DATABASE MANAGEMENT SYSTEMS LABORATORY

CS39202

Assignment 4: Mini Project

Web Application Development
Hospital Management System

Project Report

Team: The SQL Squad

Roopak Priydarshi (20CS30042)

Gaurav Malakar (20CS10029)

Abhijeet Singh (20CS30001)

Monish N (20CS30033)

Gopal (20CS30021)

➢ INTRODUCTION:

A **Hospital Management System** is a crucial web application that assists hospitals and other medical facilities in **managing their operations, patient data, and medical records**. A well-designed HMS helps to improve the quality of care and increase operational efficiency by streamlining and automating various administrative and clinical processes.

A Database Management System is an integral part of any HMS as it helps to store, **manage and retrieve large amounts of data efficiently**. The DBMS allows healthcare professionals to **access accurate and up-to-date patient information quickly and easily**, enabling them to provide timely and effective care to their patients.

In this project, we will be developing a Hospital Management System using a DBMS to ensure that medical personnel have **access to accurate and reliable data at all times**. Our goal is to create a user-friendly system that can handle **patient records, healthcare records, appointments, heath informations, and other critical functions** that are essential to the smooth functioning of a hospital or medical facility.

➢ Languages and tools used:

➔ FrontEnd

The front-end of a web application is developed using a combination of **HTML**, **CSS**, **JavaScript**, and **ReactJS.** These technologies work together to create a visually appealing and interactive user interface that allows users to interact with the application and access its various features and functions. HTML provides the **structure and content** of the web page, CSS is used for **styling and layout**, JavaScript enables **dynamic behavior and interactivity**, and ReactJS is a popular library for building **reusable UI components and managing application state**.

➔ BackEnd

For the back-end, we used **Node.js** as a server-side JavaScript runtime. To simplify the development process, we also used the **Express.js** framework which provides tools for **handling HTTP requests and responses, routing, and middleware**. To interact with our MySQL database, we used the **mysql2** library. Additionally, we used **cors** middleware to allow our application to **access resources from other domains**. Overall, this backend technology stack allowed us to build a **fast and efficient** web application with **robust database connectivity and secure resource access**.

➔ Database

In our project, we utilized **MySQL** as our preferred database management system for **storing and organizing schemas**. MySQL is a widely used, open-source relational database management system that allows us to create and manage databases, tables, and other related objects. By using MySQL, we were able to ensure that our **data is organized and easily accessible**, with the **ability to retrieve, modify, and analyze data efficiently**.

➔ Operating System

Our application is designed to be compatible with multiple operating systems, including **Windows** and **Linux**. By ensuring cross-platform compatibility, we have made our application accessible to **a wider audience** and have made it **easier for users** to access its features and functions, regardless of the device or operating system they are using.

➢ DATABASE SCHEMAS:

➔ Schema for User

```
User (
Name varchar(255) NOT NULL,
Email varchar(255) NOT NULL,
EmployeeID varchar(255) NOT NULL,
Password varchar(255) NOT NULL,
Pass_iv varchar(255) NOT NULL,
Type int NOT NULL,
PRIMARY KEY (EmployeeID, Type)
);
```

The above SQL schema defines a "User" table with five columns: Name, Email, EmployeeID, Password, Pass_iv, and Type. EmployeeID and Type together form the primary key. The table is used to store user information, including their name, email, unique employee ID, password, and encryption initialization vector. The Type column stores an integer value that identifies the user's type.

➔ Schema for Front Desk Operator

```
Front_desk_operator (
EmployeeID varchar(255) NOT NULL,
Address varchar(255) NOT NULL,
PRIMARY KEY (EmployeeID),
FOREIGN KEY (EmployeeID) REFERENCES User (EmployeeID)
);
```

The above SQL schema defines a "Front_desk_operator" table with two columns: EmployeeID and Address. EmployeeID is the primary key and also a foreign key referencing the User table's EmployeeID column. The table is used to store information about front desk operators, including their unique employee ID and address.

➔ Schema for Data Entry Operator

```
Data_entry_operator (
EmployeeID varchar(255) NOT NULL,
Address varchar(255) NOT NULL,
PRIMARY KEY (EmployeeID),
FOREIGN KEY (EmployeeID) REFERENCES User (EmployeeID)
);
```

The above SQL schema defines a table named "Data_entry_operator" with two columns: "EmployeeID" and "Address". The "EmployeeID" column is a primary key, ensuring uniqueness of each employee record. The table also has a foreign key constraint that references the "EmployeeID" column in the "User" table, ensuring data integrity between the two tables.

➔ Schema for Database Administrator

```
Database_administrator (
EmployeeID varchar(255) NOT NULL,
Address varchar(255) NOT NULL,
PRIMARY KEY (EmployeeID),
FOREIGN KEY (EmployeeID) REFERENCES User (EmployeeID)
);
```

The above SQL schema defines a table named "Database_administrator" with columns EmployeeID and Address. EmployeeID is the primary key and cannot be null. The table also has a foreign key constraint that references the EmployeeID column in the User table. The Address column cannot be null.

➔ Schema for Physician

```
Physician (
PhysicianID varchar(255) NOT NULL,
Position varchar(255) NOT NULL,
PRIMARY KEY (PhysicianID),
FOREIGN KEY (PhysicianID) REFERENCES User (EmployeeID)
);
```

The above SQL schema defines a table named "Physician" with columns PhysicianID and Position. PhysicianID is the primary key and cannot be null. The table also has a foreign key constraint that references the EmployeeID column in the User table. The Position column cannot be null.

➔ Schema for Patient

```
Patient (
Patient_SSN int NOT NULL,
Patient_Name varchar(255) NOT NULL,
Address varchar(255) NOT NULL,
Age int NOT NULL,
Gender varchar(255) NOT NULL,
Phone varchar(255) NOT NULL,
Email varchar(255) NOT NULL,
Status varchar(255) NOT NULL,
```

InsuranceID int NOT NULL,
PRIMARY KEY (Patient_SSN),
);

The above SQL schema defines a table named "Patient" with columns Patient_SSN, Patient_Name, Address, Age, Gender, Phone, Email, Status, and InsuranceID. Patient_SSN is the primary key and cannot be null. All other columns also cannot be null. The table stores information related to patients, including personal details, contact information, insurance, and status.

➔ Schema for Slot

Slot (
SlotID int NOT NULL,
Start DATETIME NOT NULL,
End DATETIME NOT NULL,
PRIMARY KEY (SlotID),
);

The above SQL schema defines a table named "Slot" with columns SlotID, Start, and End. SlotID is the primary key and cannot be null. Start and End columns store the start and end date/time for a particular slot, respectively. The table can be used to store the time slots for appointments or other scheduled events.

➔ Schema for Department

Department (
DepartmentID int NOT NULL,
Dep_Name varchar(255) NOT NULL,
Head varchar(255) NOT NULL,
PRIMARY KEY (DepartmentID),
FOREIGN KEY (Head) REFERENCES Physician (PhysicianID)
);

The above SQL schema defines a table named "Department" with columns DepartmentID, Dep_Name, and Head. DepartmentID is the primary key and cannot be null. The Head column is a foreign key that references the PhysicianID column in the Physician table. The table stores information related to departments, including department name and the physician who heads the department.

➔ Schema for Treatment Description

Treatement_Description (

```
Treatment_DescriptionID int NOT NULL,
Desc_Name varchar(255) NOT NULL,
Cost int NOT NULL,
PRIMARY KEY (Treatment_DescriptionID)
);
```

The above SQL schema defines a table named "Treatement_Description" with columns Treatment_DescriptionID, Desc_Name, and Cost. Treatment_DescriptionID is the primary key and cannot be null. The table stores information related to the different treatment descriptions, including the description name and cost. The table can be used to store the treatment options offered by a hospital or clinic.

➔ Schema for Affiliated With

```
Affiliated_with (
PhysicianID varchar(255) NOT NULL,
Department int NOT NULL,
PRIMARY KEY (PhysicianID, Department),
FOREIGN KEY (PhysicianID) REFERENCES Physician (PhysicianID),
FOREIGN KEY (Department) REFERENCES Department (DepartmentID)
);
```

The above SQL schema defines a table named "Affiliated_with" with columns PhysicianID and Department. PhysicianID and Department form a composite primary key and cannot be null. The table also has two foreign key constraints, where PhysicianID references the Physician table's PhysicianID column, and Department references the Department table's DepartmentID column. The table stores the relationship between physicians and departments they are affiliated with.

➔ Schema for Room

```
Room (
RoomID int NOT NULL,
Unavailable boolean NOT NULL,
Room_type int NULL,
PRIMARY KEY (RoomID),
);
```

The above schema describes a table named "Room" with columns for RoomID, Unavailable status, and Room_type. The RoomID column is the primary key, ensuring each room has a unique identifier. The Unavailable column indicates if the room is currently unavailable, and the Room_type column specifies the type of room.

➔ Schema for Medication

Medication (
MedicationID int NOT NULL,
Medication_Name varchar(255) NOT NULL,
Brand varchar(255) NOT NULL,
Description varchar(255) NOT NULL,
PRIMARY KEY (MedicationID),
);

The above SQL schema defines a table named "Medication" with columns MedicationID, Medication_Name, Brand, and Description. MedicationID is the primary key and cannot be null. The table stores information related to medications, including medication name, brand, and description. The table can be used to store the different medications prescribed to patients in a hospital or clinic.

➔ Schema for Appointment

Appointment (
AppointmentID int NOT NULL,
Patient_SSN int NOT NULL,
PhysicianID varchar(255) NOT NULL,
Date DATE NOT NULL,
SlotID int NOT NULL,
PRIMARY KEY (AppointmentID),
FOREIGN KEY (Patient_SSN) REFERENCES Patient (Patient_SSN),
FOREIGN KEY (PhysicianID) REFERENCES Physician (PhysicianID),
FOREIGN KEY (SlotID) REFERENCES Slot (SlotID)
);

The SQL schema defines a table named "Appointment" with columns AppointmentID, Patient_SSN, PhysicianID, Date, and SlotID. AppointmentID is the primary key and cannot be null. The table also has foreign key constraints, where Patient_SSN references the Patient table's Patient_SSN column, PhysicianID references the Physician table's PhysicianID column, and SlotID references the Slot table's SlotID column. The table stores information related to appointments, including patient, physician, date, and slot.

➔ Schema for Prescribes Medication

Prescribes_Medication (
PhysicianID varchar(255) NOT NULL,

```
Patient_SSN int NOT NULL,
MedicationID int NOT NULL,
Date DATETIME NOT NULL,
AppointmentID int NULL,
Dose varchar(255) NOT NULL,
PRIMARY KEY (PhysicianID, Patient_SSN, MedicationID, Date),
FOREIGN KEY (PhysicianID) REFERENCES Physician (PhysicianID),
FOREIGN KEY (Patient_SSN) REFERENCES Patient (Patient_SSN),
FOREIGN KEY (MedicationID) REFERENCES Medication (MedicationID),
FOREIGN KEY (AppointmentID) REFERENCES Appointment
(AppointmentID)
);
```

The above SQL schema defines a table named "Prescribes_Medication" with columns PhysicianID, Patient_SSN, MedicationID, Date, AppointmentID, and Dose. The primary key is a composite key consisting of PhysicianID, Patient_SSN, MedicationID, and Date. The table also has foreign key constraints, where PhysicianID references the Physician table's PhysicianID column, Patient_SSN references the Patient table's Patient_SSN column, MedicationID references the Medication table's MedicationID column, and AppointmentID references the Appointment table's AppointmentID column. The table stores information related to medications prescribed by physicians, including physician, patient, medication, date, and dose.

