INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR
DEPARTMENT OF COMPUTER SCICENCE AND ENGINEERING

Database Management System Laboratory

Mini-Project Report

Hospital Management System Software

Advisor: Prof. Pabitra Mitra, Prof. KS Rao

IIT KHARAGPUR, AUGUST 2023

Member List

Name	Roll No
Yashraj Singh	20CS10079
Rishi Raj	20CS30040
Vikas Vijaykumar Bastewad	20CS10073
Aditya Choudhary	20CS10005
Astitva	20CS30007

Contents

1 Hospital Management System

As the world continues to advance in technology, it is imperative that the healthcare industry follows suit. A Hospital Management System Software can greatly improve the efficiency of healthcare facilities, making patient care more effective and streamlined.

Imagine a hospital where patient records are stored digitally, easily accessible to healthcare professionals. Doctors, nurses, and administrators can access real-time information about a patient's medical history, medications, and treatment plans from anywhere in the hospital, or even remotely.

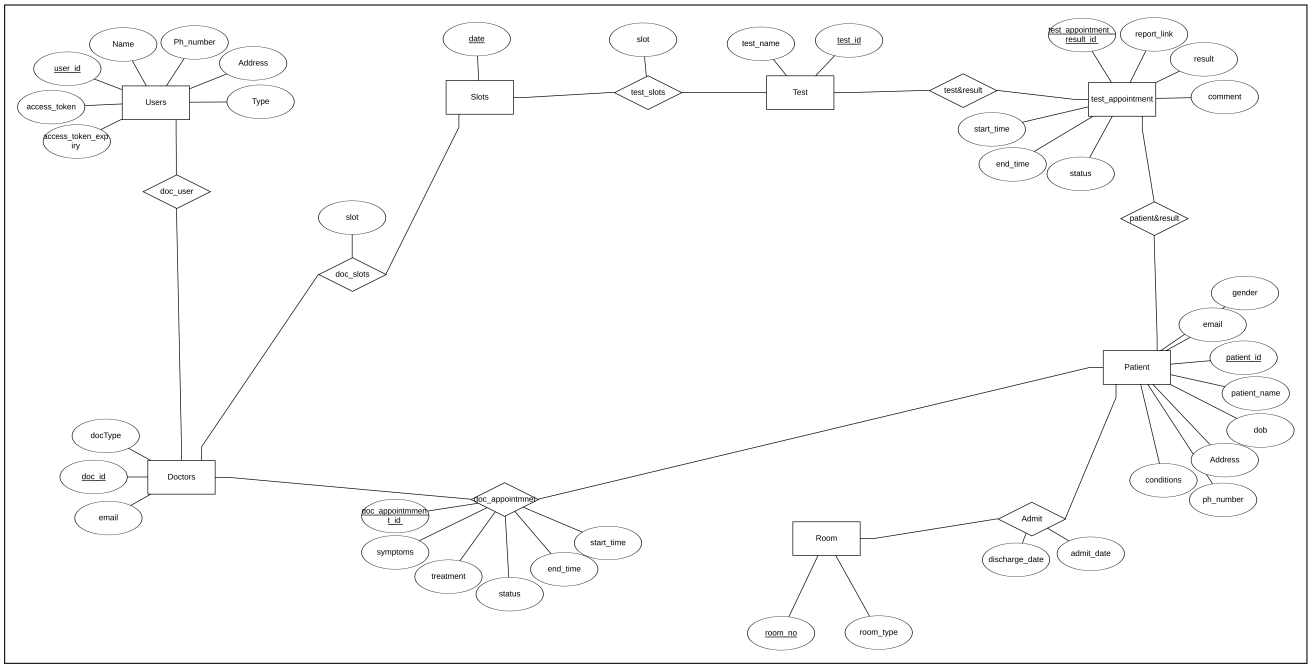
Hospital Management System Software can also improve communication and collaboration between healthcare professionals, reducing the risk of errors and improving patient outcomes. It can automate many administrative tasks, freeing up time for healthcare professionals to focus on patient care.

In short, Hospital Management System Software can transform healthcare delivery, making it more efficient, effective, and patient-centered. The potential benefits are immense, and it is up to healthcare facilities to embrace this technology and use it to provide the best possible care for their patients.

This is the link where our application is hosted: www.azadhospital.com

2 Database Design

The Entity-Relationship Diagram of our Hospital management System is as follows. It keeps in mind all the day-to-day and bonus functionalities which are offered in our Hospital Management System. This design aims to reduce the retrieval time for each query and with no redundancy.



In this ER diagram, we have 7 entities and 7 relationships. When converting this ER Diagram to SQL Schema, we end up with 10 schemas.

The schemas are namely:

- Users

Users	
name	VARCHAR(10)
user_id	VARCHAR(10)
ph_number	VARCHAR(12)
password	VARCHAR(255)
type	CHAR(3)
address	VARCHAR(255)
access_token	VARCHAR(1000)
access_token_expiry	TIMESTAMP

This table is used to store all the people who will use the application, namely, Database Administrators, Front-Desk Operators, Data-Entry Operators and Doctors.

For all these users, we store there name, phone number, address, type and password. We have an additional attribute of user_id which is unique and generated by us using an indigenous algorithm which comes into play when a new user is added in the institution.

Along with all this, we also have two attributes access_token and access_token_expiry which are used to take care of the security policy that a user cannot access any data which he or she is not intended to know. The access tokens are used to enforce the Data Security adn API security policy.

- Doctors

Doctors	
doc_id	VARCHAR(10)
user_id	VARCHAR(10)
docType	VARCHAR(20)
email	VARCHAR(255)

This table stores some additional attributes for users which are of type Ḍoctor. We store the email and docType as additional attributes because we need to email the doctor on a weekly basis and in cases of emergency and we also need to know the speciality of the doctor when booking appointments.

The doc_id is generated by us and is the primary key of the table and user_id is foreign key which references the Users table.

- Room

Room	
room_no	VARCHAR(5)
room_type	VARCHAR(20)

This table stores all the rooms available in the hospital and their type. The room_no attribute here is the primary key.

- Test

Test	
test_id	VARCHAR(10)
test_name	VARCHAR(20)

This table stores the name and ids of all the tests which are made available by the hospital.

It consists of test_id which is generated by us and is the primary key and test_name which stores the name of the test.

- Patients

Patients	
patient_id	VARCHAR(10)
patient_name	VARCHAR(100)
dob	TIMESTAMP
email	VARCHAR(255)
address	VARCHAR(255)
conditions	VARCHAR(255)
gender	VARCHAR(6)
ph_number	VARCHAR(12)

In this table we store all the basic information of the patient including his name, date-of-birth, address, email, phone number, gender and prevailing medical conditions of any.

The patient_id is generated by us when the Front-Desk Operator registers a new patient and it is the primary key.

- Admit

Admit	
patient_id	VARCHAR(10)
room_no	VARCHAR(5)
admit_date	TIMESTAMP
discharge_date	TIMESTAMP

This table stores the information of every admit entry that is made in the hospital.
It stores the patient_id, admit and dicharge date along with the room number in which the patient was admitted.

The patient_id along with the admit_date is the primary key. The patient_id is also a foreign key which references the Patients table.

- test_slots

test_slots	
test_id	VARCHAR(10)
test_date	TIMESTAMP
test_slot	VARCHAR(100)

This table contains the slot for a tests. The primary key here is the test_id and test_date.

The test_slot is a string which stores the slots of a test in a coded format where the encoding scheme is designed by us. For example, the value Í0001200,14001600ín test_slot indicates that the slot is from 10:00 to 12:00 and 14:00 to 16:00.

- doc_slots

doc_slots	
doc_id	VARCHAR(10)
doc_date	TIMESTAMP
doc_slot	VARCHAR(100)

This table contains the slot for a doctor appointments. The primary key here is the doc_id and doc_date. The doc_slot is a string which stores the slots of a appointment in a coded format where the encoding scheme is same as in the test_slot table.

- test_appointment

test_appointment	
test_appointment_result_id	VARCHAR(10)
test_id	VARCHAR(10)
patient_id	VARCHAR(10)
start_time	TIMESTAMP
end_time	TIMESTAMP
test_status	CHAR(1)
report_link	VARCHAR(255)
comment	VARCHAR(255)
result	TEXT

In this table we store the details of every test which has been conducted in the hospital.
We store the test_id which is a foreign key which refrences the Test table and is used to know which test is conducted. The patient_id is also a foreign key which denotes which patient has undergone that test.

Other attributes are used to store start and end time of the test, its report link, result, status and comments if any.

- doc_appointment

doc_appointment	
doc_appointment_id	VARCHAR(10)
doc_id	VARCHAR(10)
patient_id	VARCHAR(10)
start_time	TIMESTAMP
end_time	TIMESTAMP
appointment_status	CHAR(1)
symptoms	VARCHAR(255)
treatments	VARCHAR(255)

In this table we store the details of every appointment which has been conducted in the hospital.

We store the doc_id which is a foreign key which references the Doctor table and is used to know which doctor undertook the appointment. The patient_id is also a foreign key which patient's appointment it was.

Other attributes are used to store start and end time of the appointment, symptoms before the appointment, treatment administered and appointment status.

3 Languages and Tools

In this section we will go through the languages, tools and the methodology we used to build this Hospital Management System.

This system consists of three segments, namely Frontend, Backend and Database.
Now we will go through each of the three segments one-by-one.

