EHR for Emergency Care

Group 4:

Shiny Pidugu - 801320154

Meghna Reddy Aleti - 801326493

Lakshmi Sahithi Talluri - 801324888

Due Date: 11/24/2023

Table of Contents

Project Overview

Background Study:

The objective of this project is to establish a database for Electronic Health Records (EHR) tailored for an emergency room environment. The primary aim is to efficiently store and manage essential data encompassing patient details, provider information, clinical records, appointment details, orders, billing, and other pertinent aspects of patient care within the emergency room. The database's design is focused on optimizing the patient care process, enhancing patient outcomes, and fostering seamless care coordination among healthcare providers.

Scope:

This project's scope entails the development of a database system serving as the backend for a simplified Electronic Medical Records (EMR) system, concentrating on a specific care scenario within the emergency department. The project includes the creation of a Unified Modeling Language (UML) diagram and a database schema featuring various tables such as addresses, insurances, providers, facilities, patients, visits, clinical information, appointments, orders, beds, supplies, and billing. Each table will possess distinct attributes and primary keys, with interconnections through foreign keys to enable data relationships.

The database is intended to support a user-friendly system enabling the entry and management of patient demographic information, provider details, visit particulars, clinical care records, and other relevant data based on the given scenario. Additionally, it should facilitate the editing of existing records for correcting data entry errors or legitimate information changes, searching patient records using criteria like name, ID, and visit dates, and offer reporting functions such as generating lists of patients meeting specific criteria.

Motivation:

The driving force behind this project is the enhancement of patient care within emergency room settings by providing healthcare providers swift access to crucial patient information. The implementation of an EHR database aims to enable healthcare providers to retrieve pertinent patient information promptly, thereby reducing medical errors and improving patient outcomes. Furthermore, the comprehensive EHR database is anticipated to facilitate communication among providers, ensuring that patients receive timely and appropriate care. In summary, the project's overarching goal is to elevate patient care standards in emergency room settings and support healthcare providers in delivering high-quality care.

Database Design

This report outlines the database design for an Emergency Electronic Health Record (EHR) system, comprising 12 tables, each representing a distinct entity within the emergency EHR framework. The primary key within each table serves to uniquely identify entries, while foreign keys establish connections between tables. Notably, the "ADDRESSES" and "INSURANCES" tables incorporate foreign keys linking to the "PATIENTS" database, whereas the "VISITS" table integrates foreign keys connecting to the "PATIENTS," "PROVIDERS," and "FACILITIES" tables.

The "CLINICAL INFORMATION" table captures pertinent clinical details from a patient's visit, encompassing symptoms, discharge diagnosis, and administered medications. The "ORDERS" database documents tests prescribed for patients during visits, and the "BEDS" table furnishes information on hospital bed types.

The "USERS" table stores user credentials, including user_id, password, and role, facilitating user authentication. Meanwhile, the "BILLING" table contains data related to patient visit billing, including visit ID and billing amount. This table establishes a link to the "VISITS" table through a foreign key, streamlining the monitoring of billing specifics for each visit.

Lastly, the "LOGT" table is dedicated to logging changes or actions executed within the system. In summary, this database design offers a systematic and all-encompassing approach to storing and managing patient and clinical information within an Emergency EHR system. It ensures efficient monitoring of patient visits, clinical data, appointments, and billing, along with convenient access to information concerning healthcare providers, facilities, beds, and supplies.

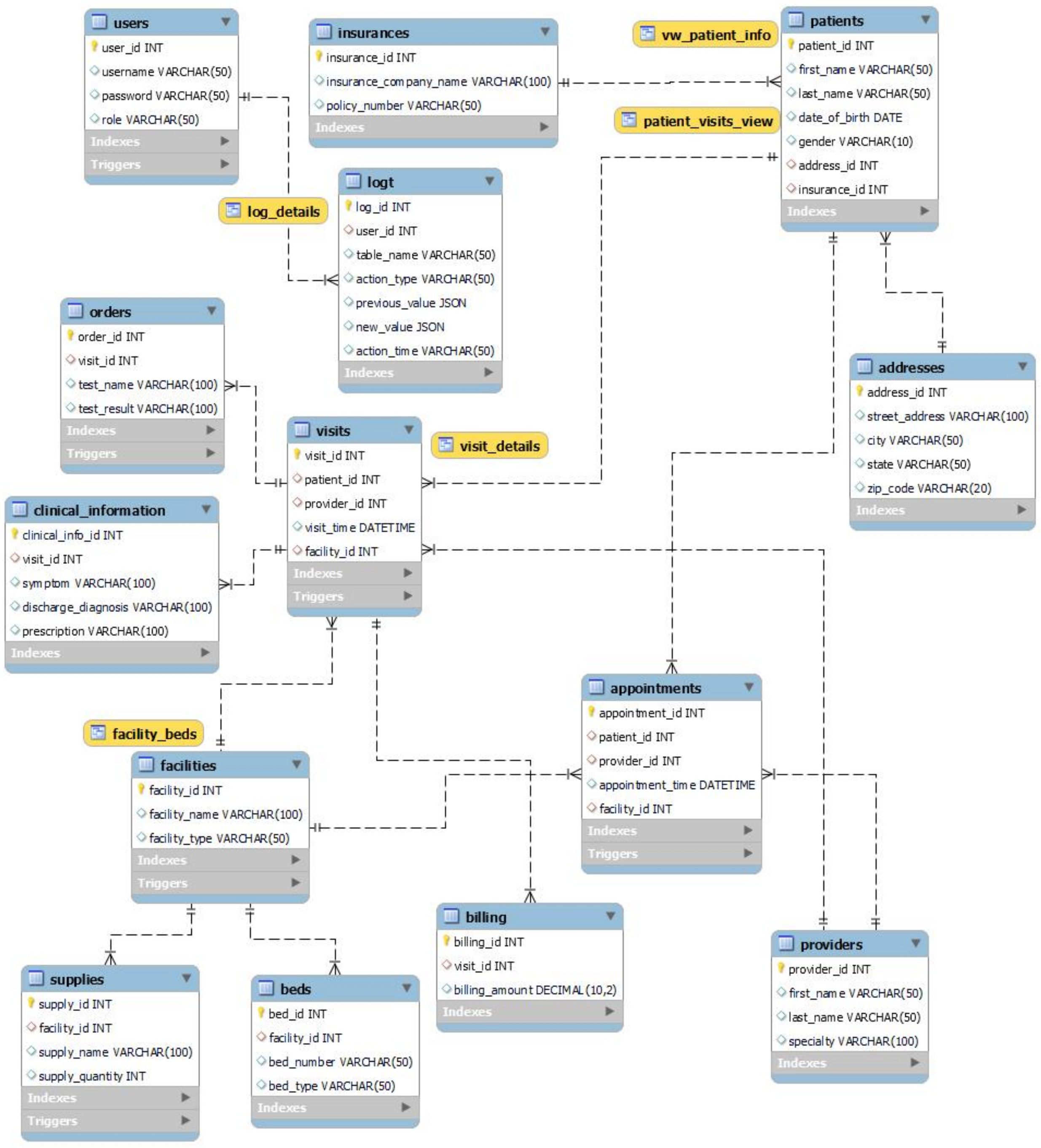
Functional Requirements

- **Patients:** This table contains demographic information for each patient, including name, date of birth information, and contact information.
- **Providers:** This table contains information about each healthcare provider, including name, specialty, and contact information.
- **Addresses:** This table contains address information for each patient, including city, state, pin code information.
- **Insurances:** This table contains insurance information, including insurance name and policy number information.
- **Visits:** This table contains information about each visit to the emergency department, including the patient ID, provider ID, time, and facility information.

- **Clinical Information:** This table contains information about the clinical care provided during each visit, including signs and symptoms, diagnoses, prescriptions, and results of exams, tests, and procedures.
- **Facilities:** This table contains information about the facilities, including facility name and type information.
- **Appointments:** This table contains information about appointments scheduled for patients, including the patient ID, provider ID, date and time, and exam room.
- **Beds:** This table contains information about the beds in the emergency department, including the bed ID, location, and bed type.
- **Supplies:** This table contains information about the supplies available in the emergency department, including the supply ID, name, quantity, and location.
- **Orders:** This table contains information on lab test, including test name and test result information.
- **Billing:** This table contains information about the billing for each visit, including the patient ID, provider ID, visit ID, and charges.
- **Users:** This table contains information about the users,including user ID, password and role.
- **Logt:** This table contains information about the changes or actions performed in the system,including log ID, user ID, table name, action type, previous value, new value, action time.

Entity Relationship Diagram

The ER diagram shows a simplified database schema for an Emergency room Electronic Health Record (EHR) system. The system is designed to manage patient information, including their addresses, insurance details, clinical information, and visits to healthcare providers.