➔ Schema for Stay

```
Stay (
StayID int NOT NULL AUTO_INCREMENT,
Patient_SSN int NOT NULL,
RoomID int NOT NULL,
Start DATETIME,
End DATETIME,
PRIMARY KEY (StayID),
FOREIGN KEY (Patient_SSN) REFERENCES Patient (Patient_SSN),
FOREIGN KEY (RoomID) REFERENCES Room (RoomID)
);
```

The above SQL schema defines a table named "Stay" with columns StayID, Patient_SSN, RoomID, Start, and End. The StayID column is an auto-incremented primary key. The table also has foreign key constraints, where Patient_SSN references the Patient table's Patient_SSN column and RoomID references the Room table's RoomID column. The table stores

information about a patient's stay, including their SSN, the room they are staying in, and the start and end dates of their stay.

➔ Schema for Treatment

```
Treatment (
TreatmentID int NOT NULL AUTO_INCREMENT,
Patient_SSN int NOT NULL,
Treatment_DescriptionID int NOT NULL,
SlotID int NULL,
PhysicianID varchar(255) NOT NULL,
Date DATE NULL,
PRIMARY KEY (TreatmentID),
FOREIGN KEY (Patient_SSN) REFERENCES Patient (Patient_SSN),
FOREIGN KEY (Treatment_DescriptionID) REFERENCES
Treatment_Description (Treatment_DescriptionID),
FOREIGN KEY (PhysicianID) REFERENCES Physician (PhysicianID),
FOREIGN KEY (SlotID) REFERENCES Slot (SlotID)
);
```

The above schema describes a table named "Treatment" with columns for TreatmentID, Patient_SSN, Treatment_DescriptionID, SlotID, PhysicianID, and Date. The TreatmentID column is the primary key and is automatically incremented for each new treatment. The table contains foreign keys that reference the Patient, Treatment_Description, Physician, and Slot tables, ensuring that each treatment is associated with a patient, treatment description, physician, and time slot. The Date column stores the date the treatment was performed.

➔ Schema for Test

```
Test (
TestID int NOT NULL,
Test_Name varchar(255) NOT NULL,
Cost int NOT NULL,
PRIMARY KEY (TestID)
);
```

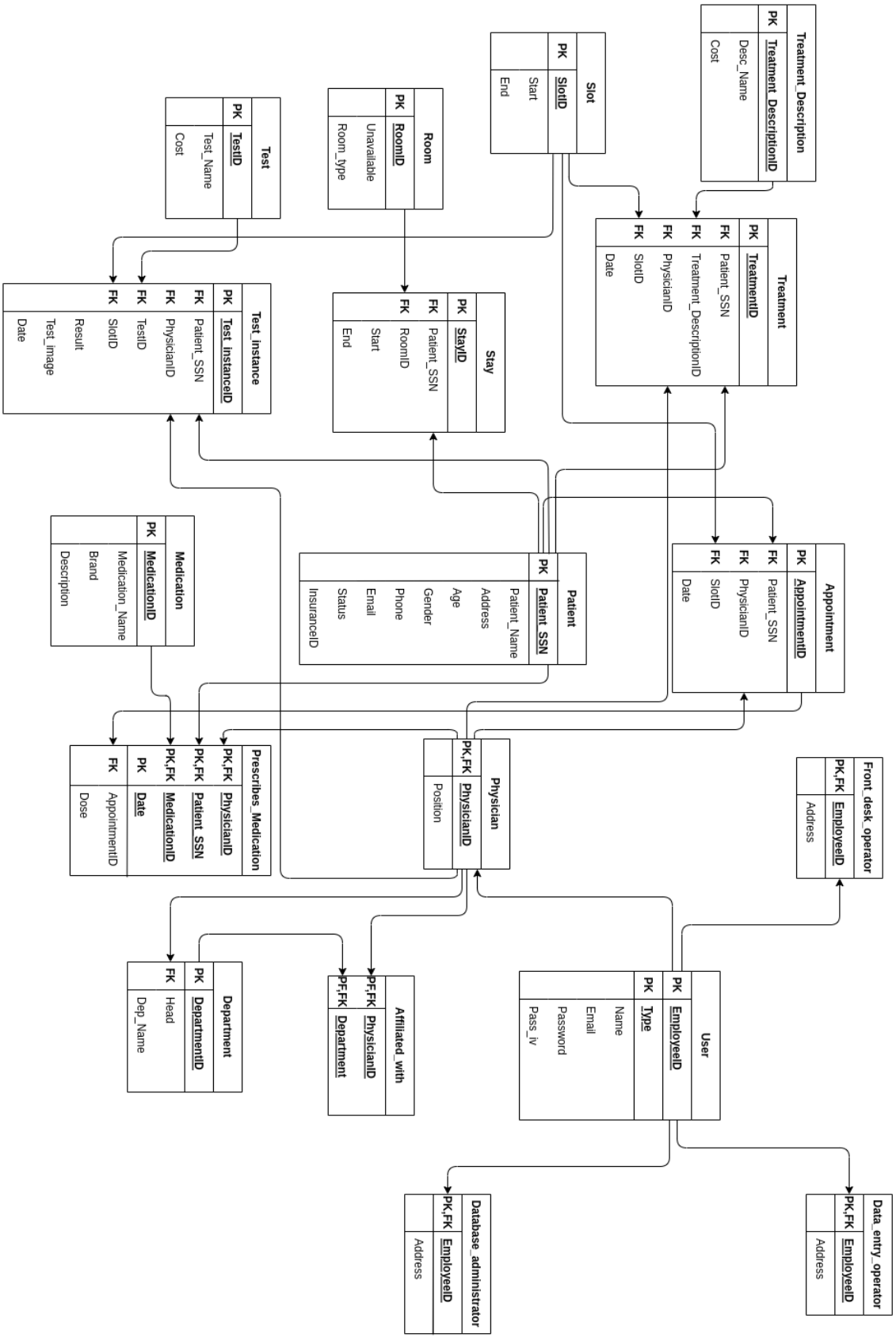
The above SQL schema "Test" has columns for TestID, Test_Name, and Cost. TestID is the primary key for the table. The table is likely used to store information about medical tests that can be performed for patients, with Test_Name being the name of the test and Cost being the cost of the test.

➔ Schema for Test Instance

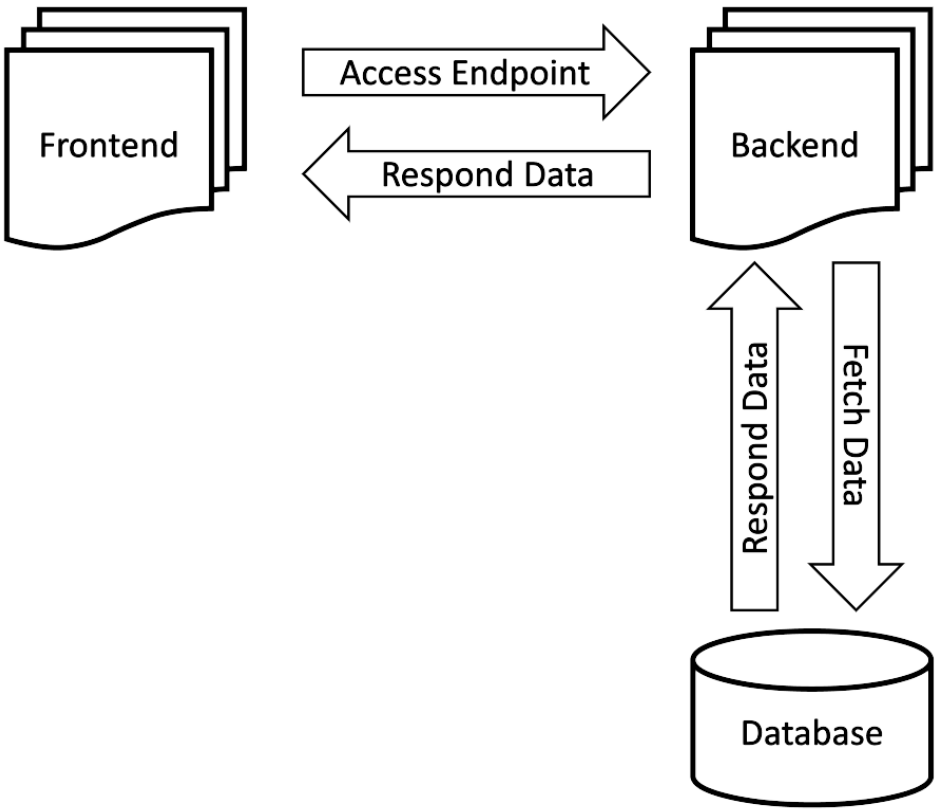
```
Test_instance (
Test_instanceID int NOT NULL AUTO_INCREMENT,
Patient_SSN int NOT NULL,
PhysicianID varchar(255) NOT NULL,
TestID int NOT NULL,
SlotID int NULL,
Result varchar(255) NULL,
Test_image LONGBLOB NULL,
Date DATE NULL,
PRIMARY KEY (Test_instanceID),
FOREIGN KEY (Patient_SSN) REFERENCES Patient (Patient_SSN),
FOREIGN KEY (PhysicianID) REFERENCES Physician (PhysicianID),
FOREIGN KEY (TestID) REFERENCES Test (TestID),
FOREIGN KEY (SotID) REFERENCES Slot (SlotID),
);
```

The above schema describes a table named "Test_instance" with columns for Test_instanceID, Patient_SSN, PhysicianID, TestID, SlotID, Result, Test_image, and Date. The Test_instanceID column is the primary key and is automatically incremented for each new test instance. The table contains foreign keys that reference the Patient, Physician, Test, and Slot tables, ensuring that each test instance is associated with a patient, physician, test, and time slot. The Result column stores the result of the test, and the Test_image column stores the image of the test if applicable. The Date column stores the date the test was taken.

➢ ENTITY RELATIONSHIP DIAGRAM:



➢ TRIGGERS AND WORKFLOWS:



➔ Creating User and Doctor Entity

User:

‘INSERT INTO User (EmployeeID, Name, Email, Type, Password, Pass_iv)’ + ‘VALUES ('${entity.employeeid}', '${entity.name}', '${entity.email}', ${entity.user_type}, '${pass}', '${iv}');’

The above SQL query inserts a new user into the database with their employee ID, name, email, user type, encrypted password, and password initialization vector (pass_iv).

Doctor:

‘INSERT INTO Physician (PhysicianID, Position)’ + ‘VALUES ('${entity.physicianid}', '${entity.position}');’

‘INSERT INTO Affiliated_with (PhysicianID, Department)’ + ‘VALUES ('${entity.physicianid}', ${entity.department});’

The above SQL query inserts a new doctor into the database with their physician ID and position. Additionally, it creates an entry in the "Affiliated_with" table that links the physician to their department, with the physician ID as a foreign key.

➔ Scheduling Appointment

Selecting Doctor:

‘SELECT Name, PhysicianID AS id, Position FROM Physician INNER JOIN User ON Physician.PhysicianID = User.EmployeeID;’

The above query is used to retrieve the details of doctors (physicians) from the database. It uses an INNER JOIN on the Physician and User tables to combine their attributes. The resulting columns are 'Name' and 'Position' from the User table and 'PhysicianID' as 'id' from the Physician table.

Selecting Patient:

‘SELECT Patient_Name, Patient_SSN AS id, Age, Gender FROM Patient;’

The above query is used to retrieve the details of patients from the database. It selects 'Patient_Name', 'Patient_SSN' as 'id', 'Age' and 'Gender' from the 'Patient' table.

Finding Slots:

'SELECT SlotID, Start, End FROM Slot WHERE SlotID NOT IN’ +
‘(SELECT SlotID FROM Appointment WHERE Date = '${req_date}' and PhysicianID = '${doc_id}')’ + ‘AND SlotID NOT IN (SELECT SlotID FROM Treatment WHERE Date = '${req_date}' and PhysicianID = '${doc_id}');’

The above query is used to find available time slots for a physician on a given date. It selects 'SlotID', 'Start', and 'End' from the 'Slot' table. The query checks the availability of the slots by ensuring that they are not already occupied by an appointment or treatment on the same date and physician.