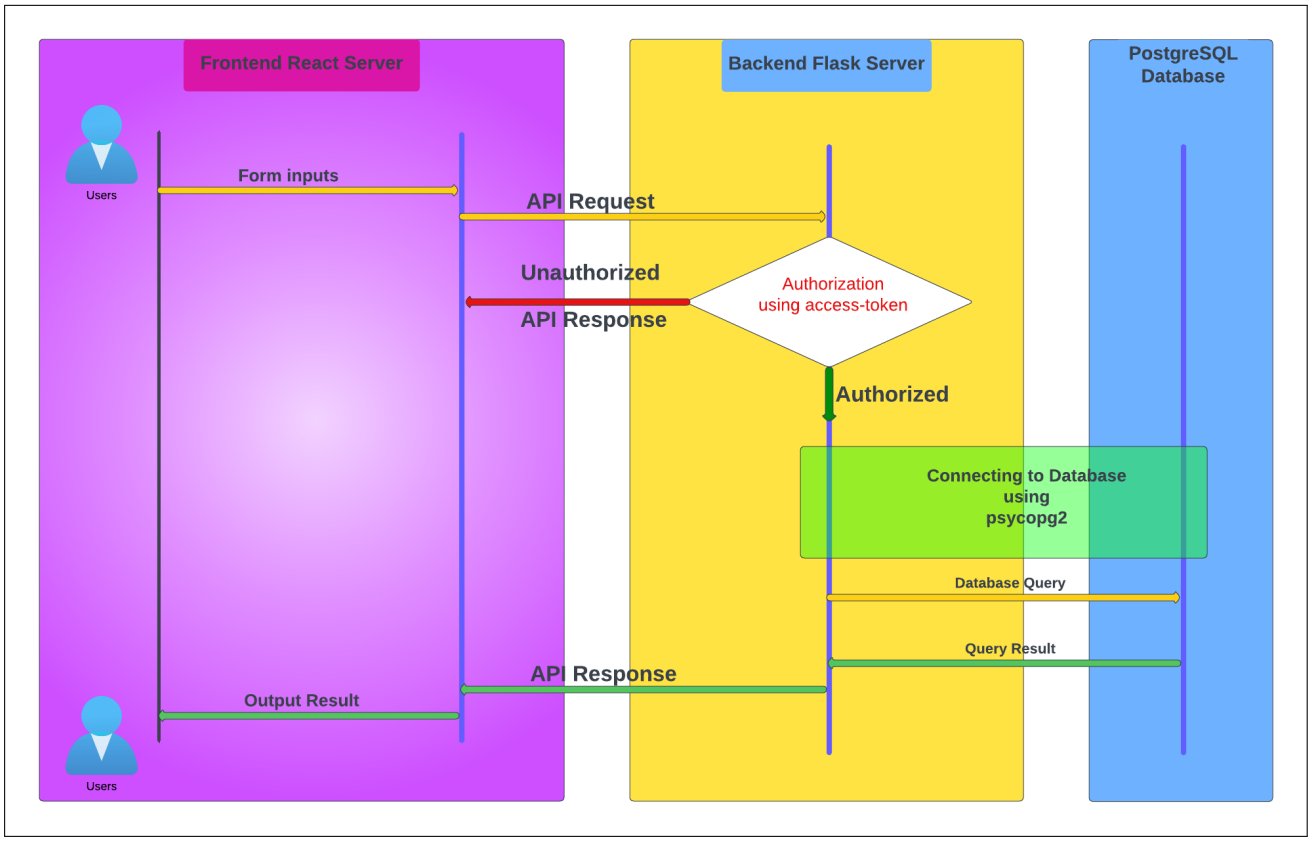
- Frontend
  For this segment, we have used React framework of the JavaScript language. The front-end is what the user sees and any interaction with the user results in change of the website which is done by us using React. Any data which is displayed on the web page is fetched by the front-end by sending an API request to the backend server.

- Backend
  In this segment, we have used Flask framework of Python to make the backend server. This server handles the API (Application Program Interface) calls which are made by the frontend React web page.
  Whenever this server receives an API call from the React Server, we parse the request and implement API and Data security policy. If the request is able to pass through the security screenings, we fetch the required data from our database server using the psycopg2, a PostgreSQL database adapter for Python and then passes it as a response to the frontend React Server.

- Database
  This segment consists of a PostgreSQL database which is hosted on SUPABASE (an online Open-Source Database service which is used to host PostgreSQL databases).
  This database is queried for information by the Backend Flask Server and returns the queried data to it.

4 Implemented Workflows and Triggers

The workflow diagram of our system is as follows:



This diagram shows the working of our application in brief. Whenever a User interacts with the Web Application, a API Request consisting of access_token is sent to the Backend Flask Server to take action corresponding to the User's interaction. The `access_token` then gets checked for correctness. If authorization is not granted, suitable message is sent to the Frontend. If authorization is granted, the Backend then connects to the database and performs the required function of SELECT, UPDATE, or DELETE and then closes the connection. After completing its task, the Backend sends the API response to the Frontend and this is reflected in the User Interface.

We store the `access_token` and `user_id` of the current user in the local storage of the browser.

The triggers which we added are:

- When a User is added, he gets a personalized email to change his default password.
- When the Test Result of a particular test is added, the patient gets an email that his test report has been added.
- The doctors automatically get an email in cases of emergency or otherwise they get a weekly email of their patients well being.
- When Treatment corresponding to an Appointment is added, the patient gets an email about the Treatment.

5 Code Listings of Backend APIs

In this section we have listed all the APIs which we are using to implement the functionality of the system and also how the APIs query the database for the required data.

- Function to connect to online PostgreSQL(SUPABASE) Database

```
def get_db_connection():
    conn = psycopg2.connect(host=getenv('SUPABASE_HOST'),
    database=getenv("SUPABASE_DATABASE"), user=getenv("SUPABASE_USER"),
    password=getenv("SUPABASE_PASSWORD"), port=getenv('SUPABASE_PORT'))
    return conn
```

- Logout API

```
@auth.route('/logout', methods=['POST'])
def logout_user():
    # delete the access token or update the expiry date
    ########################################
    req = request.get_json()
    access_token = req['access_token']

    val, current_user_id = check_token(access_token, ['fdo','dba', 'doc', 'deo'])
    ########################################
    # we have an access token if we come here
    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute("UPDATE users SET access_token = '' WHERE user_id =
    '"+current_user_id+"';")

    conn.commit()
    conn.close()

    return jsonify(message="Logout Success"), 200
```

- Login API

```
@auth.route('/login', methods=['POST'])
def login():
    req = request.get_json()        # request should contain "user_id" and "password"

    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute("SELECT user_id, type FROM users WHERE user_id = %s AND password = %s", (req['user_id'], req['password']))
    data = cur.fetchall()

    if len(data) != 1:
        return jsonify(message = "Login Failed"), 401

    # this contains info of the login entity
    temp = data[0]
    user_type = temp[1]
    # create the access token
    access_token = create_access_token(identity=req['user_id'])
    access_token = user_type + access_token

    # save access_token in the user table along with expiration of 5 mins later
    time_ex = datetime.now() + timedelta(minutes=100)
    new_expiry_time = time_ex.strftime('%Y-%m-%d %H:%M:%S')

    # update the table
    cur.execute("UPDATE users SET access_token = \'"+access_token+"\', access_token_expiry = \'"+new_expiry_time+ "\' WHERE user_id = \'"+temp[0]+"\';")
    conn.commit()
    conn.close()

    return jsonify(message="Login Success",access_token=access_token, user_id=temp[0]), 200
```

- Code Snippet which does user authentication using `access_token`

```
1       req = request.get_json()
2       access_token = req['access_token']
3
4       val, current_user_id = check_token(access_token, ['dba', 'fdo'])
5       if val == 401:
6           return jsonify(message = "Unidentified User"), 401
7       elif val == 69:
8           return jsonify(message = "User Session Expired"), 401
9       elif val == 403:
10          return jsonify(message = "Page Forbidden for user"), 403
```

Here in line no 4, we specify that which type of users can access a particular backend API.

- API to get number of each user type for DBA Dashboard

```
1       conn = get_db_connection()
2       cur = conn.cursor()
3
4       cur.execute("SELECT count(*) FROM users WHERE users.type='dba';")
5       val_dba = cur.fetchone()[0]
6       cur.execute("SELECT count(*) FROM users WHERE users.type='deo';")
7       val_deo = cur.fetchone()[0]
8       cur.execute("SELECT count(*) FROM users WHERE users.type='fdo';")
9       val_fdo = cur.fetchone()[0]
10      cur.execute("SELECT count(*) FROM users WHERE users.type='doc';")
11      val_doc = cur.fetchone()[0]
12
13      conn.close()
14
15      return jsonify({"DBA": val_dba, "DEO": val_deo, "FDO": val_fdo, "DOC":
        val_doc}), 200
```

- API to get information of all users for DBA Dashboard

```
    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute("SELECT user_id, name, ph_number, type, address FROM users;")
    val = cur.fetchall()
    conn.close()

    list=[]
    for i in val:
        dict = {"user_id":i[0], "name":i[1], "ph_number":i[2], "type":i[3],
        "address":i[4]}
        list.append(dict)

    return jsonify(list), 200
```

- API to implement search functionality for DBA Dashboard

```
    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute("SELECT user_id, name, ph_number, type, address FROM users;")
    val = cur.fetchall()
    conn.close()

    searchString = request.args.get('search_string')
    list=[]
    if any(chr.isdigit() for chr in searchString):
        for i in val:
            if searchString.lower() in i[0].lower():
                dict = {"user_id":i[0], "name":i[1], "ph_number":i[2], "type":i[3],
                "address":i[4]}
                list.append(dict)
    else:
        for i in val:
            if searchString.lower() in i[1].lower():
                dict = {"user_id":i[0], "name":i[1], "ph_number":i[2], "type":i[3],
                "address":i[4]}
                list.append(dict)
    return jsonify(list), 200
```

- API for adding User by DBA

```
    req = request.get_json()
    name = req['name']
    ph_number = req['ph_number']
    type = req['type']
    address = req['address']
    password = req['password']
    user_id = generate_user_id(type)

    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute("INSERT INTO users (user_id, name, ph_number, password, type,
    address) VALUES (%s, %s, %s, %s, %s, %s);", (user_id, name, ph_number,
    password, type, address))
    conn.commit()
    conn.close()

    if type == 'doc':
        email = req['email']
        docType = req['docType']

        conn = get_db_connection()
        cur = conn.cursor()
        cur.execute("INSERT INTO doctors (doc_id, user_id, email, docType) VALUES
        (%s, %s, %s, %s);", (user_id, user_id, email, docType))
        conn.commit()
        conn.close()

    return jsonify({"message": "User added successfully", "user_id":user_id}), 200
```

- API for deleting User by DBA

```
    req = request.get_json()
    user_id = req['user_id']

    if user_id == current_user_id:
        return jsonify(message = "Cannot delete self"), 403

    conn = get_db_connection()
    cur = conn.cursor()
    if user_id.startswith('DOC'):
        cur.execute("DELETE FROM doctors WHERE doc_id=%s;", (user_id,))
```

```

    cur.execute("DELETE FROM users WHERE user_id=%s;", (user_id,))
    conn.commit()
    conn.close()

    return jsonify({"message": "User deleted successfully"}), 200
```

- API for changing password

```
req = request.get_json()
user_id = req['user_id']
password = req['password']

if user_id!=current_user_id and current_user_id[1] != 'B':
    return jsonify(message = "Cannot update password of other users"), 403

conn = get_db_connection()
cur = conn.cursor()

cur.execute("SELECT count(*) FROM users WHERE user_id=%s;", (user_id,))
val = cur.fetchone()[0]
if(val == 0):
    return jsonify({"message": "Invalid user_id"}), 400

cur.execute("UPDATE users SET password=%s WHERE user_id=%s;", (password, user_id))

conn.commit()
conn.close()

return jsonify({"message": "Password updated successfully"}), 200
```

- API for registering a new patient by Front Desk Operator

```
    data = request.get_json()
    conn = get_db_connection()
    cur = conn.cursor()

    patient_id = generate_patient_id()

    dob = reqDate_to_SQLdate(data['dob'])

    cur.execute("INSERT INTO patients (patient_id, patient_name, dob, email,
    address, conditions, ph_number, gender) VALUES (%s, %s, %s, %s, %s, %s, %s,
    %s)", (patient_id, data['patient_name'], dob, data['email'], data['address'],
    data['conditions'], data['ph_number'], data['gender']))
    conn.commit()
    conn.close()

    return jsonify({"message": "Patient added successfully", "patient_id":
    patient_id}), 200
```

- API for getting list of Tests offered by Front Desk Operator

```
    conn = get_db_connection()
    cur = conn.cursor()
    cur.execute("SELECT * FROM test")
    val = cur.fetchall()

    list=[]
    for i in val:
        dict = {"test_id":i[0], "test_name":i[1]}
        list.append(dict)

    conn.close()

    return jsonify(list), 200
```

