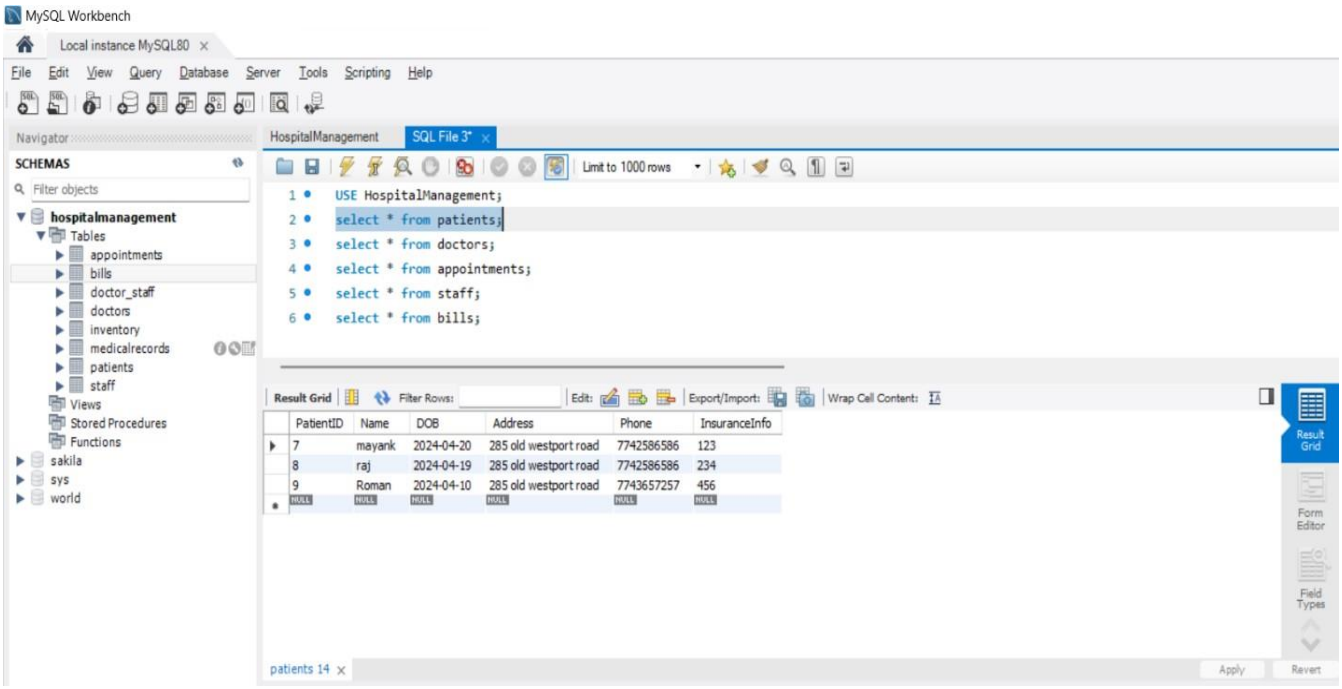




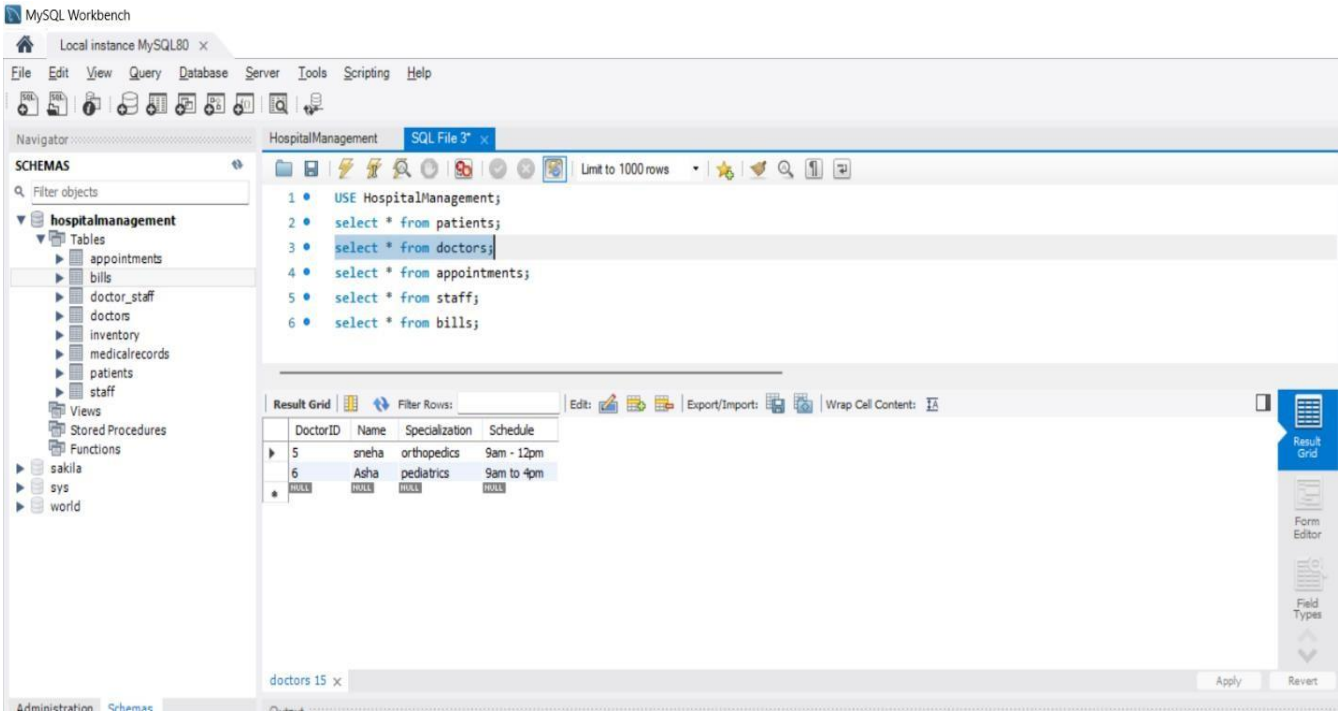


Below are some of the **CRUD** operations we did during the presentation and its evidence that itgot saved in the database as intended to work.

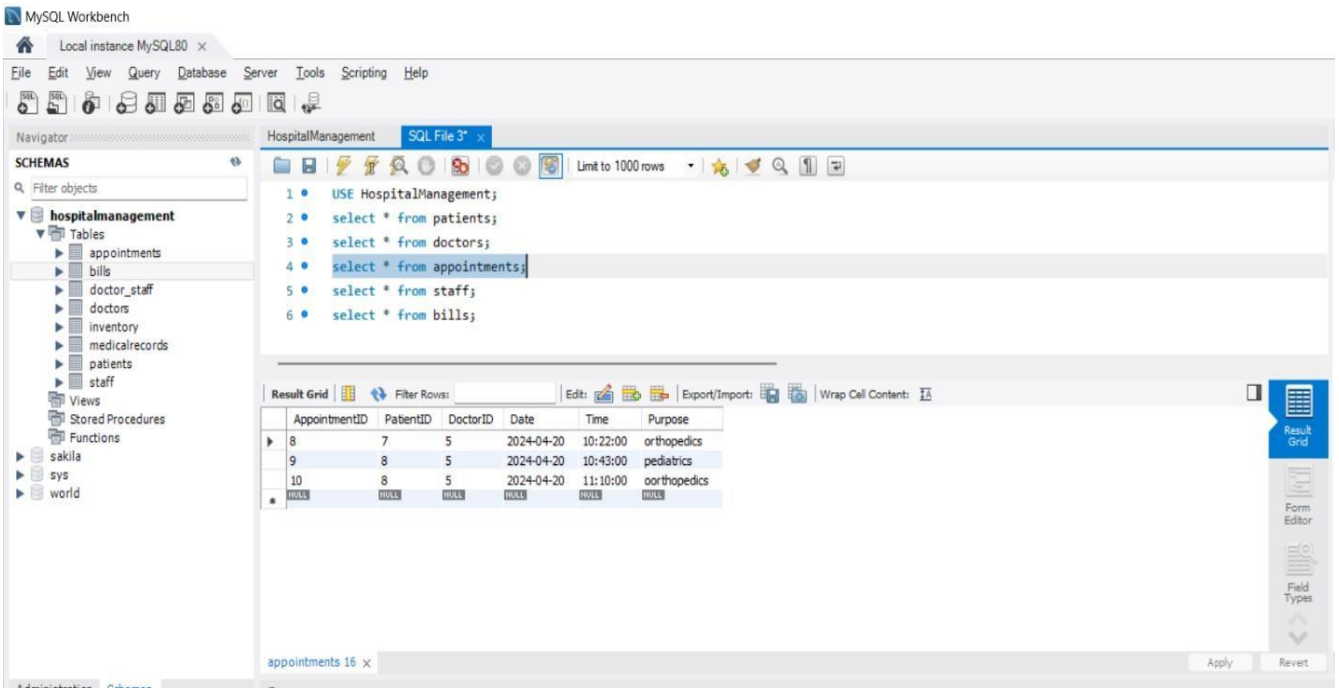
As in the presentation, I added **PatientID 9 as Roman** with all the mentioned detailed on thedatabase as can be seen in the screenshot below. The **PatientID is unique**.



As seen in the screenshot, I added the **DoctorID as 6 with name Asha** during the live presentation in class. **DoctorID is unique** for every doctor.

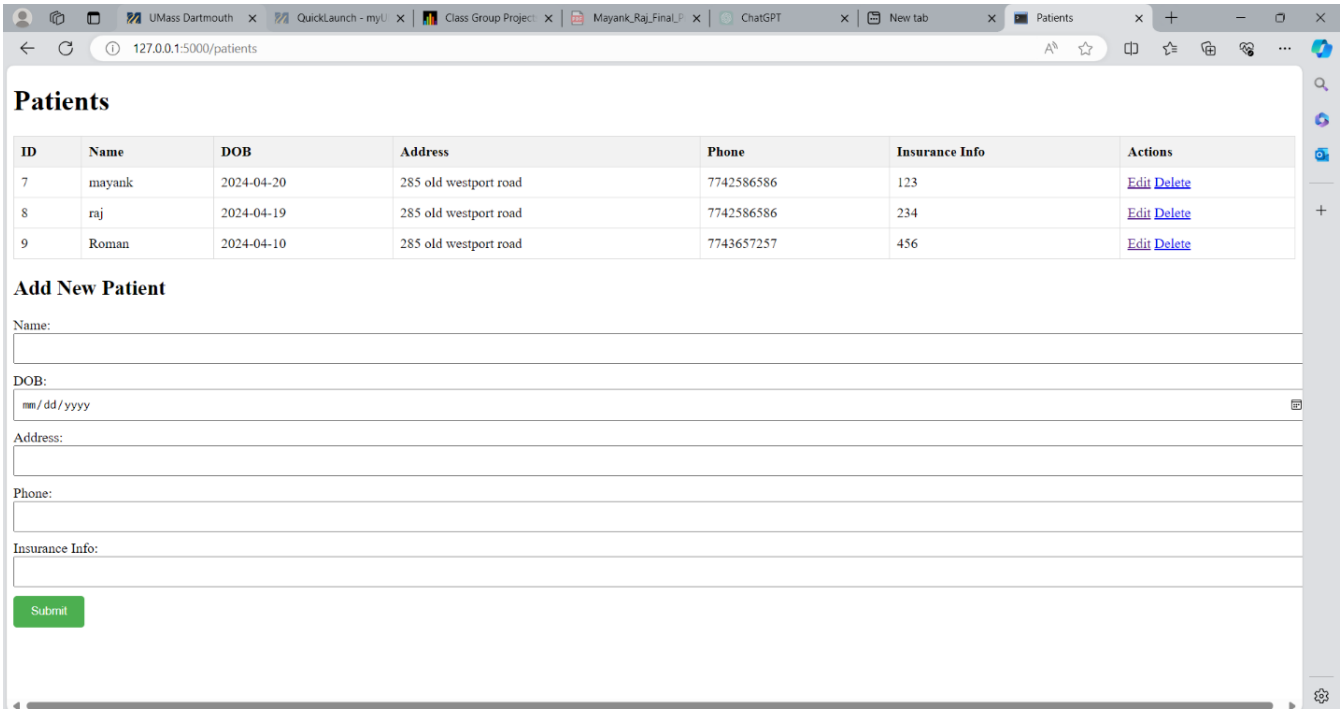


The **PatientID** and **DoctorId** are unique as can be seen in the screenshots attached with the database code, Here as shown in the live demonstration in the class the DoctorID and **PatientID**needs to be inserted with other details needed to make an appointment. This can be seen below as well.