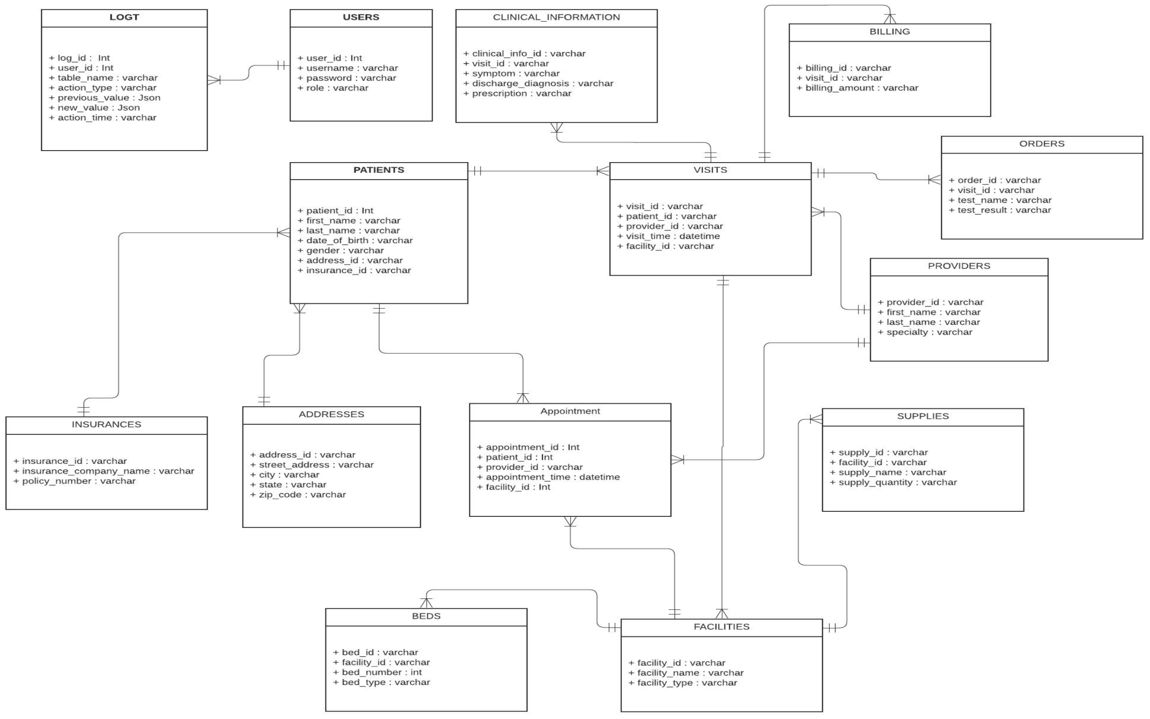


The ER diagram includes the following entities and their relationships:

- **PATIENTS:** This entity represents patients' personal information, including their first and last names, date of birth, gender, and their address and insurance IDs. A patient can have multiple visits to healthcare providers, and each visit is associated with a unique visit ID.
- **ADDRESSES:** This entity represents the addresses of patients. Each address is identified by a unique address ID and includes the street address, city, state, and zip code.
- **INSURANCES:** This entity represents the insurance details of patients, including the name of the insurance company and the policy number. Each insurance is identified by a unique insurance ID.
- **PROVIDERS:** This entity represents healthcare providers' information, including their first and last names and specialty. Each provider is identified by a unique provider ID.
- **FACILITIES:** This entity represents the healthcare facilities' information, including their name and type. Each facility is identified by a unique facility ID.
- **VISITS:** This entity represents patients' visits to healthcare providers, including the visit ID, patient ID, provider ID, visit time, and facility ID. Each visit can have multiple clinical information, orders, and billing associated with it.
- **CLINICAL_INFORMATION:** This entity stores clinical information associated with a patient's visit, including the visit ID, symptoms, discharge diagnosis, and prescription. Each clinical information has a unique clinical info ID.
- **APPOINTMENTS:** This entity represents appointments made by patients with healthcare providers, including the patient ID, provider ID, facility ID, and the date and time of the appointment. Each appointment has a unique appointment ID.
- **ORDERS:** This entity stores orders made by healthcare providers during a patient's visit, including the visit ID, test name, and test result. Each order has a unique order ID.
- **BEDS:** This entity stores information about beds in healthcare facilities, including the facility ID, bed number, and bed type. Each bed has a unique bed ID.
- **SUPPLIES:** This entity stores information about supplies in healthcare facilities, including the facility ID, supply name, and supply quantity. Each supply has a unique supply ID.

- **BILLING:** This entity stores information about the billing associated with a patient's visit, including the visit ID and billing amount. Each billing has a unique billing ID.
- **USERS:** This entity stores information about the users associated with a hospital, including the user ID and username,password,role. Each user has a unique user ID.
- **LOGT:** This entity stores information about the Logs associated with a hospital, including the log ID and user ID, table name,action type,previous value,new value,action time.Each log has a unique log ID.

UML Diagram



The table relationships for the above UML diagram can be described as follows:

- **PATIENTS** table has a many-to-one relationship with ADDRESSES and INSURANCES tables, as each patient can have only one address and one insurance, but each address and insurance can be associated with multiple patients.

- **VISITS** table has a many-to-one relationship with PATIENTS, PROVIDERS, and FACILITIES tables, as each visit can have only one patient, one provider, and one facility, but each patient, provider, and facility can have multiple visits.
- **CLINICAL_INFORMATION** table has a many-to-one relationship with VISITS table, as each visit can have only one set of clinical information.
- **APPOINTMENTS** table has a many-to-one relationship with PATIENTS, PROVIDERS, and FACILITIES tables, as each appointment can have only one patient, one provider, and one facility, but each patient, provider, and facility can have multiple appointments.
- **ORDERS** table has a many-to-one relationship with VISITS table, as each order can be associated with only one visit, but each visit can have multiple orders.
- **BEDS** and **SUPPLIES** tables have a many-to-one relationship with FACILITIES table, as each bed and supply can be associated with only one facility, but each facility can have multiple beds and supplies.
- **ADDRESSES** table has a one-to-many relationship with the PATIENTS table. This means that a patient can have only one address, but an address can be associated with multiple patients.
- **INSURANCES** table has a one-to-many relationship with the PATIENTS table. This means that a patient can have only one insurance, but an insurance company can be associated with multiple patients.
- **PROVIDERS** table has a many-to-many relationship with the VISITS table. This means that multiple providers can be associated with multiple visits, and vice versa.
- **FACILITIES** table has a many-to-one relationship with the BEDS, SUPPLIES, VISITS, and APPOINTMENTS tables. This means that a facility can have multiple beds, supplies, visits, and appointments, but each bed, supply, visit, and appointment can only be associated with one facility.
- **SUPPLIES** table has a many-to-one relationship with the FACILITIES table. This means that a facility can have multiple supplies, but each supply can only be associated with one facility.
- **BILLING** table has a many-to-one relationship with the VISITS table. This means that a visit can have only one billing, but each billing can only be associated with one visit.
- **USERS** table has a one-to-many relationship with the LOGT table. This means that a user can perform multiple actions that are recorded in the log, but each log entry can only be associated with one user.

- **LOGT** table has a many-to-one relationship with the USERS table, as multiple log entries can be associated with a single user. The user_id column in the LOGT table serves as a foreign key referencing the user_id column in the USERS table.

Proof of BCNF

Proving that a set of tables are in Boyce-Codd Normal Form (BCNF) involves verifying that each table meets the following criteria:

- Each non-trivial functional dependency in the table is a dependency on a super key.
- There are no non-trivial dependencies between candidate keys.
- All attributes are functionally dependent on the primary key.

Based on our schema, we can verify that the tables are in BCNF by examining each table and its relationships to other tables in the schema:

- **ADDRESSES** table has a primary key address_id, and there are no non-trivial functional dependencies in the table.
- **INSURANCES** table has a primary key insurance_id, and there are no non-trivial functional dependencies in the table.
- **PROVIDERS** table has a primary key provider_id, and there are no non-trivial functional dependencies in the table.
- **FACILITIES** table has a primary key facility_id, and there are no non-trivial functional dependencies in the table.
- **PATIENTS** table has a primary key patient_id. The address_id and insurance_id columns are foreign keys that reference the ADDRESSES and INSURANCES tables, respectively. All non-key attributes in the table are functionally dependent on the primary key.
- **VISITS** table has a primary key visit_id. The patient_id, provider_id, and facility_id columns are foreign keys that reference the PATIENTS, PROVIDERS, and FACILITIES tables, respectively. All non-key attributes in the table are functionally dependent on the primary key.
- **CLINICAL_INFORMATION** table has a primary key clinical_info_id. The visit_id column is a foreign key that references the VISITS table. All non-key attributes in the table are functionally dependent on the primary key.
- **APPOINTMENTS** table has a primary key appointment_id. The patient_id, provider_id, and facility_id columns are foreign keys that reference the

PATIENTS, PROVIDERS, and FACILITIES tables, respectively. All non-key attributes in the table are functionally dependent on the primary key.

- **ORDERS** table has a primary key order_id. The visit_id column is a foreign key that references the VISITS table. All non-key attributes in the table are functionally dependent on the primary key.
- **BEDS** table has a primary key bed_id. The facility_id column is a foreign key that references the FACILITIES table. All non-key attributes in the table are functionally dependent on the primary key.
- **SUPPLIES** table has a primary key supply_id. The facility_id column is a foreign key that references the FACILITIES table. All non-key attributes in the table are functionally dependent on the primary key.
- **BILLING** table has a primary key billing_id. The visit_id column is a foreign key that references the VISITS table. All non-key attributes in the table are functionally dependent on the primary key.
- **USERS** table has a primary key user_id uniquely identifies each user. The username column has a unique constraint, indicating that each username must be unique. This ensures that there are no duplicate usernames in the table. The password and role columns are functionally dependent on the user_id column, which is the primary key. There are no non-key attributes that depend on non-key attributes, satisfying the requirements of BCNF.
- **LOGT** table has a primary key log_id uniquely identifies each log entry. The user_id column is a foreign key that references the user_id column in the USERS table. This establishes a relationship between the two tables. The table_name, action_type, previous_value, new_value and action_time columns are all functionally dependent on the log_id column, which is the primary key. Again, there are no non-key attributes that depend on non-key attributes, meeting the conditions of BCNF.

Therefore, we can conclude that all tables in the database are in BCNF.

Table Information

The database of Electronic Health Records (EHRs) in emergency rooms contains digital records of patient health information collected during emergency room visits. These records include details such as patient demographics, medical history, medications, allergies, vital signs, lab results, imaging studies, diagnoses, and treatment plans.