- API for getting dates of a particular Test by Front Desk Operator

```
    args = request.args
    test_id = args['test_id']

    conn = get_db_connection()
    cur = conn.cursor()

    cur.execute("SELECT DISTINCT test_date FROM test_slots WHERE test_id = %s AND
    test_date >= %s;", (test_id, datetime.date.today()))
    val = cur.fetchall()

    list=[]
    for i in val:
        list.append({"date":i[0].strftime('%Y-%m-%d')})

    conn.close()

    return jsonify(list), 200
```

- API for getting slots of a Test on a particular date by Front Desk Operator

```
    args = request.args
    test_id = args['test_id']
    test_date = reqDate_to_SQLdate(args['date'])
    test_date_begin = str(test_date) + " 00:00:00"
    test_date_end = str(test_date) + " 23:59:59"

    conn = get_db_connection()
    cur = conn.cursor()

    uniqueSet = {}

    cur.execute("SELECT test_slot FROM test_slots WHERE test_id = %s AND test_date
    = %s", (test_id, test_date))
    allSlots = cur.fetchall()
    for mid in allSlots:
        slots = mid[0].split(',')
        for slot in slots:
            n1 = int(slot[0:4])
            n2 = int(slot[4:8])
            while n1 < n2:
                uniqueSet[n1] = 1
```

```
21                  n1 += 100
22      print (uniqueSet)
23
24      cur.execute("SELECT start_time FROM test_appointment WHERE test_id = %s AND
        start_time >= %s AND start_time <= %s", (test_id, test_date_begin,
        test_date_end))
25      bookedSlots = cur.fetchall()
26      for mid in bookedSlots:
27          midx = str(mid[0])
28          bookedSlot = midx[11]+midx[12]+midx[14]+midx[15]
29          n1 = int(bookedSlot[0:4])
30          uniqueSet[n1] = 0
31
32      list=[]
33      for slot in uniqueSet:
34          if uniqueSet[slot] == 1:
35              start = str(slot)
36              end = str(slot+100)
37              list.append({"slot":start+"-"+end})
38
39      conn.close()
40
41      return jsonify(list), 200
```

In this code snippet we have demonstrated the code snippet where we convert our encoded form of slots to the more general form. This is done in codes from line number 12 through 22.

In lines from 24 to 30, we remove the slots that have already been booked by other patients.

In lines from 32 to 39, we have stored the slots nicely in a list and passed as response of the API.

Similar code is written for Appointment Booking of Doctor.

- API for admitting a patient by Front Desk Operator

```
    data = request.get_json()               # request contains patient_id,
    room_type and admit_date
    conn = get_db_connection()
    cur = conn.cursor()
    patient_id = data['patient_id']
    room_type = data['room_type']
    admit_date = reqDate_to_SQLdate(data['admit_date'])
    # print (admit_date)

    cur.execute("SELECT count(*) from patients WHERE patient_id =
    '"+patient_id+"';")
    val = cur.fetchone()[0]
    if val == 0:
        return jsonify(message="Patient Id does not exist"), 404


    cur.execute("SELECT count(*) FROM admit WHERE patient_id = %s AND admit_date <=
    %s AND discharge_date IS NULL;", (patient_id, admit_date))
    val = cur.fetchone()[0]

    if val>0:
        conn.commit()
        conn.close()
        return jsonify({"message": "Patient already admitted on that date"}), 400

    cur.execute("SELECT count(*) FROM admit WHERE patient_id = %s AND admit_date >
    %s;", (patient_id, admit_date))
    val = cur.fetchone()[0]

    if val>0:
        conn.commit()
        conn.close()
        return jsonify({"message": "Patient already admitted on a later date"}),
        400

    cur.execute("(SELECT room_no FROM room WHERE room_type = %s) EXCEPT ((SELECT
    room_no FROM admit WHERE admit_date <= %s AND discharge_date IS NULL) UNION
    (SELECT room_no FROM admit WHERE discharge_date IS NOT NULL AND admit_date <=
    %s AND %s <= discharge_date))", (room_type, admit_date, admit_date,
    admit_date))
```

```
32
33      if "SELECT 0" in cur.statusmessage:
34          conn.commit()
35          conn.close()
36          return jsonify({"message": "No rooms available"}), 400
37
38      fin_room = cur.fetchone()[0]
39
40      cur.execute("INSERT INTO admit (patient_id, room_no, admit_date,
        discharge_date) VALUES (%s, %s, %s, NULL);", (patient_id, fin_room,
        admit_date))
41
42      conn.commit()
43      conn.close()
44
45      return jsonify({"message": "Patient admitted successfully", "room_no":
        fin_room}), 200
```

In this segment we have handles all the errors that can happen while admitting a patient, such as whether he is already admitted on that date or no rooms are available.

- API for discharging a Patient by Front Desk Operator

```
    data = request.get_json()               # request contains patient_id and
    discahrge date is the current date

    conn = get_db_connection()
    cur = conn.cursor()

    patient_id = data['patient_id']
    discharge_date = datetime.date.today()

    cur.execute("SELECT count(*) FROM admit WHERE patient_id = %s AND admit_date <=
    %s AND discharge_date IS NULL;", (patient_id, discharge_date))
    val = cur.fetchone()[0]
    # print(val + " " + patient_id + " " + discharge_date)
    if val==0:
        conn.commit()
        conn.close()
        return jsonify({"message": "Patient not admitted on that date"}), 400

    cur.execute("UPDATE admit SET discharge_date = %s WHERE patient_id = %s AND
    admit_date <= %s AND discharge_date IS NULL;", (discharge_date, patient_id,
    discharge_date))

    conn.commit()
    conn.close()

    return jsonify({"message": "Patient discharged successfully"}), 200
```

- API for adding test result to a already conducted test by Data Entry Operator

```
    conn = get_db_connection()
    cur = conn.cursor()

    cur.execute("SELECT patient_id FROM test_appointment WHERE
    test_appointment_result_id = \'"+req['test_appointment_result_id']+"\';")
    data = cur.fetchall()

    if len(data) != 1:
        conn.close()
        return jsonify(message="Invalid Test Id"), 401

    temp = data[0]
    patient_id = temp[0]
    cur.execute("SELECT patient_id, email, patient_name FROM patients WHERE
    patient_id = \'"+patient_id+"\';")
    data = cur.fetchall()
    temp = data[0]

    id = req['test_appointment_result_id']
    email = temp[1]
    name = temp[2]
    link = req['report_link']
    result = req['result']

    cur.execute("UPDATE test_appointment SET test_status = \'1\', report_link =
    \'"+req['report_link']+"\', result = \'"+req['result']+"\', comment = \'"+
    req['comment']+ "\' WHERE test_appointment_result_id =
    \'"+req['test_appointment_result_id']+"\';")

    conn.commit()
    conn.close()
```

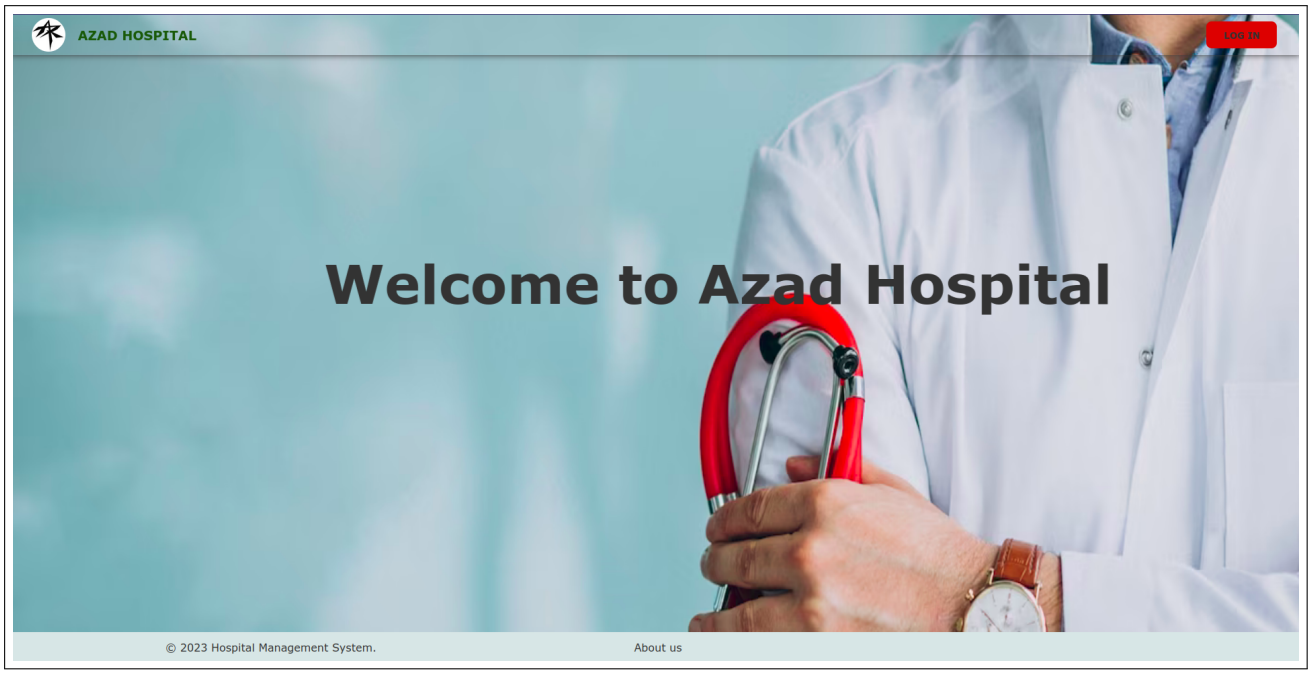
Similarly, the Doctor and the Data Entry Operator can add treatments for a patient corresponding to a specific appointment with the doctor.

