Contributors:
Hamna Inam Abro (27113)
Rafsha Rahim (26639)
Zara Masood (26928)

Submitted To:
Ms. Abeera Tariq

TABLE OF CONTENTS:

Business Scenario

HealthWave is a comprehensive hospital management system developed to streamline hospital operations and improve how data is managed. It brings together information about key hospital entities, such as departments, doctors, patients, pharmacies, prescriptions, and appointments, into a single, organized platform. By centralizing data, HealthWave ensures that essential information can be accessed, updated, and retrieved quickly and reliably, reducing errors and inefficiencies.

The system also generates real-time analytics, providing hospital administrators with a clear overview of daily operations. These insights enable informed decision-making, better resource allocation, and improved overall efficiency. HealthWave is built on a robust and adaptable database structure, allowing it to scale with the needs of the hospital while maintaining reliability and accuracy. Its goal is to support healthcare facilities in delivering better patient care by simplifying complex processes and improving data-driven operations.

Business Rules

1. Each hospital is uniquely identified by a Hospital_ID and contains multiple departments, each linked to the hospital through Hospital_id.
2. Departments are uniquely identified by an id and manage several doctors, where each doctor belongs to only one department via Department_id.
3. Doctors are identified by Doctor_ID and can handle multiple appointments, each linked to a specific patient.
4. Patients, uniquely identified by Patient_id, can book multiple appointments with different doctors.
5. Each patient may have multiple prescriptions, where each prescription contains a unique Id and specifies a Med_Name, Date, Cost, and associations with the patient (Patient_id) and pharmacy (Pharmacy_id).
6. Pharmacies, uniquely identified by Pharmacy_ID, store information about available medicines and issue prescriptions.
7. Real-time analytics will track metrics such as the number of appointments, most prescribed medications, and doctor-to-patient ratios.
8. Data integrity is enforced with mandatory relationships between entities (e.g., a prescription cannot exist without an associated patient and pharmacy).
9. Updates to doctor, patient, or prescription records cascade appropriately across the database to maintain consistency.

Key Use Cases and User Interactions

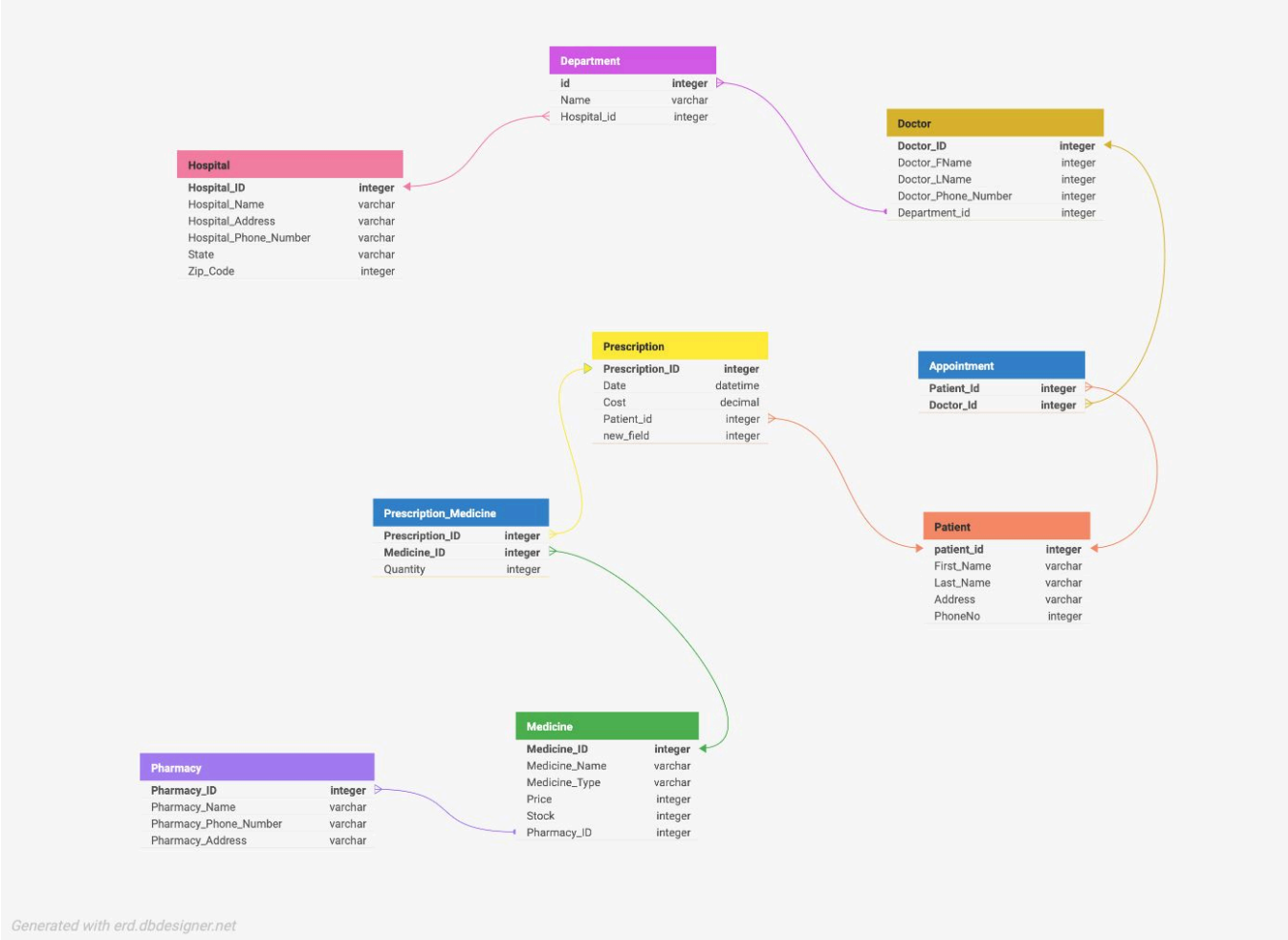
- Booking Appointments
  Patients can log into the system to view the list of available doctors and select appointment slots based on their preferences and the doctor's schedule.
- Viewing Doctor Schedules
  Real-time doctor schedules are accessible to both patients and hospital staff, enabling efficient planning and appointment management.
- Managing Pharmacy Inventory
  The system automatically updates medicine stock levels whenever prescriptions are fulfilled. Hospital administrators and pharmacy staff can monitor inventory levels to ensure the availability of critical medications.
- Accessing Insights
  Administrators can use the dashboard to track key operational metrics, such as appointment trends, patient statistics, and pharmacy performance, providing data-driven insights for decision-making.
- Managing Patient Records
  Authorized personnel can update and maintain patient records after consultations, creating an accurate and reliable repository for future use.