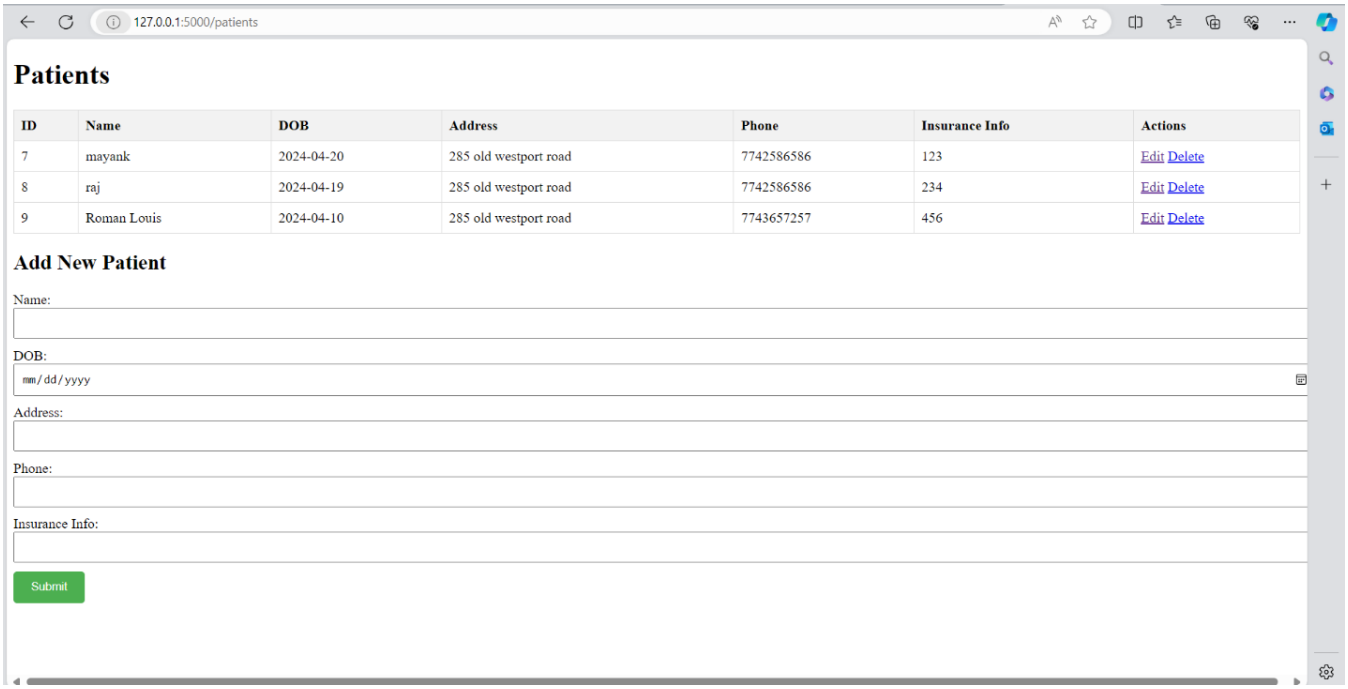


PART-3 FRONTEND INTERFACE

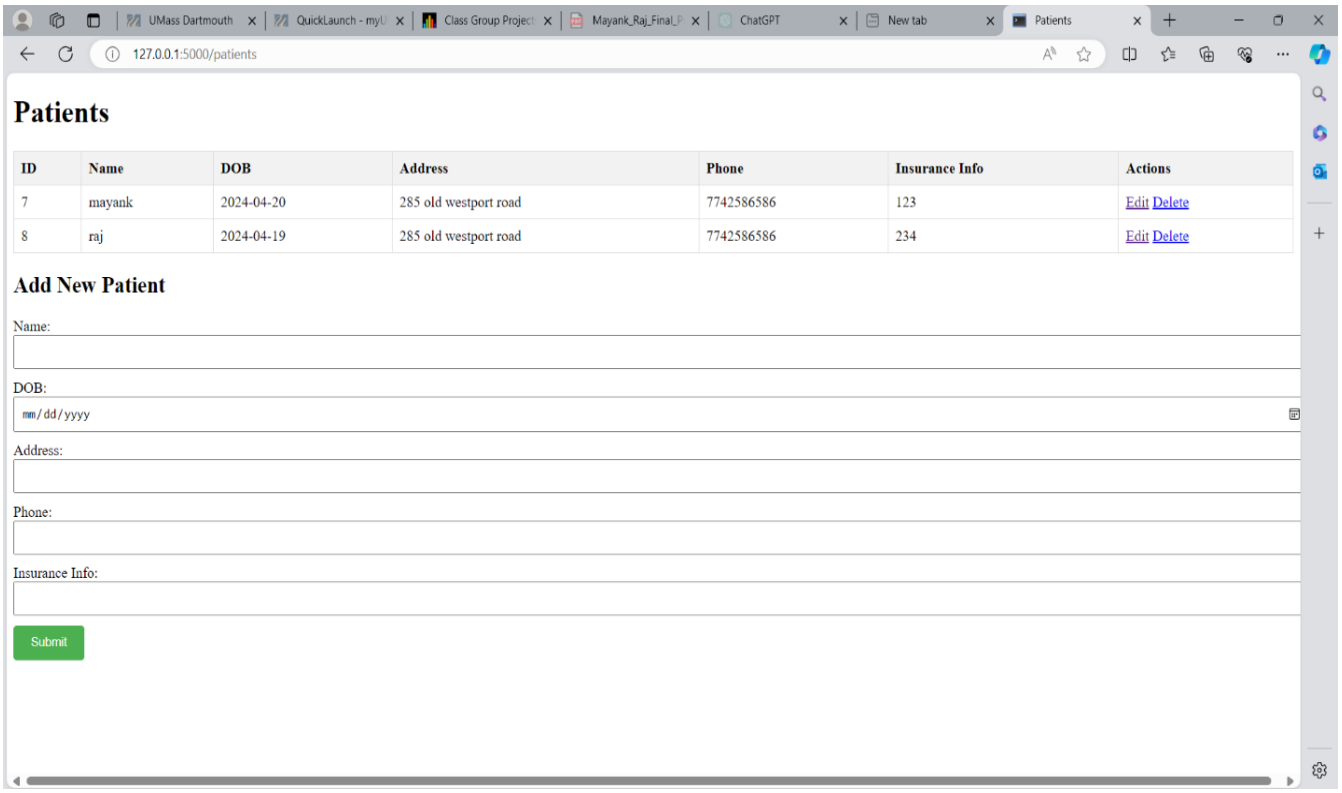
As can be seen in the screenshot below, **patient 9 named Roman** is added or **CREATED** in thepatients list with all the details as mentioned above in the database section.



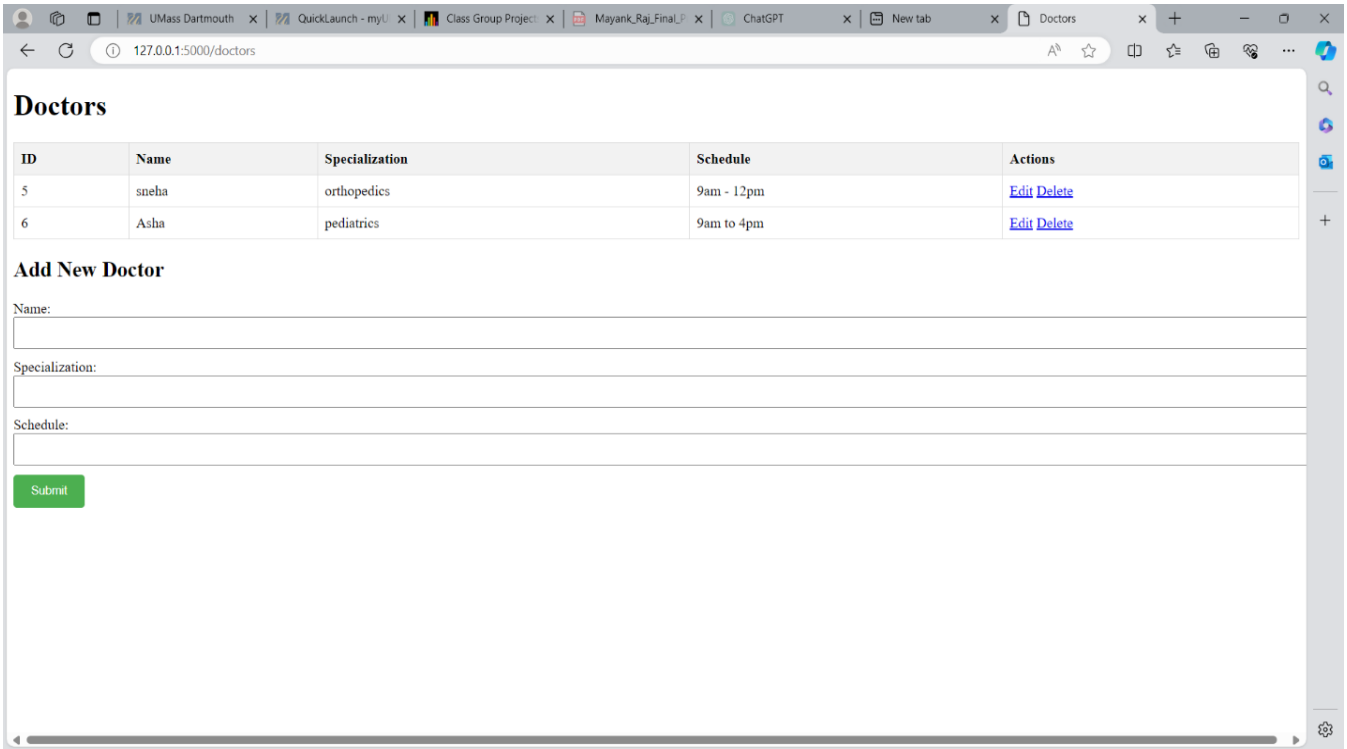
As seen in the screenshot below, I **UPDATED** patient **9** by adding **the last name** to his **firstname** seen in the above screenshot.



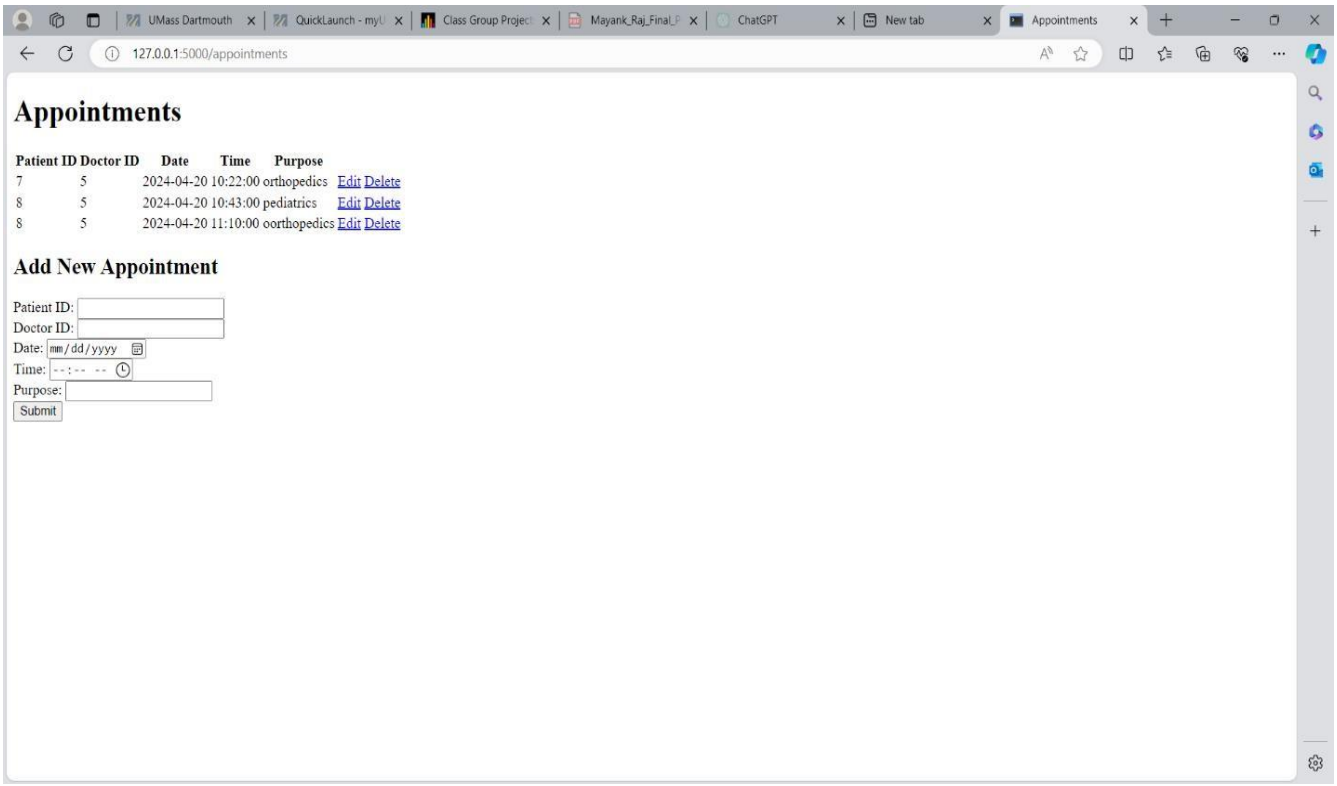
As seen in the below screenshot, **I DELETED Patient 9** from the patient list we created duringthe live demonstration in the class presentations.



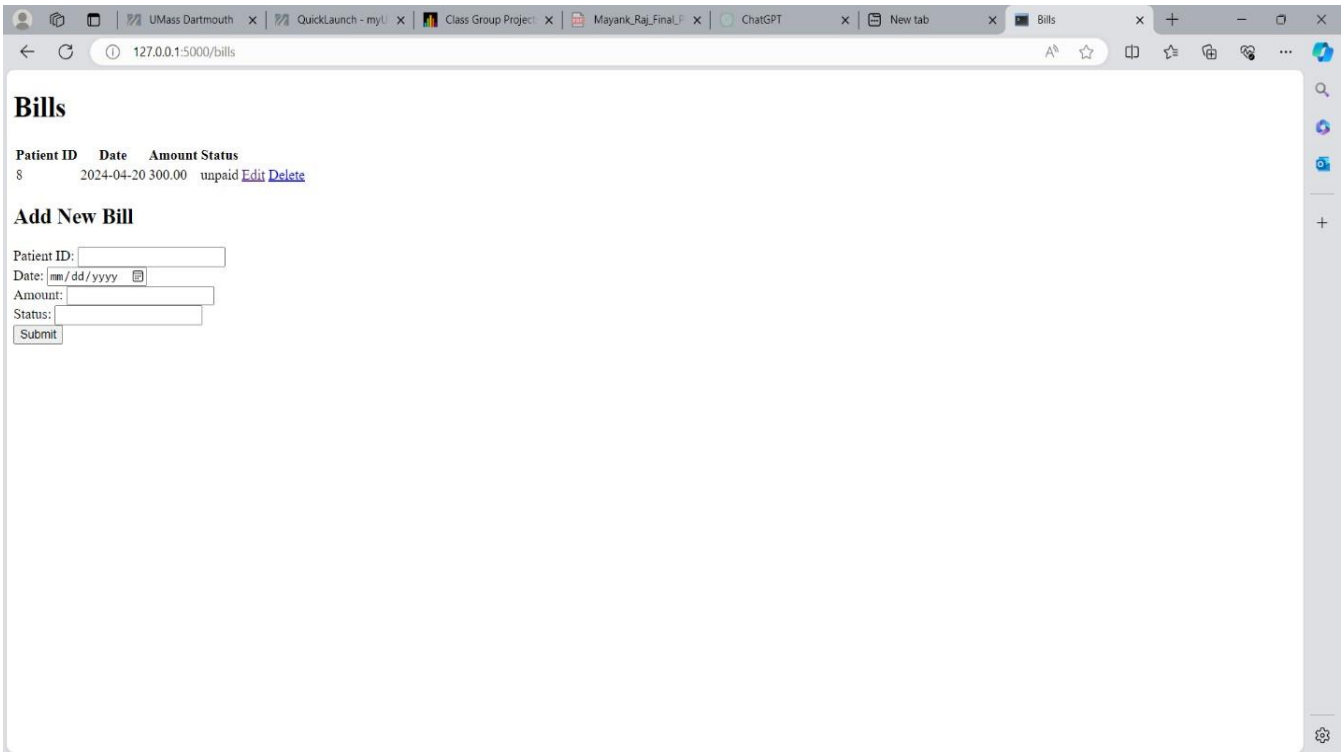
Similarly, the screenshot below shows the Doctors List as per the database and the names wecreated in the live demonstration.



Below is the **Appointments** page with the mentioned **PatientID** and **DoctorID** which are uniqueand their scheduled appointments.



Below is the **Bills** Page as can be seen in the screenshot below. It also shows the bills currently inthe database as created during the live demonstration in the class.



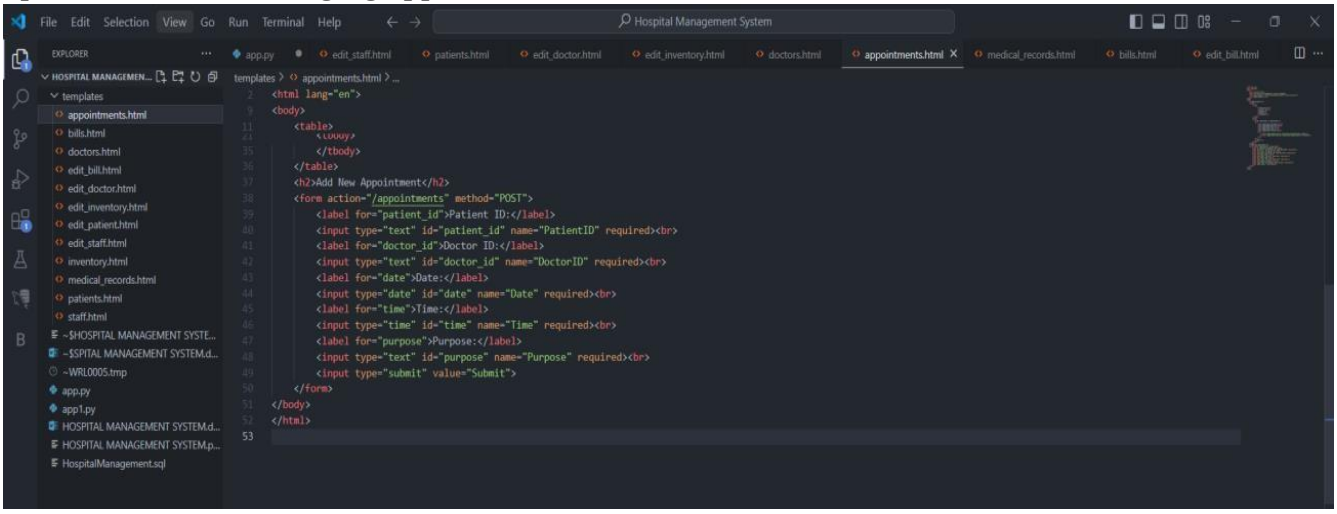
Below screenshot shows the **Staff Management** page and can see a staff added to the list thatshows the exact copy of the data stored in the database.