Scheduling Appointment:

‘SELECT Patient.Patient_SSN, Email FROM Appointment NATURAL JOIN Patient WHERE Date = '${date}' AND SlotID = ${slotID} AND PhysicianID = '${docID}';’

This query is used to schedule an appointment for a patient with a selected physician on a specific date and time slot. It selects the 'Patient_SSN' and 'Email' from the 'Appointment' table, where the 'Date', 'SlotID', and 'PhysicianID' match the input parameters. It also joins with the 'Patient' table to retrieve the 'Patient_SSN' and 'Email'.

➔ Prescribing Treatment, Test and Medication:

Treatment:

‘SELECT User.Name as 'Physician Name', Desc_Name as 'Treatment Name', Date FROM Treatment_Description NATURAL JOIN Treatment NATURAL JOIN Patient NATURAL JOIN Physician NATURAL JOIN Slot,

User WHERE User.EmployeeID = PhysicianID and Patient_SSN =' + patient_id + ';'

The above query retrieves the treatment details of a patient, including the physician's name, treatment name, and treatment date. It uses a NATURAL JOIN on multiple tables and selects specific attributes from those tables to combine the necessary information. The query also joins with the 'User' table to retrieve the physician's name.

Test:

'SELECT Test_Name, Result, Date, Age, Gender FROM Test_instance NATURAL JOIN Test NATURAL JOIN Patient WHERE Patient_SSN = ${patient_id};'

The above query retrieves the test details of a patient, including the test name, result, date, age, and gender. It uses a NATURAL JOIN on the 'Test_instance', 'Test', and 'Patient' tables to combine their attributes and retrieve the necessary information. The query selects specific attributes from these tables and joins them based on the patient's SSN.

Medication:

'SELECT User.Name as 'Physician Name', Medication_Name as 'Medication Name', Brand as Brand, Date, AppointmentID as 'Appointment ID' FROM Prescribes_Medication NATURAL JOIN Medication NATURAL JOIN Physician NATURAL JOIN Patient, User WHERE Patient_SSN = ${patient_id} and User.EmployeeID = PhysicianID;'

The above query retrieves the medication details prescribed to a patient, including the physician's name, medication name, brand, date, and appointment ID. It uses a NATURAL JOIN on multiple tables and selects specific attributes from those tables to combine the necessary information. The query also joins with the 'User' table to retrieve the physician's name and filters the results based on the patient's SSN.

➔ Admitting Patient

Selecting Patient:

'SELECT Patient_SSN AS id, Patient_Name, Address, Age, Gender FROM Patient WHERE Status='not admitted';'

The above query selects patients who are not yet admitted to the hospital and retrieves their SSN, name, address, age, and gender from the Patient table.

Getting Available Room:

'SELECT RoomID AS id FROM Room WHERE Unavailable = false;'

The above query selects room IDs from the Room table where the room is currently available (i.e., Unavailable is false).

Admitting Patient:

‘INSERT INTO Stay (Patient_SSN, RoomID, Start, End) VALUES ('${patient}', ${room}, '${date}', NULL)’

The above query selects room IDs from the Room table where the room is currently available (i.e., Unavailable is false).

Updating Room Availability:

‘UPDATE Room SET Unavailable = true WHERE RoomID = ${room}’

The above query updates the Unavailable field of the Room table to true for the room that was selected for the admitted patient.

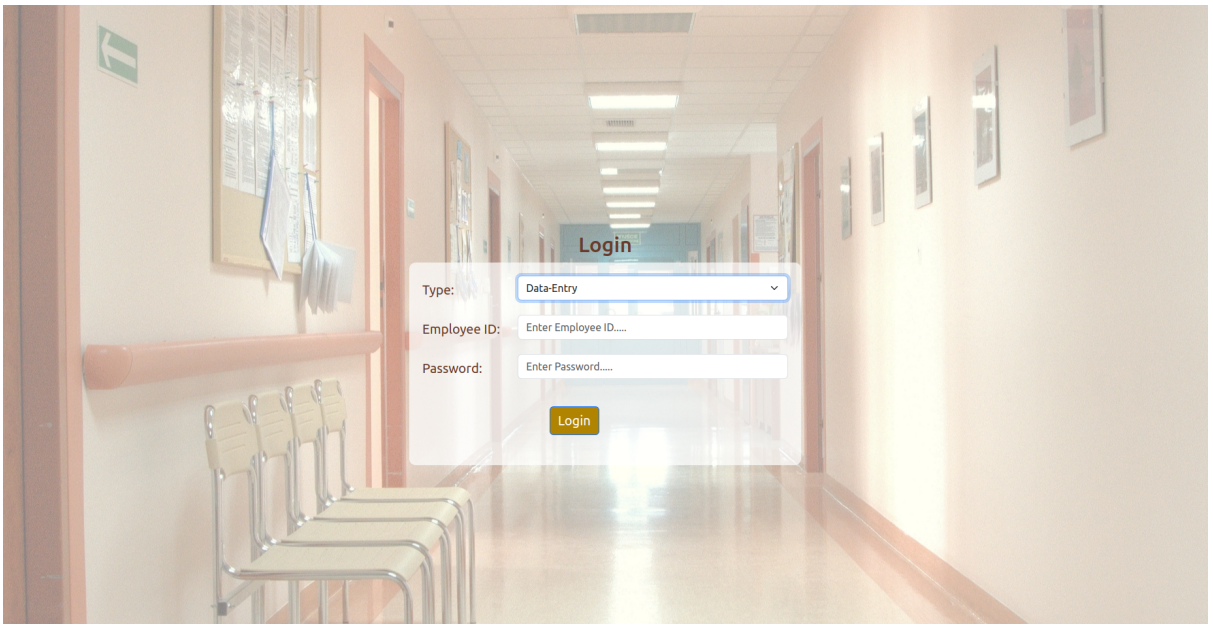
Updating Patient Status:

‘UPDATE Patient SET Status = 'admitted' WHERE Patient_SSN = '${patient}’’

The above query updates the Status field of the Patient table to 'admitted' for the patient with the specified SSN, indicating that they are now admitted to the hospital.

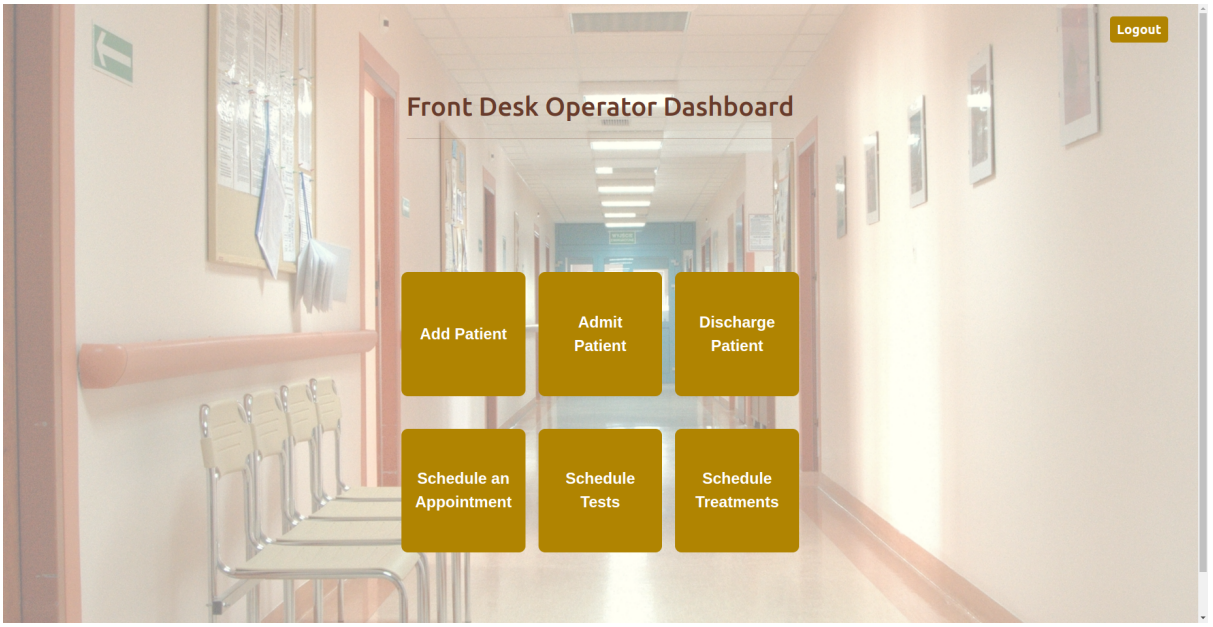
➢ WEB INTERFACE:

➔ Login Page



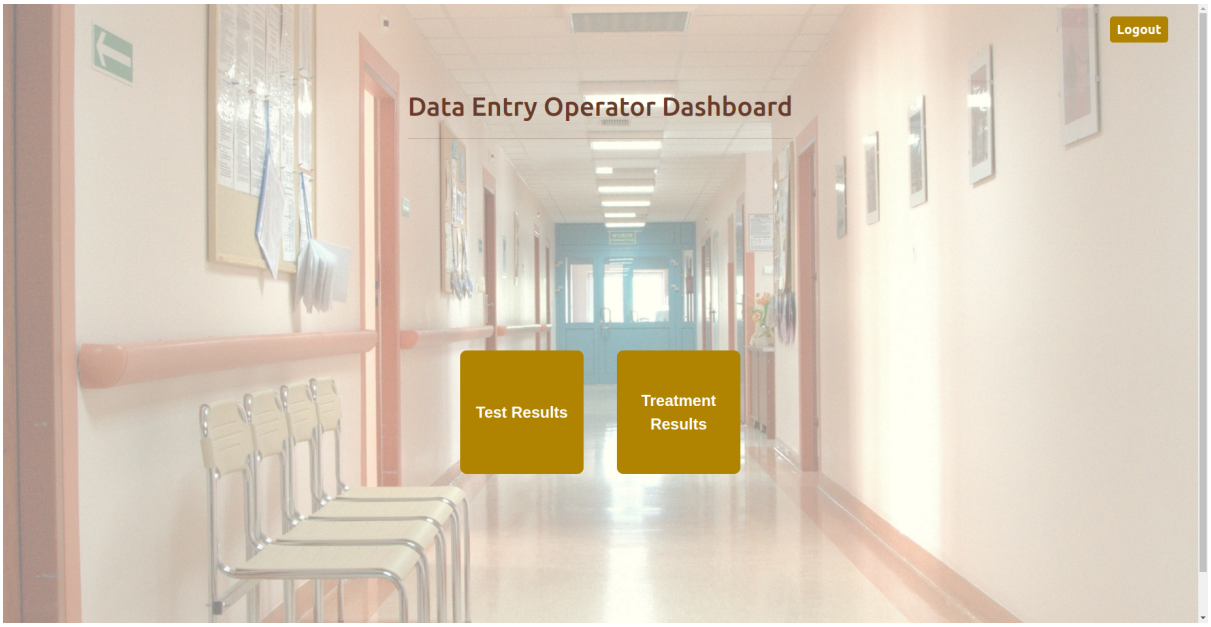
➔ Front Desk Operator Dashboard

Login as Front Desk Operator



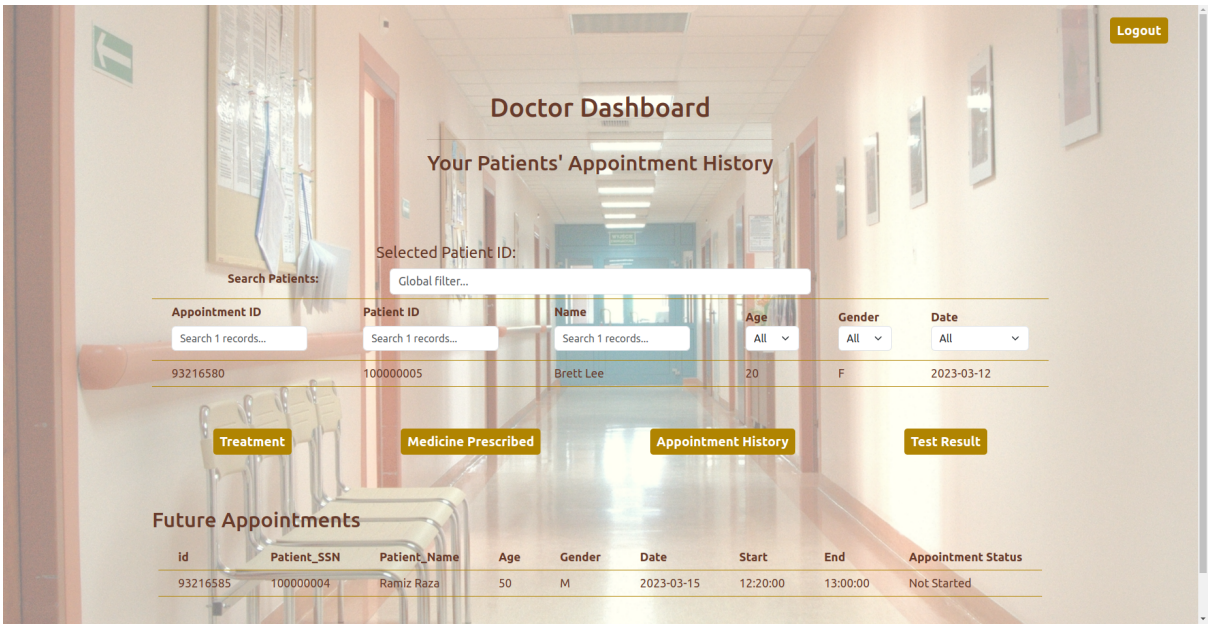
➔ Data Entry Operator Dashboard

Login as Data Entry Operator



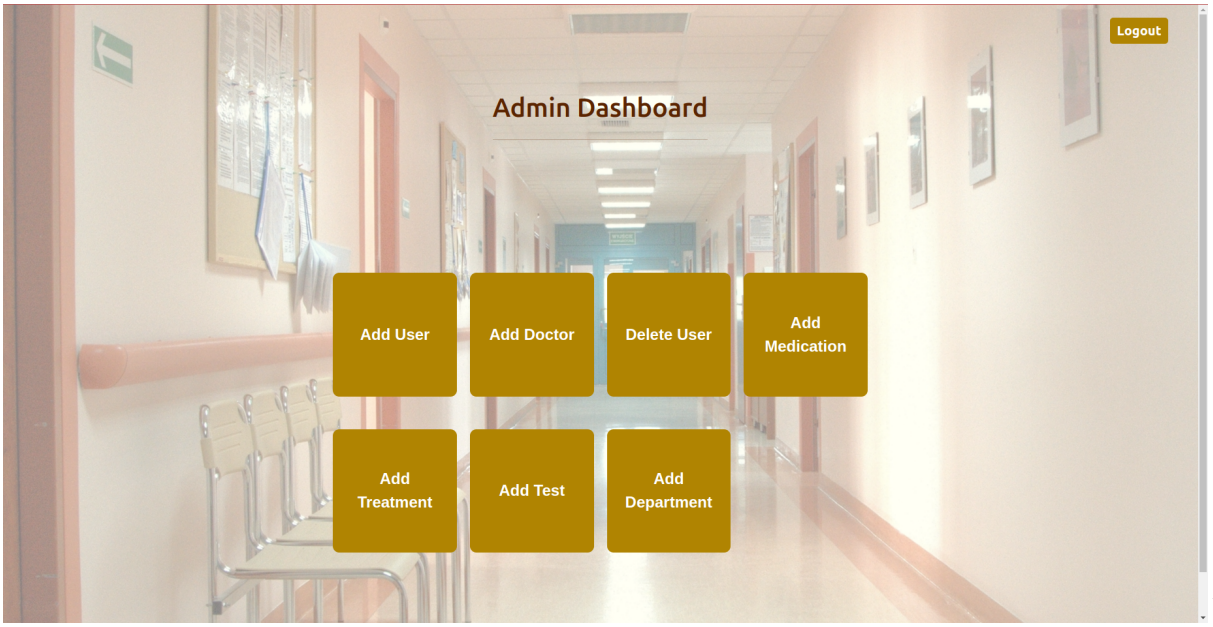
➔ Doctor Dashboard

Login as Doctor



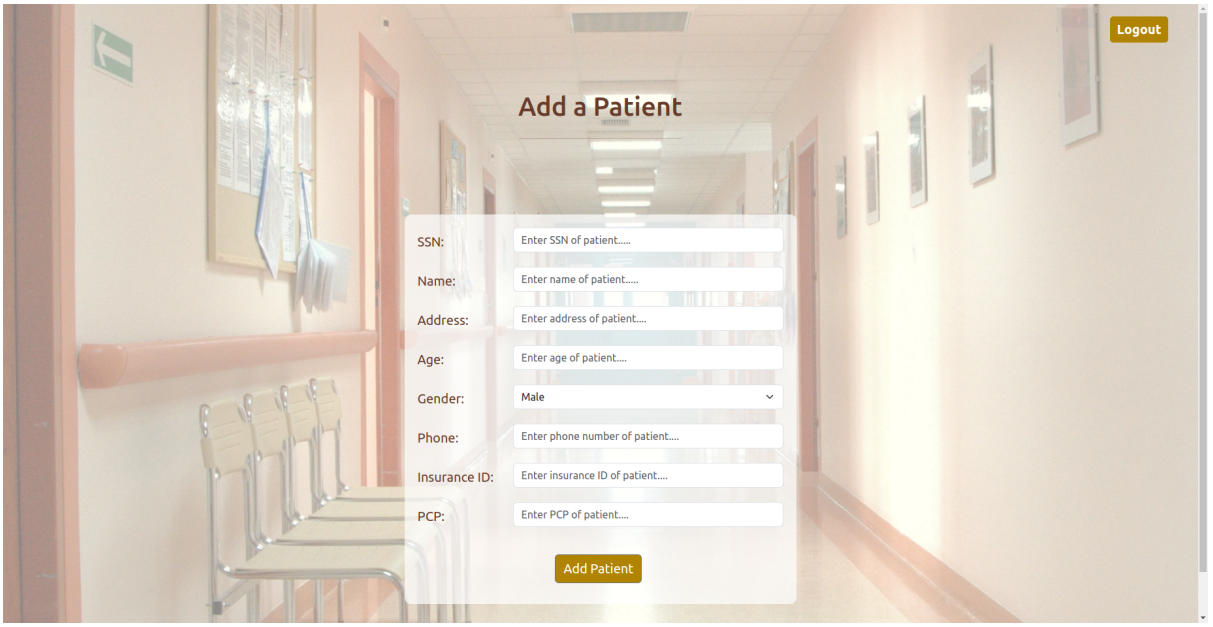
➔ Admin Dashboard

Login as Administrator



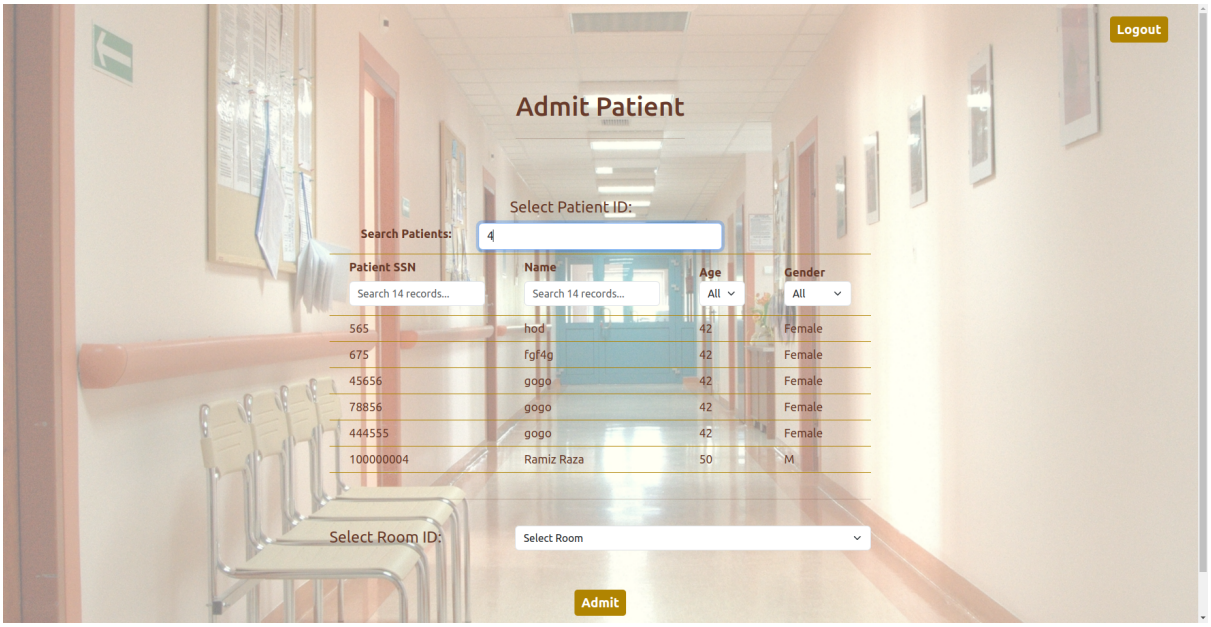
➔ Add a Patient

Home Page > Front Desk Operator > Add Patient



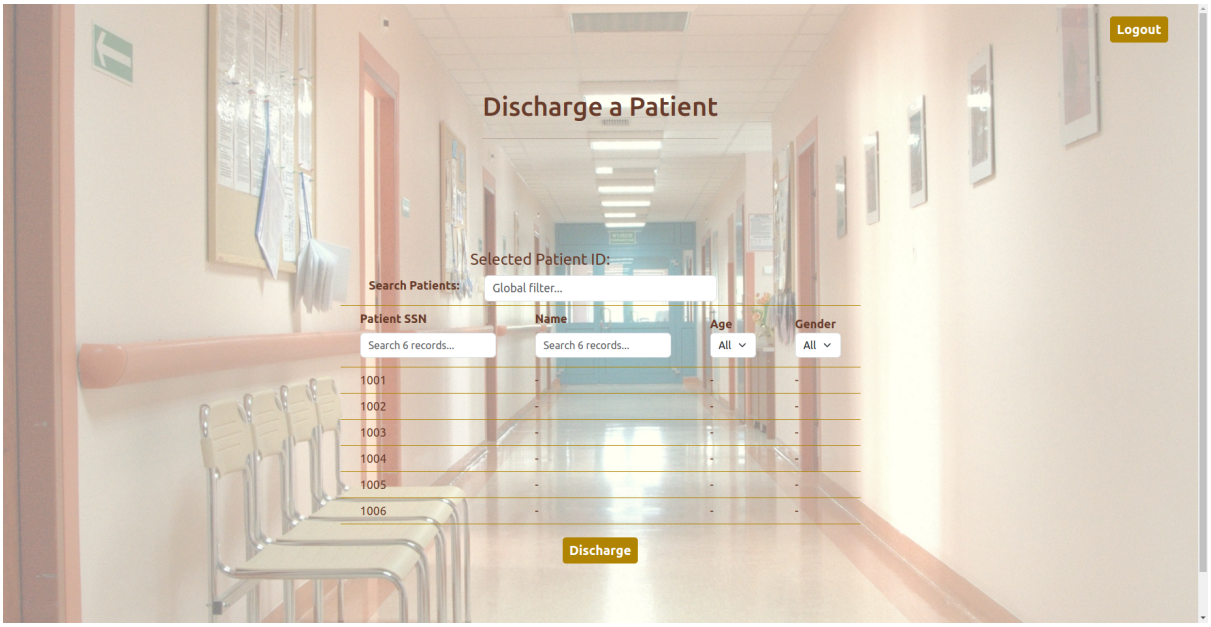