Example Workflow

- Appointment Booking:
  - A patient logs into the system and selects an available doctor based on their specialty and personal preference.
  - The system allows the patient to choose an appointment time that aligns with the doctor’s schedule.
- Post-Appointment Management:
  - After the consultation, doctors issue prescriptions directly within the system.
  - The Pharmacy Module stores the prescriptions and updates the Medicine Table to reflect any issued medications.
- Real-Time Monitoring and Insights:
  - The system’s Dashboard provides hospital administrators with up-to-date metrics, such as the number of booked appointments, patient check-ins, doctor utilization rates, and pharmacy inventory.
- Updating Patient Contact Information:

    - A patient can log into the system and input their updated contact details, such as a new phone number or address, along with their patient ID. Once the necessary fields are filled, the system will validate the information and update the contact details in the patient's record
- Cancelling Appointments
    - When a patient wishes to cancel an appointment, they can simply click a button to access the cancellation form. After selecting the specific appointment, the system allows the user to delete the patient's appointment by referencing the respective doctor ID.
- Searching for Specific Appointment Details
    - Users can search for specific appointment details by entering criteria such as the doctor's name, appointment date, time, department, or doctor's phone number. The system will fetch and display the relevant appointment information, including the doctor's name, the time and date of the appointment, and the doctor's department and contact details, ensuring users have all the necessary information at hand.

Entities, Attributes, and Relationships:



1. Hospital:
   Attributes:
   - Hospital_ID (Primary Key): Uniquely identifies each hospital.
   - Hospital_Name: The name of the hospital.
   - Hospital_Address, Hospital_Phone_Number, State, and Zip_Code: Provide detailed contact and location information.
2. Department:
   Attributes:
   - id (Primary Key): Uniquely identifies each department.
   - Name: Name of the department.

    - Hospital_id (Foreign Key): Links the department to a hospital.
3. Doctor:
   Attributes:
    - Doctor_ID (Primary Key): Uniquely identifies each doctor.
    - Doctor_FName and Doctor_LName: Store the first and last names of doctors.
    - Doctor_Phone_Number: Contact information.
    - Department_id (Foreign Key): Links a doctor to a department.
4. Patient:
   Attributes:
    - Patient_id (Primary Key): Uniquely identifies each patient.
    - First_Name, Last_Name, Address, and PhoneNo: Personal details of the patient.
5. Prescription:
   Attributes:
    - Prescription_ID (Primary Key): Unique identifier for each prescription.
    - Date, Cost, and Patient_id (Foreign Key): Provide details of the prescription.
    - new_field: Placeholder for additional information.
6. Prescription_Medicine (Associative Entity):
   Attributes:
    - Prescription_ID and Medicine_ID (Composite Primary Keys): Represent the relationship between prescriptions and medicines.
    - Quantity: Number of medicine units prescribed.
7. Pharmacy:
   Attributes:
    - Pharmacy_ID (Primary Key): Unique identifier for each pharmacy.
    - Pharmacy_Name, Pharmacy_Phone_Number, and Pharmacy_Address: Detailed pharmacy information.
8. Medicine:
   Attributes:
    - Medicine_ID (Primary Key): Unique identifier for medicines.
    - Medicine_Name, Medicine_Type, Price, Stock, and Pharmacy_ID (Foreign Key): Details regarding the medicine and its availability.
9. Appointment:
   Attributes:
    - Patient_id and Doctor_id (Composite Primary Keys): Represent the relationship between patients and doctors for scheduling.

Relationships and Multiplicity Constraints

1. Hospital-Department (1:N):
    - A single hospital can have multiple departments (1:N relationship).
    - Each department is associated with one hospital through the Hospital_id foreign key.
2. Department-Doctor (1:N):
    - One department can have multiple doctors, but each doctor belongs to only one department.
3. Doctor-Appointment (1:N):
    - A doctor can have multiple appointments, but each appointment is linked to only one doctor.
4. Patient-Appointment (1:N):
    - A patient can schedule multiple appointments with different doctors.
5. Prescription-Patient (1:N):
    - A patient can have multiple prescriptions, but each prescription is linked to one patient.
6. Prescription-Prescription_Medicine (1:N):
    - A single prescription can include multiple medicines, as captured in the associative entity Prescription_Medicine.
7. Pharmacy-Medicine (1:N):
    - A pharmacy can stock multiple medicines, while each medicine is tied to one pharmacy.

Key Observations on Constraints

- Primary Keys ensure the uniqueness of each record within an entity.
- Foreign Keys enforce referential integrity, linking related entities.
- Multiplicity is critical in defining the cardinality of relationships:
    - 1:N (e.g., Hospital-Department, Department-Doctor): Highlights hierarchical relationships.
    - M:N relationships are resolved using associative entities like Prescription_Medicine for many-to-many mapping.

DDL Script, Procedures, Functions & Triggers:

Procedures:

1) This stored procedure, CancelAppointment, deletes an appointment from the Appointment table based on the provided patientId and doctorId. If either of the input values is NULL, it raises an error with the message "Both patientId and doctorId must be provided."
2) This stored procedure, GetAppointmentHistory, retrieves the appointment history for a specific patient identified by patientId. It first checks the number of appointments for the patient, and if no appointments are found, it returns a message saying "No appointments found for this patient." If appointments exist, it selects detailed information about each appointment, including the doctor's name, phone number, department, appointment date/time, and status, ordered by the most recent appointment.
3) This stored procedure, ScheduleAppointment, schedules an appointment for a patient with a specific doctor at a given time. It performs the following checks:
    1. Verifies that the patient exists by checking the Patient table.
    2. Verifies that the doctor exists by checking the Doctor table.
    3. Checks if the doctor is already booked at the specified time by querying the Appointment table. If any of these conditions fail, it raises an error with a specific message. If all checks pass, the procedure inserts the appointment into the Appointment table and returns a confirmation message with the appointment details.

4) This stored procedure, UpdatePatientContact, allows updating a patient's contact information, specifically their phone number and address. The procedure works as follows:

1. Check if Patient Exists: It first checks if the patient with the given patientId exists in the Patient table. If not, it returns an error message indicating that the patient does not exist.
2. Update Contact Information:
    - If a new phone number is provided (and not empty), it updates the patient's phone number.
    - If a new address is provided (and not empty), it updates the patient's address.
3. Return Confirmation:
    - If neither the phone number nor the address was updated, it informs the user that no updates were made.

- If updates are successfully made, it returns a success message.

5) This stored procedure, AddNewPatient, adds a new patient record to the Patient table. It performs the following steps:

1. Check Mandatory Fields: It checks if the required fields (patientId, firstName, and lastName) are provided. If any of these are missing, it returns an error message.
2. Check for Existing Patient: It checks if a patient with the provided patientId already exists in the Patient table. If so, it returns an error message indicating that the patient already exists.
3. Insert New Patient: If the above checks pass, the procedure inserts the new patient's details into the Patient table. The address and phoneNo are set to empty strings if not provided (using IFNULL).
4. Return Confirmation: It returns a success message confirming the addition of the new patient record.

Functions:

This stored function, Get_Medicines_In_Prescription, returns a comma-separated list of medicines associated with a given prescription ID. Here's how it works:

1. Cursor Declaration: A cursor is declared to fetch the names of medicines from the Medicine table, joined with the Prescription_Medicine table, based on the provided prescription_id.
2. Looping Through Medicines: The function opens the cursor and loops through the result set, fetching each medicine name one by one. It concatenates the names into a string (medicine_list), separated by commas.
3. No Medicines Handling: If no medicines are found (i.e., the list remains empty), it sets the result as 'No medicines found'.
4. Return: The function returns the final concatenated string of medicine names, or a message indicating that no medicines are found.