Below screenshot shows the **Manage Inventory** page and can see an item added to the list thatshows the exact copy of the data stored in the database.

PART- 4: SOURCE CODES BACKEND AND FRONTEND CODES

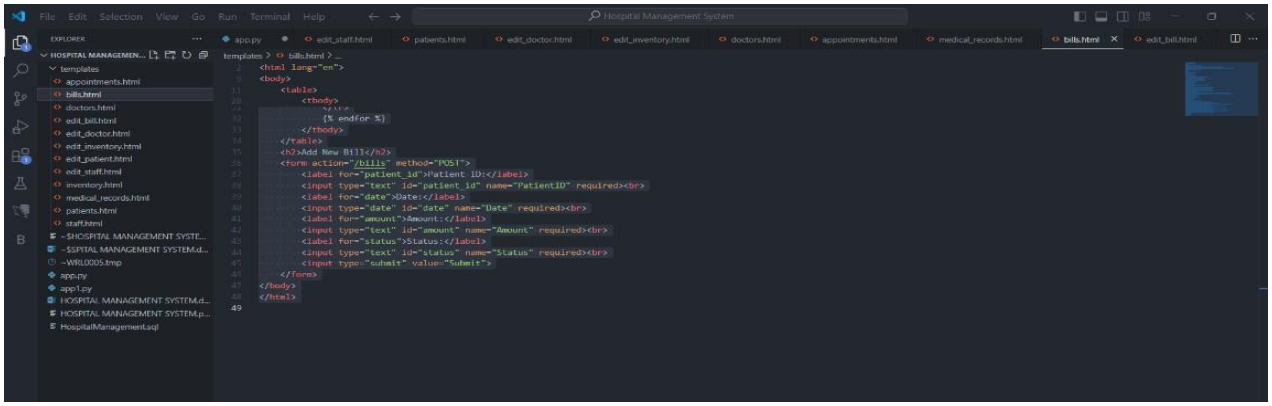
FRONTEND CODES

Below is the screenshot of the **appointment page** HTML codes that makes CRUD operations onthe managing appointment side of the HMS.

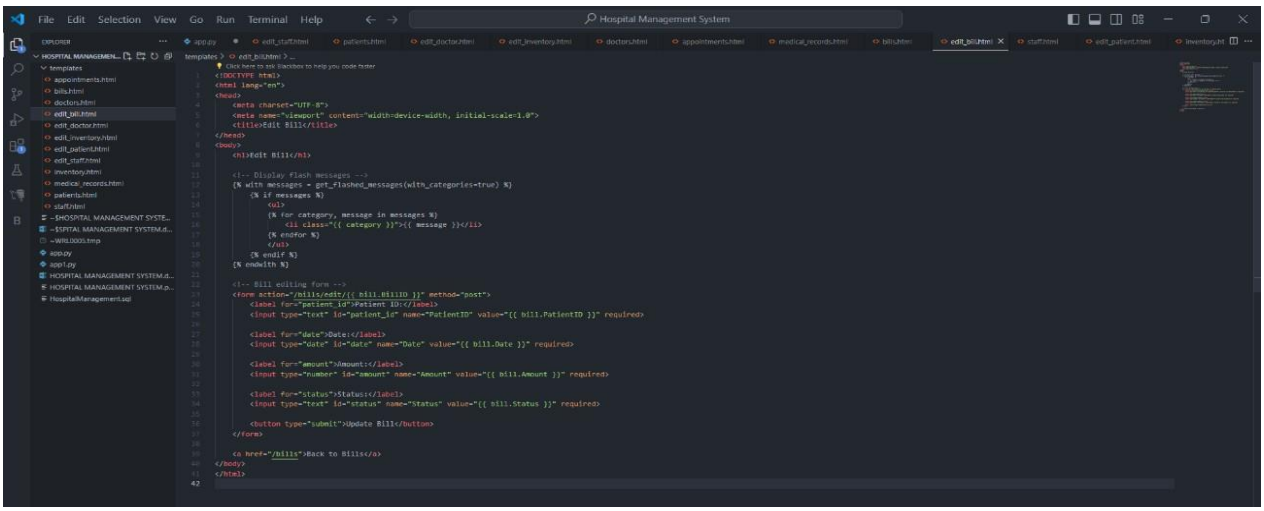


```
<html lang="en">
<body>
    <table>
        <tbody>
        </tbody>
    </table>
    <h2>Add New Appointment</h2>
    <form action="/appointments" method="POST">
        <label for="patient_id">Patient ID:</label>
        <input type="text" id="patient_id" name="PatientID" required><br>
        <label for="doctor_id">Doctor ID:</label>
        <input type="text" id="doctor_id" name="DoctorID" required><br>
        <label for="date">Date:</label>
        <input type="date" id="date" name="Date" required><br>
        <label for="time">Time:</label>
        <input type="time" id="time" name="Time" required><br>
        <label for="purpose">Purpose:</label>
        <input type="text" id="purpose" name="Purpose" required><br>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```

Below is the screenshot of the **Bills page** and **Edit Bill** HTML codes that makes CRUD operationson the managing bills side of the HMS.

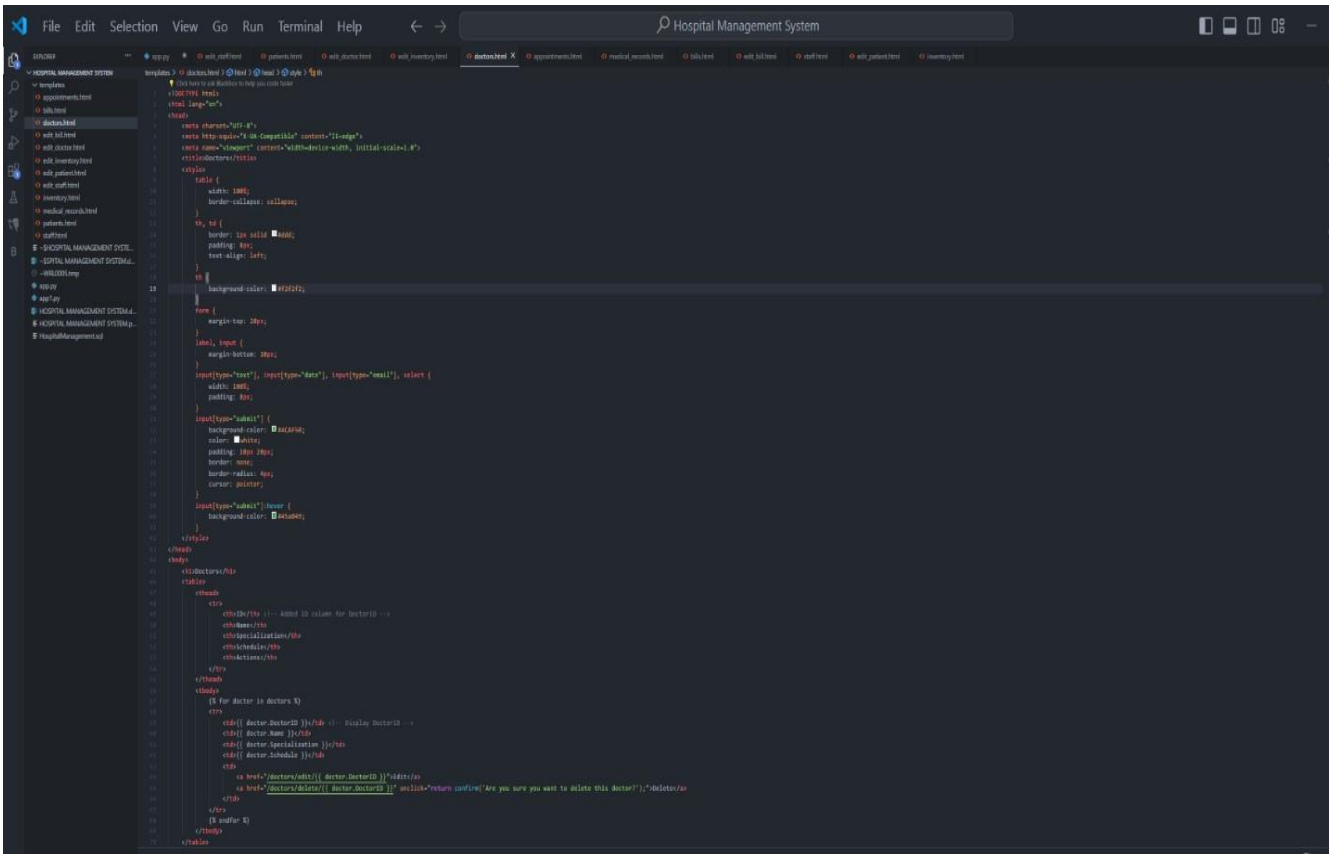


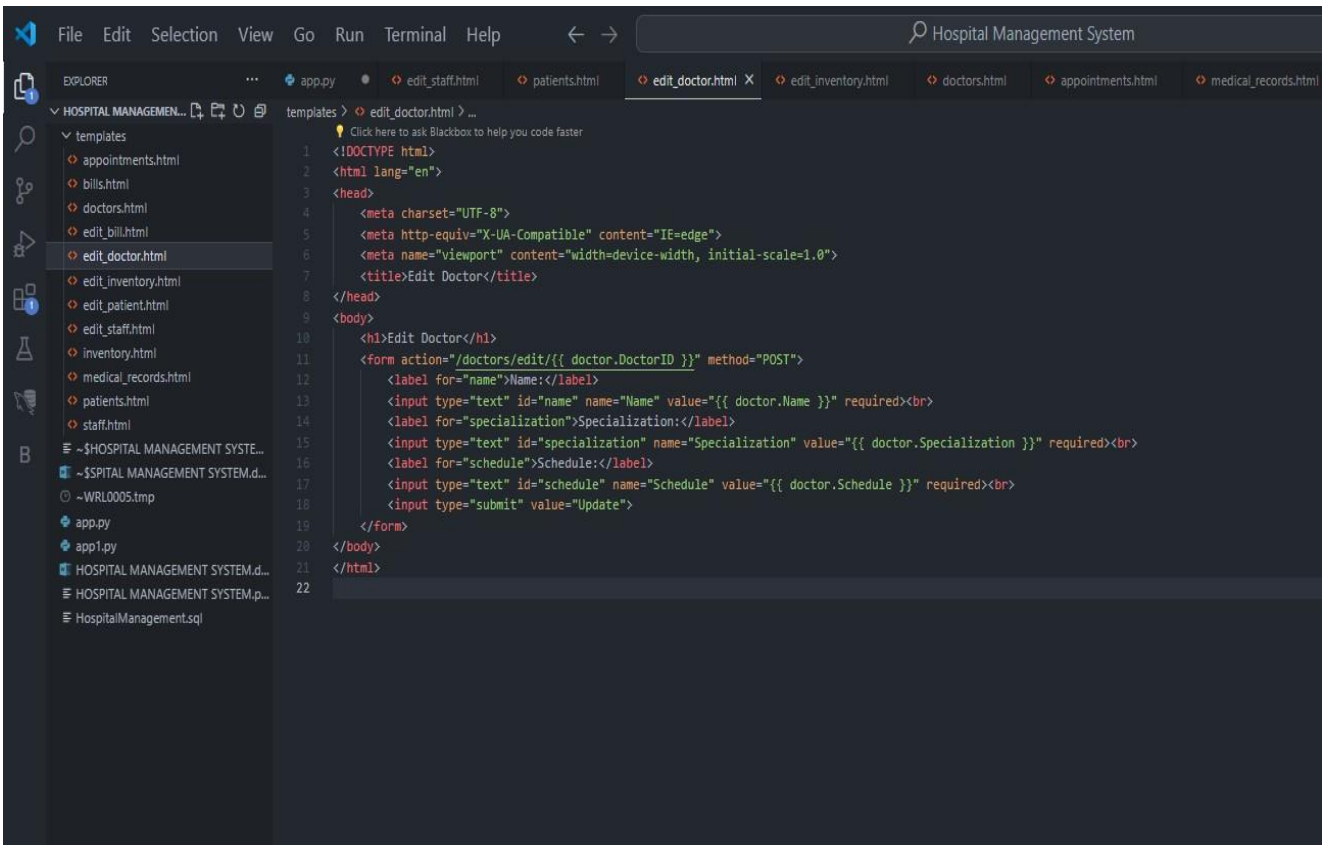
```
<html lang="en">
<body>
    <table>
        <tbody>
            {% endfor %}
        </tbody>
    </table>
    <h2>Add New Bill</h2>
    <form action="/bills" method="POST">
        <label for="patient_id">Patient ID:</label>
        <input type="text" id="patient_id" name="PatientID" required><br>
        <label for="date">Date:</label>
        <input type="date" id="date" name="Date" required><br>
        <label for="amount">Amount:</label>
        <input type="text" id="amount" name="Amount" required><br>
        <label for="status">Status:</label>
        <input type="text" id="status" name="Status" required><br>
        <input type="submit" value="Submit">
    </form>
</body>
</html>
```



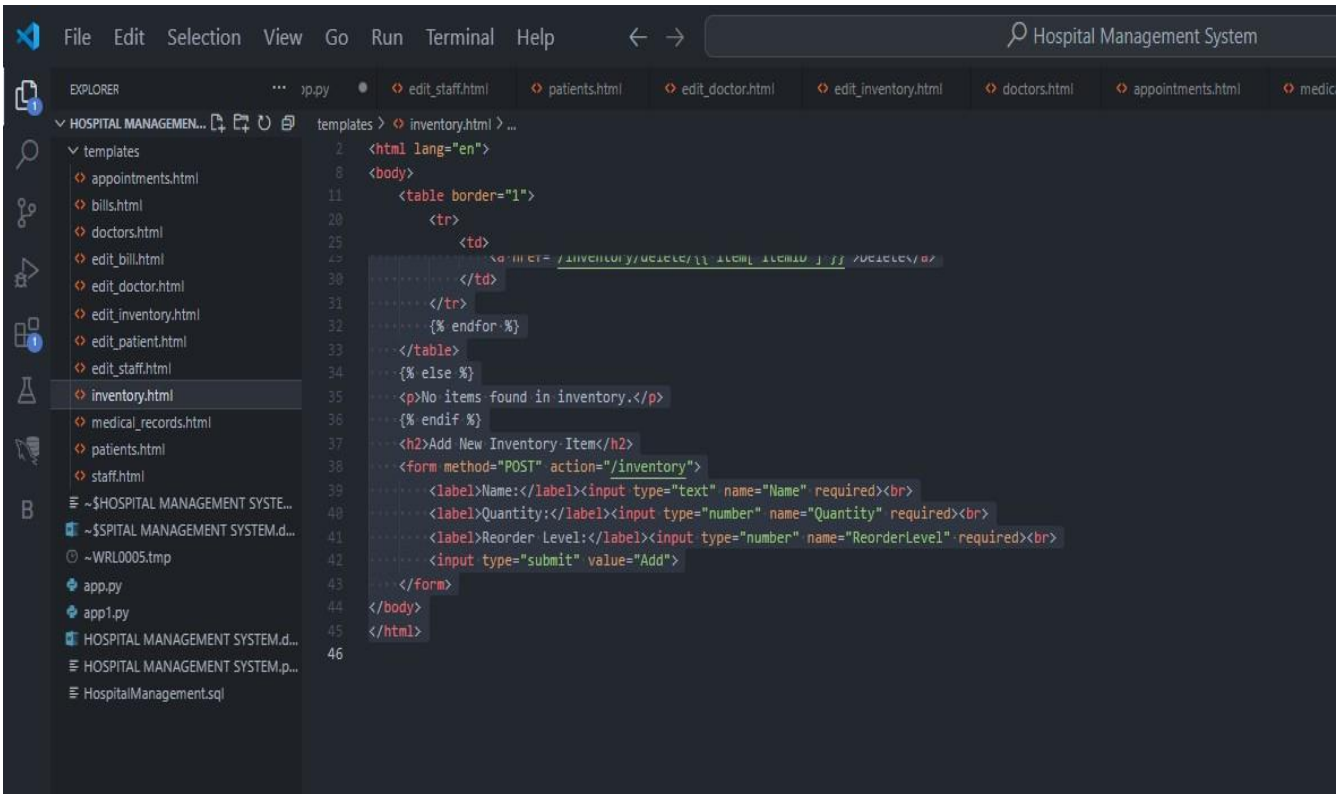
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Edit Bill</title>
</head>
<body>
    <h1>Edit Bill</h1>

    <!-- Display flash messages -->
    {% with messages = get_flashed_messages(with_categories=true) %}
        {% if messages %}
            <ul>
            {% for category, message in messages %}
                <li class="{{ category }}">{{ message }}</li>
            {% endfor %}
            </ul>
        {% endif %}
    {% endwith %}

    <!-- Bill editing form -->
    <form action="/bills/edit/{{ bill.BillID }}" method="post">
        <label for="patient_id">Patient ID:</label>
        <input type="text" id="patient_id" name="PatientID" value="{{ bill.PatientID }}" required>

        <label for="date">Date:</label>
        <input type="date" id="date" name="Date" value="{{ bill.Date }}" required>

        <label for="amount">Amount:</label>
        <input type="number" id="amount" name="Amount" value="{{ bill.Amount }}" required>

        <label for="status">Status:</label>
        <input type="text" id="status" name="Status" value="{{ bill.Status }}" required>

        <button type="submit">Update Bill</button>
    </form>

    <a href="/bills">Back to Bills</a>
</body>
</html>
```