➔ Admit a Patient

Front Desk Operator > Admit Patient



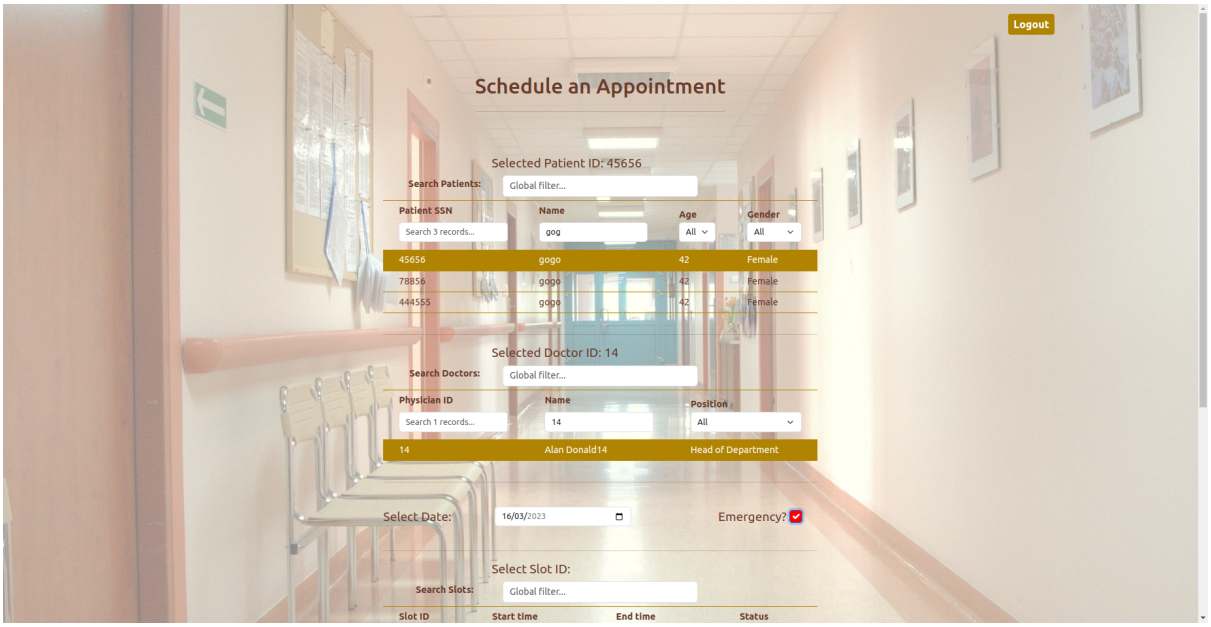
➔ Discharge a Patient

Front Desk Operator > Discharge Patient



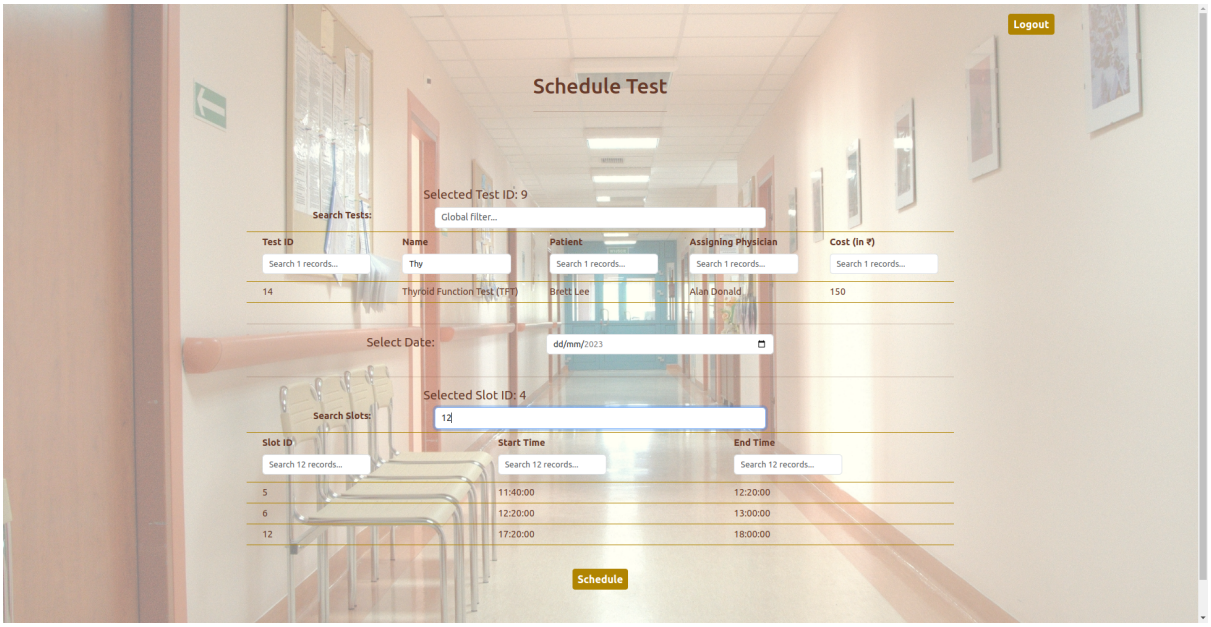
➔ Schedule an Appointment

Front Desk Operator > Schedule an Appointment



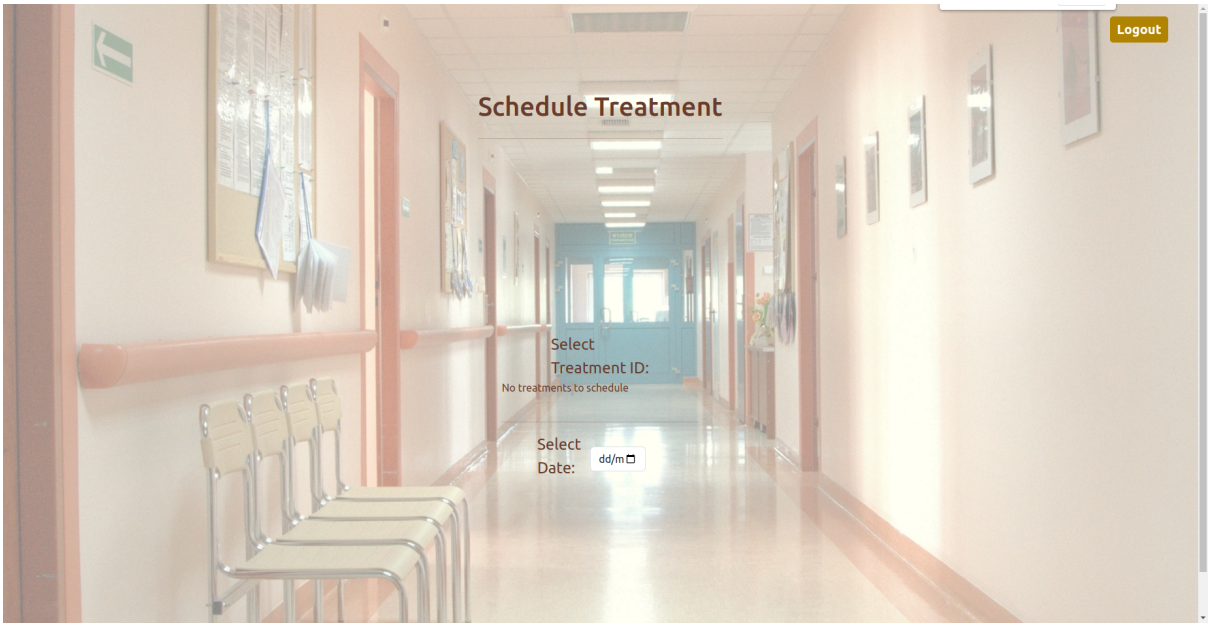
➔ Schedule Tests

Front Desk Operator > Schedule Tests



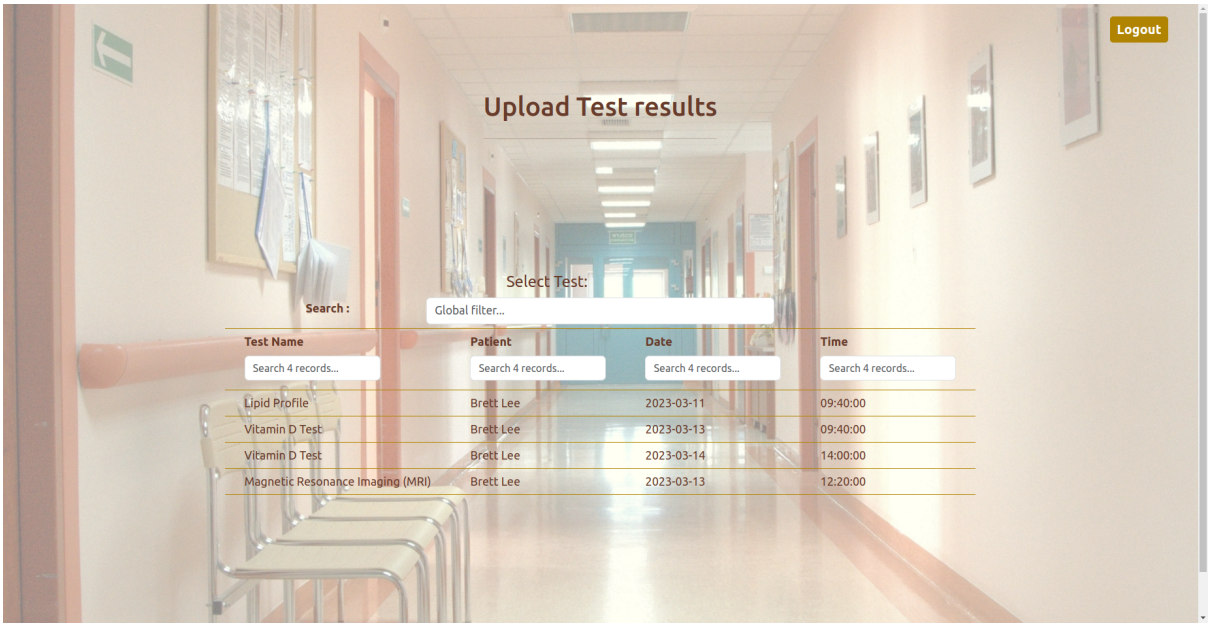
➔ Schedule Treatments

Front Desk Operator > Schedule Treatments