   This stored function, get_doctor_count, returns the number of doctors in a specific department, identified by p_department_id. Here's how it works:

   Input Parameter: The function takes a department ID (p_department_id) as input.

Query Execution: It executes a SELECT COUNT(*) query on the Doctor table to count the number of doctors belonging to the specified department (Department_ID = p_department_id).

Return Value: The function returns the count of doctors in that department as an integer.

This function is useful for retrieving the total number of doctors in a particular department, providing a quick way to gather department-specific doctor data.

This stored function, get_department_count, returns the number of departments in a specified hospital. Here's a breakdown of its functionality:

1. Input Parameter: The function takes a hospital ID (p_hospital_id) as input.
2. Query Execution: It executes a SELECT COUNT(*) query on the Department table to count how many departments are associated with the given Hospital_ID.
3. Return Value: The count of departments is stored in the department_count variable, and the function returns this value as an integer.

Triggers:

1. The Prescription_BEFORE_INSERT_date_validate trigger ensures that the `Date` field for any new prescription does not point to a future date. It is executed before an `INSERT` operation and checks whether `NEW.Date > CURRENT_TIMESTAMP`. If the date is in the future, it raises an error (SQLSTATE '45000') with the message: *"Prescription date cannot be in the future."* This ensures that all prescriptions adhere to real-world constraints by restricting entries to current or past dates, thus avoiding invalid records.
2. The Prescription_Medicine_AFTER_INSERT trigger automatically updates the `Cost` field in the `Prescription` table whenever a medicine is added to a prescription. It runs after an `INSERT` operation on the `Prescription_Medicine` table and always triggers on an insert. The trigger calculates the total cost of medicines in a prescription by summing the price and quantity for all associated medicines, then updates the `Cost` field in the `Prescription` table. This ensures that the `Cost` field reflects accurate data as new medicines are added, reducing manual updates.

3. The Prescription_Medicine_AFTER_UPDATE trigger keeps the `Cost` field in the `Prescription` table up to date whenever prescription medicines are modified. It is triggered after an `UPDATE` operation on the `Prescription_Medicine` table and always triggers on an update. The trigger recalculates the total cost of all medicines in the prescription based on their updated prices and quantities, then updates the `Cost` field in the `Prescription` table. This ensures that the `Cost` field remains accurate and consistent, reflecting any changes made to prescription details.
4. The Prescription_Medicine_AFTER_DELETE trigger ensures that the `Cost` field in the `Prescription` table is updated accurately when a medicine is removed from a prescription. It is executed after a `DELETE` operation on the `Prescription_Medicine` table and always triggers on a deletion. The trigger recalculates the total cost of the remaining medicines in the prescription and updates the `Cost` field accordingly. This prevents inaccuracies in the `Cost` field when a medicine is deleted, maintaining data integrity.
5. The update_stock_after_prescription trigger manages the stock levels of medicines whenever a new prescription is created. It is triggered after an `INSERT` operation on the `Prescription_Medicine` table and always triggers when a medicine is prescribed. The trigger verifies that there is sufficient stock of the prescribed medicine. If the available stock is less than `NEW.Quantity`, it raises an error (SQLSTATE '45000') with the message: *"Not enough stock for the prescribed medicine."* If stock is sufficient, it deducts the prescribed quantity from the `Stock` field in the `Medicine` table. This ensures accurate inventory levels and prevents overselling medicines.
6. The update_stock_on_prescription_update trigger ensures medicine stock levels are updated correctly when prescription quantities are modified. It runs before an `UPDATE` operation on the `Prescription_Medicine` table. The trigger checks whether there is enough stock to accommodate an increase in the prescription quantity (`quantity_difference > 0`). If stock is inadequate, it raises an error (SQLSTATE '45000') with the

message: *"Not enough stock for the updated prescription."* If stock is sufficient, it adjusts the stock in the `Medicine` table by subtracting the quantity difference. This prevents inconsistencies in stock levels caused by prescription updates, ensuring accurate inventory management.

7. The prevent_overlapping_appointments trigger prevents scheduling conflicts for both patients and doctors. It is executed before an `INSERT` operation on the `Appointment` table. The trigger verifies that the patient does not already have an appointment at the same time (`Patient_ID` and `Appointment_Time`) and checks that the doctor is not double-booked at the same time (`Doctor_ID` and `Appointment_Time`). If either condition is met, it raises an error (SQLSTATE '45000') with one of the following messages: *"A patient cannot have overlapping appointments"* or *"A doctor cannot have overlapping appointments."* This ensures the scheduling system avoids double bookings for both patients and doctors, maintaining operational efficiency.
8. The prevent_overlapping_appointments_update trigger safeguards against scheduling conflicts when updating appointment details. It runs before an `UPDATE` operation on the `Appointment` table. The trigger confirms that the updated time does not conflict with other appointments for the same patient (`Patient_ID` and `Appointment_Time`), excluding the current appointment being updated. It also ensures the doctor is not double-booked at the updated time (`Doctor_ID` and `Appointment_Time`), excluding the current appointment. If any conflicts are found, it raises an error (SQLSTATE '45000') with one of the following messages: *"A patient cannot have overlapping appointments"* or *"A doctor cannot have overlapping appointments."*