- Code Snippet which shows automated Mailing

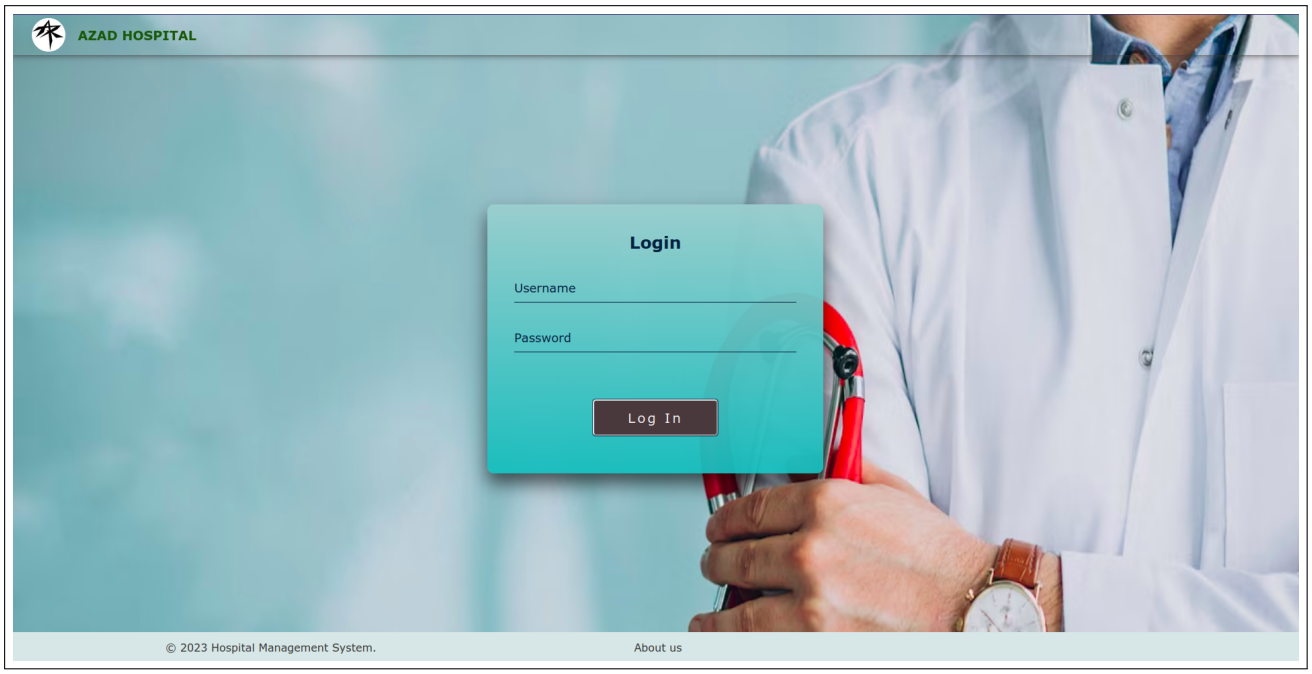
```
def send_email(receiver, subject, body, attachment):
    sender = 'admin@AzadHospital.email'
    msg = MIMEMultipart()
    msg['Subject'] = subject
    msg['To'] = ','.join(receiver)
    msg['From'] = sender
    msg.attach(MIMEText(body, 'plain'))
    for file in attachment:
        filename = file
        with open(filename, 'rb') as f:
            part = MIMEApplication(f.read(),Name=basename(filename))
        part['Content-Disposition'] = 'attachment;
        filename="{}"'.format(basename(filename))
        msg.attach(part)
    # print(basename(filename))
    link = getenv('MAILGUN_LINK')
    port = getenv('MAILGUN_PORT')
    user = getenv('MAILGUN_USER')
    password = getenv('MAILGUN_PASSWORD')
    with smtplib.SMTP(link, port) as server:
        server.login(user, password)
        server.sendmail(sender, receiver, msg.as_string())
        print("Successfully sent email")
```

6 Frontend Dashboards and Web pages

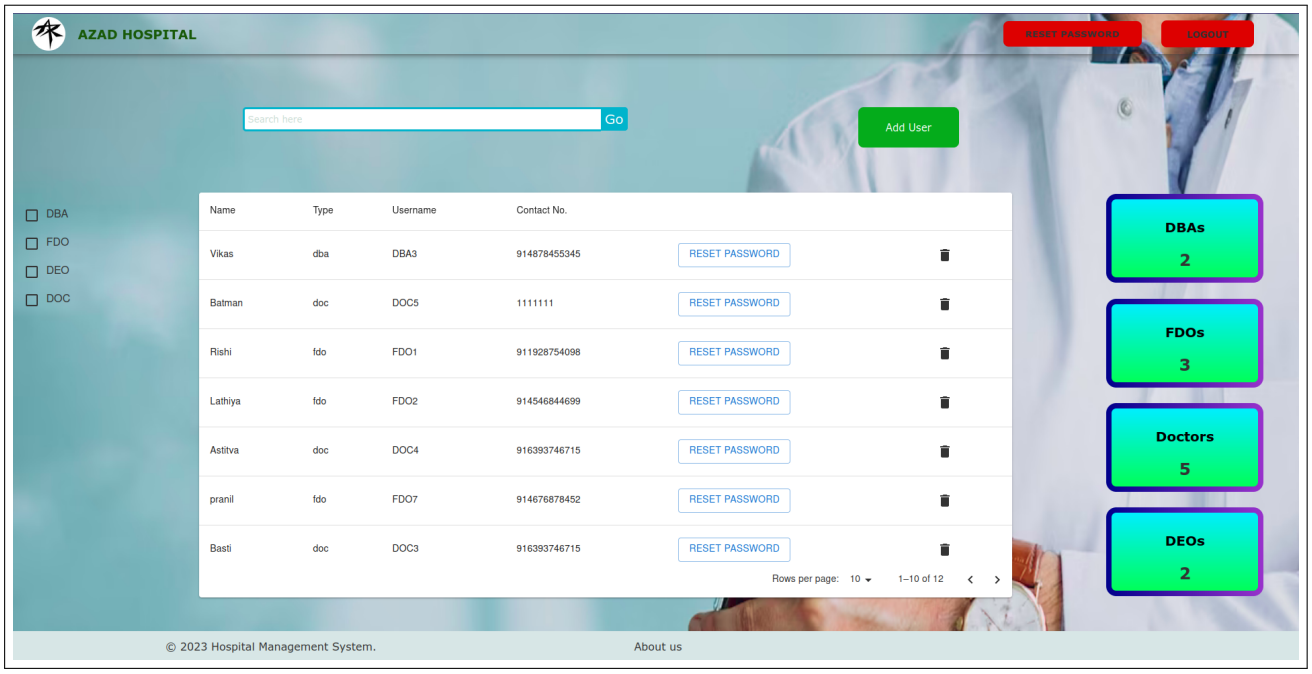
➜ This is the homepage of our website which appears when we open the site:



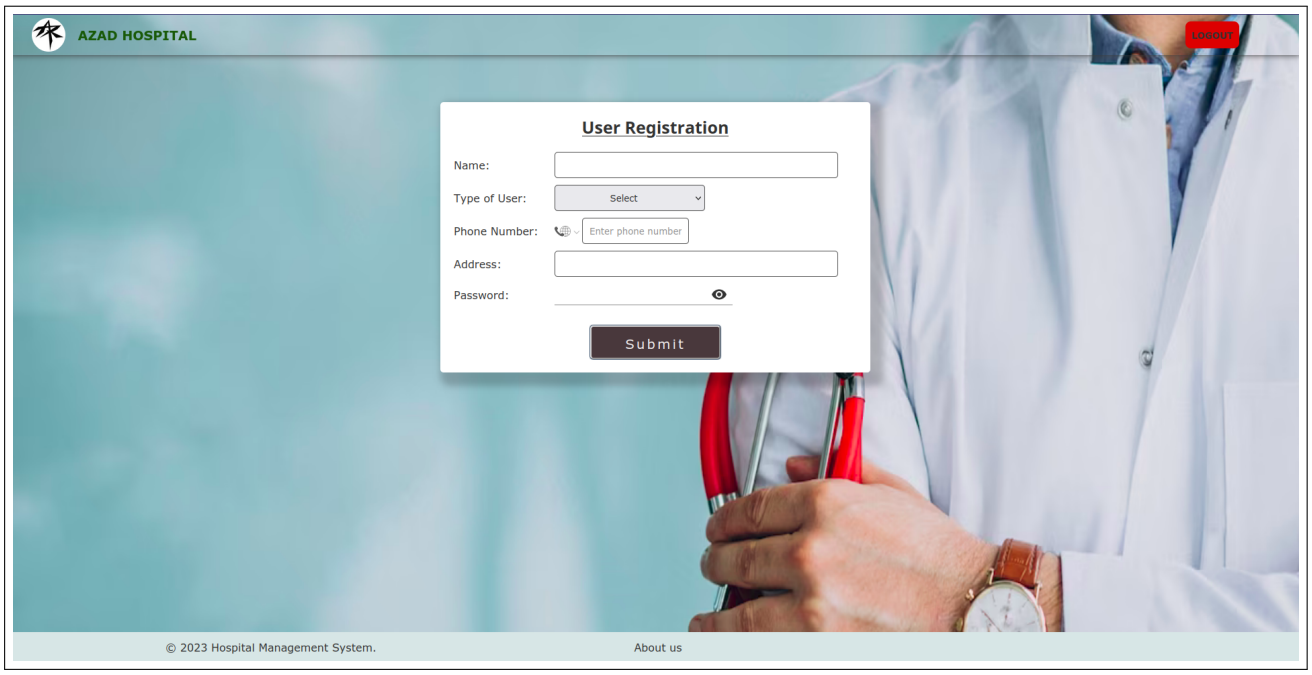
➜ Following is the Login page which opens when we press the `LOG IN` button on the home page.



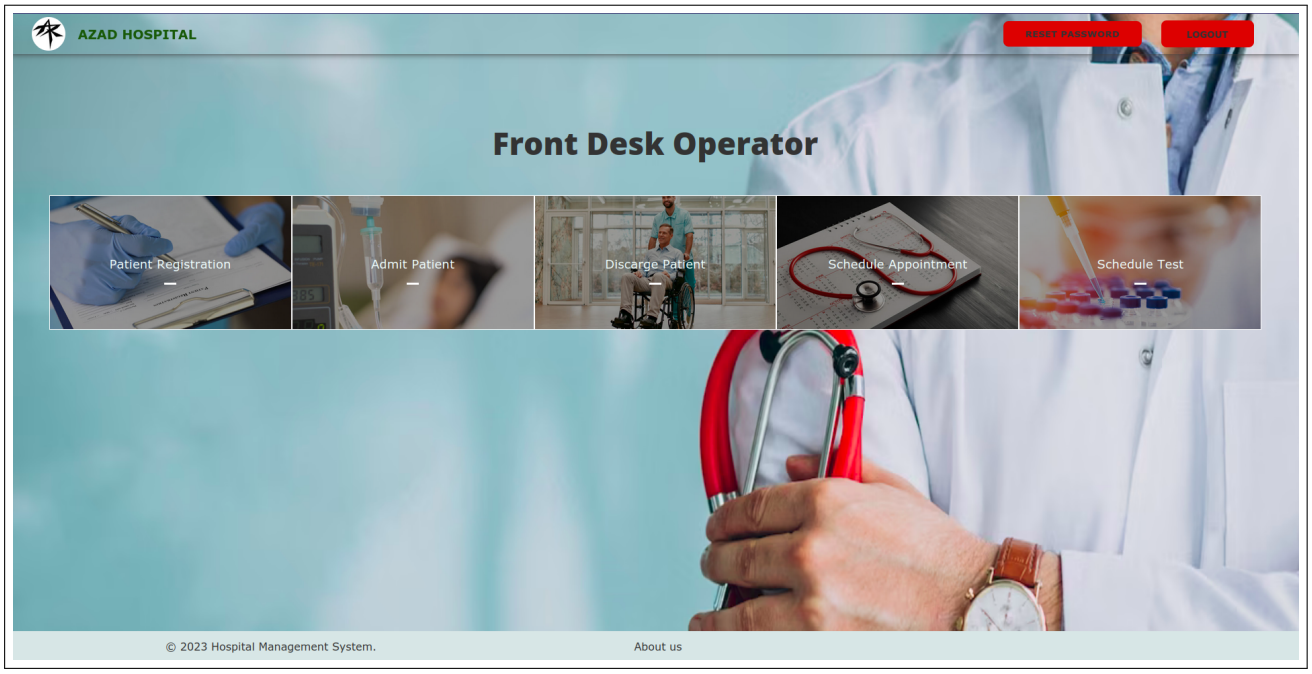
➜ This is the Dashboard which the Database Administrator (DBA) encounters when he logs in. This page has the search functionality where the DBA can search using the `username` or the `Name` of an employee. He can also filter the employees using the check-boxes given on the left-hand side.