Table 1: Patients

- patient_id
- first_name
- last_name
- date_of_birth
- gender
- address_id
- insurance_id

Table 2: Providers

- provider_id
- first_name
- last_name
- specialty

Table 3: Addresses

- address_id
- street_address
- city
- state
- zip_code

Table 4: Insurances

- insurance_id
- insurance_company_name
- policy_number

Table 5: Visits

- visit_id

- patient_id
- provider_id
- visit_time
- facility_id

Table 6: Clinical Information

- clinical_info_id
- visit_id
- symptom
- discharge_diagnosis
- prescription

Table 7: Facilities

- facility_id
- facility_name
- facility_type

Table 8: Appointments

- appointment_id
- patient_id
- provider_id
- appointment_time
- facility_id

Table 9: Beds

- bed_id
- facility_id
- bed_number
- bed_type

Table 10: Supplies

- supply_id
- facility_id
- supply_name
- supply_quantity

Table 11: Orders

- order_id
- visit_id
- test_name
- test_result

Table 12: Billing:

- billing_id
- visit_id
- billing_amount

Table 13: Users:

- user_id
- username
- password
- role

Table 14: Logt:

- log_id
- user_id
- table_name
- action_type
- previous_value
- new_value
- action_time

Stored Procedures

Stored procedures are pre-compiled and stored database programs that can be invoked and executed as needed. They serve to encapsulate a set of SQL statements into a reusable routine, callable by applications or other stored procedures. The benefits of using stored procedures include improved performance, heightened security, and the promotion of code reusability.

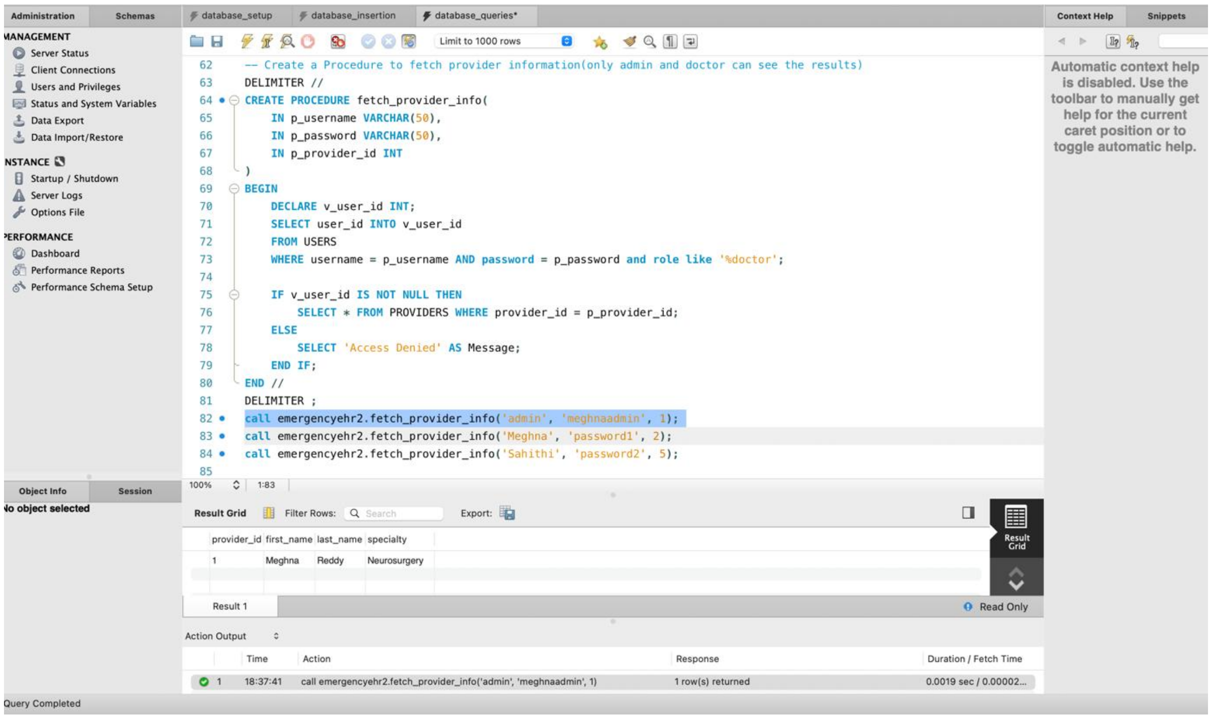
fetch_provider_info

Within the system, there exists a stored procedure named "fetch_provider_info," designed to retrieve provider information based on the provided username, password, and provider ID. This procedure adheres to a specific logic for user authentication and subsequent retrieval processes.

The "fetch_provider_info" stored procedure operates as follows:

- Acceptance of three input parameters: p_username (VARCHAR), p_password (VARCHAR), and p_provider_id (INT).
- Declaration of a variable named v_user_id to store the user ID.
- User authentication through a query to the USERS table, verifying a match for the provided username, password, and the user's role as a doctor.
- Upon successful authentication, assignment of the user's ID to the v_user_id variable.
- Verification of v_user_id not being null, indicating successful authentication.
- If authenticated, execution of a SELECT statement to retrieve provider details from the PROVIDERS table based on the provided provider ID.
- If a match is found, the result of the SELECT statement is returned, presenting the provider information.
- In cases where the user is not authenticated or lacks authorization as a doctor, the procedure returns a result set with the message "Access Denied."

In summary, the "fetch_provider_info" stored procedure in the system performs user authentication, verifies authorization based on the user's role, and retrieves provider information if the user is authenticated and authorized as a doctor. This ensures access control and yields either the provider details or an "Access Denied" message accordingly.



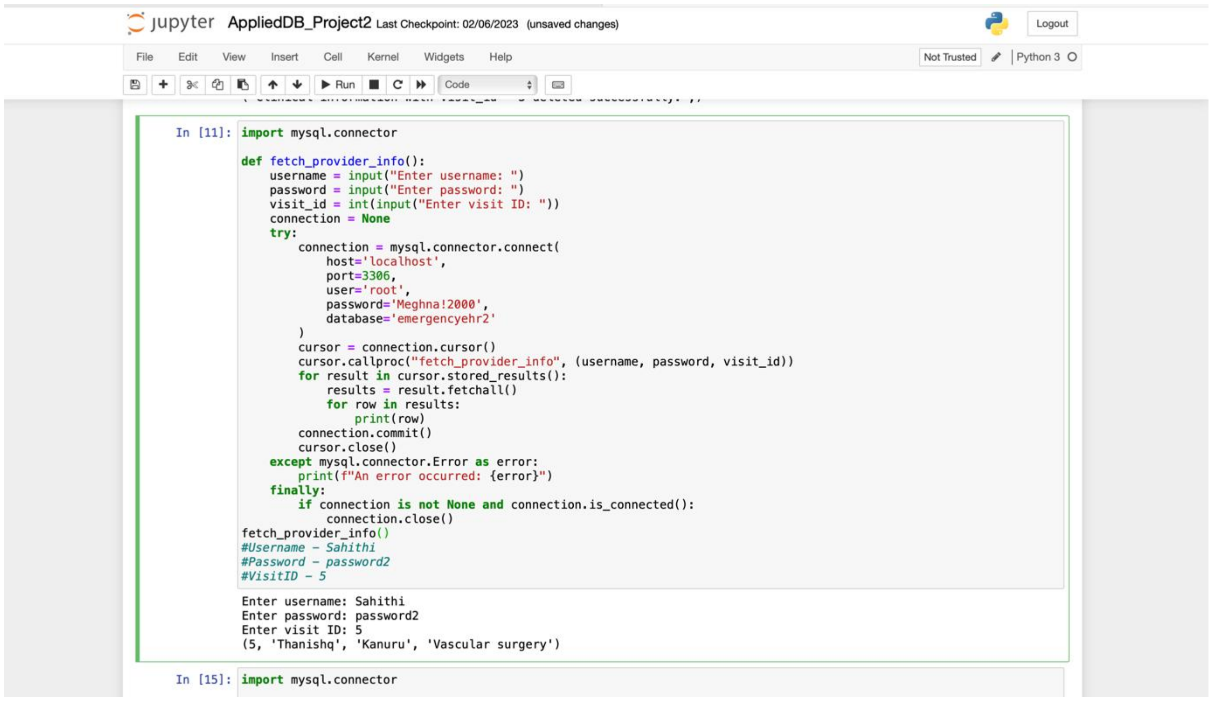
```sql
-- Create a Procedure to fetch provider information(only admin and doctor can see the results)
DELIMITER //
CREATE PROCEDURE fetch_provider_info(
    IN p_username VARCHAR(50),
    IN p_password VARCHAR(50),
    IN p_provider_id INT
)
BEGIN
    DECLARE v_user_id INT;
    SELECT user_id INTO v_user_id
    FROM USERS
    WHERE username = p_username AND password = p_password and role like '%doctor';

    IF v_user_id IS NOT NULL THEN
        SELECT * FROM PROVIDERS WHERE provider_id = p_provider_id;
    ELSE
        SELECT 'Access Denied' AS Message;
    END IF;
END //
DELIMITER ;
call emergencyehr2.fetch_provider_info('admin', 'meghnaadmin', 1);
call emergencyehr2.fetch_provider_info('Meghna', 'password1', 2);
call emergencyehr2.fetch_provider_info('Sahithi', 'password2', 5);
```

provider_id	first_name	last_name	specialty
1	Meghna	Reddy	Neurosurgery

Action Output

		Time	Action	Response	Duration / Fetch Time
✓	1	18:37:41	call emergencyehr2.fetch_provider_info('admin', 'meghnaadmin', 1)	1 row(s) returned	0.0019 sec / 0.00002...