Tables:

```
CREATE TABLE `Appointment` (
  `Patient_id` int NOT NULL,
  `Doctor_id` int NOT NULL,
  `Appointment_Time` datetime DEFAULT NULL,
  `Status` varchar(50) DEFAULT 'Scheduled',
  PRIMARY KEY (`Patient_id`,`Doctor_id`),
  KEY `FK2_idx` (`Doctor_id`),
  CONSTRAINT `FK1` FOREIGN KEY (`Patient_id`) REFERENCES `Patient` (`patient_id`),
  CONSTRAINT `FK2` FOREIGN KEY (`Doctor_id`) REFERENCES `Doctor` (`Doctor_ID`)
```

```
CREATE TABLE `Department` (
  `Department_ID` int NOT NULL,
  `Name` varchar(255) DEFAULT NULL,
  `Hospital_ID` int DEFAULT NULL,
  PRIMARY KEY (`Department_ID`),
  KEY `Hospital_ID_idx` (`Hospital_ID`),
  CONSTRAINT `FK_Hospital_ID` FOREIGN KEY (`Hospital_ID`) REFERENCES `Hospital` (`Hospital_id`)
```

```
CREATE TABLE `Doctor` (
  `Doctor_ID` int NOT NULL,
  `First_Name` varchar(255) DEFAULT NULL,
  `Last_Name` varchar(255) DEFAULT NULL,
  `Phone_Number` varchar(20) DEFAULT NULL,
  `Department_ID` int DEFAULT NULL,
  PRIMARY KEY (`Doctor_ID`),
  KEY `Department_ID_idx` (`Department_ID`),
  CONSTRAINT `FK_Department_ID` FOREIGN KEY (`Department_ID`) REFERENCES `Department` (`Department_ID`)
```

```sql
CREATE TABLE `Hospital` (
  `Hospital_id` int NOT NULL,
  `Hospital_Name` varchar(255) DEFAULT NULL,
  `Hospital_Address` varchar(255) DEFAULT NULL,
  `Hospital_PhoneNo` varchar(20) DEFAULT NULL,
  `State` varchar(255) DEFAULT NULL,
  `Zip_Code` int DEFAULT NULL,
  PRIMARY KEY (`Hospital_id`),
  CONSTRAINT `check_zip` CHECK (((`Zip_Code` is null) or ((`Zip_Code` >= 10000) and (`Zip_Code` <= 99999))))
```

```sql
CREATE TABLE `Medicine` (
  `Medicine_ID` int NOT NULL,
  `Medicine_Name` varchar(255) NOT NULL,
  `Medicine_Type` varchar(100) DEFAULT NULL,
  `Price` int NOT NULL,
  `Stock` int NOT NULL,
  `Pharmacy_ID` int NOT NULL,
  PRIMARY KEY (`Medicine_ID`),
  CONSTRAINT `chk_price_nonnegative` CHECK ((`Price` >= 0)),
  CONSTRAINT `chk_stock_nonnegative` CHECK ((`stock` >= 0))
```

```sql
CREATE TABLE `Patient` (
  `patient_id` int NOT NULL,
  `First_Name` varchar(225) DEFAULT NULL,
  `Last_Name` varchar(225) DEFAULT NULL,
  `Address` varchar(225) DEFAULT NULL,
  `PhoneNo` varchar(220) DEFAULT NULL,
  PRIMARY KEY (`patient_id`)
```

```sql
CREATE TABLE `Pharmacy` (
  `pharmacy_id` int NOT NULL,
  `pharmacy_name` varchar(255) DEFAULT NULL,
  `Phone_Number` varchar(20) DEFAULT NULL,
  `Address` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`pharmacy_id`)
```

```sql
CREATE TABLE `Prescription` (
  `prescription_id` int NOT NULL,
  `Date` datetime DEFAULT CURRENT_TIMESTAMP,
  `Cost` decimal(10,2) DEFAULT NULL,
  `Patient_id` int NOT NULL,
  PRIMARY KEY (`prescription_id`),
  KEY `Patient_id_idx` (`Patient_id`),
  CONSTRAINT `Patient_id` FOREIGN KEY (`Patient_id`) REFERENCES `Patient` (`patient_id`),
  CONSTRAINT `chk cost positive` CHECK ((`Cost` >= 0))
```

```sql
CREATE TABLE `Prescription_Medicine` (
  `Prescription_ID` int NOT NULL,
  `Medicine_ID` int NOT NULL,
  `Quantity` int DEFAULT NULL,
  PRIMARY KEY (`Prescription_ID`,`Medicine_ID`)
```

Triggers:

```sql
CREATE*/ /*!50017 DEFINER=`avnadmin`@`%`*/ /*!50003 TRIGGER `prevent_overlapping_appointments`
BEFORE INSERT ON `Appointment` FOR EACH ROW BEGIN
  -- Check if the patient already has an appointment at the same time
  IF EXISTS (
    SELECT 1
    FROM Appointment
    WHERE Patient_ID = NEW.Patient_ID
      AND Appointment_Time = NEW.Appointment_Time
  ) THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'A patient cannot have overlapping appointments.';
  END IF;

  -- Check if the doctor already has an appointment at the same time
  IF EXISTS (
    SELECT 1
    FROM Appointment
    WHERE Doctor_ID = NEW.Doctor_ID
      AND Appointment_Time = NEW.Appointment_Time
  ) THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'A doctor cannot have overlapping appointments.';
  END IF;
END */;;
DELIMITER ;
```

```
DELIMITER ;;
/*!50003 CREATE*/ /*!50017 DEFINER=`avnadmin`@`%`*/ /*!50003 TRIGGER `prevent_overlapping_appointments_update`
BEFORE UPDATE ON `Appointment` FOR EACH ROW BEGIN
  -- Check if the patient already has an appointment at the same time, excluding the current appointment being updated
  IF EXISTS (
    SELECT 1
    FROM Appointment
    WHERE Patient_ID = NEW.Patient_ID
      AND Appointment_Time = NEW.Appointment_Time
  ) THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'A patient cannot have overlapping appointments.';
  END IF;

  -- Check if the doctor already has an appointment at the same time, excluding the current appointment being updated
  IF EXISTS (
    SELECT 1
    FROM Appointment
    WHERE Doctor_ID = NEW.Doctor_ID
      AND Appointment_Time = NEW.Appointment_Time
  ) THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'A doctor cannot have overlapping appointments.';
  END IF;
END */;;
```

```
/*!50003 CREATE*/ /*!50017 DEFINER=`avnadmin`@`%`*/ /*!50003 TRIGGER `Prescription_BEFORE_INSERT_date_validate`
BEFORE INSERT ON `Prescription` FOR EACH ROW BEGIN
    IF NEW.Date > CURRENT_TIMESTAMP THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Prescription date cannot be in the future';
    END IF;
END */;;
DELIMITER ;
```

```
/*!50003 CREATE*/ /*!50017 DEFINER=`avnadmin`@`%`*/ /*!50003 TRIGGER `Prescription_Medicine_AFTER_INSERT`
AFTER INSERT ON `Prescription_Medicine` FOR EACH ROW BEGIN
    DECLARE total_cost INT;

    SELECT SUM(m.Price * pm.Quantity)
    INTO total_cost
    FROM Prescription_Medicine pm
    JOIN Medicine m ON pm.Medicine_ID = m.Medicine_ID
    WHERE pm.Prescription_ID = NEW.Prescription_ID;

    -- Update the cost in the Prescription table
    UPDATE Prescription
    SET Cost = IFNULL(total_cost, 0)
    WHERE Prescription_ID = NEW.Prescription_ID;
END */;;
```

```
/*!50003 CREATE*/ /*!50017 DEFINER=`avnadmin`@`%`*/ /*!50003 TRIGGER `update_stock_after_prescription`
AFTER INSERT ON `Prescription_Medicine` FOR EACH ROW BEGIN
  -- Declare a variable to hold the current stock
  DECLARE current_stock INT;

  -- Fetch the current stock for the prescribed medicine
  SELECT `Stock` INTO current_stock
  FROM `Medicine`
  WHERE `Medicine_ID` = NEW.Medicine_ID;

  -- Check if there is sufficient stock
  IF current_stock < NEW.Quantity THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Not enough stock for the prescribed medicine.';
  ELSE
    -- Update the stock by subtracting the prescribed quantity
    UPDATE `Medicine`
    SET `Stock` = `Stock` - NEW.Quantity
    WHERE `Medicine_ID` = NEW.Medicine_ID;
  END IF;
END */;;
DELIMITER ;
```

```
DELIMITER ;;
/*!50003 CREATE*/ /*!50017 DEFINER=`avnadmin`@`%`*/ /*!50003 TRIGGER `update_stock_on_prescription_update`
 BEFORE UPDATE ON `Prescription_Medicine` FOR EACH ROW BEGIN
  -- Declare variables for current stock and the difference in quantity
  DECLARE current_stock INT;
  DECLARE quantity_difference INT;

  -- Fetch the current stock for the medicine
  SELECT `Stock` INTO current_stock
  FROM `Medicine`
  WHERE `Medicine_ID` = NEW.Medicine_ID;

  -- Calculate the difference between the new quantity and the old quantity
  SET quantity_difference = NEW.Quantity - OLD.Quantity;

  -- Check if there is sufficient stock if the quantity is being increased
  IF quantity_difference > 0 AND current_stock < quantity_difference THEN
    SIGNAL SQLSTATE '45000'
      SET MESSAGE_TEXT = 'Not enough stock for the updated prescription.';
  ELSE
    -- Update the stock by adjusting for the quantity difference
    UPDATE `Medicine`
    SET `Stock` = `Stock` - quantity_difference
    WHERE `Medicine_ID` = NEW.Medicine_ID;
  END IF;
END */;;
DELIMITER ;
```

```
/*!50003 CREATE*/ /*!50017 DEFINER=`avnadmin`@`%`*/ /*!50003 TRIGGER `Prescription_Medicine_AFTER_DELETE`
AFTER DELETE ON `Prescription_Medicine` FOR EACH ROW BEGIN
 DECLARE total_cost INT;

    SELECT SUM(m.Price * pm.Quantity)
    INTO total_cost
    FROM Prescription_Medicine pm
    JOIN Medicine m ON pm.Medicine_ID = m.Medicine_ID
    WHERE pm.Prescription_ID = OLD.Prescription_ID;

    -- Update the cost in the Prescription table
    UPDATE Prescription
    SET Cost = IFNULL(total_cost, 0)
    WHERE Prescription_ID = OLD.Prescription_ID;
END */;;
DELIMITER ;
```