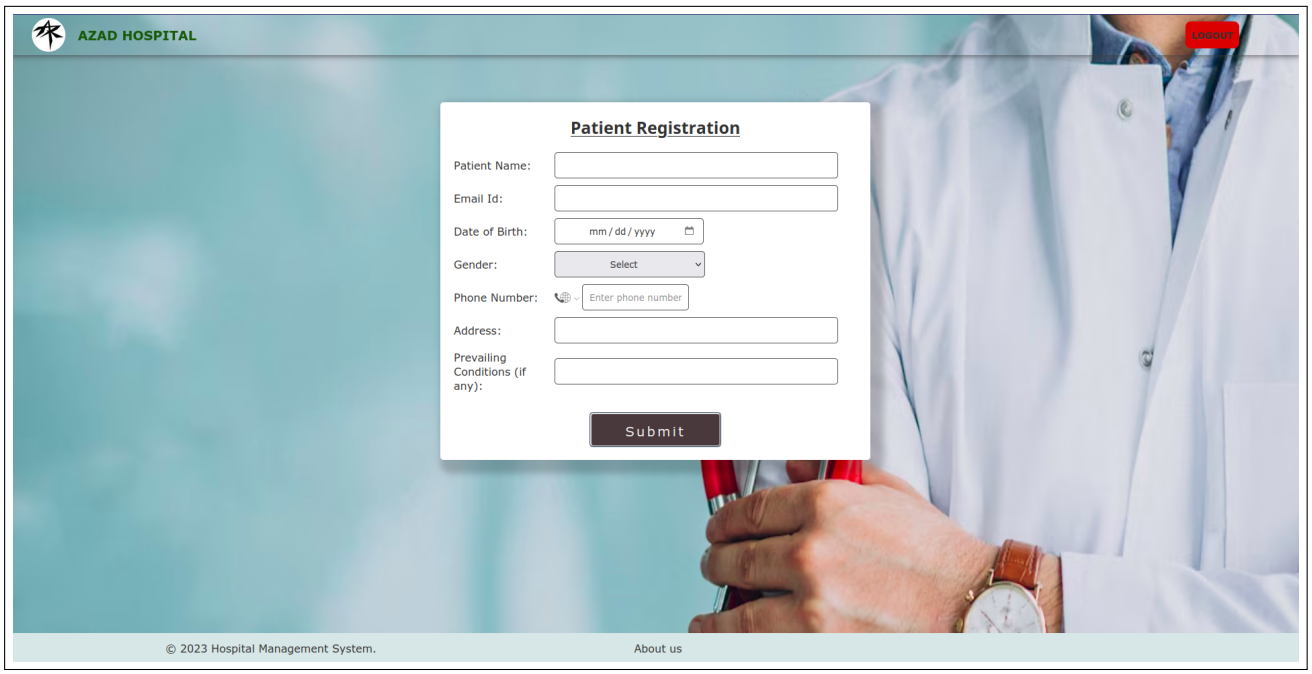
➜ Following is the page which opens when the DBA presses the `Add User` button on his Dashboard.



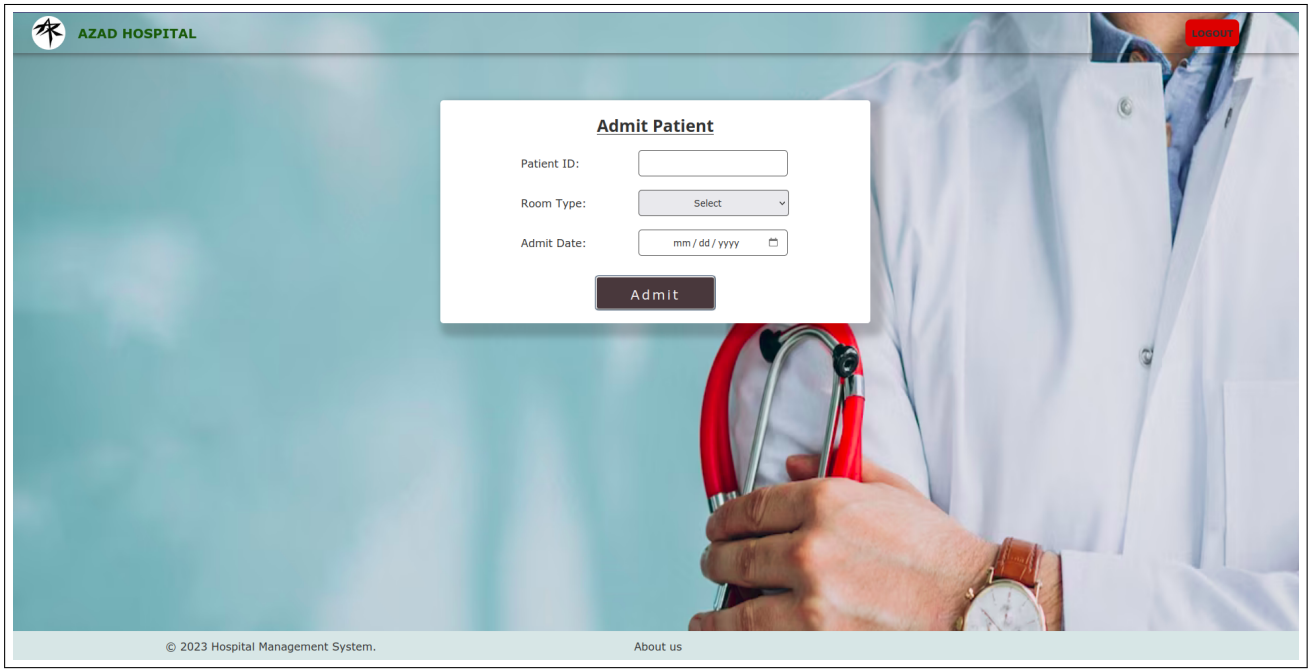
➜ This is the Dashboard which the Front Desk Operator (FDO) encounters when he logs in.
Here the FDO can do any of his 5 duties, ie, Register a Patient, Admit or Discharge a Patient or Schedule and Appointment or Test for a Patient.



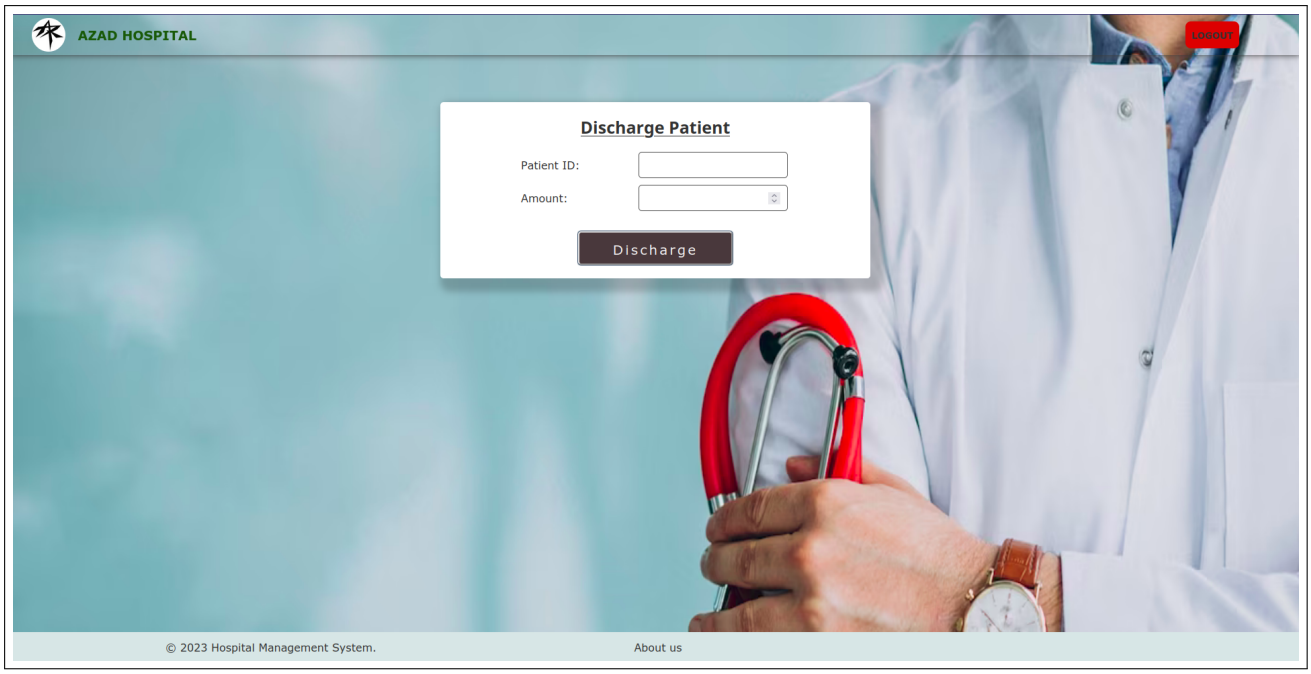
➜ Following is the page encountered by FDO when he wants to Register a Patient.



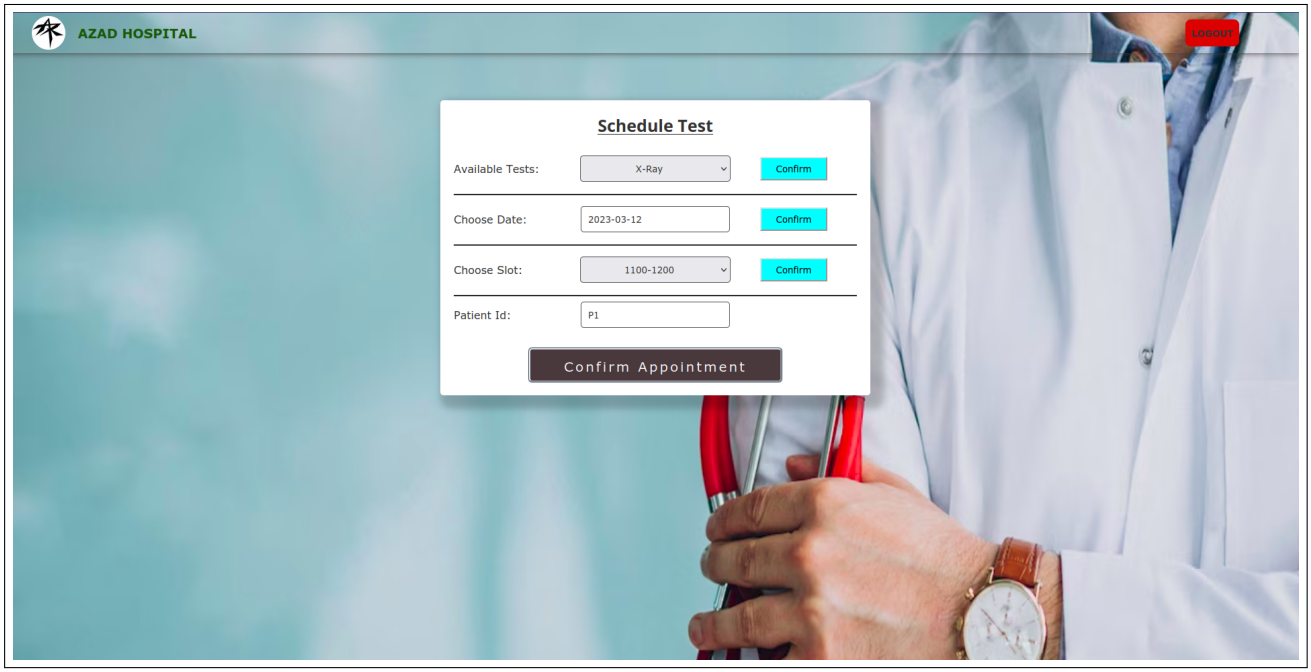
➜ Following is the page encountered by FDO when he wants to Admit a Patient.