Query Completed



jupyter AppliedDB_Project2 Last Checkpoint: 02/06/2023 (unsaved changes)

```python
In [11]: import mysql.connector

         def fetch_provider_info():
             username = input("Enter username: ")
             password = input("Enter password: ")
             visit_id = int(input("Enter visit ID: "))
             connection = None
             try:
                 connection = mysql.connector.connect(
                     host='localhost',
                     port=3306,
                     user='root',
                     password='Meghna!2000',
                     database='emergencyehr2'
                 )
                 cursor = connection.cursor()
                 cursor.callproc("fetch_provider_info", (username, password, visit_id))
                 for result in cursor.stored_results():
                     results = result.fetchall()
                     for row in results:
                         print(row)
                 connection.commit()
                 cursor.close()
             except mysql.connector.Error as error:
                 print(f"An error occurred: {error}")
             finally:
                 if connection is not None and connection.is_connected():
                     connection.close()
         fetch_provider_info()
         #Username - Sahithi
         #Password - password2
         #VisitID - 5
```

```
Enter username: Sahithi
Enter password: password2
Enter visit ID: 5
(5, 'Thanishq', 'Kanuru', 'Vascular surgery')
```

```python
In [15]: import mysql.connector
```

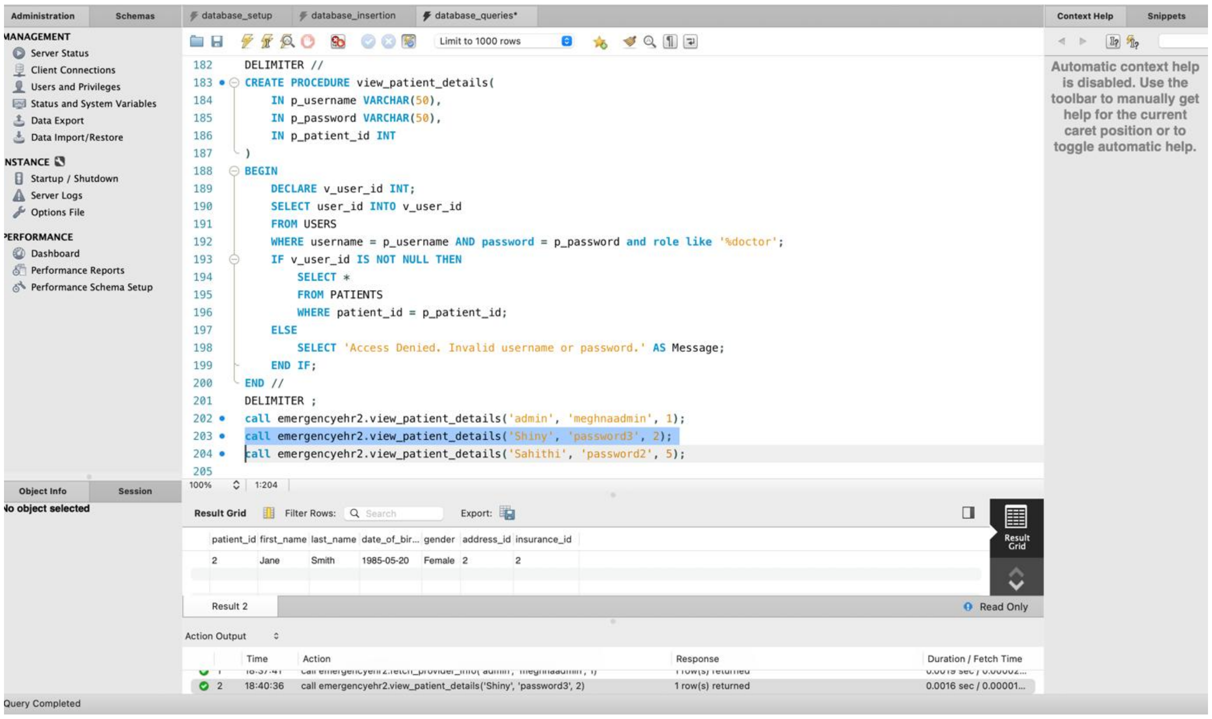
view_patient_details:

The "view_patient_details" stored procedure within the system is designed to fetch patient details by utilizing the provided username, password, and patient ID. This procedure adheres to a specific logic for user authentication and subsequent data retrieval.

Here is an overview of how the "view_patient_details" stored procedure operates:

- The procedure takes three input parameters: p_username (VARCHAR), p_password (VARCHAR), and p_patient_id (INT).
- A variable named v_user_id is declared within the procedure to store the user ID.
- User authentication is performed by querying the USERS table. It verifies if there is a match for the given username, password, and the user's role as a doctor.
- Upon finding a match, the user is considered authenticated, and their user ID is assigned to the v_user_id variable.
- The procedure then checks if v_user_id is not null, signifying successful authentication.
- If the user is authenticated, the procedure executes a SELECT statement to retrieve patient details from the PATIENTS table based on the provided patient ID.
- In the event of a match, the result of the SELECT statement is returned, providing the patient details.
- If the user is not authenticated or lacks authorization as a doctor, the procedure returns a result set with the message "Access Denied. Invalid username or password."

In summary, the "view_patient_details" stored procedure ensures user authentication, verifies authorization based on the user's role, and retrieves patient details if the user is authenticated and authorized as a doctor. It upholds access control and delivers either the patient details or an "Access Denied. Invalid username or password." message accordingly.

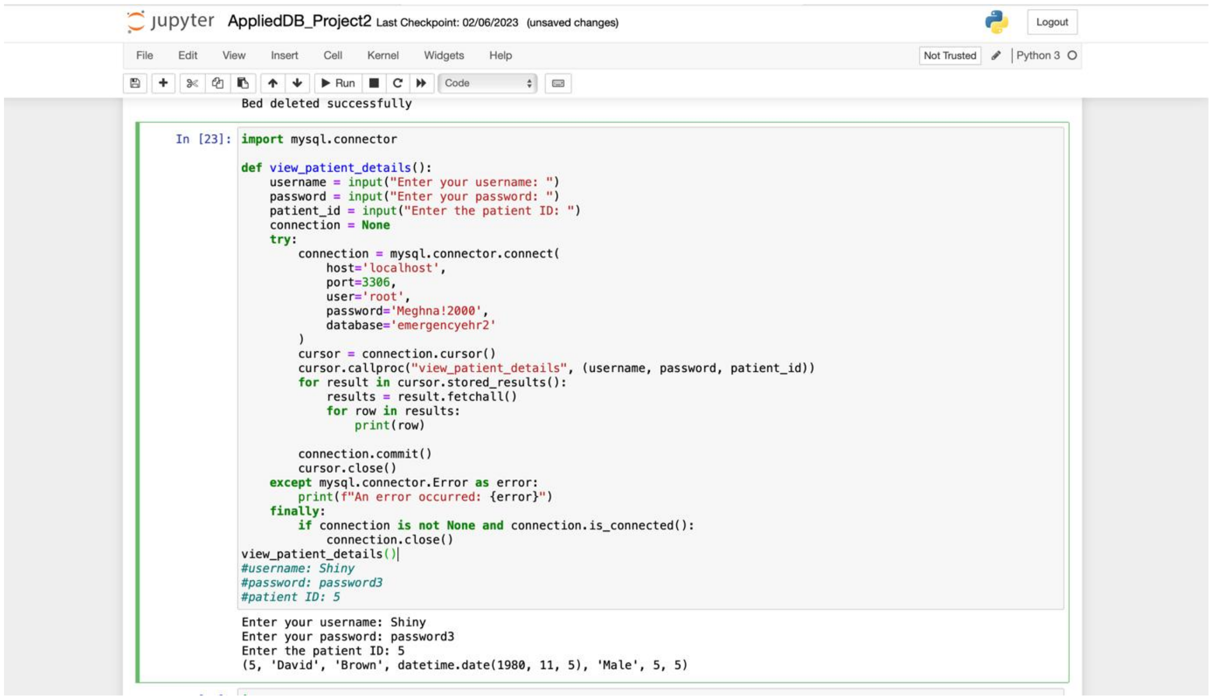


```sql
DELIMITER //
CREATE PROCEDURE view_patient_details(
    IN p_username VARCHAR(50),
    IN p_password VARCHAR(50),
    IN p_patient_id INT
)
BEGIN
    DECLARE v_user_id INT;
    SELECT user_id INTO v_user_id
    FROM USERS
    WHERE username = p_username AND password = p_password and role like '%doctor';
    IF v_user_id IS NOT NULL THEN
        SELECT *
        FROM PATIENTS
        WHERE patient_id = p_patient_id;
    ELSE
        SELECT 'Access Denied. Invalid username or password.' AS Message;
    END IF;
END //
DELIMITER ;
call emergencyehr2.view_patient_details('admin', 'meghnaadmin', 1);
call emergencyehr2.view_patient_details('Shiny', 'password3', 2);
call emergencyehr2.view_patient_details('Sahithi', 'password2', 5);
```

patient_id	first_name	last_name	date_of_bir...	gender	address_id	insurance_id
2	Jane	Smith	1985-05-20	Female	2	2

```
   Time      Action                                                        Response           Duration / Fetch Time
2  18:40:36  call emergencyehr2.view_patient_details('Shiny', 'password3', 2)  1 row(s) returned  0.0016 sec / 0.00001...
```

Query Completed



```
Bed deleted successfully
```

In [23]:

```python
import mysql.connector

def view_patient_details():
    username = input("Enter your username: ")
    password = input("Enter your password: ")
    patient_id = input("Enter the patient ID: ")
    connection = None
    try:
        connection = mysql.connector.connect(
            host='localhost',
            port=3306,
            user='root',
            password='Meghna!2000',
            database='emergencyehr2'
        )
        cursor = connection.cursor()
        cursor.callproc("view_patient_details", (username, password, patient_id))
        for result in cursor.stored_results():
            results = result.fetchall()
            for row in results:
                print(row)

        connection.commit()
        cursor.close()
    except mysql.connector.Error as error:
        print(f"An error occurred: {error}")
    finally:
        if connection is not None and connection.is_connected():
            connection.close()
view_patient_details()
#username: Shiny
#password: password3
#patient ID: 5
```