Functions:

```
CREATE DEFINER="avnadmin"@"%" FUNCTION "get_department_count"(p_hospital_id INT) RETURNS int
    DETERMINISTIC
BEGIN
    DECLARE department_count INT;

    SELECT COUNT(*)
    INTO department_count
    FROM Department
    WHERE Hospital_ID = p_hospital_id;

    RETURN department_count;
END ;;

        ;;
CREATE DEFINER="avnadmin"@"%" FUNCTION "get_doctor_count"(p_department_id INT) RETURNS int
    DETERMINISTIC
BEGIN
    RETURN (SELECT COUNT(*)
            FROM Doctor
            WHERE Department_ID = p_department_id);
END ;;
```

```
CREATE DEFINER="avnadmin"@"%" FUNCTION "Get_Medicines_In_Prescription"(prescription_id INT) RETURNS text CHARSET utf8mb4
    DETERMINISTIC
BEGIN
    DECLARE medicine_list TEXT DEFAULT '';
    DECLARE medicine_name VARCHAR(255);
    DECLARE done INT DEFAULT FALSE;

    -- Select medicines associated with the prescription
    DECLARE cur CURSOR FOR
        SELECT m.Medicine_Name
        FROM Medicine m
        JOIN Prescription_Medicine pm ON m.Medicine_ID = pm.medicine_id
        WHERE pm.prescription_id = prescription_id;

    -- Declare continue handler for when the cursor reaches the end of the result set
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    -- Open cursor
    OPEN cur;
    -- Fetch and process rows from cursor
    read_loop: LOOP
        FETCH cur INTO medicine_name;
        IF done THEN
            LEAVE read_loop;
        END IF;

        -- Concatenate the medicine names
        IF medicine_list = '' THEN
            SET medicine_list = medicine_name;
        ELSE
            SET medicine_list = CONCAT(medicine_list, ', ', medicine_name);
        END IF;
    END LOOP;

    -- Close cursor
    CLOSE cur;

    -- If no medicines were found, set appropriate message
    IF medicine_list = '' THEN
        SET medicine_list = 'No medicines found';
    END IF;

    -- Return the list
    RETURN medicine_list;
END ;;
```

Procedures:

```
CREATE DEFINER="avnadmin"@"%" PROCEDURE "AddNewPatient"(
    IN patientId INT,
    IN firstName VARCHAR(225),
    IN lastName VARCHAR(225),
    IN address VARCHAR(225),
    IN phoneNo VARCHAR(220)
)
BEGIN
    -- Check if mandatory fields are provided
    IF patientId IS NULL OR firstName IS NULL OR lastName IS NULL THEN
        SELECT 'Error: Patient ID, First Name, and Last Name are required fields.' AS Message;
       -- RETURN;
    END IF;

    -- Check if the patient already exists
    IF EXISTS (SELECT 1 FROM Patient WHERE patient_id = patientId) THEN
        SELECT CONCAT('Error: Patient with ID ', patientId, ' already exists.') AS Message;
        -- RETURN;
    END IF;


    INSERT INTO Patient (patient_id, First_Name, Last_Name, Address, PhoneNo)
    VALUES (patientId, firstName, lastName, IFNULL(address, ''), IFNULL(phoneNo, ''));

     SELECT 'New patient record added successfully.' AS Message;
END ;;


CREATE DEFINER="avnadmin"@"%" PROCEDURE "CancelAppointment"(
     IN patientId INT,
     IN doctorId INT
)
BEGIN
     -- Check if patientId or doctorId is provided
     IF patientId IS NULL OR doctorId IS NULL THEN
         -- Raise an error if either patientId or doctorId is NULL
         SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Both patientId and doctorId must be provided.';
     ELSE
         -- Remove the appointment for the given patientId and doctorId
         DELETE FROM Appointment WHERE patient_id = patientId AND doctor_id = doctorId;
     END IF;
END ;;
```

```
CREATE DEFINER="avnadmin"@"%" PROCEDURE "GetAppointmentHistory"(
    IN patientId INT
)
BEGIN

    DECLARE appointmentCount INT;


    SELECT COUNT(*) INTO appointmentCount
    FROM Appointment
    WHERE Patient_id = patientId;


    IF appointmentCount = 0 THEN
        SELECT 'No appointments found for this patient.' AS Message;
    ELSE

        SELECT
            d.Doctor_ID,
            d.First_Name AS Doctor_First_Name,
            d.Last_Name AS Doctor_Last_Name,
            d.Phone_Number AS Doctor_Phone,
            dep.Name AS Department_Name,
            a.Appointment_Time AS Appointment_DateTime,
            IFNULL(a.Status, 'Scheduled') AS Appointment_Status
        FROM Appointment a
        JOIN Doctor d ON a.Doctor_id = d.Doctor_ID
        JOIN Department dep ON d.Department_ID = dep.Department_ID
        WHERE a.Patient_id = patientId
        ORDER BY a.Appointment_Time DESC; -- Sorting appointments by most recent
    END IF;
```

```sql
CREATE DEFINER="avnadmin"@"%" PROCEDURE "ScheduleAppointment"(
    IN patientId INT,
    IN doctorId INT,
    IN appointmentDateTime DATETIME
)
BEGIN
    DECLARE doctorName VARCHAR(255);
    DECLARE patientName VARCHAR(255);
    DECLARE doctorAvailability INT;

    -- Check if the patient exists
    SELECT CONCAT(First_Name, ' ', Last_Name) INTO patientName
    FROM Patient
    WHERE Patient_ID = patientId;

    IF patientName IS NULL THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Patient does not exist.';
    END IF;

    -- Check if the doctor exists
    SELECT CONCAT(First_Name, ' ', Last_Name) INTO doctorName
    FROM Doctor
    WHERE Doctor_ID = doctorId;

    IF doctorName IS NULL THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Doctor does not exist.';
    END IF;

    -- Check if the doctor is available at the given time
    SELECT COUNT(*) INTO doctorAvailability
    FROM Appointment
    WHERE Doctor_id = doctorId AND Appointment_Time = appointmentDateTime;

    IF doctorAvailability > 0 THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'Doctor is not available at the specified time.';
    END IF;

    -- Insert the appointment into the Appointment table
    INSERT INTO Appointment (Patient_id, Doctor_id, Appointment_Time)
    VALUES (patientId, doctorId, appointmentDateTime);
```

```
CREATE DEFINER="avnadmin"@"%" PROCEDURE "UpdatePatientContact"(
    IN patientId INT,
    IN newPhoneNumber VARCHAR(15),
    IN newAddress VARCHAR(255)
)
BEGIN
    -- Check if the patient exists
    IF NOT EXISTS (SELECT 1 FROM Patient WHERE Patient_ID = patientId) THEN
        SELECT CONCAT('Error: Patient with ID ', patientId, ' does not exist.') AS Message;
    ELSE
        -- Update the phone number if provided
        IF newPhoneNumber IS NOT NULL AND newPhoneNumber != '' THEN
            UPDATE Patient
            SET PhoneNo = newPhoneNumber
            WHERE Patient_ID = patientId;
        END IF;

        -- Update the address if provided
        IF newAddress IS NOT NULL AND newAddress != '' THEN
            UPDATE Patient
            SET Address = newAddress
            WHERE Patient_ID = patientId;
        END IF;

        -- Check if any updates were made
        IF (newPhoneNumber IS NULL OR newPhoneNumber = '') AND (newAddress IS NULL OR newAddress = '') THEN
            SELECT 'No updates were made. Both phone number and address inputs are empty.' AS Message;
        ELSE
            SELECT 'Patient contact information updated successfully.' AS Message;
        END IF;
    END IF;
```