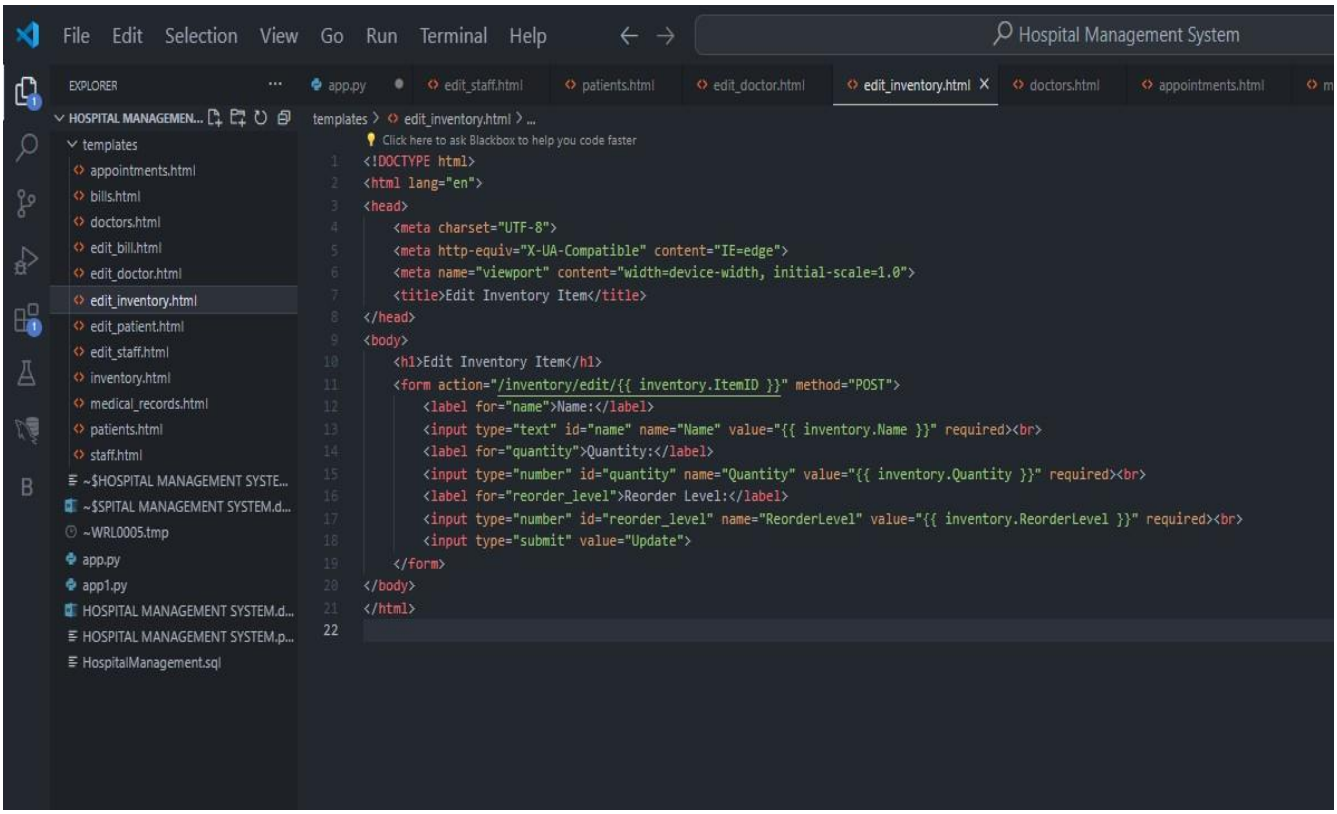
Below is the screenshot of the **Doctors page and Edit Doctors** HTML codes that makes CRUDoperations on the managing doctors side of the HMS.



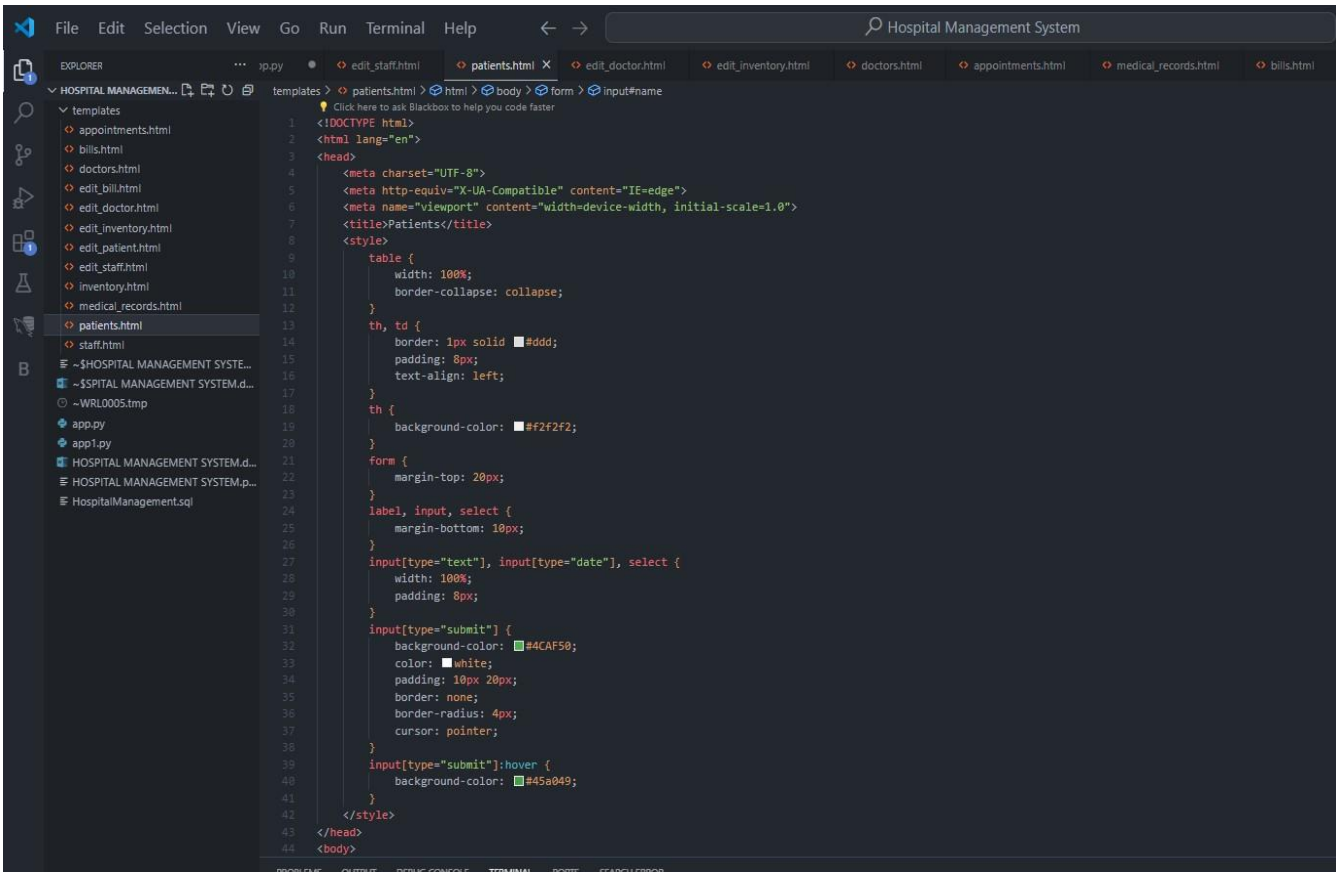


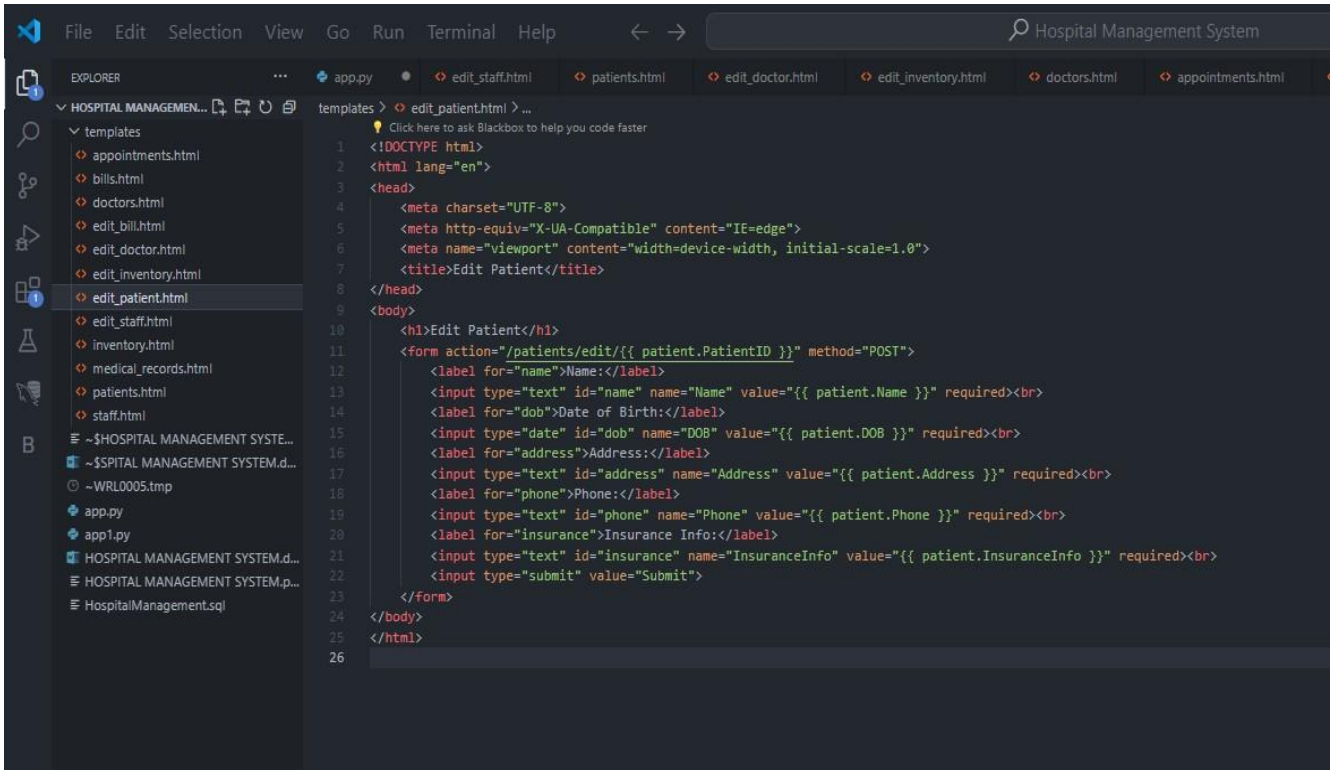
Below is the screenshot of the **Inventory and Edit Inventory** page HTML codes that makes CRUD operations on the managing inventory side of the HMS.