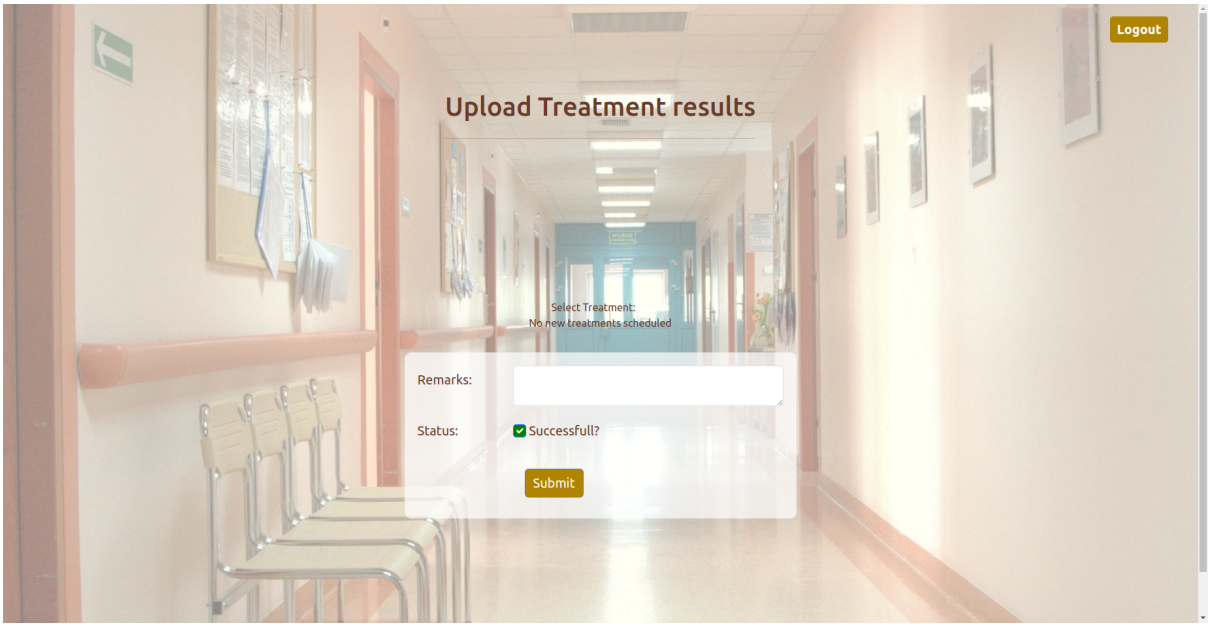
➔ Test Results

Data Entry Operator > Test Results



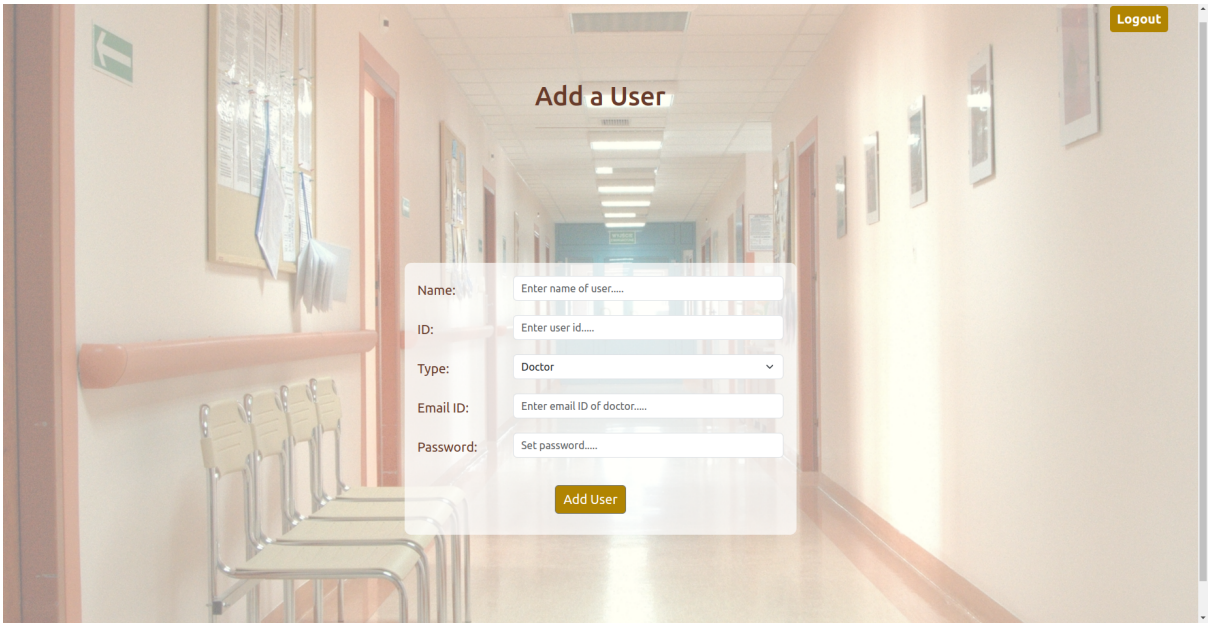
➔ Treatment Results

Data Entry Operator > Treatment Results



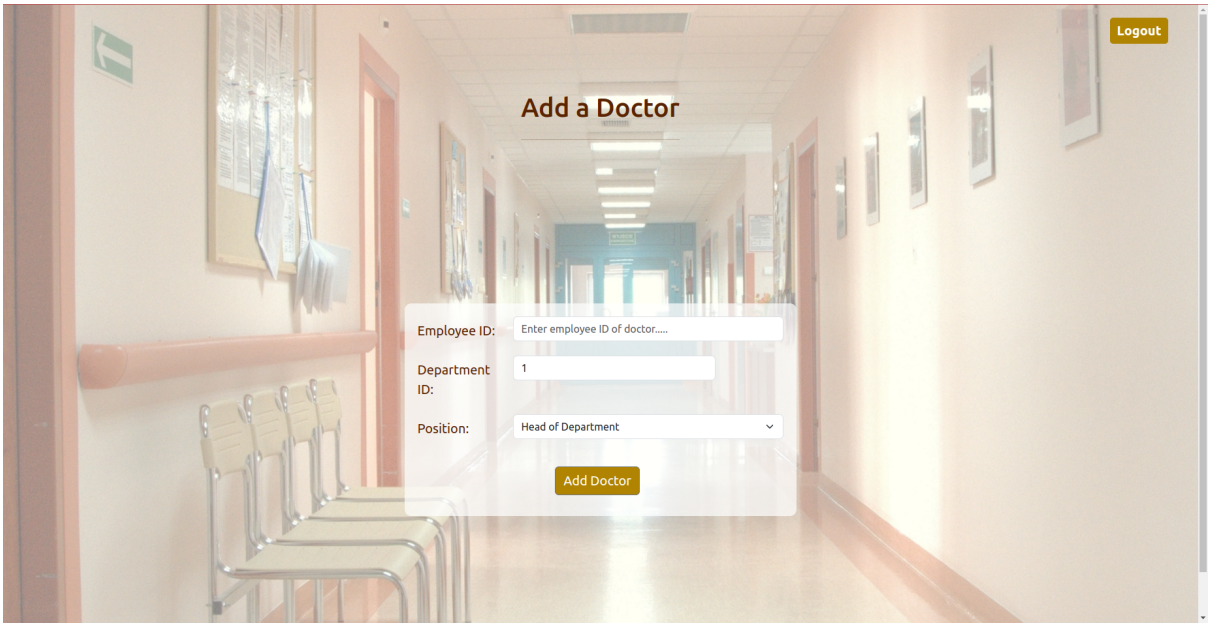
➔ Add a User

Administrator > Add User



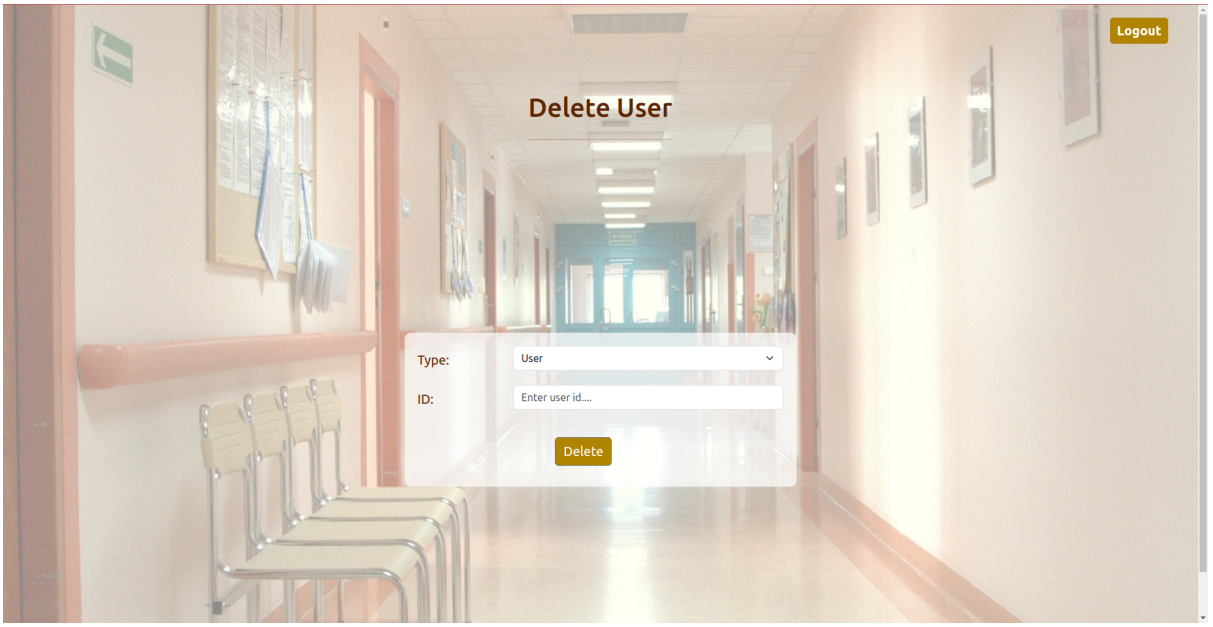
➔ Add a Doctor

Administrator > Add Doctor



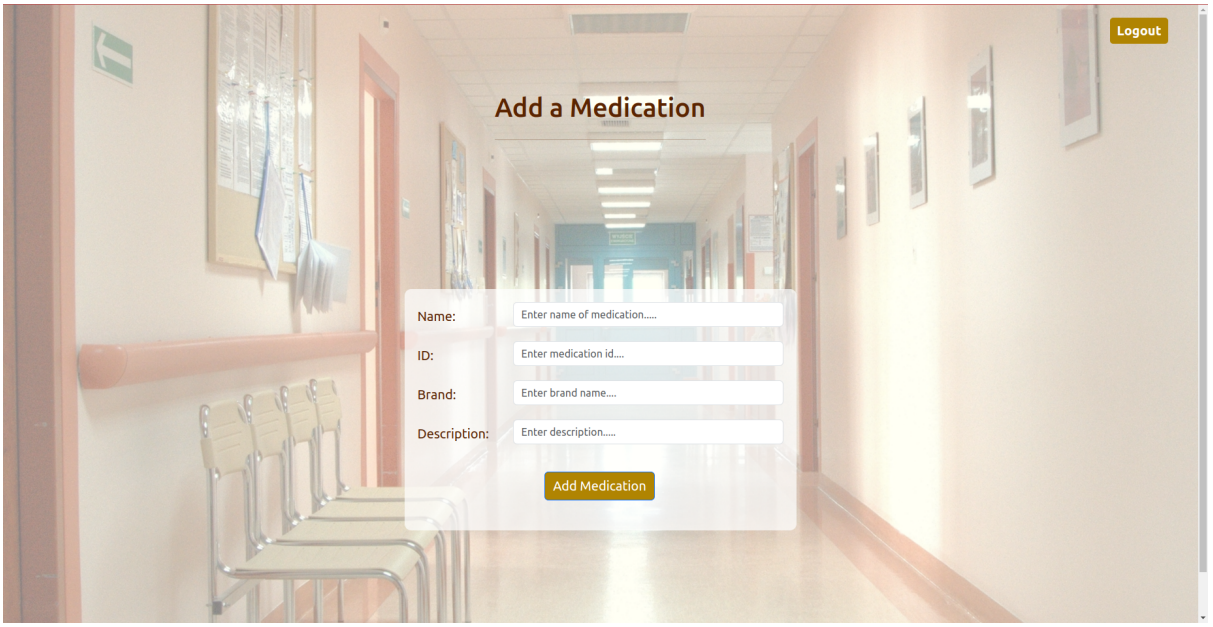
➔ Delete User

Administrator > Delete User



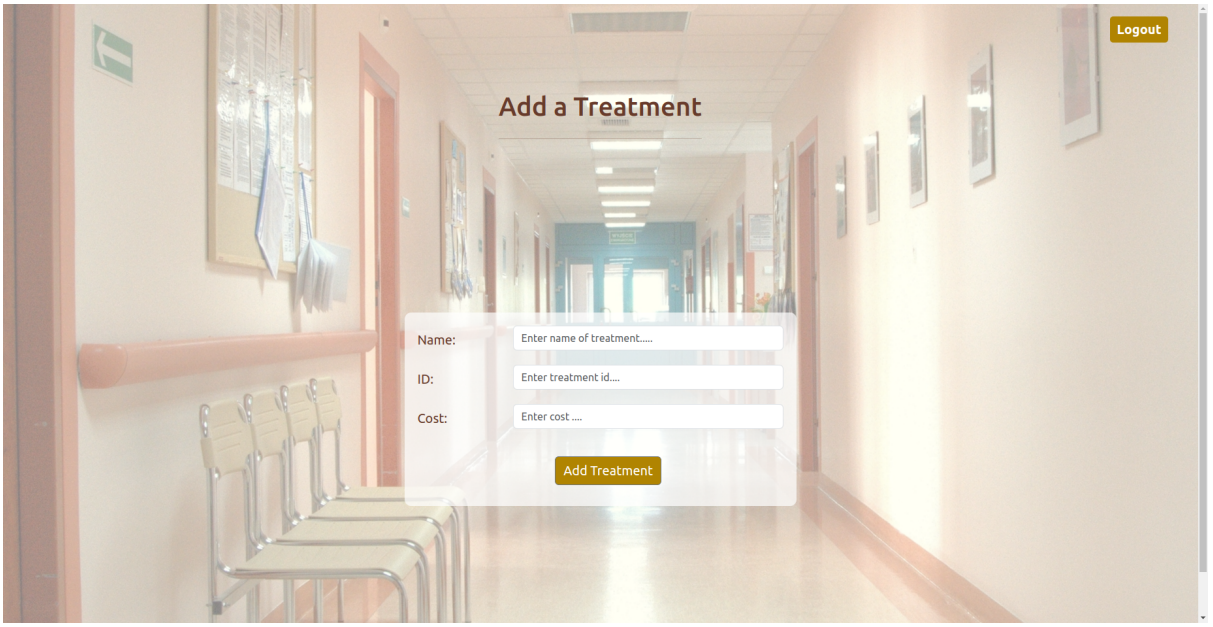
➔ Add a Medication

Administrator > Add Medication