Application Flow:

1. Dashboard
    - Functionality:
        - Central hub for accessing different sections of the HMS.
        - Quick overview of appointments, patients, doctors, pharmacy, prescription.
        - Available actions like "Book Appointment" (for patients), "View Appointments" (for doctors), or "Manage Patients" (for admins).
2. Patient Management
    - Functionality for Admin:
        - View patient details (name, contact, history).
        - Update patient information (edit details like contact info, medical history).
        - Add patients in the system.
3. Doctor Management
    - Functionality for Admin:
        - View doctor details (name, contact, specialization).
        - Searching information based on Specialization, and Name of the Doctor.
        - Add doctors in the system.
4. Book Appointment (Patient)
    - Functionality for Patients:
        - Select a department and choose an available doctor.
        - Pick a date and time for the appointment.
        - Appointment booking confirms the doctor’s availability based on their schedule.
        - View appointment schedule for a particular patient.
5. Prescription Management
    - Functionality for Doctors:
        - After an appointment, doctors can write prescriptions for patients.
        - The prescription includes the medication, dosage, and pharmacy for fulfilling the order.
6. Pharmacy Management
    - Functionality for Pharmacists/Admins:
        - Manage medicine inventory.
        - Track prescriptions issued and medicines dispensed. (Get Medicines in Prescription)
        - Update stock and verify availability for patients.

Using DBDesigner and React/Node.js/MySQL:

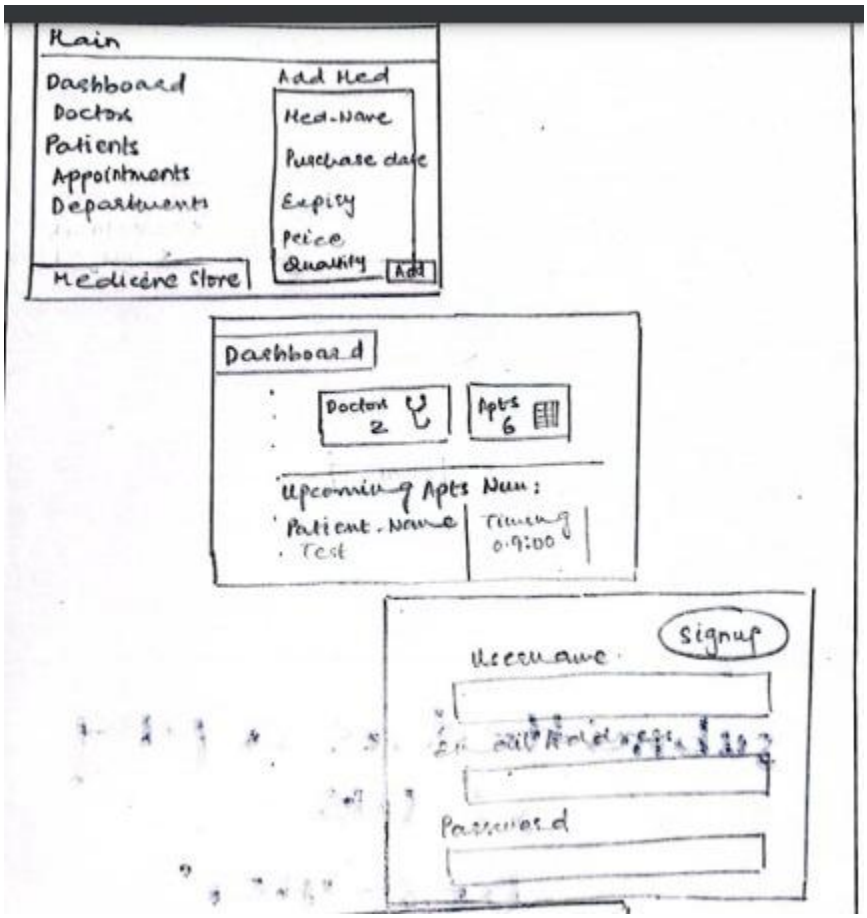
We have used DB Designer to create the ERD (Entity Relationship Diagram) which will help us visualize how the backend structure is set up, ensuring smooth interaction between tables like Patients, Doctors, Appointments, Prescriptions, and Pharmacy.

Moreover, React Frontend will interact with Node.js Backend, fetching and posting data to the MySQL Database.

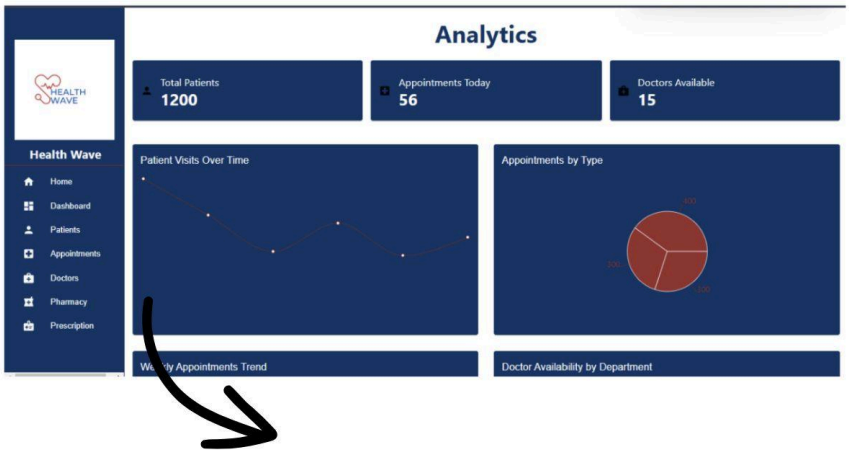
The system will use APIs (from Node.js) to fetch patient data, doctor schedules, and other records, and React will render these in the frontend.