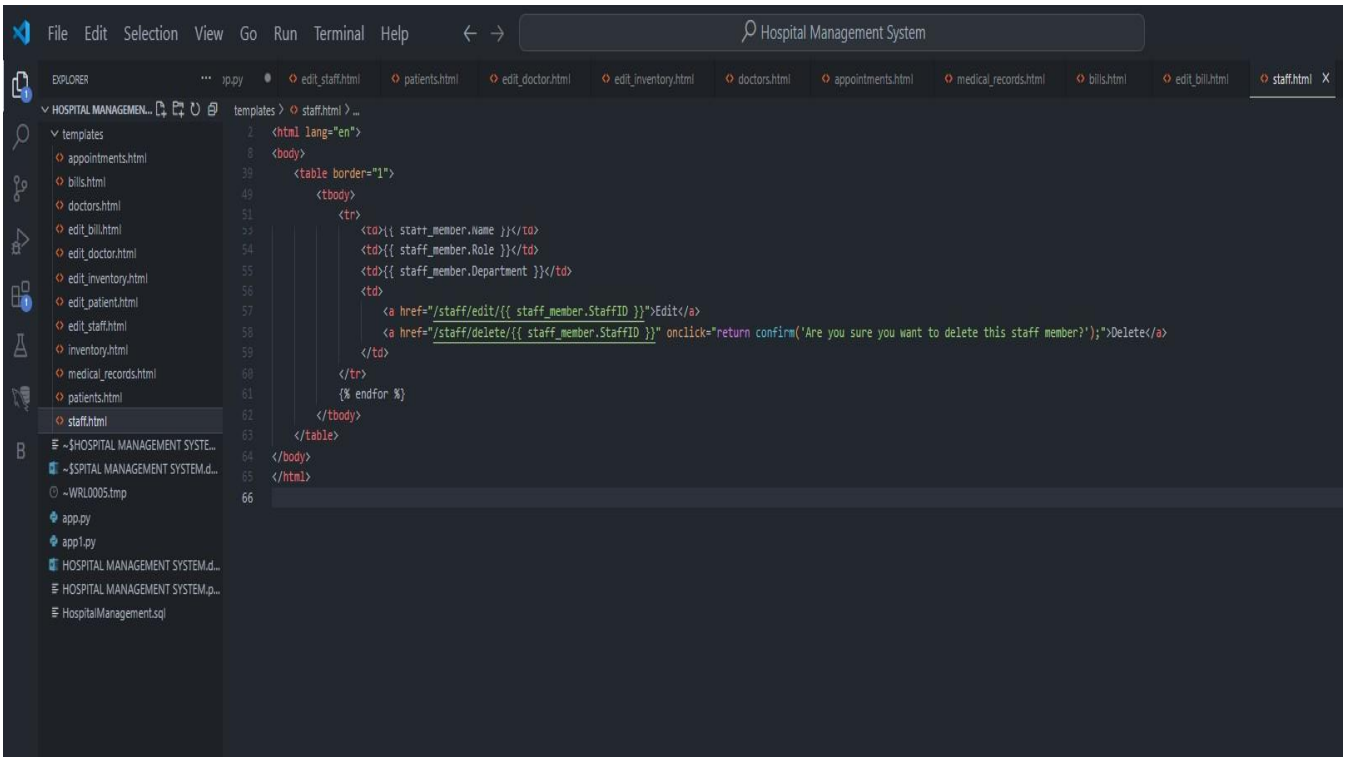


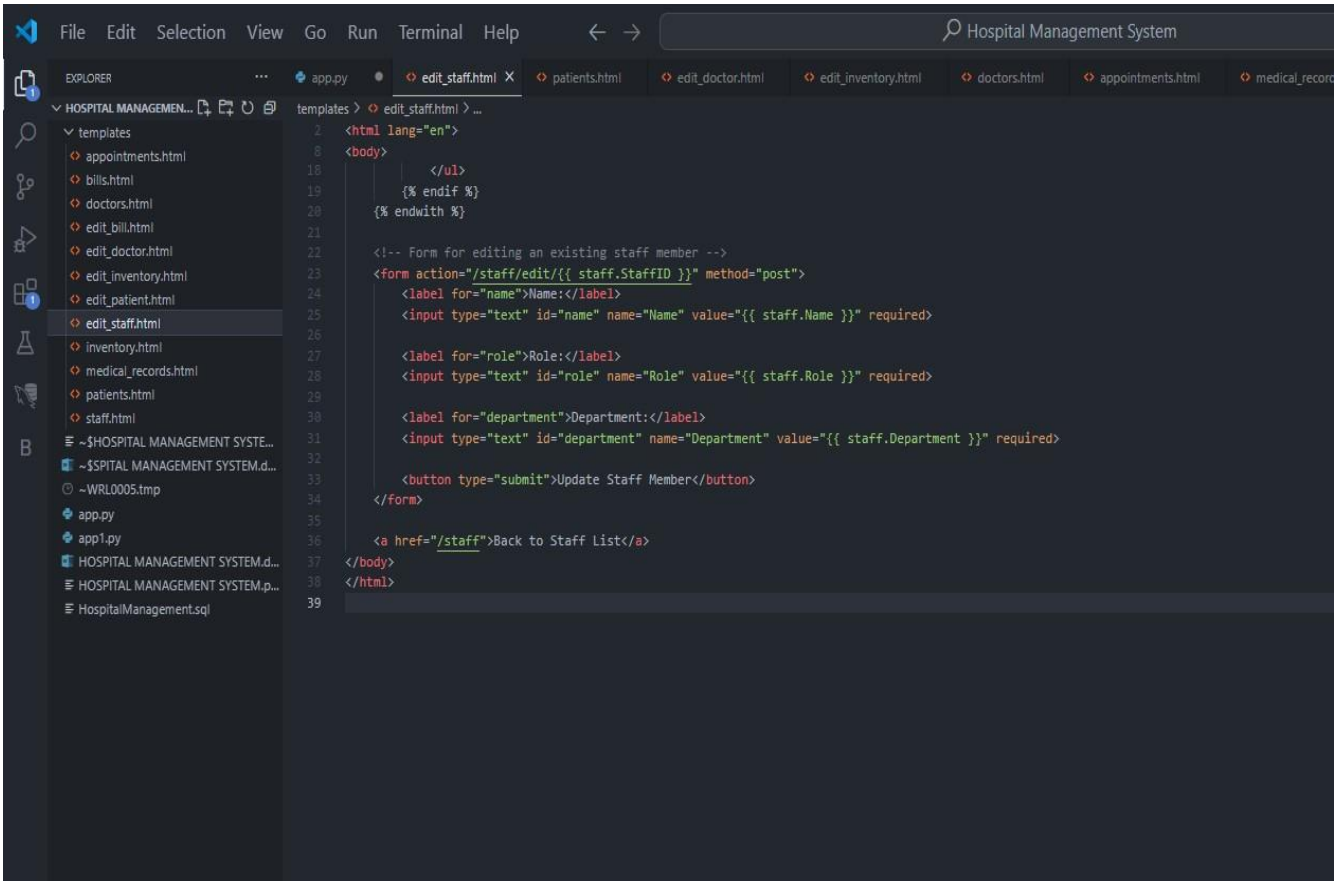
Below is the screenshot of the **Patient and edit Patients page** HTML codes that makes CRUDoperations on the managing patients' side of the HMS.



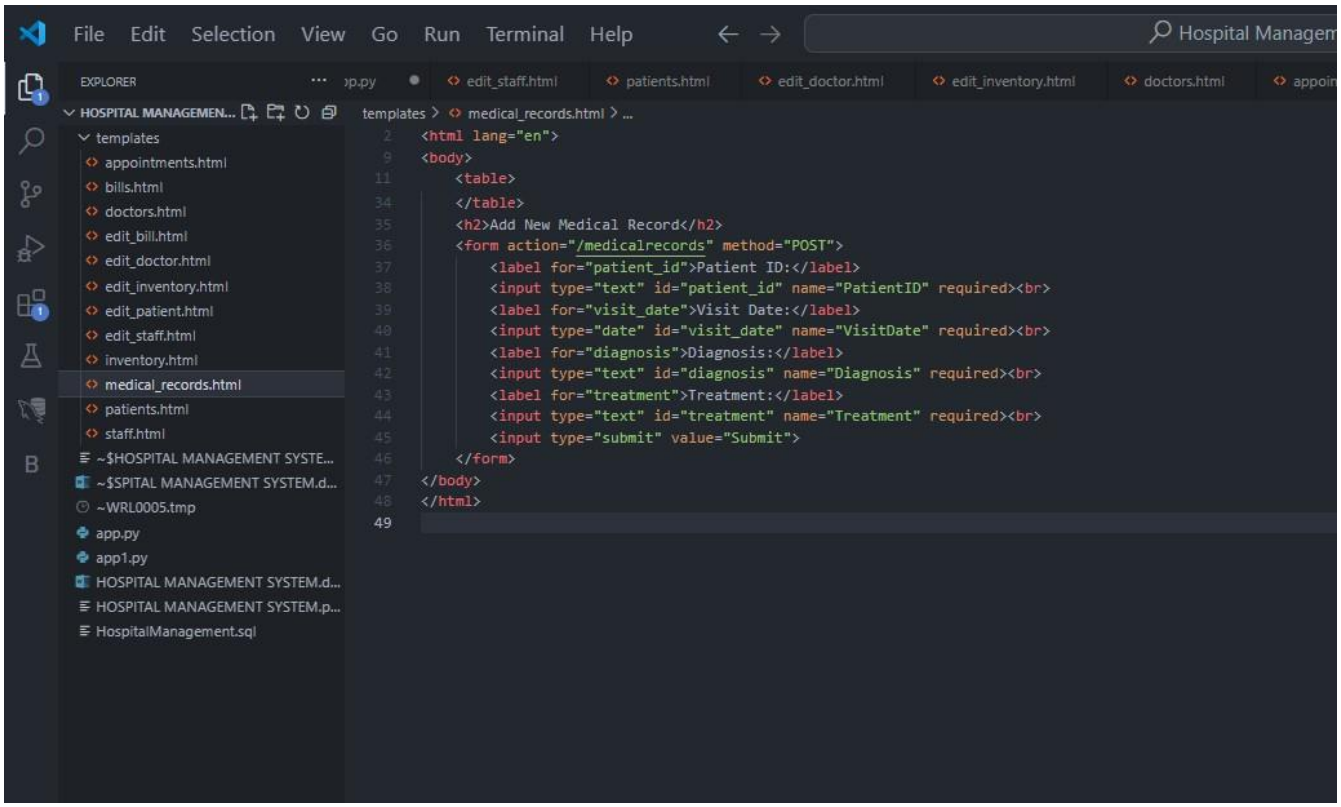


Below is the screenshot of the **Staff page and edit Staff** HTML codes that makes CRUDoperations on the managing Staff side of the HMS.



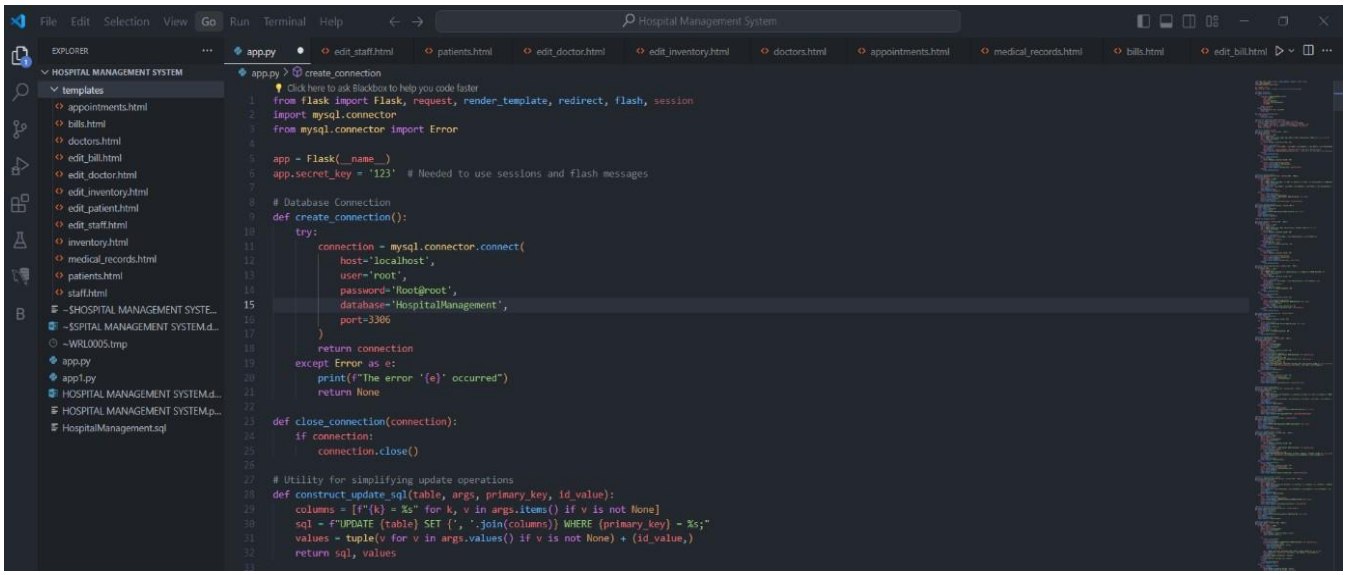


Below is the screenshot of the **medical records** page HTML codes that makes CRUD operationson the managing medical records side of the HMS.



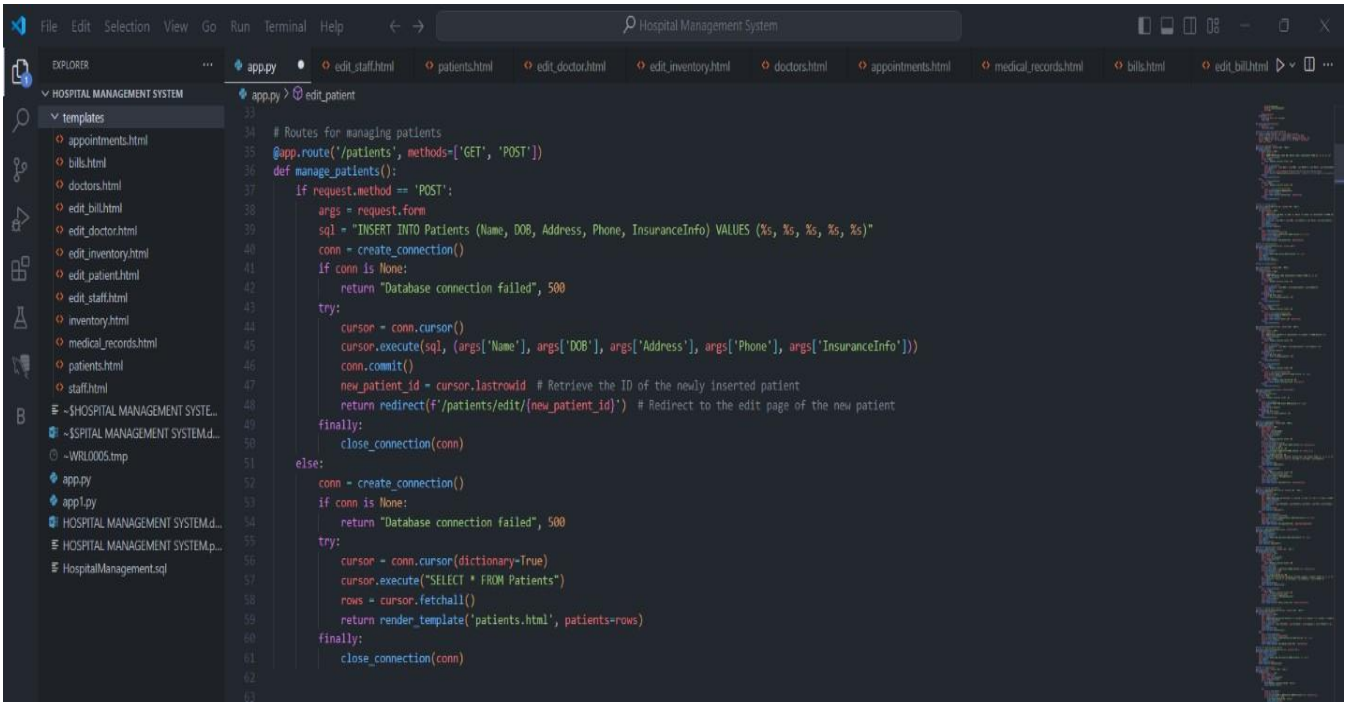
BACKEND CODES

As seen in the screenshot below, the database connection can be seen as connected to the **local host with the user name as "user" and password as "Root@root" also database connected to is "Hospital Management" with the port id as "3306".** As the database is on thesame computer as the backend codes, the connection host is mentioned as **"Localhost".**

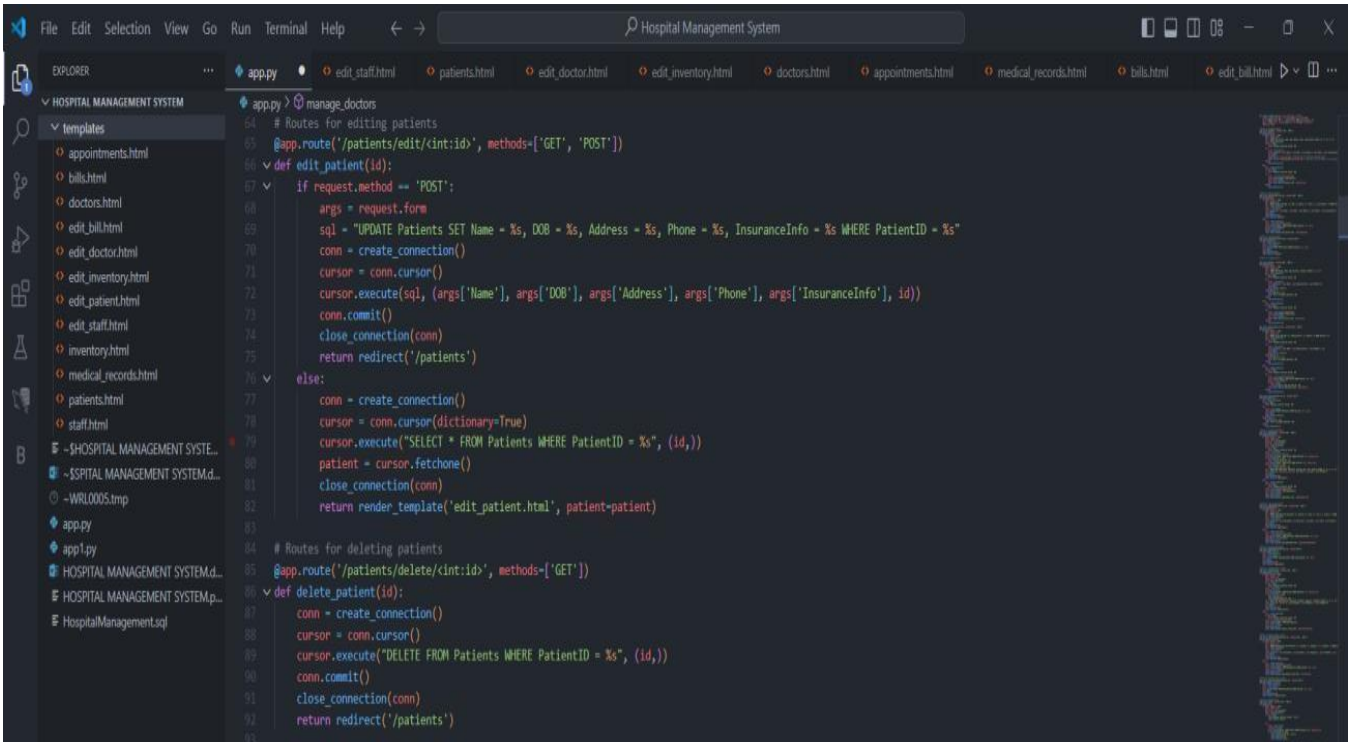


Also, can be seen in the screenshot above, only the update codes is given in the backend so asto give the update commands from the backend as the database was created explicitly as mentioned above in the database codes above.