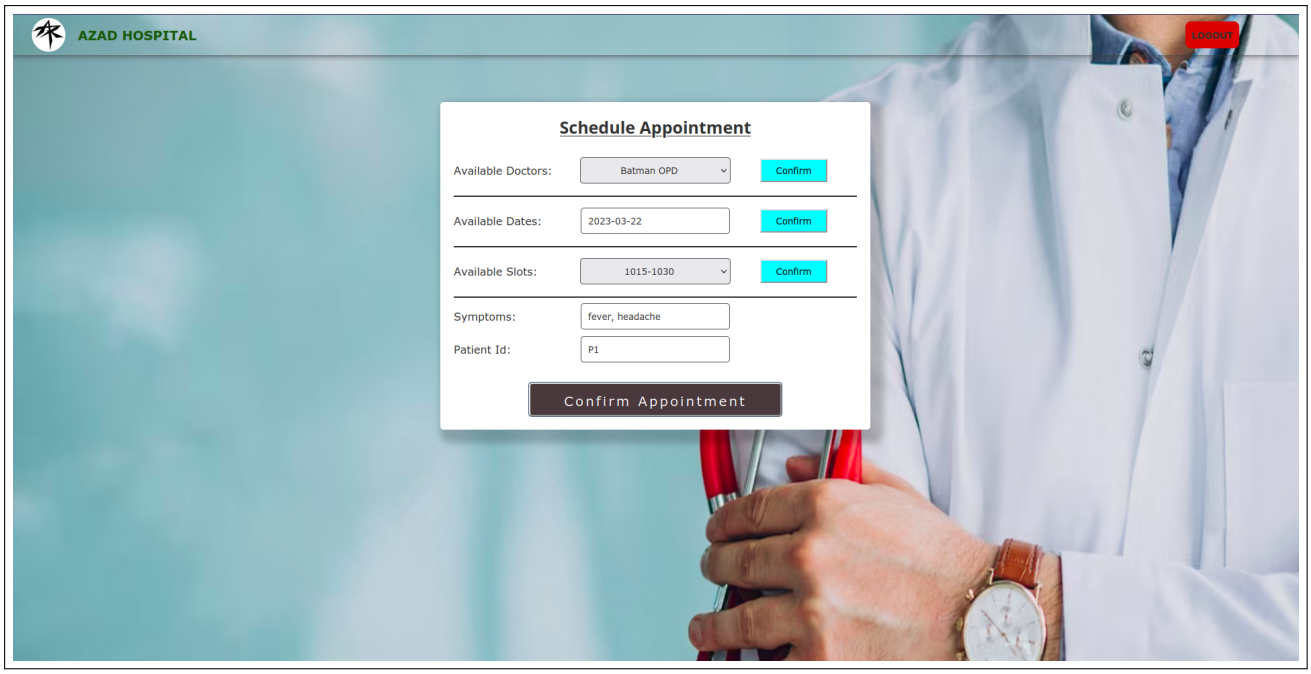
➜ Following is the page encountered by FDO when he wants to Discharge a Patient.



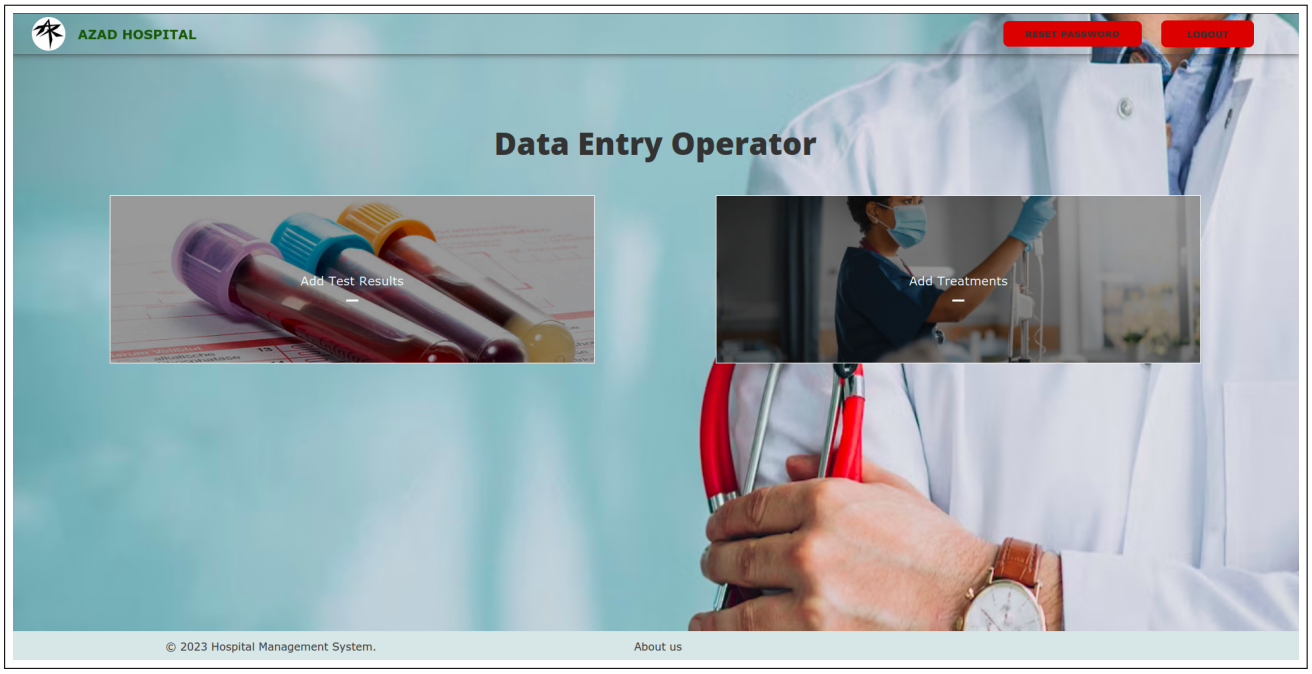
➜ Following is the page encountered by FDO when he wants to Schedule an Test for a Patient.



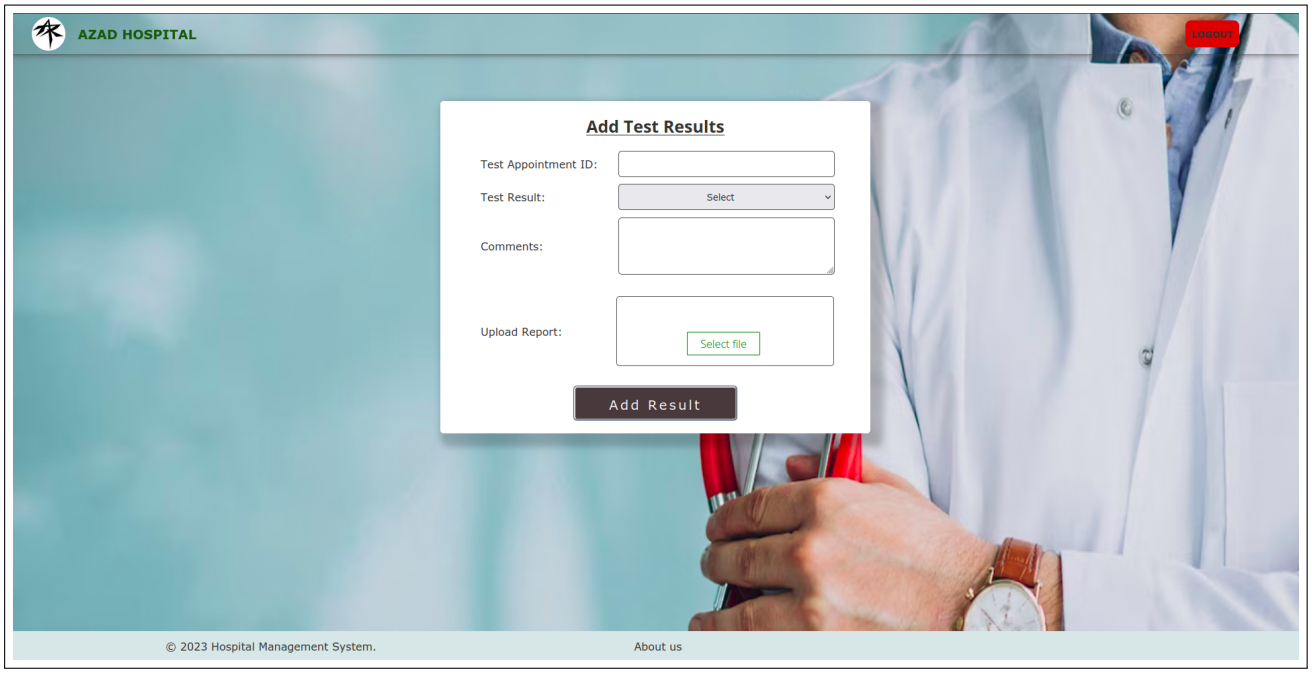
➜ Following is the page encountered by FDO when he wants to Schedule an Appointment for a Patient.