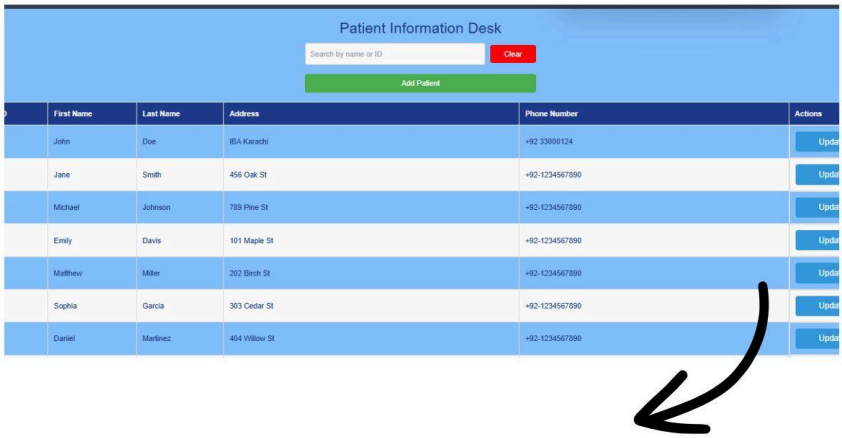
Used Axios in React for communicating with the Node.js server.

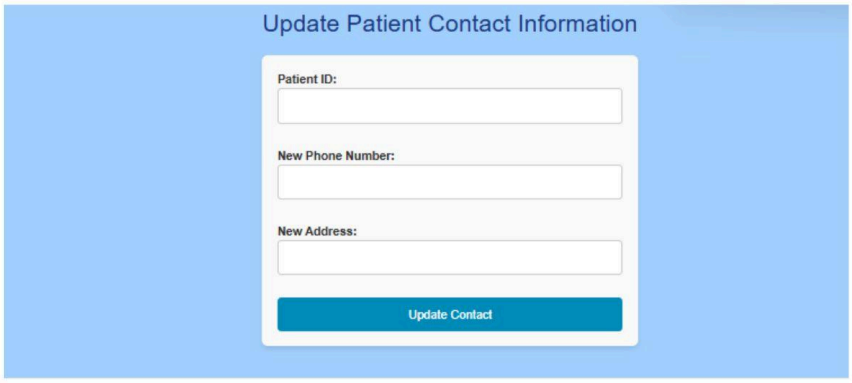
Wireframes:

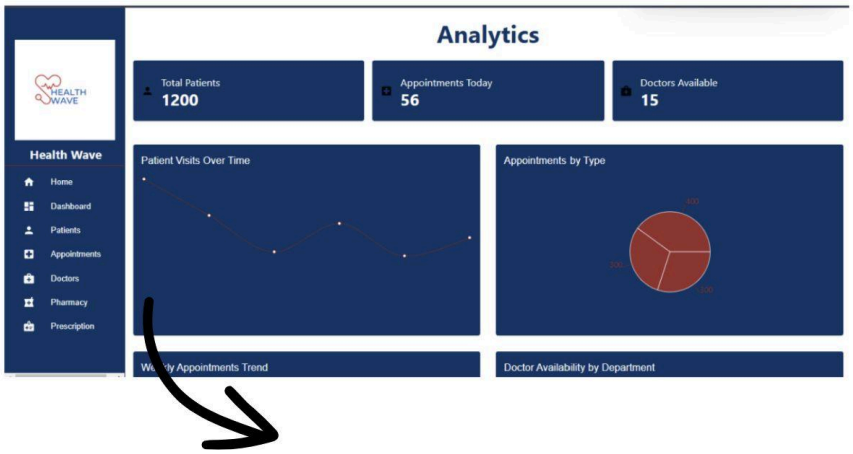

Main
Dashboard
Doctors
Patients
Appointments
Departments
Medicine Store
Add Med
Med-Name
Purchase date
Expiry
Price
Quantity
Add
Dashboard
Doctors 2
Apts 6
Upcoming Apts Num:
Patient. Name
Test
Timing
0:9:00
Signup
Username
Password

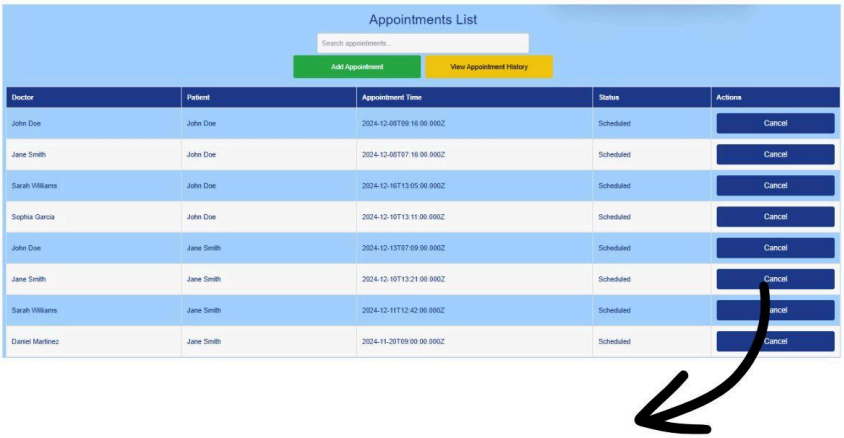
Navigating page by page:
(Note that we have discussed the Workflows above)