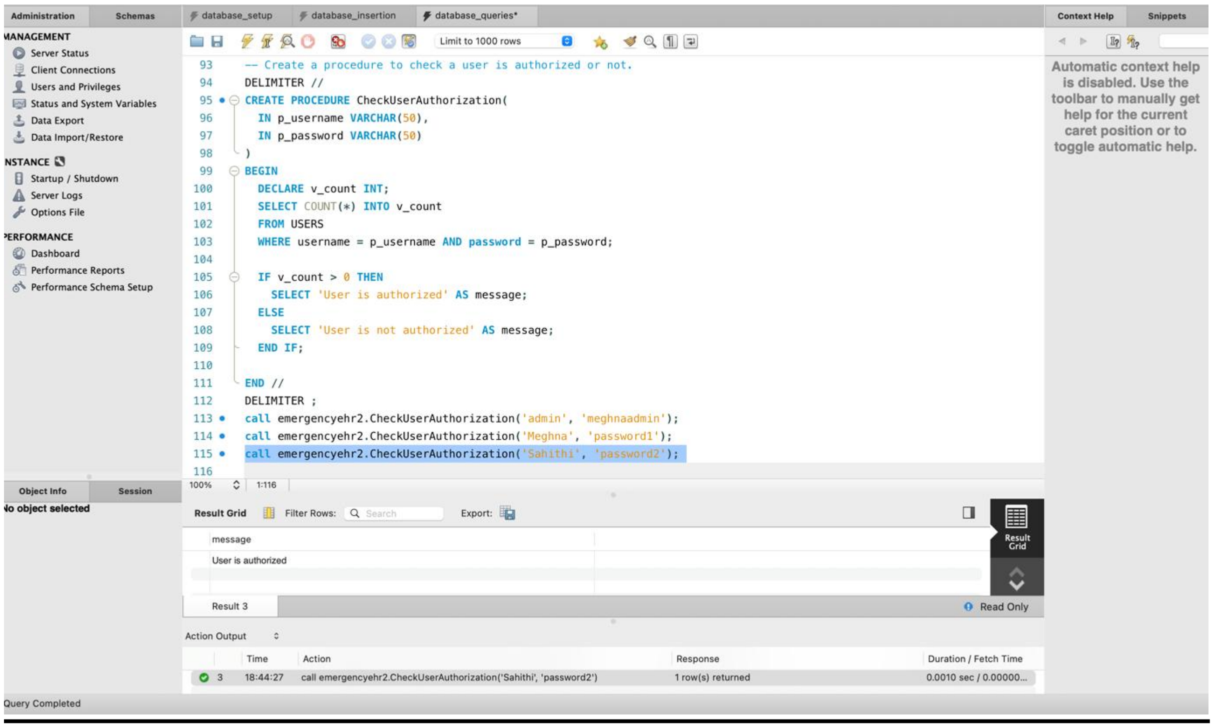
```
Enter your username: Shiny
Enter your password: password3
Enter the patient ID: 5
(5, 'David', 'Brown', datetime.date(1980, 11, 5), 'Male', 5, 5)
```

User Authentication:

User Authentication stands as a pivotal component in systems managing user access and privileges, ensuring that only duly authorized users gain entry to the system and its functionalities. Within the outlined system, the process of user authentication unfolds in the following manner:

- A Users table functions as the repository for user data, encompassing usernames, passwords, and roles.
- Authentication of a user hinges on the utilization of the "**CheckUserAuthorization**" stored procedure.
- This stored procedure takes two input parameters: "p_username" and "p_password," representing the user-provided username and password.
- Within the procedure, a variable named "v_count" is instantiated to store the count of matching records identified in the "USERS" table for the specified username and password.
- The SELECT statement embedded in the procedure conducts a query on the "USERS" table to verify the existence of a match for the provided username and password.
- The count of matching records is then assigned to the "v_count" variable.
- Subsequently, the procedure evaluates the value of "v_count" to ascertain the user's authorization status.
- If "v_count" exceeds 0, signifying a match, the user is deemed authorized. In this scenario, a result set conveying the message "User is authorized" is returned.
- Conversely, if "v_count" is 0, indicating no match, the user is not authorized. In this case, a result set featuring the message "User is not authorized" is returned.

In the system, users with the "admin_doctor" role have administrative privileges, while users with the "doctor" role have doctor-level access. Users with the "patient" role have patient-level access. Users with the "biller" role have biller-level access.

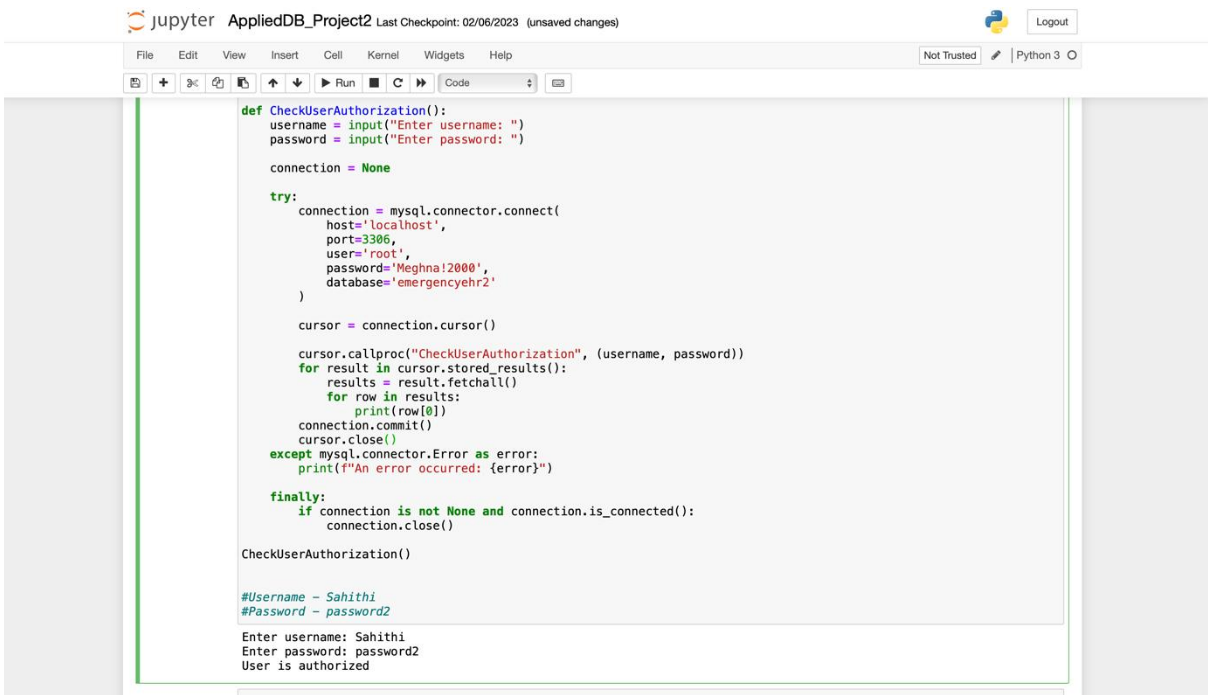


```sql
-- Create a procedure to check a user is authorized or not.
DELIMITER //
CREATE PROCEDURE CheckUserAuthorization(
  IN p_username VARCHAR(50),
  IN p_password VARCHAR(50)
)
BEGIN
  DECLARE v_count INT;
  SELECT COUNT(*) INTO v_count
  FROM USERS
  WHERE username = p_username AND password = p_password;

  IF v_count > 0 THEN
    SELECT 'User is authorized' AS message;
  ELSE
    SELECT 'User is not authorized' AS message;
  END IF;

END //
DELIMITER ;
call emergencyehr2.CheckUserAuthorization('admin', 'meghnaadmin');
call emergencyehr2.CheckUserAuthorization('Meghna', 'password1');
call emergencyehr2.CheckUserAuthorization('Sahithi', 'password2');
```

message
User is authorized

	Time	Action	Response	Duration / Fetch Time
3	18:44:27	call emergencyehr2.CheckUserAuthorization('Sahithi', 'password2')	1 row(s) returned	0.0010 sec / 0.00000...

Query Completed



```python
def CheckUserAuthorization():
    username = input("Enter username: ")
    password = input("Enter password: ")

    connection = None

    try:
        connection = mysql.connector.connect(
            host='localhost',
            port=3306,
            user='root',
            password='Meghna!2000',
            database='emergencyehr2'
        )

        cursor = connection.cursor()

        cursor.callproc("CheckUserAuthorization", (username, password))
        for result in cursor.stored_results():
            results = result.fetchall()
            for row in results:
                print(row[0])
        connection.commit()
        cursor.close()
    except mysql.connector.Error as error:
        print(f"An error occurred: {error}")

    finally:
        if connection is not None and connection.is_connected():
            connection.close()

CheckUserAuthorization()


#Username - Sahithi
#Password - password2
```

```
Enter username: Sahithi
Enter password: password2
User is authorized
```