Below are the codes for managing the patients and the Rest codes for GET and POST requests, also includes error handling codes and validation of the details entered in the frontend for thepatients. The codes also include the **EDIT codes** to perform the CRUD operations.

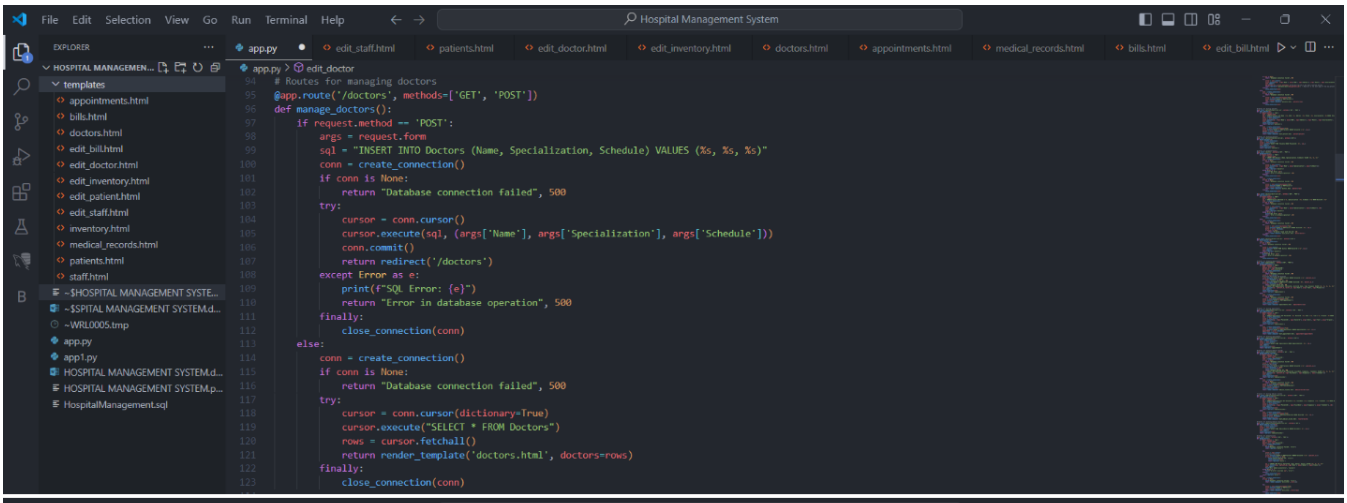


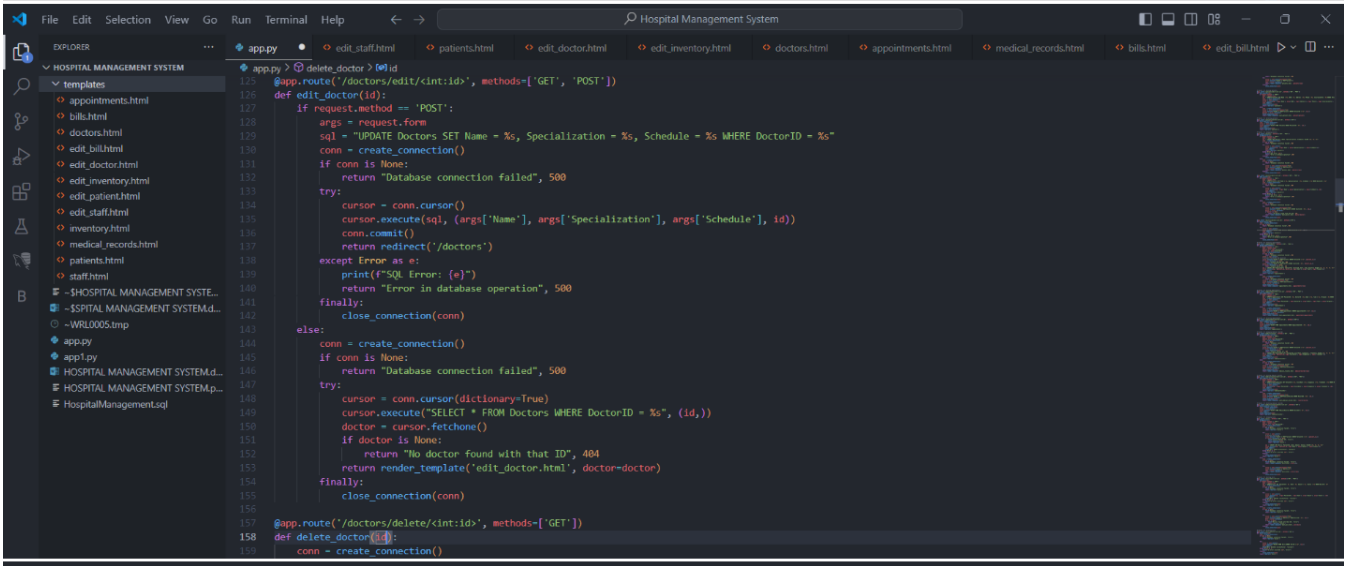
```python
# Routes for managing patients
@app.route('/patients', methods=['GET', 'POST'])
def manage_patients():
    if request.method == 'POST':
        args = request.form
        sql = "INSERT INTO Patients (Name, DOB, Address, Phone, InsuranceInfo) VALUES (%s, %s, %s, %s, %s)"
        conn = create_connection()
        if conn is None:
            return "Database connection failed", 500
        try:
            cursor = conn.cursor()
            cursor.execute(sql, (args['Name'], args['DOB'], args['Address'], args['Phone'], args['InsuranceInfo']))
            conn.commit()
            new_patient_id = cursor.lastrowid  # Retrieve the ID of the newly inserted patient
            return redirect(f'/patients/edit/{new_patient_id}')  # Redirect to the edit page of the new patient
        finally:
            close_connection(conn)
    else:
        conn = create_connection()
        if conn is None:
            return "Database connection failed", 500
        try:
            cursor = conn.cursor(dictionary=True)
            cursor.execute("SELECT * FROM Patients")
            rows = cursor.fetchall()
            return render_template('patients.html', patients=rows)
        finally:
            close_connection(conn)
```

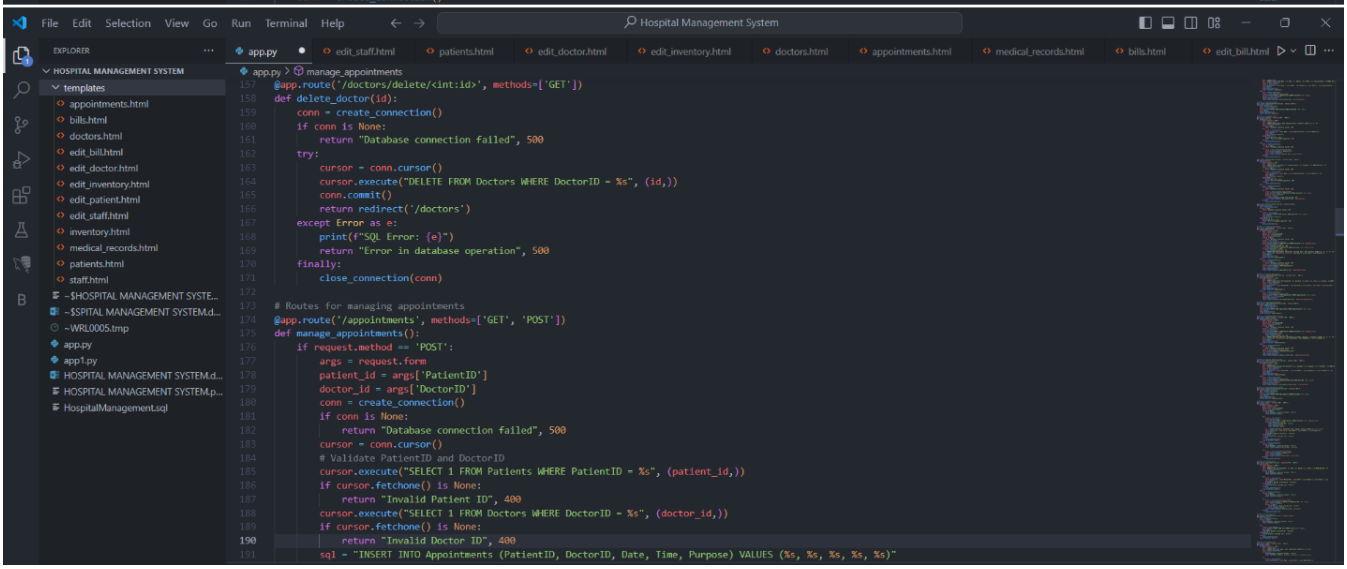


```python
# Routes for editing patients
@app.route('/patients/edit/<int:id>', methods=['GET', 'POST'])
def edit_patient(id):
    if request.method == 'POST':
        args = request.form
        sql = "UPDATE Patients SET Name = %s, DOB = %s, Address = %s, Phone = %s, InsuranceInfo = %s WHERE PatientID = %s"
        conn = create_connection()
        cursor = conn.cursor()
        cursor.execute(sql, (args['Name'], args['DOB'], args['Address'], args['Phone'], args['InsuranceInfo'], id))
        conn.commit()
        close_connection(conn)
        return redirect('/patients')
    else:
        conn = create_connection()
        cursor = conn.cursor(dictionary=True)
        cursor.execute("SELECT * FROM Patients WHERE PatientID = %s", (id,))
        patient = cursor.fetchone()
        close_connection(conn)
        return render_template('edit_patient.html', patient=patient)

# Routes for deleting patients
@app.route('/patients/delete/<int:id>', methods=['GET'])
def delete_patient(id):
    conn = create_connection()
    cursor = conn.cursor()
    cursor.execute("DELETE FROM Patients WHERE PatientID = %s", (id,))
    conn.commit()
    close_connection(conn)
    return redirect('/patients')
```

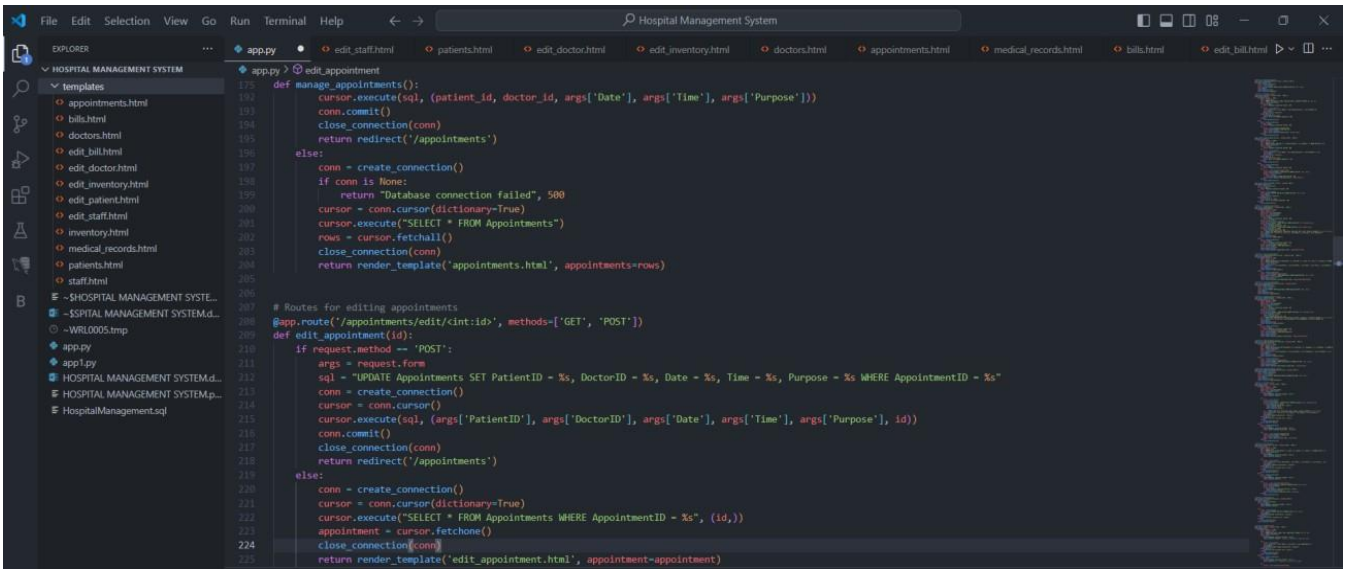
Below are the codes for managing the **doctors** and the Rest codes for GET and POST requests, also includes error handling codes and validation of the details entered in the frontend for the doctors. The codes also include the **EDIT codes** to perform the CRUD operations.



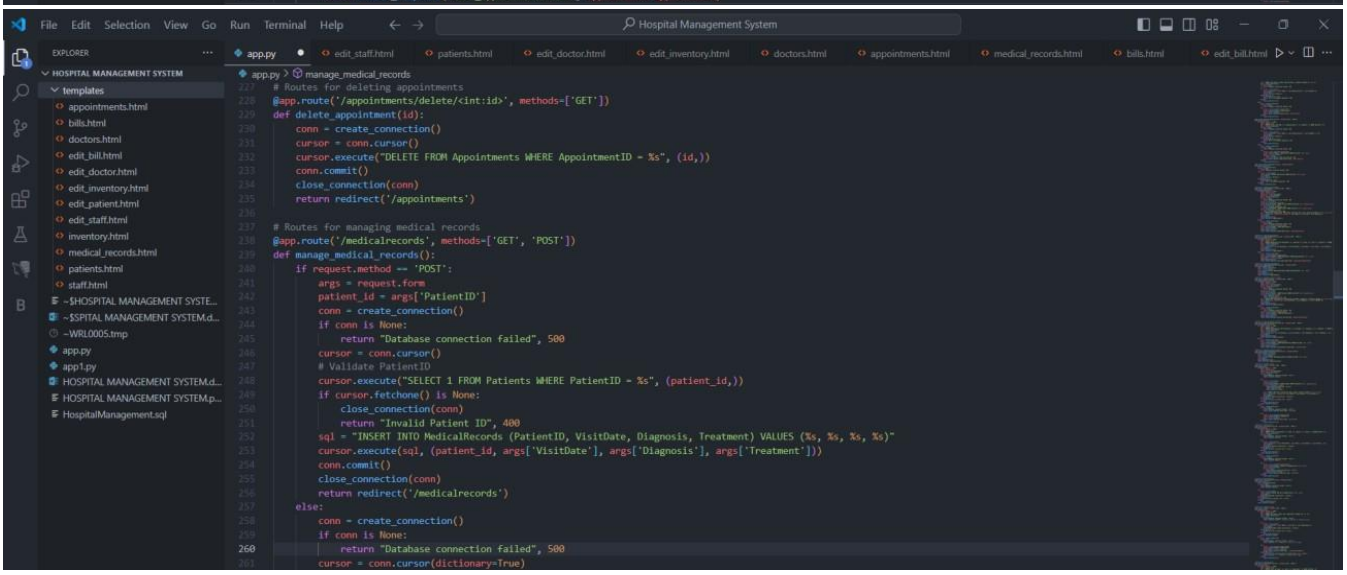




Below are the codes for managing the **Appointments** and the Rest codes for GET and POST requests, also includes error handling codes and validation of the details entered in the frontend for the appointments. The codes also include the **EDIT codes** to perform the CRUDoperations.