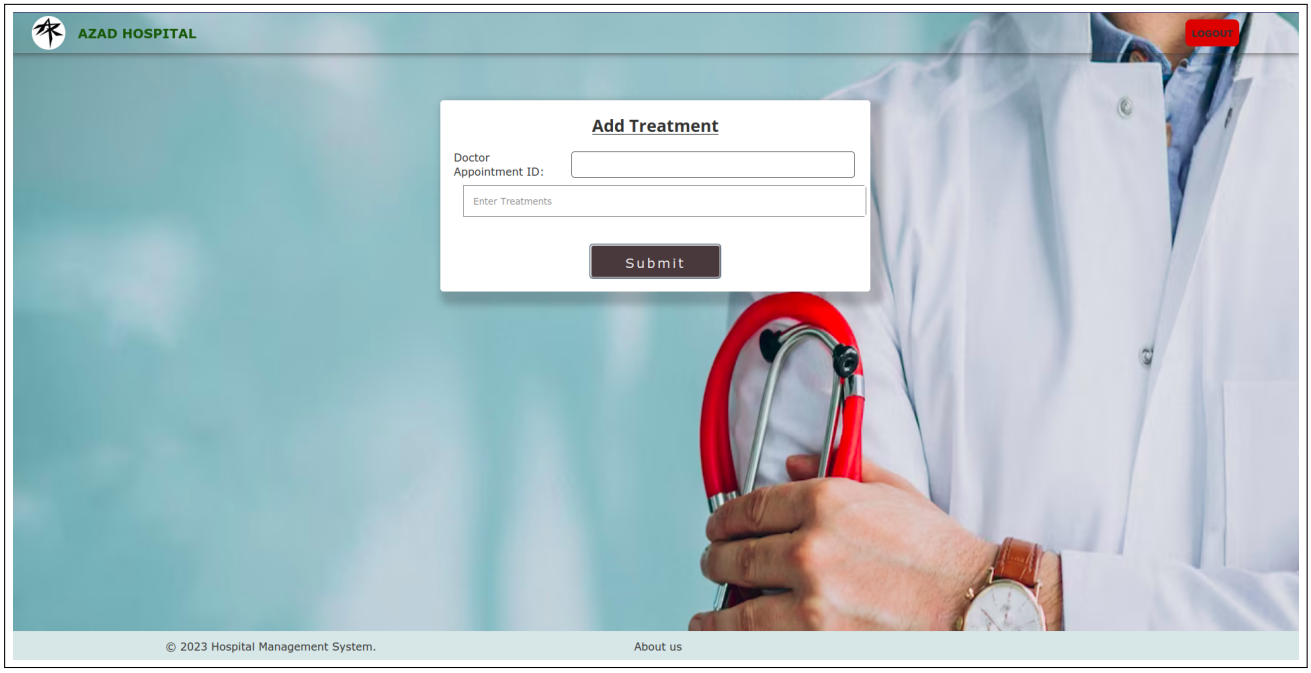
➜ This is the Dashboard which the Data Entry Operator (DEO) encounters when he logs in. Here the DEO can do any of his 2 duties, ie, add result for a Test which is already conducted or add treatment corresponding to a Patient's appointment with a Doctor.



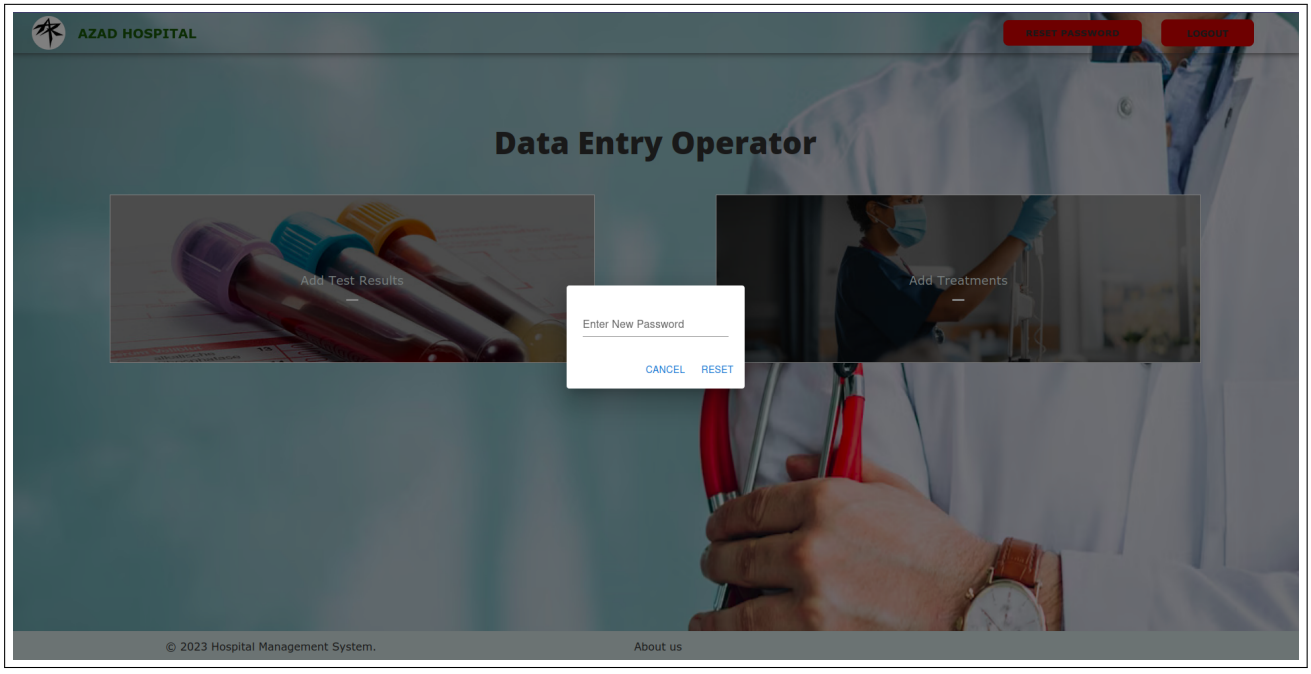
➜ Following is the page encountered by DEO when he wants to add result corresponding to a Test. Here we can see that he can add an image or PDF file for the result along with any comments and a `Test Result` which can be either positive or negetive or none.



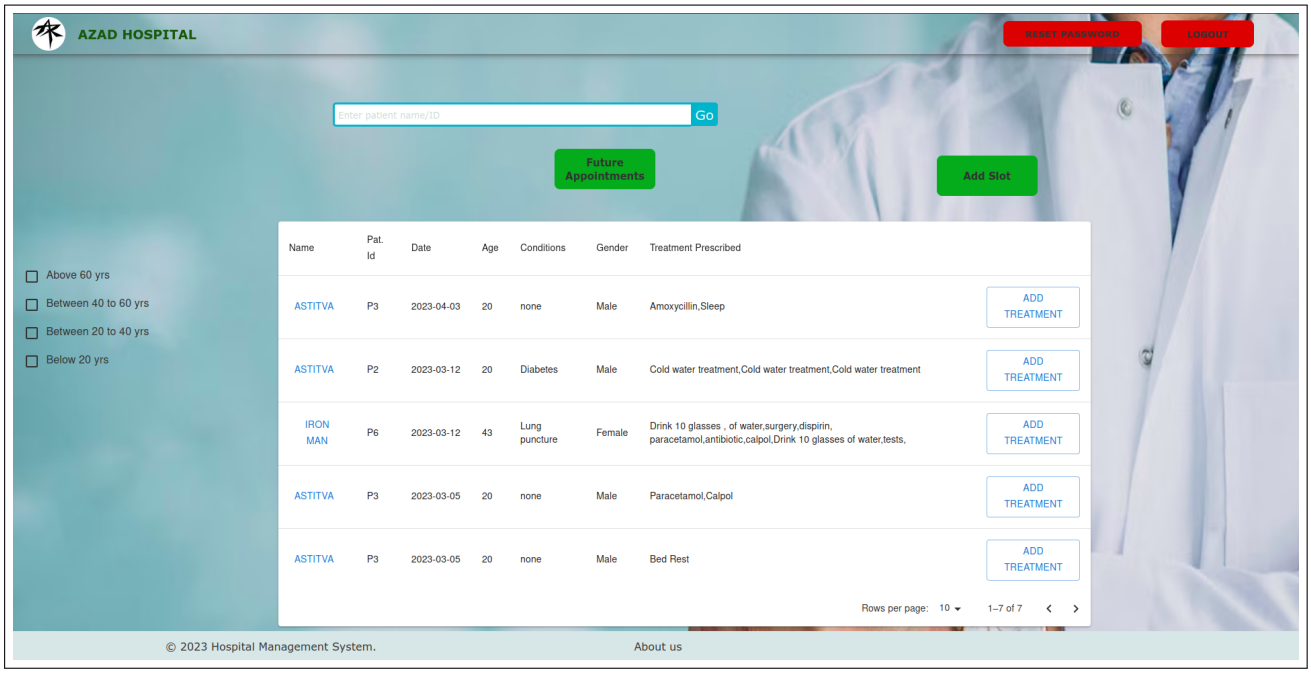
➜ Following is the page encountered by DEO when he wants to add treatment corresponding to an appointment with the doctor.



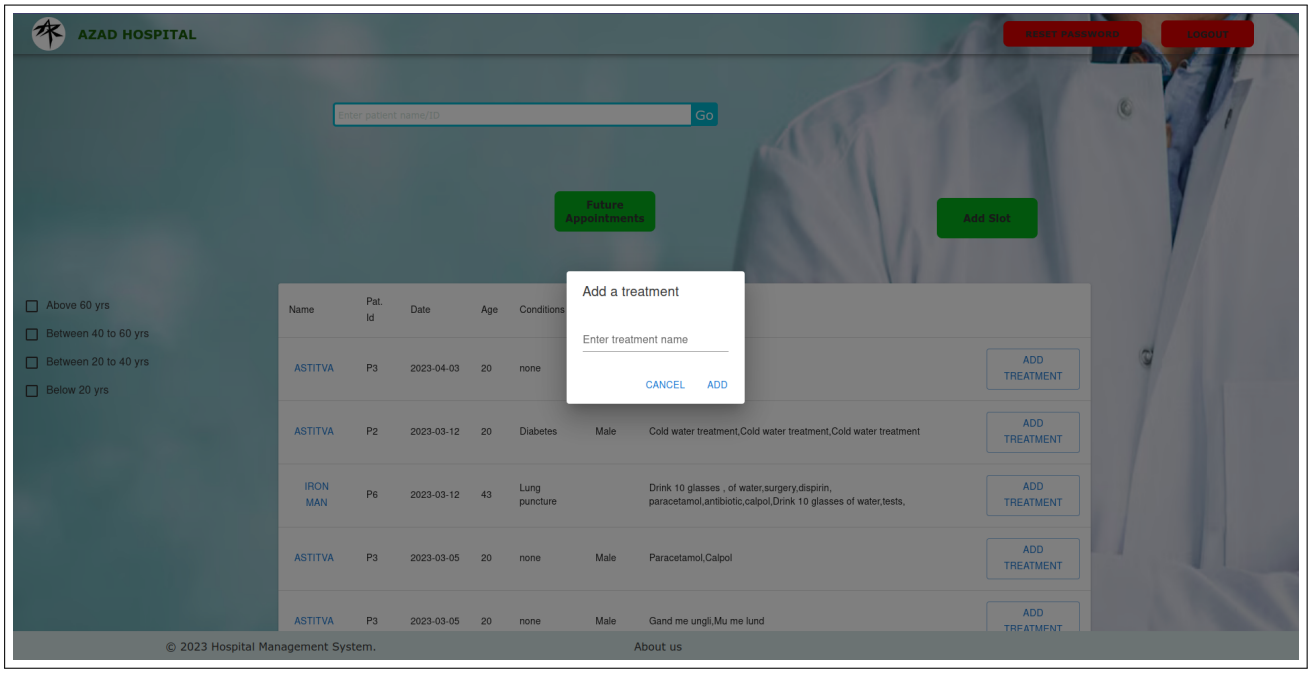
➜ This is the dialog-box which appers when any User has to change his/her password using the `Reset Password` button on the top right corner of the webpage.