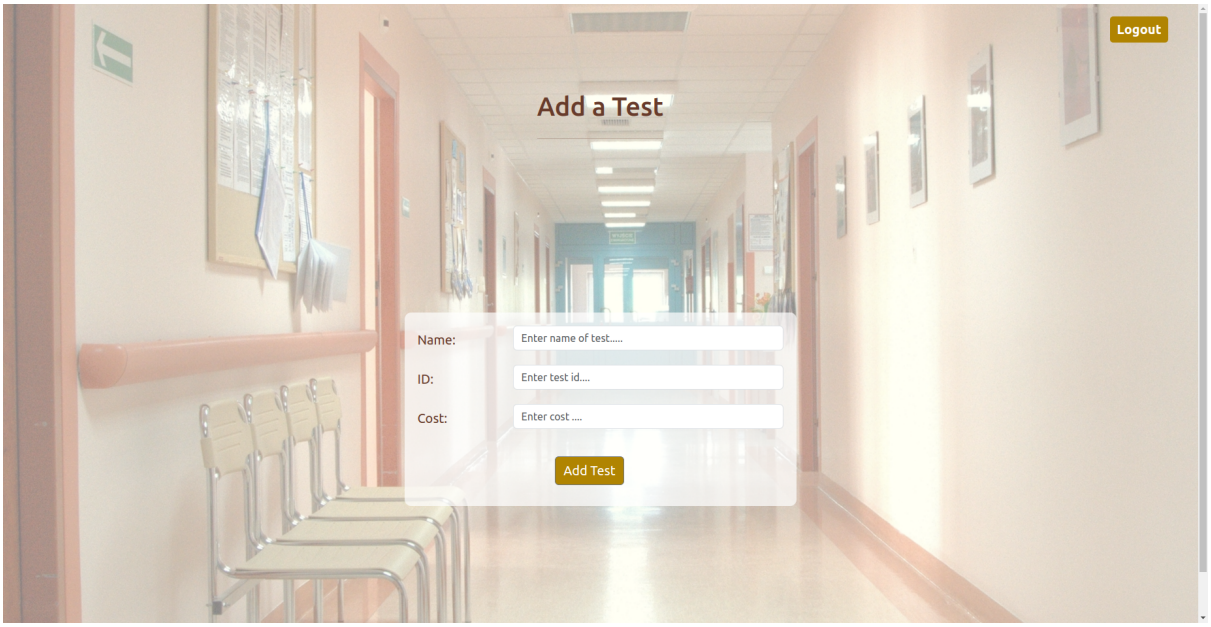
➔ Add a Treatment

Administrator > Add Treatment



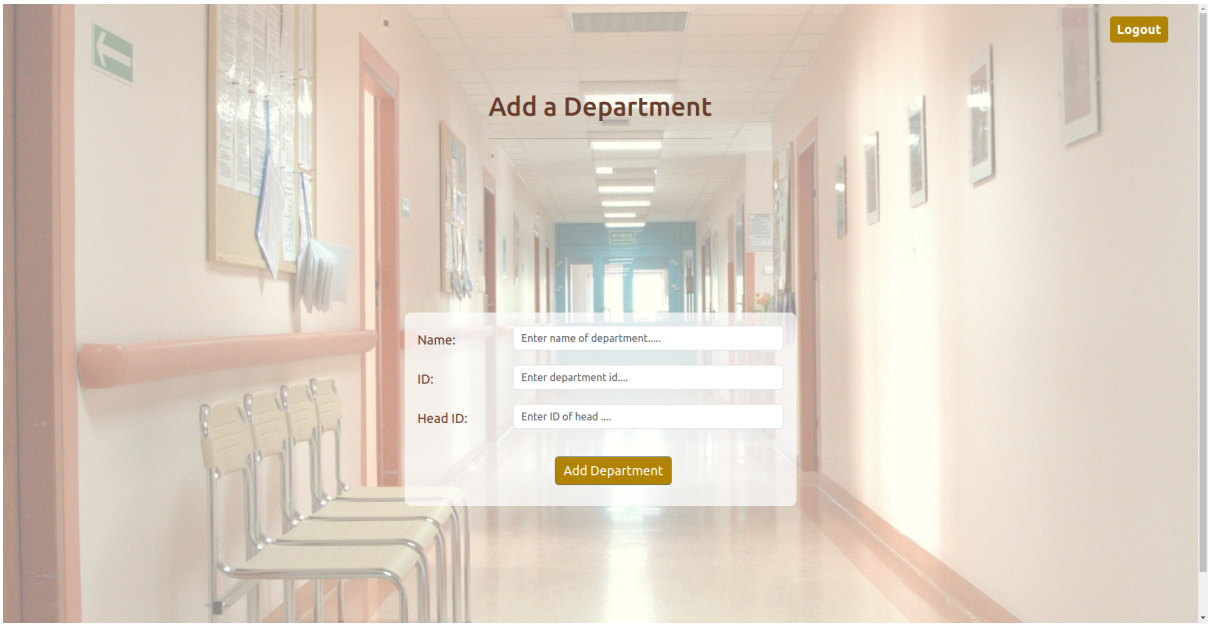
➔ Add a Test

Administrator > Add Test



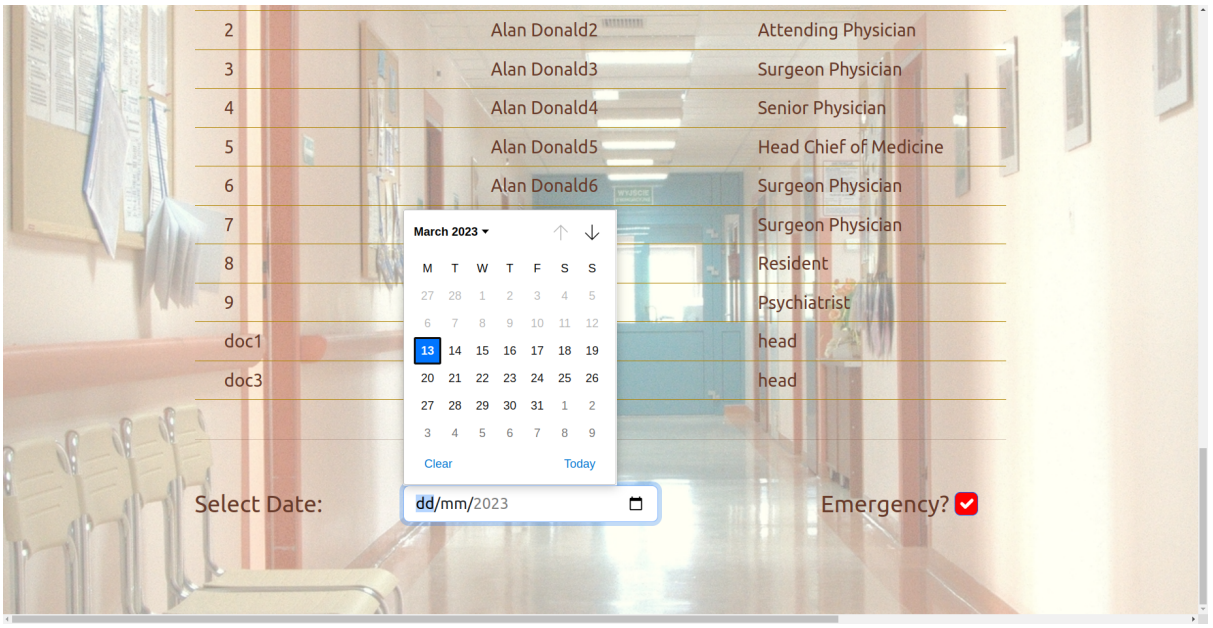
➔ Add a Department

Administrator > Add Department



➢ ADDITIONAL FUNCTIONALITIES:

➔ Implementation of Calendar and Emergency



➔ Implementation of Automated Email

Email received by Patient on Appointment Cancellation:

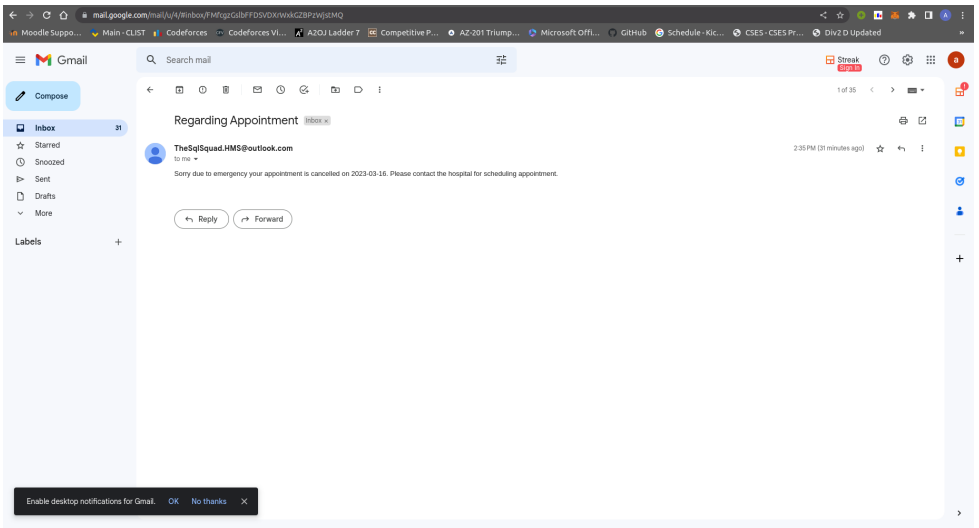
Implementation in Backend:

'SELECT Email FROM Patient NATURAL JOIN Appointment WHERE Appointment.Date = '${date}' and Appointment.SlotID = ${slotID}'

mailText = 'Sorry due to an emergency your appointment is cancelled on ${date}. Please contact the hospital for scheduling appointment.'

sendMail(patient_email,"Regarding Appointment", mailText)

Result:



Email received by Doctor on Appointment Cancellation:

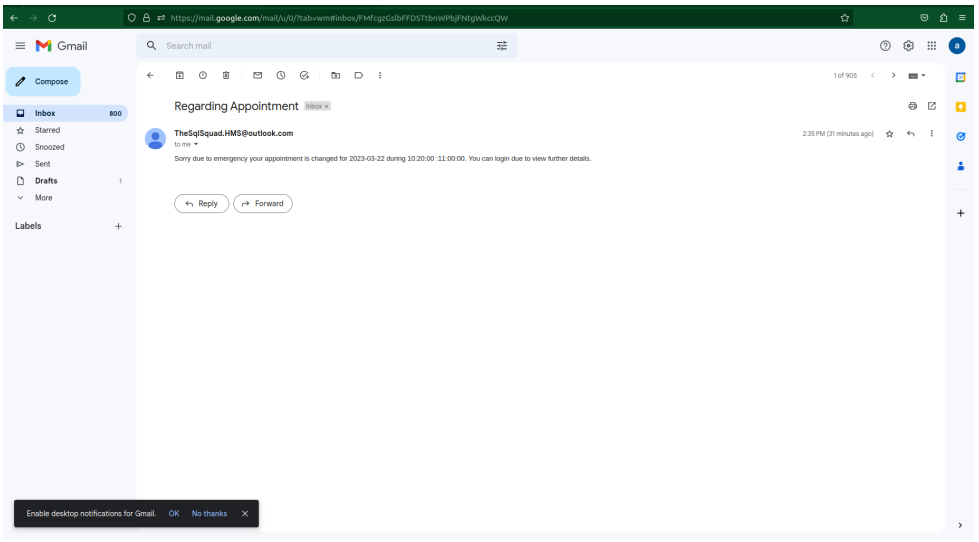
Implementation in Backend:

‘Select Email FROM Physician, User, Appointment WHERE Appointment.Date = '${date}' and Appointment.SlotID = ${slotID} and Appointment.PhysicianID = Physician.PhysicianID and Physician.PhysicianID = User.EmployeeID`

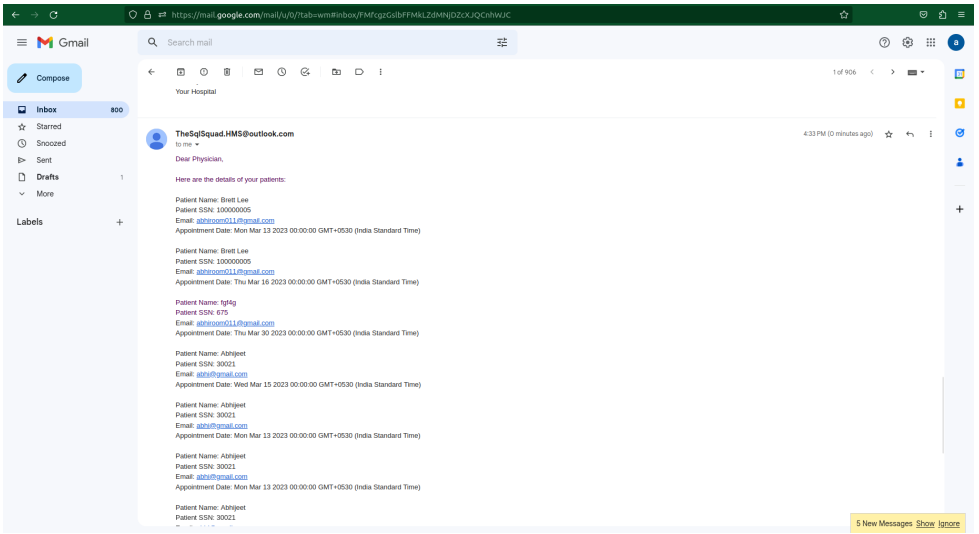
mailText2 = ‘Sorry due to an emergency your appointment is changed for ${date} during ${slotID}. You can login to view further details.’

sendMail(physician_email,"Regarding Appointment", mailText2)

Result:



Email to Doctor about Weekly Report of his/her Patients:



➔ Handling Encryption

```
const crypto = require('crypto');
const secret = 'konstantynopolitanczykowianeczka';
```

```
const encrypt = (password) => {
   const iv = Buffer.from(crypto.randomBytes(16));
   const cipher = crypto.createCipheriv('aes-256-ctr', Buffer.from(secret), iv);
   const encrypted_password = Buffer.concat([cipher.update(password),
cipher.final()]);
   return { iv: iv.toString('hex'), encrypted_password:
encrypted_password.toString('hex') };
};
const decrypt = (encryption) => {
   const decipher = crypto.createDecipheriv('aes-256-ctr', Buffer.from(secret),
Buffer.from(encryption.iv, 'hex'));
   const decrypted_password =
Buffer.concat([decipher.update(Buffer.from(encryption.encrypted_password, 'hex')),
decipher.final()]);
   return decrypted_password.toString();
};
module.exports = { encrypt, decrypt };
```

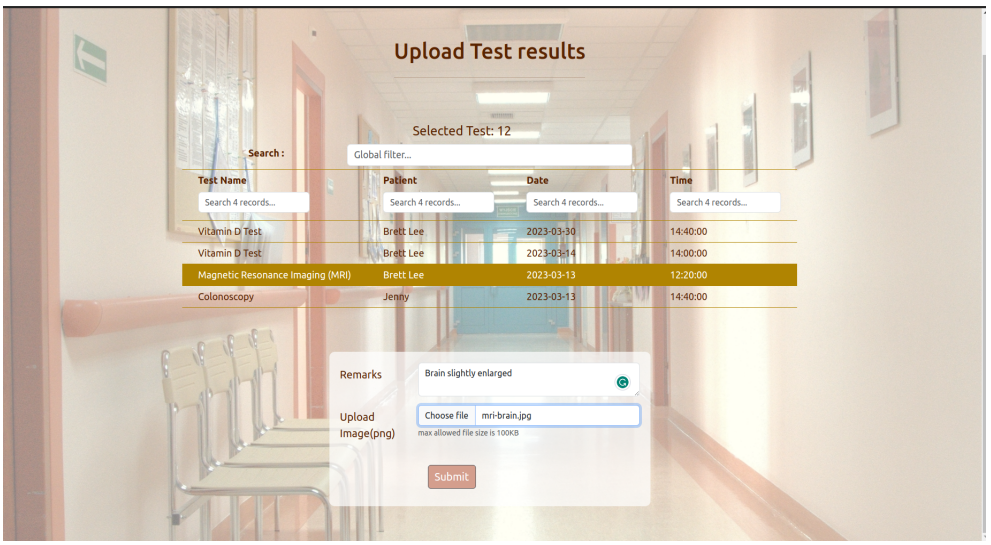
The above code is a Node.js module that exports two functions, **encrypt** and **decrypt**. These functions are used for encrypting and decrypting passwords using the **Advanced Encryption Standard (AES)** algorithm with a 256-bit key length in counter (CTR) mode.

The **encrypt** function takes a password as input and generates a random initialization vector (IV) using **crypto.randomBytes(16)**, which is a cryptographically secure random number generator provided by Node.js. The IV is then used along with a secret key to create a **Cipher** object using **crypto.createCipheriv('aes-256-ctr', Buffer.from(secret), iv)**. This cipher object is used to encrypt the password using the **update** and **final** methods, and the resulting ciphertext is returned as a hexadecimal string along with the IV.

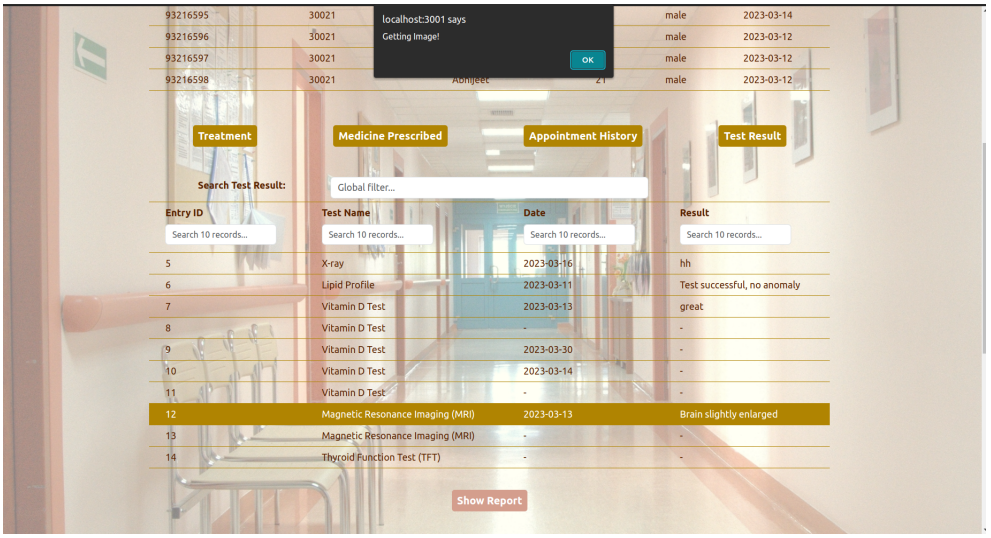
The **decrypt** function takes the output of the **encrypt** function as input, which includes the encrypted password and the IV. It uses the same secret key and IV to create a **Decipher** object using **crypto.createDecipheriv('aes-256-ctr', Buffer.from(secret), Buffer.from(encryption.iv, 'hex'))**. This decipher object is used to decrypt the password using the **update** and **final** methods, and the resulting plaintext password is returned as a string.

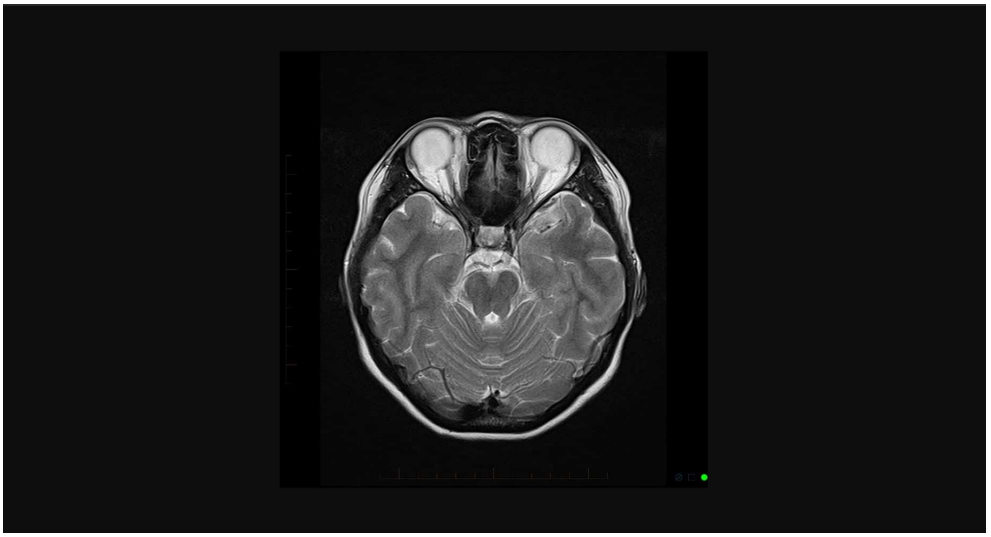
➔ Implementation of Storage and Display Image

Uploading Image (X-Ray) from Frontend:



Reviewing image from Frontend:





-----: THE END :-----