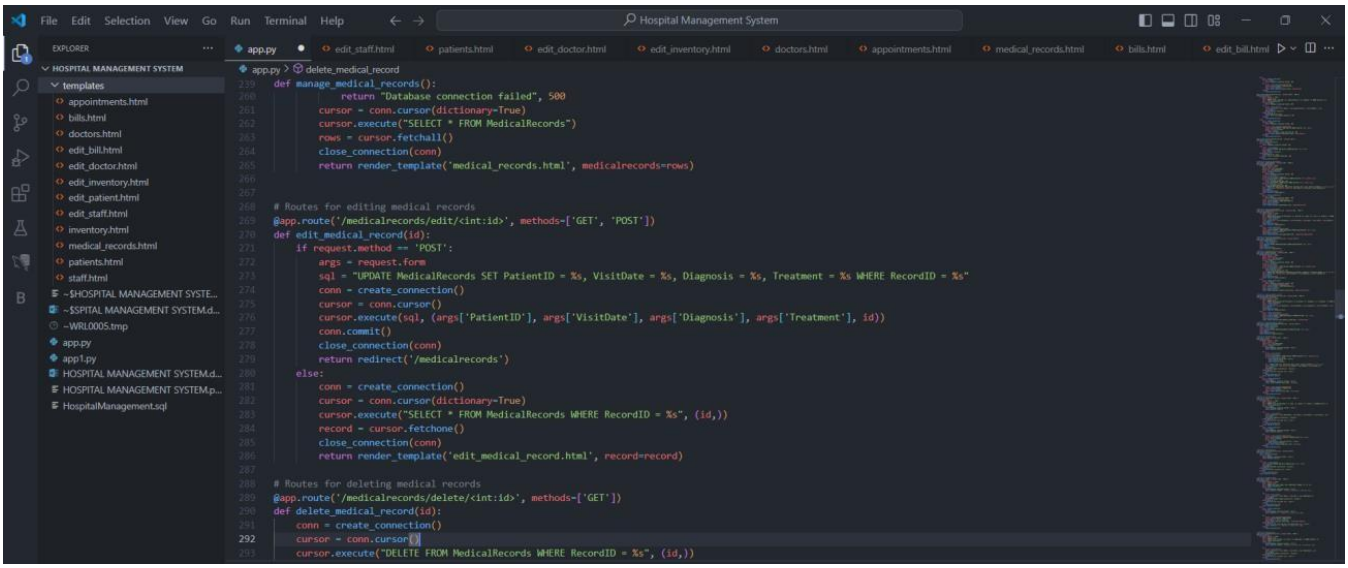
```python
def manage_appointments():
        cursor.execute(sql, (patient_id, doctor_id, args['Date'], args['Time'], args['Purpose']))
        conn.commit()
        close_connection(conn)
        return redirect('/appointments')
    else:
        conn = create_connection()
        if conn is None:
            return "Database connection failed", 500
        cursor = conn.cursor(dictionary=True)
        cursor.execute("SELECT * FROM Appointments")
        rows = cursor.fetchall()
        close_connection(conn)
        return render_template('appointments.html', appointments=rows)


# Routes for editing appointments
@app.route('/appointments/edit/<int:id>', methods=['GET', 'POST'])
def edit_appointment(id):
    if request.method == 'POST':
        args = request.form
        sql = "UPDATE Appointments SET PatientID = %s, DoctorID = %s, Date = %s, Time = %s, Purpose = %s WHERE AppointmentID = %s"
        conn = create_connection()
        cursor = conn.cursor()
        cursor.execute(sql, (args['PatientID'], args['DoctorID'], args['Date'], args['Time'], args['Purpose'], id))
        conn.commit()
        close_connection(conn)
        return redirect('/appointments')
    else:
        conn = create_connection()
        cursor = conn.cursor(dictionary=True)
        cursor.execute("SELECT * FROM Appointments WHERE AppointmentID = %s", (id,))
        appointment = cursor.fetchone()
        close_connection(conn)
        return render_template('edit_appointment.html', appointment=appointment)
```



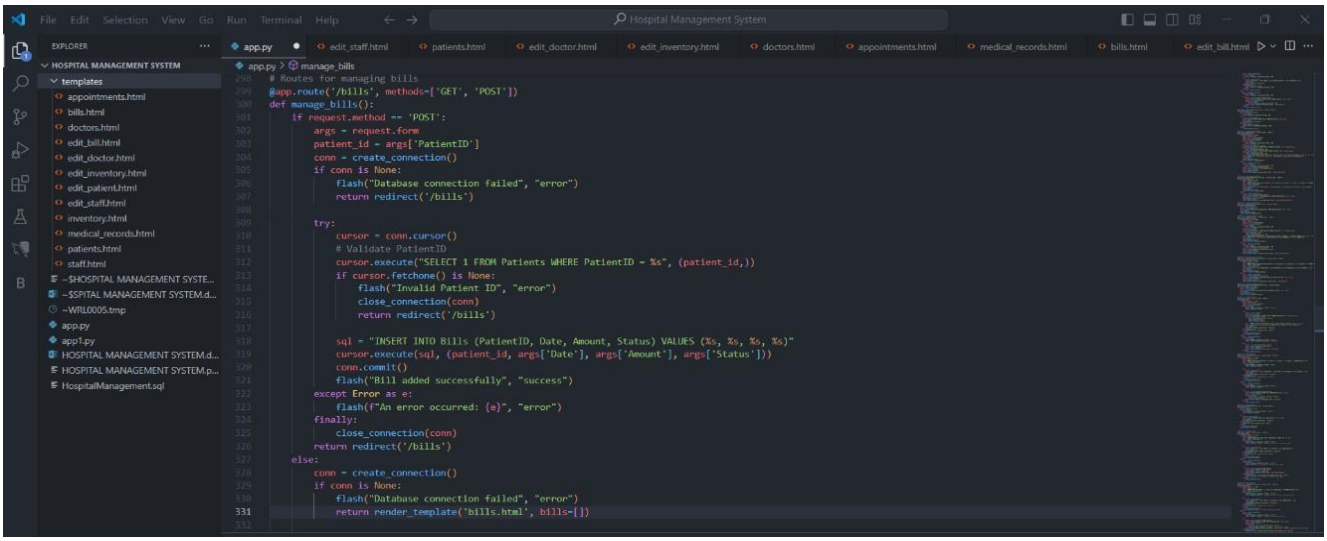
```python
# Routes for deleting appointments
@app.route('/appointments/delete/<int:id>', methods=['GET'])
def delete_appointment(id):
    conn = create_connection()
    cursor = conn.cursor()
    cursor.execute("DELETE FROM Appointments WHERE AppointmentID = %s", (id,))
    conn.commit()
    close_connection(conn)
    return redirect('/appointments')

# Routes for managing medical records
@app.route('/medicalrecords', methods=['GET', 'POST'])
def manage_medical_records():
    if request.method == 'POST':
        args = request.form
        patient_id = args['PatientID']
        conn = create_connection()
        if conn is None:
            return "Database connection failed", 500
        cursor = conn.cursor()
        # Validate PatientID
        cursor.execute("SELECT 1 FROM Patients WHERE PatientID = %s", (patient_id,))
        if cursor.fetchone() is None:
            close_connection(conn)
            return "Invalid Patient ID", 400
        sql = "INSERT INTO MedicalRecords (PatientID, VisitDate, Diagnosis, Treatment) VALUES (%s, %s, %s, %s)"
        cursor.execute(sql, (patient_id, args['VisitDate'], args['Diagnosis'], args['Treatment']))
        conn.commit()
        close_connection(conn)
        return redirect('/medicalrecords')
    else:
        conn = create_connection()
        if conn is None:
            return "Database connection failed", 500
        cursor = conn.cursor(dictionary=True)
```

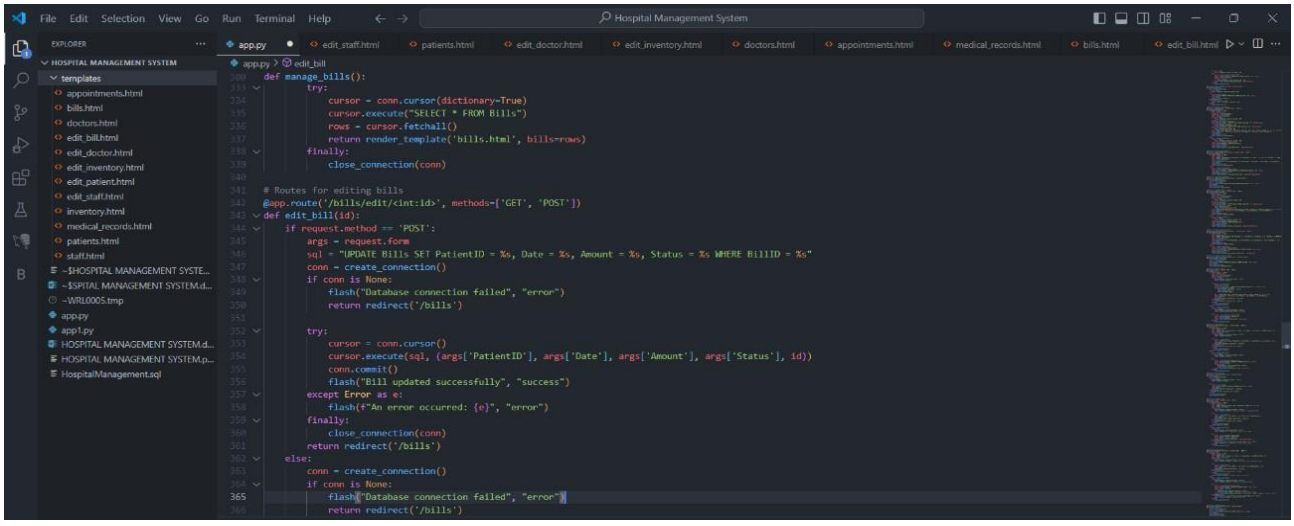
Above are the codes for managing the **medical records** and the Rest codes for GET and POSTrequests, also includes error handling codes and validation of the details entered in the frontend for the medical records.

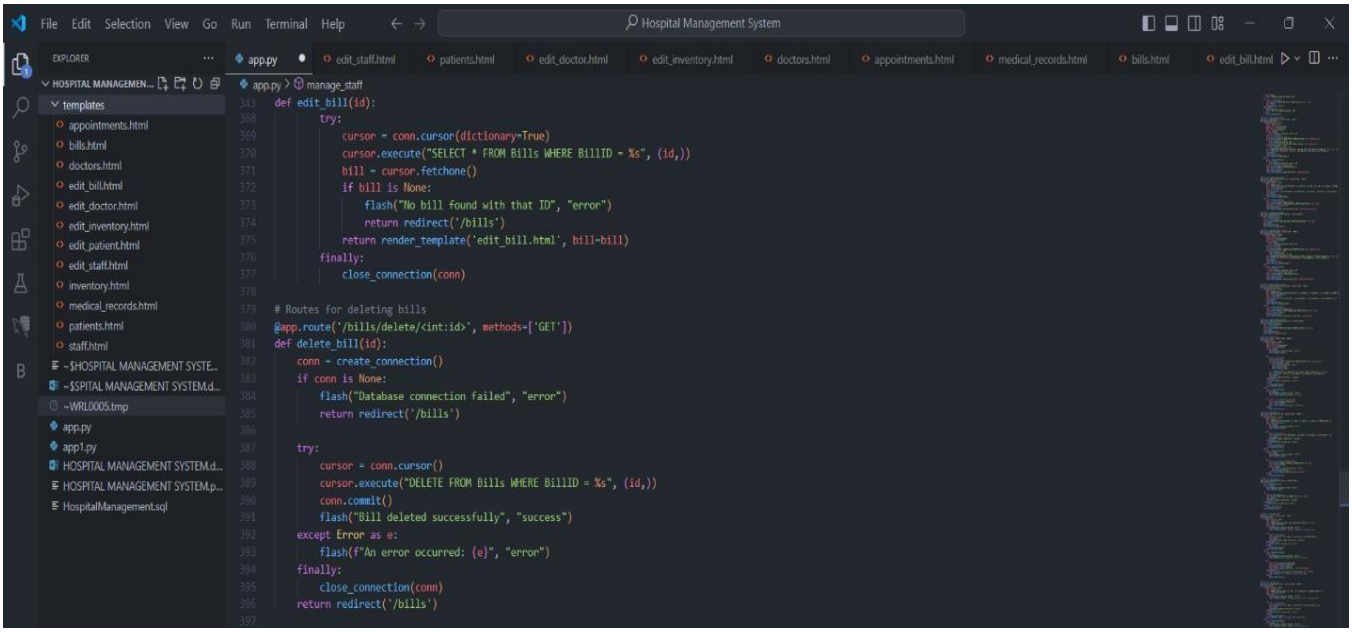
The codes below also include the **EDIT codes** to perform the CRUD operations.



Below are the codes for managing the **bills** and the Rest codes for GET and POST requests, also includes error handling codes and validation of the details entered in the frontend for the bills. The codes also include the **EDIT codes** to perform the CRUD operations.