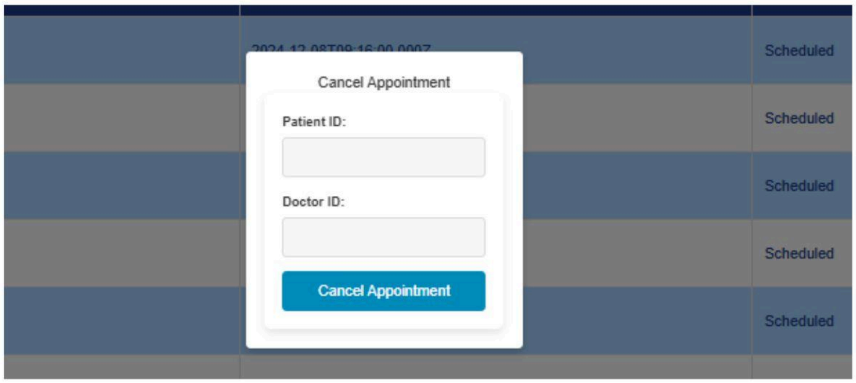


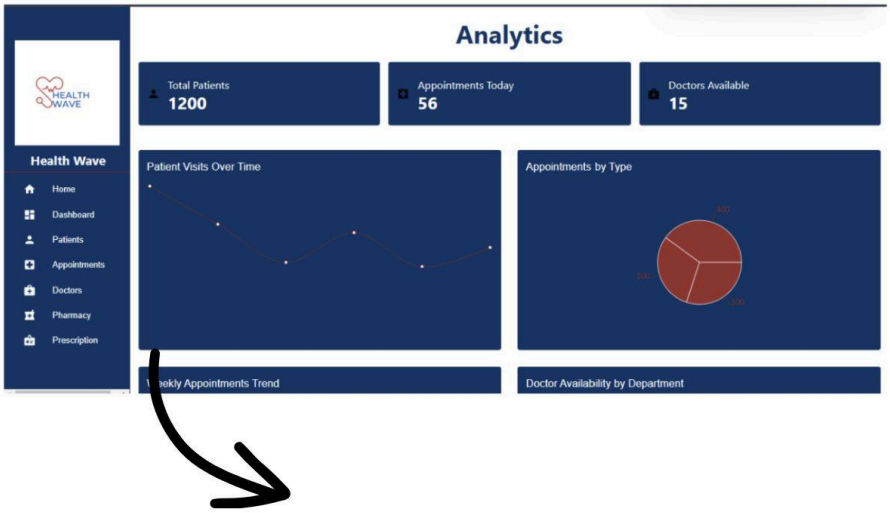


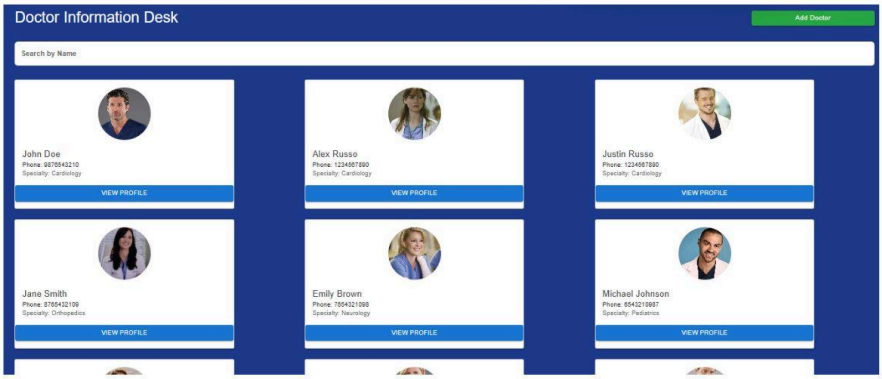


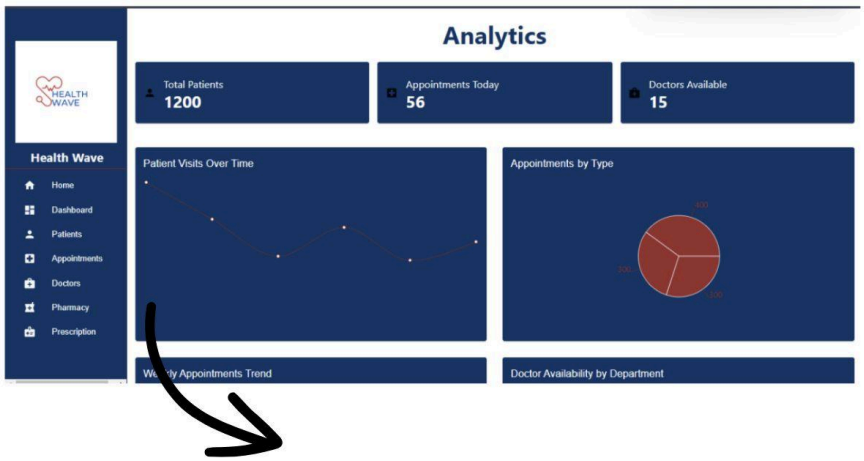

Analytics
Total Patients
1200
Appointments Today
56
Doctors Available
15
Health Wave
Home
Dashboard
Patients
Appointments
Doctors
Pharmacy
Prescription
Patient Visits Over Time
Appointments by Type
Doctor Availability by Department


Appointments List
Add Appointment
View Appointment History
Doctor
Patient
Appointment Time
Status
Actions
Cancel

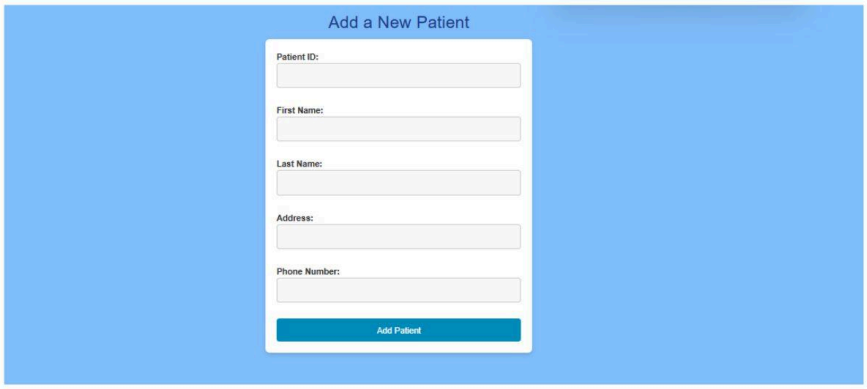

Cancel Appointment
Patient ID:
Doctor ID:
Cancel Appointment
Scheduled


Analytics
Total Patients
1200
Appointments Today
56
Doctors Available
15
Health Wave
Home
Dashboard
Patients
Appointments
Doctors
Pharmacy
Prescription
Patient Visits Over Time
Appointments by Type
Doctor Availability by Department


Doctor Information Desk
Add Doctor
Search by Name
John Doe
Alex Russo
Justin Russo
Jane Smith
Emily Brown
Michael Johnson
VIEW PROFILE


Analytics
Total Patients
1200
Appointments Today
56
Doctors Available
15
Health Wave
Home
Dashboard
Patients
Appointments
Doctors
Pharmacy
Prescription
Patient Visits Over Time
Appointments by Type
Doctor Availability by Department


Patient Information Desk
Search by name or ID
Clear
Add Patient
First Name
Last Name
Address
Phone Number
Actions
John
Doe
IBA Karachi
+92 33000124
Jane
Smith
456 Oak St
+92-1234567890
Michael
Johnson
789 Pine St
+92-1234567890
Emily
Davis
101 Maple St
+92-1234567890
Matthew
Miller
202 Birch St
+92-1234567890
Sophia
Garcia
303 Cedar St
+92-1234567890
Daniel
Martinez
404 Willow St
+92-1234567890


Add a New Patient
Patient ID:
First Name:
Last Name:
Address:
Phone Number:
Add Patient

Work Contribution:

Names	Database	Backend	Frontend
Hamna Inam Abro	Created Tables Hospital, Prescription, Prescription_Medicine Created Triggers	Set up Express server and initial routing, Created CRUD operations controllers and Routes	Set up React app structure Made Prescription, doctor pages and set up routing in the dashboard and the rest of the pages
Rafsha Rahim	Created Tables Pharmacy, Department, Medicine Created Procedures	Created Functions Controllers and Routes	Made AboutUs, Dashboard, Medicine, and Pharmacy pages
Zara Masood	Created Tables Appointment, Doctor, Patient Created Functions	Created Procedures Controllers and Routes	Made Appointment and Patients pages