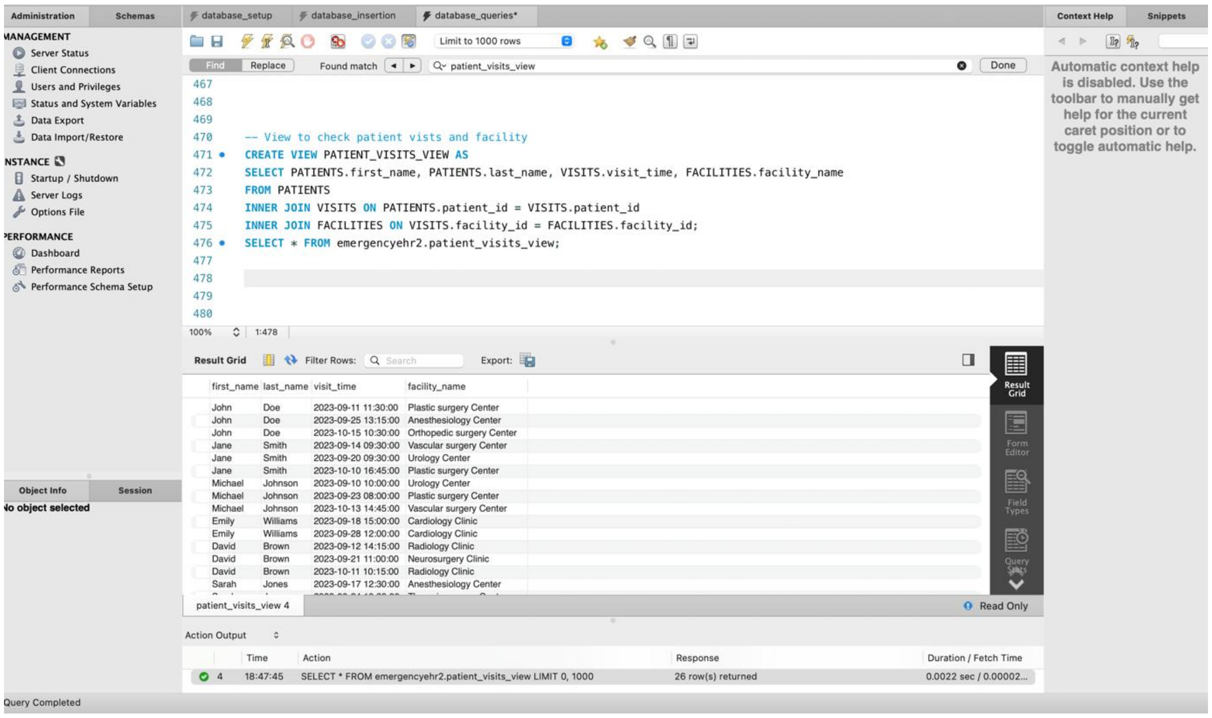
Views:

Views within a database serve as virtual tables, generated from query results. They offer a means to streamline intricate queries, encapsulate business logic, and furnish users with a tailored perspective of the data. In the outlined system, views play a pivotal role in facilitating data access and delivering convenient avenues for obtaining specific information.

Within this framework, the PATIENT_VISITS_VIEW provides several advantages:

- **Streamlined Access to Patient Visit Information:** The view simplifies the retrieval of patient visit details by consolidating pertinent data from the PATIENTS, VISITS, and FACILITIES tables. Users can query the view to extract the patient's first and last name, visit time, and the associated facility name.
- **Enhanced Data Readability:** Through the amalgamation of essential tables and selective column choices, the view presents data in a lucid and succinct manner. Users can readily comprehend and interpret patient visit information without navigating through multiple tables or deciphering complex query structures.
- **Data Consolidation:** The view merges data from disparate tables into a unified logical entity, obviating the need for users to manually conduct joins and manage relationships. This consolidation enhances data access efficiency and convenience.
- **Data Integrity Assurance:** Maintaining referential integrity between the PATIENTS, VISITS, and FACILITIES tables, the view ensures that only valid and interrelated data is included in its result set. The joins in the view are grounded in patient ID and facility ID.
- **Query Flexibility:** Users can exploit the view to execute queries related to patient visit information, without grappling with the intricacies of the underlying table structure. The view abstracts the complexities of joins, presenting a simplified interface for querying patient visit data.
- **Potential for Performance Optimization:** Depending on the intricacy of underlying queries, the view may enhance performance by pre-computing and caching results. Storing the view's result set allows subsequent queries to be executed more efficiently, thereby reducing overall processing time.

In summary, the PATIENT_VISITS_VIEW in the system streamlines data access, improves data readability, consolidates related information, ensures data integrity, provides query flexibility, and potentially optimizes performance. It offers an efficient and user-friendly means of retrieving patient visit information from the database.



```sql
-- View to check patient vists and facility
CREATE VIEW PATIENT_VISITS_VIEW AS
SELECT PATIENTS.first_name, PATIENTS.last_name, VISITS.visit_time, FACILITIES.facility_name
FROM PATIENTS
INNER JOIN VISITS ON PATIENTS.patient_id = VISITS.patient_id
INNER JOIN FACILITIES ON VISITS.facility_id = FACILITIES.facility_id;
SELECT * FROM emergencyehr2.patient_visits_view;
```

first_name	last_name	visit_time	facility_name
John	Doe	2023-09-11 11:30:00	Plastic surgery Center
John	Doe	2023-09-25 13:15:00	Anesthesiology Center
John	Doe	2023-10-15 10:30:00	Orthopedic surgery Center
Jane	Smith	2023-09-14 09:30:00	Vascular surgery Center
Jane	Smith	2023-09-20 09:30:00	Urology Center
Jane	Smith	2023-10-10 16:45:00	Plastic surgery Center
Michael	Johnson	2023-09-10 10:00:00	Urology Center
Michael	Johnson	2023-09-23 08:00:00	Plastic surgery Center
Michael	Johnson	2023-10-13 14:45:00	Vascular surgery Center
Emily	Williams	2023-09-18 15:00:00	Cardiology Clinic
Emily	Williams	2023-09-28 12:00:00	Cardiology Clinic
David	Brown	2023-09-12 14:15:00	Radiology Clinic
David	Brown	2023-09-21 11:00:00	Neurosurgery Clinic
David	Brown	2023-10-11 10:15:00	Radiology Clinic
Sarah	Jones	2023-09-17 12:30:00	Anesthesiology Center

```
   Time      Action                                                           Response              Duration / Fetch Time
4  18:47:45  SELECT * FROM emergencyehr2.patient_visits_view LIMIT 0, 1000    26 row(s) returned    0.0022 sec / 0.00002...
```

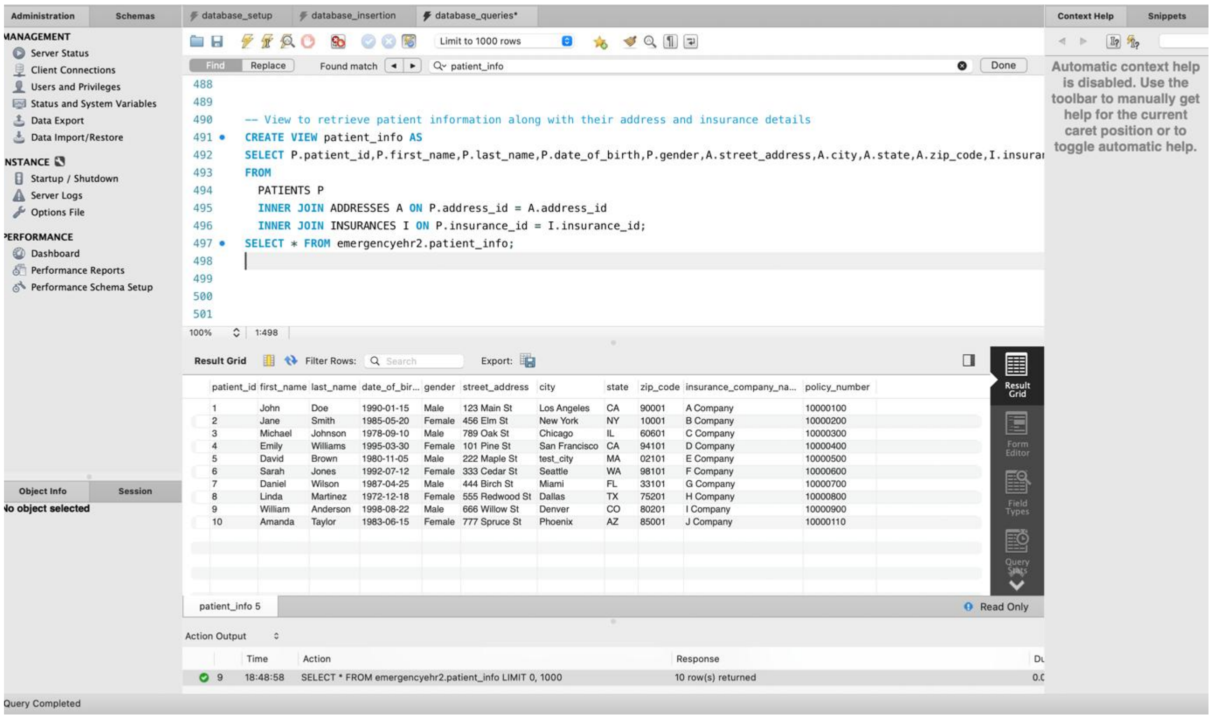
patient_info view in the system provides the following benefits:

- **Comprehensive Patient Information:** This view amalgamates data from the PATIENTS, ADDRESSES, and INSURANCES tables, encompassing crucial details like patient ID, first and last names, date of birth, gender, complete address (including street, city, state, and zip code), insurance company name, and policy number. This consolidation facilitates the seamless retrieval and analysis of exhaustive patient records.
- **Streamlined Data Retrieval:** Users can effortlessly query the patient_info view to access all pertinent patient information, eliminating the need to manually join multiple tables. The view takes care of the requisite joins between the PATIENTS, ADDRESSES, and INSURANCES tables, presenting a user-friendly interface for obtaining patient details, addresses, and insurance information.
- **Improved Data Readability:** The view organizes patient information in a clear format, enhancing user understanding and analysis. By selectively choosing columns and joining pertinent tables, the view eliminates extraneous data, offering a concise representation of patient records.
- **Data Integrity:** The patient_info view upholds data integrity by preserving referential integrity between the PATIENTS, ADDRESSES, and INSURANCES

tables. The joins are based on corresponding IDs (address_id and insurance_id), ensuring that data retrieved from the view is both valid and consistent.