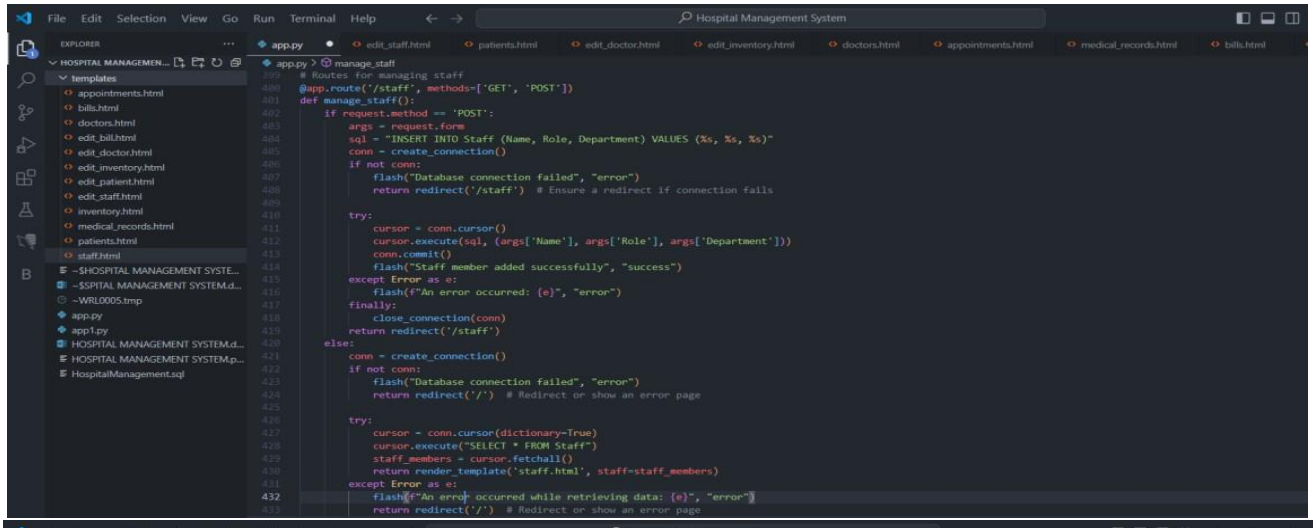




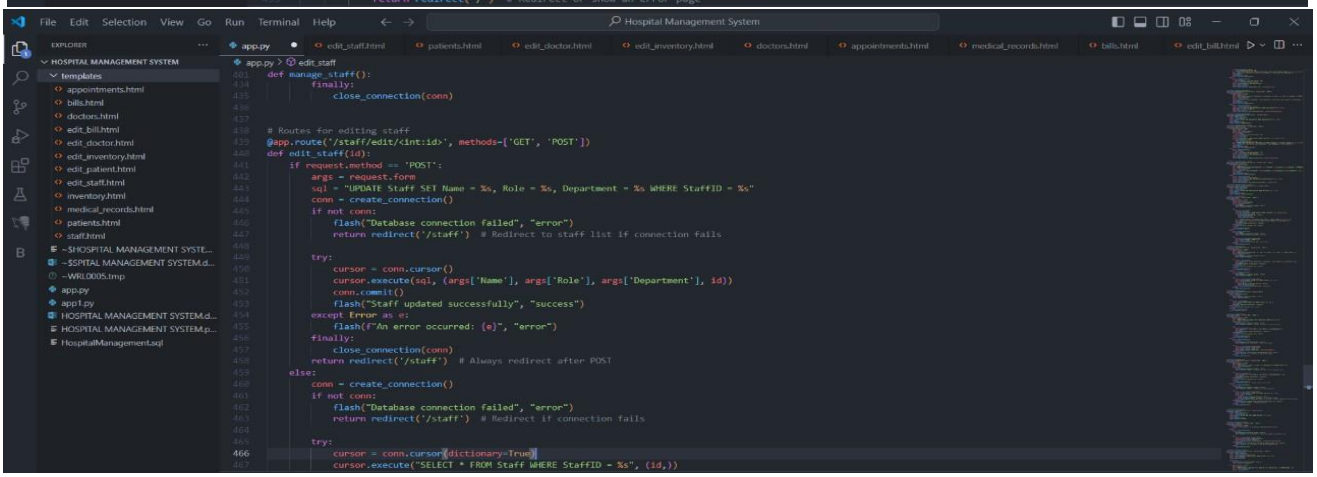


```
def edit_bill(id):
        try:
            cursor = conn.cursor(dictionary=True)
            cursor.execute("SELECT * FROM Bills WHERE BillID = %s", (id,))
            bill = cursor.fetchone()
            if bill is None:
                flash("No bill found with that ID", "error")
                return redirect('/bills')
            return render_template('edit_bill.html', bill=bill)
        finally:
            close_connection(conn)

# Routes for deleting bills
@app.route('/bills/delete/<int:id>', methods=['GET'])
def delete_bill(id):
    conn = create_connection()
    if conn is None:
        flash("Database connection failed", "error")
        return redirect('/bills')

    try:
        cursor = conn.cursor()
        cursor.execute("DELETE FROM Bills WHERE BillID = %s", (id,))
        conn.commit()
        flash("Bill deleted successfully", "success")
    except Error as e:
        flash(f"An error occurred: {e}", "error")
    finally:
        close_connection(conn)
    return redirect('/bills')
```

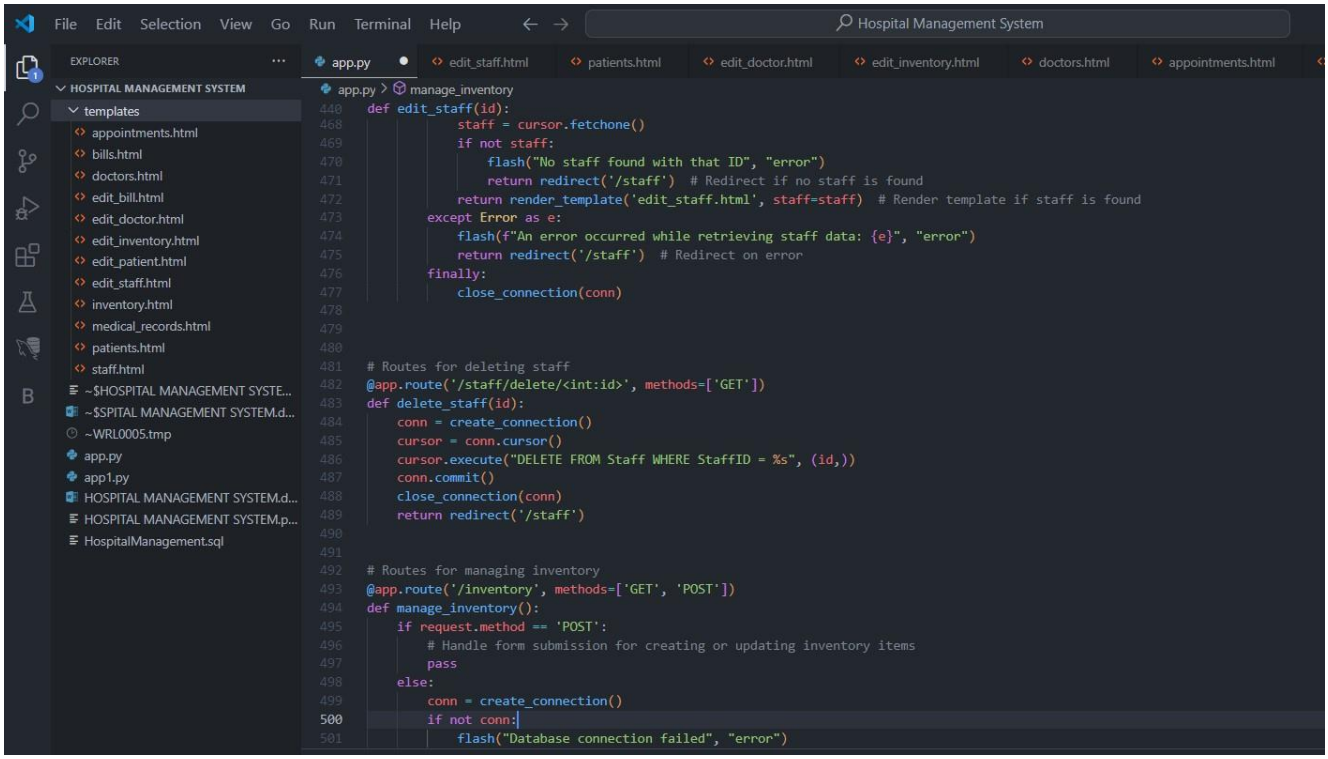
Below are the codes for managing the **Staffs** and the Rest codes for GET and POST requests, also includes error handling codes and validation of the details entered in the frontend for theStaffs. The codes also include the **EDIT codes** to perform the CRUD operations.



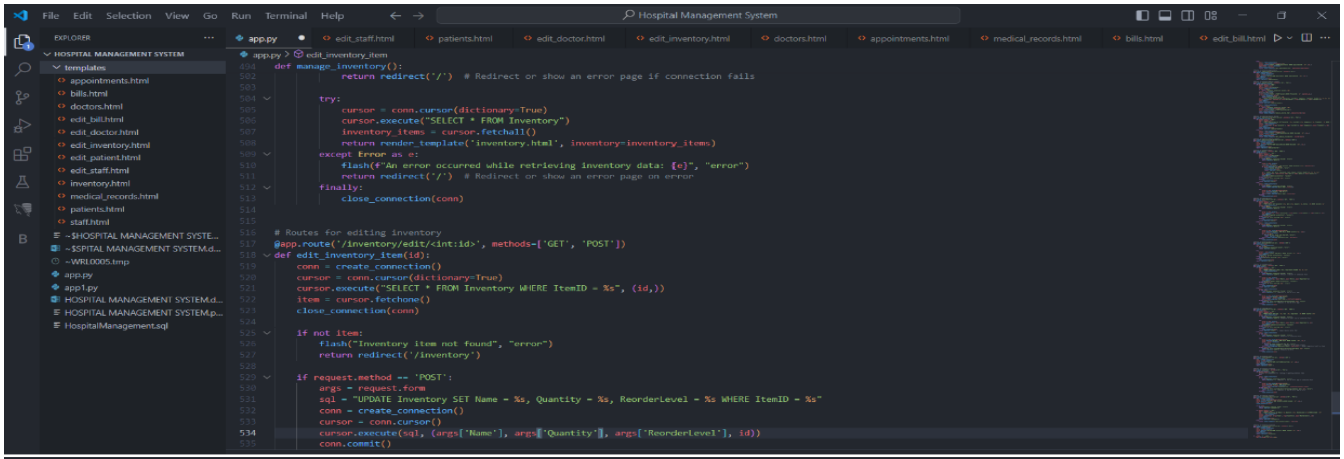
```
# Routes for managing staff
@app.route('/staff', methods=['GET', 'POST'])
def manage_staff():
    if request.method == 'POST':
        args = request.form
        sql = "INSERT INTO Staff (Name, Role, Department) VALUES (%s, %s, %s)"
        conn = create_connection()
        if not conn:
            flash("Database connection failed", "error")
            return redirect('/staff')  # Ensure a redirect if connection fails

        try:
            cursor = conn.cursor()
            cursor.execute(sql, (args['Name'], args['Role'], args['Department']))
            conn.commit()
            flash("Staff member added successfully", "success")
        except Error as e:
            flash(f"An error occurred: {e}", "error")
        finally:
            close_connection(conn)
        return redirect('/staff')
    else:
        conn = create_connection()
        if not conn:
            flash("Database connection failed", "error")
            return redirect('/')  # Redirect or show an error page

        try:
            cursor = conn.cursor(dictionary=True)
            cursor.execute("SELECT * FROM Staff")
            staff_members = cursor.fetchall()
            return render_template('staff.html', staff=staff_members)
        except Error as e:
            flash(f"An error occurred while retrieving data: {e}", "error")
            return redirect('/')  # Redirect or show an error page
```

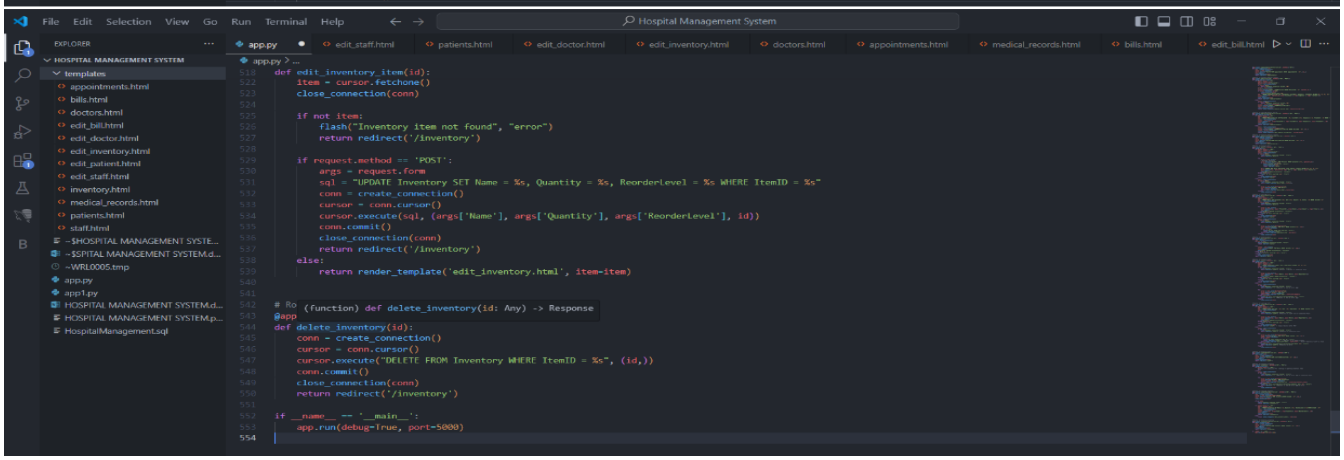


```
def manage_staff():
        finally:
            close_connection(conn)


# Routes for editing staff
@app.route('/staff/edit/<int:id>', methods=['GET', 'POST'])
def edit_staff(id):
    if request.method == 'POST':
        args = request.form
        sql = "UPDATE Staff SET Name = %s, Role = %s, Department = %s WHERE StaffID = %s"
        conn = create_connection()
        if not conn:
            flash("Database connection failed", "error")
            return redirect('/staff')  # Redirect to staff list if connection fails

        try:
            cursor = conn.cursor()
            cursor.execute(sql, (args['Name'], args['Role'], args['Department'], id))
            conn.commit()
            flash("Staff updated successfully", "success")
        except Error as e:
            flash(f"An error occurred: {e}", "error")
        finally:
            close_connection(conn)
        return redirect('/staff')  # Always redirect after POST
    else:
        conn = create_connection()
        if not conn:
            flash("Database connection failed", "error")
            return redirect('/staff')  # Redirect if connection fails

        try:
            cursor = conn.cursor(dictionary=True)
            cursor.execute("SELECT * FROM Staff WHERE StaffID = %s", (id,))
```



Below are the codes for managing the **Inventory** and the Rest codes for GET and POST requests,also includes error handling codes and validation of the details entered in the frontend for the Inventory. The codes also include **EDIT codes** to perform CRUD operations.





CONCLUSION SUMMARY

In our journey to develop a Hospital Management System (HMS), we've created a comprehensive database-driven application that revolutionizes how healthcare facilities manage their operations. Our HMS encompasses a relational database schema that efficiently organizes and interconnects various entities critical for hospital operations. These entities include Patients, Doctors, Appointments, Medical Records, Bills, Staff, and Inventory. Each entity is defined with specific attributes, ensuring a detailed and structured approach to data management.

The application provides essential functionalities:

1. Patient Management: Registering patients, updating profiles, and managing medical histories.
2. Appointment Scheduling: Facilitating seamless booking, rescheduling, and cancellation of appointments.
3. Medical Records Management: Creating and accessing detailed medical records securely.
4. Billing and Insurance Processing: Automating bill generation and managing insurance claims.
5. Staff and Schedule Management: Maintaining records of staff roles, departments, and schedules.
6. Inventory Management: Tracking medical supplies and medications.

Our database structure ensures efficient relationships between entities, optimizing hospital workflows and ensuring comprehensive patient care. We gained hands-on experience on below:

1. Database Design: We learned the importance of a well-structured relational database schema in managing complex data relationships. Designing entities, attributes, and relationships helped us understand how data flows within an application.

2. Application Development: Building an application from scratch taught us valuable lessons in coding, including creating frontend interfaces and backend functionalities. We gained hands-on experience in CRUD (Create, Read, Update, Delete) operations.

3. User Experience (UX): We emphasized user-friendly interfaces to ensure easy navigation for hospital staff. Integrating functionalities into a cohesive platform enhanced user experience and operational efficiency.

4. Data Security and Privacy: Managing patient data required strict adherence to security and privacy standards. We implemented role-based access control and ensured data encryption to maintain confidentiality.

5. Error Handling and Validation: Developing robust error handling mechanisms and data validation processes was crucial for maintaining data integrity and application reliability.

6. Collaboration: Working in a group taught us the importance of effective communication and collaboration. Dividing tasks, sharing responsibilities, and coordinating efforts were key to the project's success.

7. Adaptability: As we encountered challenges and requirements evolved, we learned to adapt our designs and codebase. This flexibility is essential in real-world application development.

In conclusion, our HMS project represents a culmination of database design principles, application development, and teamwork. We've not only created a functional system but also gained valuable insights and skills that will serve us well in our future endeavors in the field of technology and healthcare.

REFERENCES:

1. https://code.visualstudio.com/ - VS STUDIO – For Python Coding
2. https://www.mysql.com/products/workbench/ - My SQL WorkBench – for SQL queries
3. https://palletsprojects.com/p/jinja/ - Jinja2 – For Templating
4. https://flask.palletsprojects.com/en/3.0.x/ - Flask Framework
5. https://www.ibm.com/topics/rest-apis - REST API