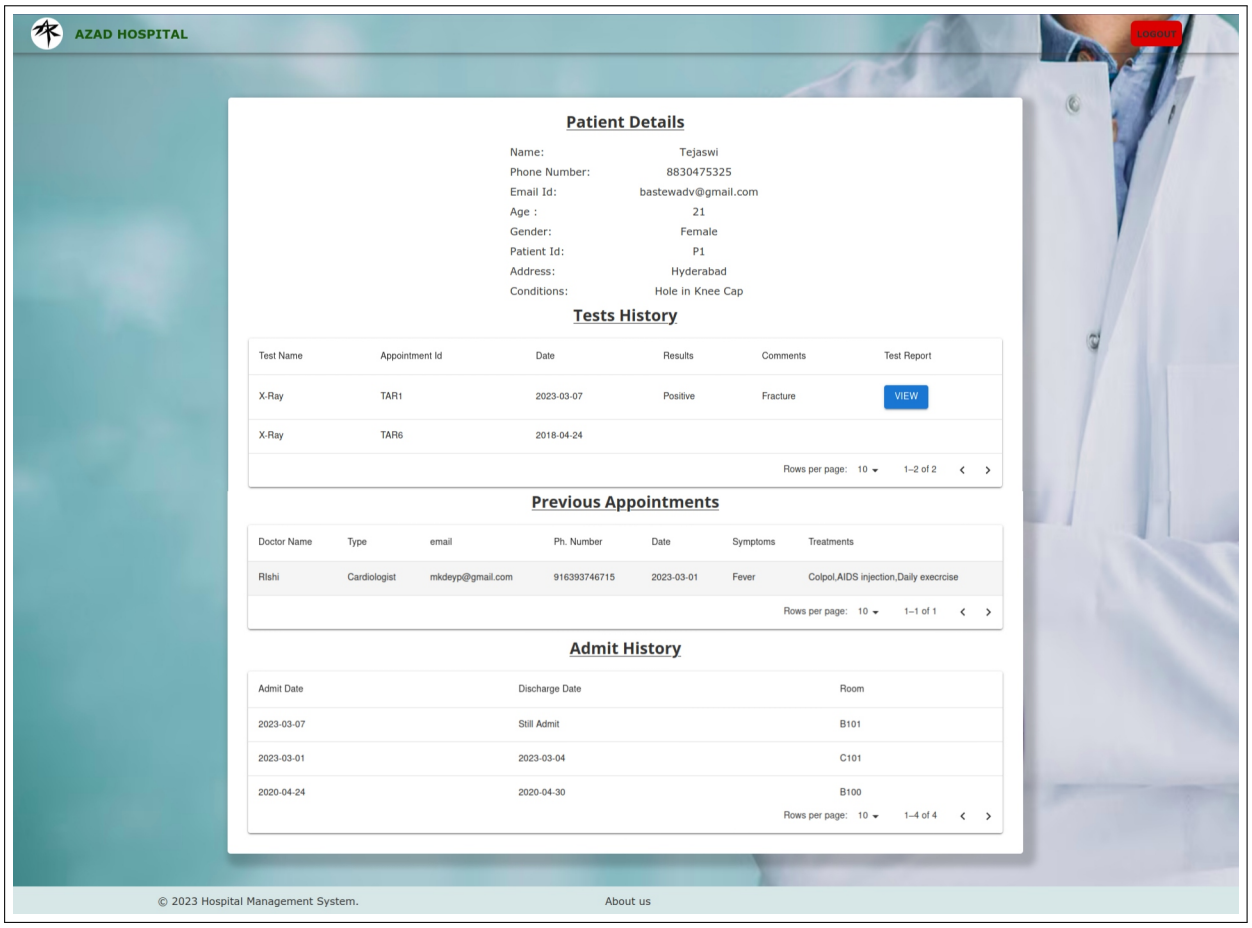
➜ This is the Dashboard which the Doctor encounters when he logs in.
Here the Doctor can see all the Patients he has treating in descending order of Appointments. He also has the option of `Add Treatment`, see `Future Appointments` and `Add Slot` for himself. By clicking on a Patient's name, he will be able to see detailed information about a Patient



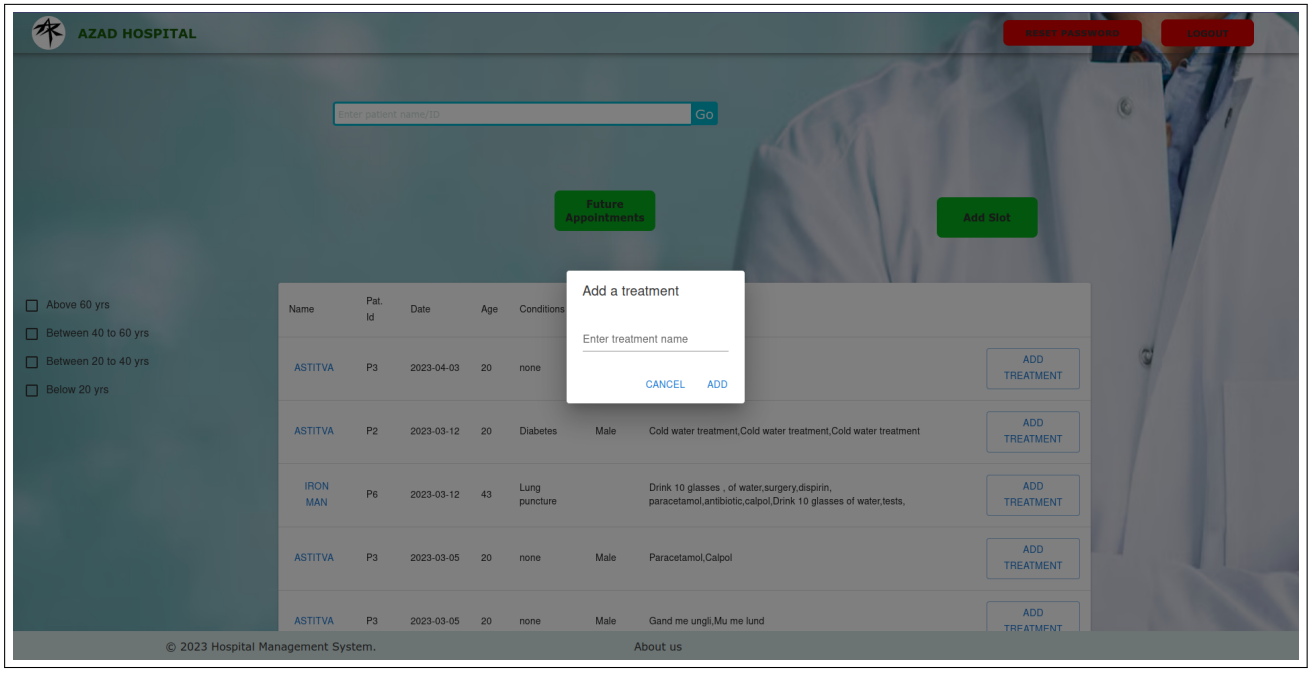
➜ This is the dialog-box which the Doctor encounters when we wants to add a specific Treatment for a Patient against an Appointment.



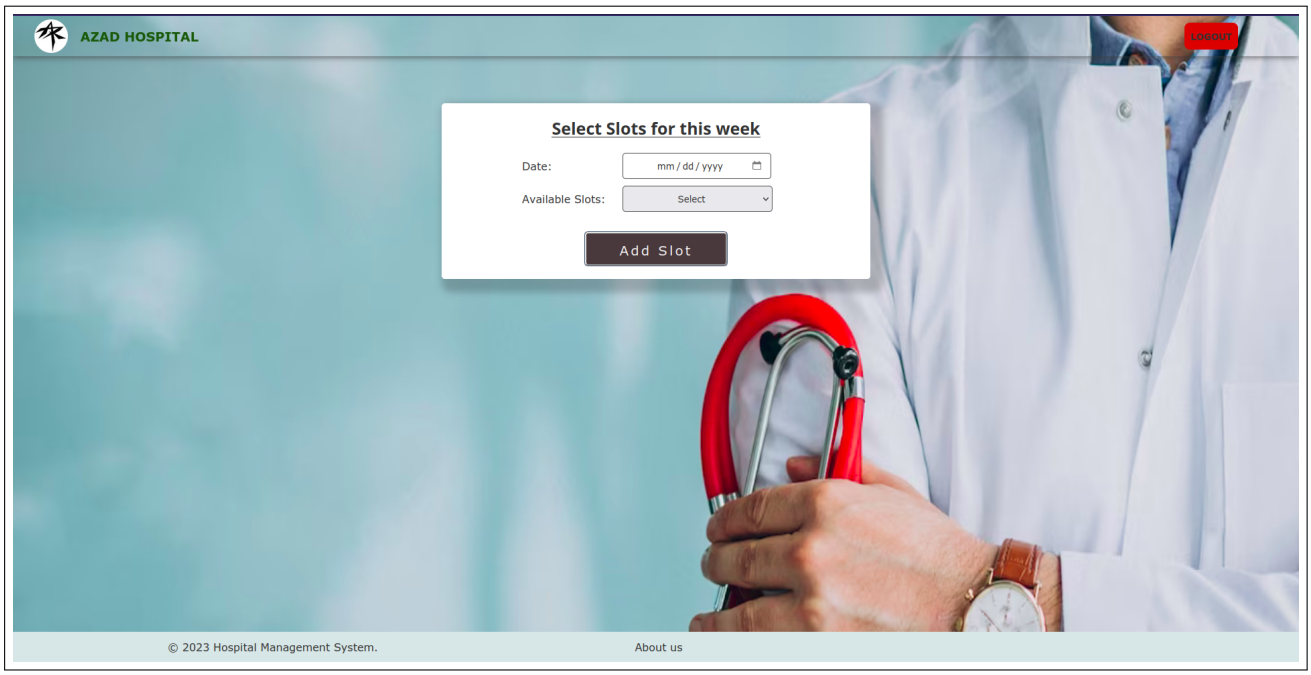
➜ This is page the Doctor is directed to when he wants to see details pertaining to a particular patient. Here, he can see the basic details of the patient along with any medical conditions he is suffering from. The doctor can also see the tests which the patient has undergone along with the respective test reports in image or pdf format. He also sees the admit and appointment of the history along with the cause and treatment of the same.



➜ This is the dialog-box which the Doctor encounters when we wants to add a specific Treatment for a Patient against an Appointment.



➜ This is the page the Doctor is redirected to when he wants to add his Appointment Slots for the upcoming week.



7 Future

In future, the Hospital Management System can extend its functionality to even let the Patients use the Web Application so that they can themselves book appointments, schedule tests, view their test results and add any of their symptoms which are personal and can't be disclosed to the Front Desk Operator while registering.

Also, we can make an interface through which the Patient and Doctor can contact each other via chat through the Web Application. This will help the Patient contact their doctor if he/she faces any difficulty, be it due to a treatment administered or any other thing. The doctor can also stay updated about his patients by asking them about their well-being.

THANK YOU