- **Data Consistency:** Derived from underlying tables, the view automatically reflects any updates or changes made to the PATIENTS, ADDRESSES, or INSURANCES tables. This guarantees that the patient_info view consistently provides current and coherent patient information.
- **Enhanced Efficiency:** Through pre-computing join operations and storing results in the view, subsequent queries for patient information can be executed more efficiently. The view negates the need for repetitive complex joins, thereby improving query performance and reducing processing time.

In summary, the patient_info view streamlines data retrieval, enhances data readability, ensures data integrity and consistency, and improves query efficiency. It offers a comprehensive and organized representation of patient information, including address and insurance details, facilitating user access and analysis of patient records within the system.



```
-- View to retrieve patient information along with their address and insurance details
CREATE VIEW patient_info AS
SELECT P.patient_id,P.first_name,P.last_name,P.date_of_birth,P.gender,A.street_address,A.city,A.state,A.zip_code,I.insurai
FROM
  PATIENTS P
  INNER JOIN ADDRESSES A ON P.address_id = A.address_id
  INNER JOIN INSURANCES I ON P.insurance_id = I.insurance_id;
SELECT * FROM emergencyehr2.patient_info;
```

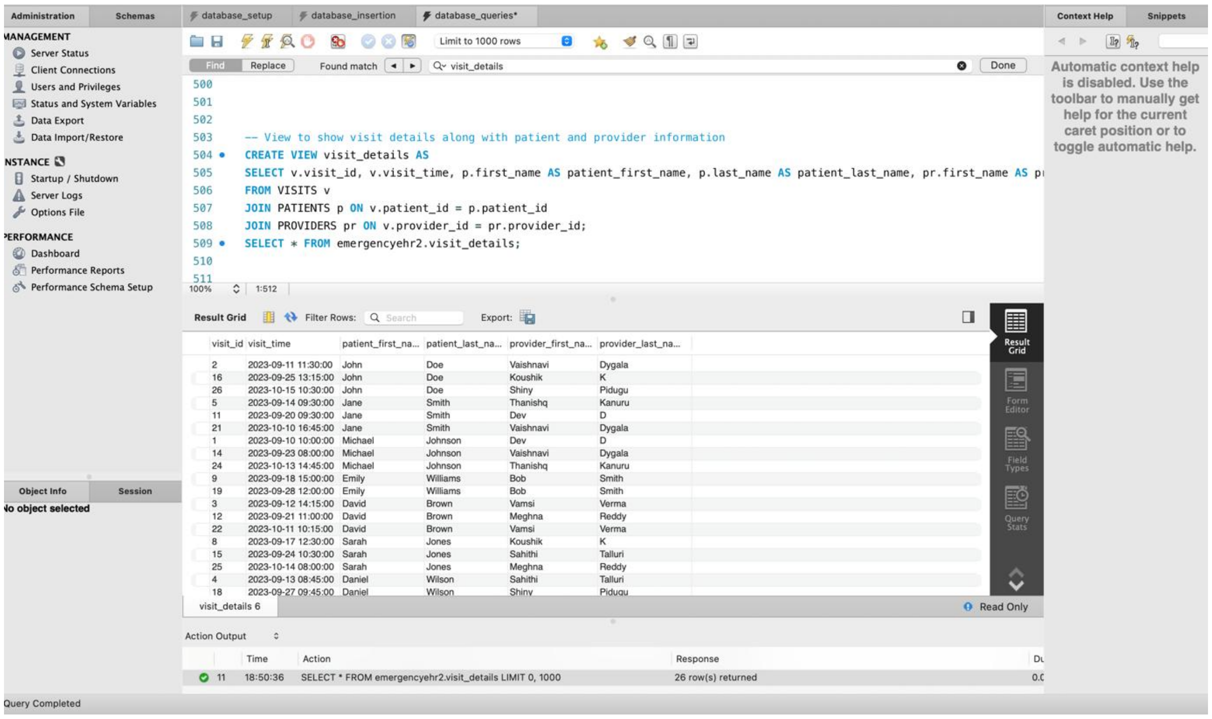
patient_id	first_name	last_name	date_of_bir...	gender	street_address	city	state	zip_code	insurance_company_na...	policy_number
1	John	Doe	1990-01-15	Male	123 Main St	Los Angeles	CA	90001	A Company	10000100
2	Jane	Smith	1985-05-20	Female	456 Elm St	New York	NY	10001	B Company	10000200
3	Michael	Johnson	1978-09-10	Male	789 Oak St	Chicago	IL	60601	C Company	10000300
4	Emily	Williams	1995-03-30	Female	101 Pine St	San Francisco	CA	94101	D Company	10000400
5	David	Brown	1980-11-05	Male	222 Maple St	test_city	MA	02101	E Company	10000500
6	Sarah	Jones	1992-07-12	Female	333 Cedar St	Seattle	WA	98101	F Company	10000600
7	Daniel	Wilson	1987-04-25	Male	444 Birch St	Miami	FL	33101	G Company	10000700
8	Linda	Martinez	1972-12-18	Female	555 Redwood St	Dallas	TX	75201	H Company	10000800
9	William	Anderson	1998-08-22	Male	666 Willow St	Denver	CO	80201	I Company	10000900
10	Amanda	Taylor	1983-06-15	Female	777 Spruce St	Phoenix	AZ	85001	J Company	10000110

		Time	Action	Response	Du
✓	9	18:48:58	SELECT * FROM emergencyehr2.patient_info LIMIT 0, 1000	10 row(s) returned	0.0

visit_details view in the system provides the following benefits:

- **Comprehensive Visit Information:** By amalgamating data from the VISITS, PATIENTS, and PROVIDERS tables, this view presents a thorough overview of visit details in one consolidated display. This includes the visit ID, visit time, patient's first and last name, and provider's first and last name.
- **Effortless Retrieval of Patient and Provider Information:** Users can swiftly access patient and provider details linked to each visit through the visit_details view, eliminating the manual process of joining multiple tables and streamlining data retrieval.
- **Enhanced Data Analysis:** The view facilitates data analysis and reporting by presenting visit details alongside pertinent patient and provider information. This empowers users to analyze visit patterns, discern trends, and assess the performance of specific providers within the context of patient visits.
- **Streamlined Querying:** Rather than crafting intricate join queries each time to fetch visit, patient, and provider details, users can directly query the visit_details view. This simplifies the process of query development, contributing to an overall improvement in system usability.

In summary, the visit_details view enhances the system's efficiency, convenience, and analytical capabilities by providing a consolidated and easily accessible presentation of visit information along with associated patient and provider details.



```sql
-- View to show visit details along with patient and provider information
CREATE VIEW visit_details AS
SELECT v.visit_id, v.visit_time, p.first_name AS patient_first_name, p.last_name AS patient_last_name, pr.first_name AS pı
FROM VISITS v
JOIN PATIENTS p ON v.patient_id = p.patient_id
JOIN PROVIDERS pr ON v.provider_id = pr.provider_id;
SELECT * FROM emergencyehr2.visit_details;
```

visit_id	visit_time	patient_first_na...	patient_last_na...	provider_first_na...	provider_last_na...
2	2023-09-11 11:30:00	John	Doe	Vaishnavi	Dygala
16	2023-09-25 13:15:00	John	Doe	Koushik	K
26	2023-10-15 10:30:00	John	Doe	Shiny	Pidugu
5	2023-09-14 09:30:00	Jane	Smith	Thanishq	Kanuru
11	2023-09-20 09:30:00	Jane	Smith	Dev	D
21	2023-10-10 16:45:00	Jane	Smith	Vaishnavi	Dygala
1	2023-09-10 10:00:00	Michael	Johnson	Dev	D
14	2023-09-23 08:00:00	Michael	Johnson	Vaishnavi	Dygala
24	2023-10-13 14:45:00	Michael	Johnson	Thanishq	Kanuru
9	2023-09-18 15:00:00	Emily	Williams	Bob	Smith
19	2023-09-28 12:00:00	Emily	Williams	Bob	Smith
3	2023-09-12 14:15:00	David	Brown	Vamsi	Verma
12	2023-09-21 11:00:00	David	Brown	Meghna	Reddy
22	2023-10-11 10:15:00	David	Brown	Vamsi	Verma
8	2023-09-17 12:30:00	Sarah	Jones	Koushik	K
15	2023-09-24 10:30:00	Sarah	Jones	Sahithi	Talluri
25	2023-10-14 08:00:00	Sarah	Jones	Meghna	Reddy
4	2023-09-13 08:45:00	Daniel	Wilson	Sahithi	Talluri
18	2023-09-27 09:45:00	Daniel	Wilson	Shiny	Pidugu

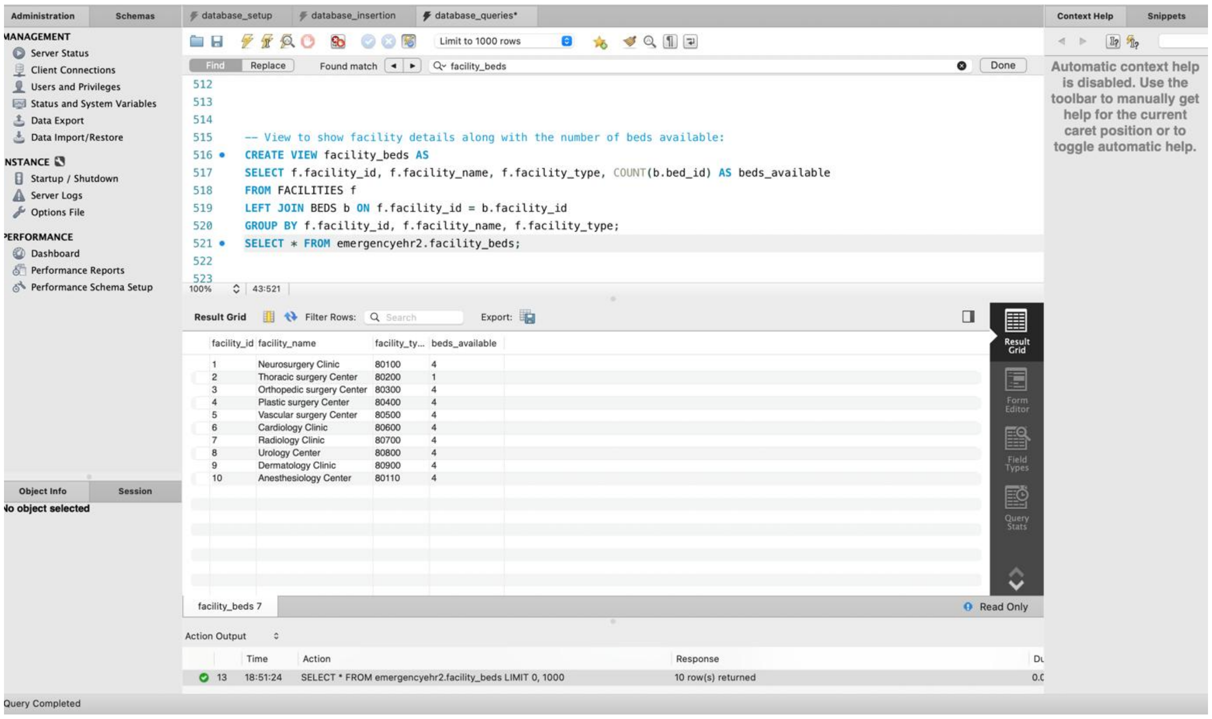
```
Action Output
     Time      Action                                              Response
✓ 11 18:50:36  SELECT * FROM emergencyehr2.visit_details LIMIT 0, 1000  26 row(s) returned
```

Query Completed

facility_beds view in the system offers the following benefits:

- **Facility Details:** The view combines data from the FACILITIES table with the number of available beds from the BEDS table. It includes information such as facility ID, facility name, and facility type. This allows users to quickly access details about each facility in the system.
- **Bed Availability:** The view provides the count of available beds for each facility. By performing a LEFT JOIN between the FACILITIES and BEDS tables and grouping the results by facility ID, name, and type, the view calculates the number of beds available at each facility. This information is valuable for tracking bed capacity and managing resource allocation.
- **Real-Time Updates:** Since the view is based on a query that combines the FACILITIES and BEDS tables, any changes in the bed availability or addition/removal of beds will be reflected in the view automatically. This ensures that the information presented in the view is up to date and accurate.
- **Simplified Data Analysis:** With the facility_beds view, users can easily analyze and compare the bed availability across different facilities. They can identify facilities with high bed occupancy or those that may require additional resources. This simplifies the process of monitoring and optimizing bed utilization in the system.
- **Streamlined Querying:** Instead of writing complex join queries and aggregations to retrieve facility details and bed availability, users can directly query the facility_beds view. This improves query performance and simplifies the process of retrieving this specific information.

Overall, the facility_beds view provides a consolidated and easily accessible view of facility details along with the number of available beds. It facilitates data analysis, resource management, and simplifies querying, making it a valuable tool in the system.



```sql
-- View to show facility details along with the number of beds available:
CREATE VIEW facility_beds AS
SELECT f.facility_id, f.facility_name, f.facility_type, COUNT(b.bed_id) AS beds_available
FROM FACILITIES f
LEFT JOIN BEDS b ON f.facility_id = b.facility_id
GROUP BY f.facility_id, f.facility_name, f.facility_type;
SELECT * FROM emergencyehr2.facility_beds;
```

facility_id	facility_name	facility_ty...	beds_available
1	Neurosurgery Clinic	80100	4
2	Thoracic surgery Center	80200	1
3	Orthopedic surgery Center	80300	4
4	Plastic surgery Center	80400	4
5	Vascular surgery Center	80500	4
6	Cardiology Clinic	80600	4
7	Radiology Clinic	80700	4
8	Urology Center	80800	4
9	Dermatology Clinic	80900	4
10	Anesthesiology Center	80110	4

```
   Time      Action                                                Response
13 18:51:24  SELECT * FROM emergencyehr2.facility_beds LIMIT 0, 1000  10 row(s) returned
```

log_details view in the system offers the following benefits:

- **Log Details:** The presented view amalgamates information from the LOGT table with the corresponding usernames retrieved from the USERS table. This includes details such as log ID, username, table name, action type, and action time. The purpose is to facilitate easy access and review of diverse actions executed within the system.

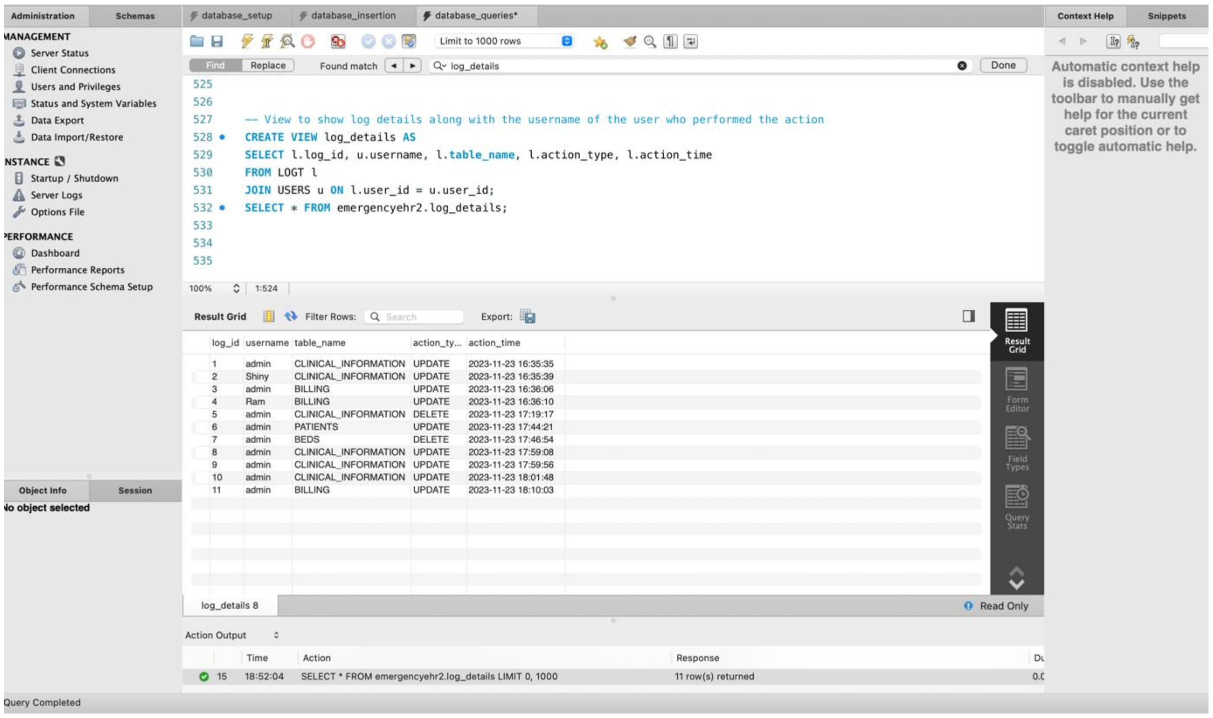
- **User Identification:** Through the association of the LOGT table with the USERS table using the user_id column, each log entry in the view is linked to the respective username. This linkage proves instrumental in identifying and attributing specific actions to individual users, thereby establishing accountability and traceability.

- **Audit Trail:** The log_details view effectively serves as an audit trail, documenting actions performed in the system. It empowers administrators or authorized users to scrutinize and monitor activities, ensuring adherence to compliance standards, bolstering security, and establishing accountability. This feature aids in investigating discrepancies, identifying unauthorized actions, and troubleshooting issues.

- **User Activity Analysis:** This view facilitates the analysis of user activity by presenting a consolidated overview of log details alongside associated usernames. Users can discern patterns, trends, or anomalies in user actions, contributing to the detection of suspicious or abnormal behavior.

- **Simplified Log Access:** Eliminating the need to directly query the LOGT table and perform subsequent joins with the USERS table, users can efficiently query the log_details view to retrieve log information paired with corresponding usernames. This streamlined approach enhances the efficiency of the querying process.
- **Data Privacy:** A crucial aspect of the view is its safeguarding of sensitive user IDs stored in the LOGT table. External exposure is mitigated by presenting usernames linked to each log entry, ensuring the preservation of user identification privacy.

In summary, the log_details view offers a comprehensive and user-friendly means of accessing log details alongside the usernames of individuals executing actions. It supports auditing, facilitates user activity analysis, and simplifies log access, rendering it an invaluable tool for monitoring system activities and upholding data security..



```
-- View to show log details along with the username of the user who performed the action
CREATE VIEW log_details AS
SELECT l.log_id, u.username, l.table_name, l.action_type, l.action_time
FROM LOGT l
JOIN USERS u ON l.user_id = u.user_id;
SELECT * FROM emergencyehr2.log_details;
```

log_id	username	table_name	action_ty...	action_time
1	admin	CLINICAL_INFORMATION	UPDATE	2023-11-23 16:35:35
2	Shiny	CLINICAL_INFORMATION	UPDATE	2023-11-23 16:35:39
3	admin	BILLING	UPDATE	2023-11-23 16:36:06
4	Ram	BILLING	UPDATE	2023-11-23 16:36:10
5	admin	CLINICAL_INFORMATION	DELETE	2023-11-23 17:19:17
6	admin	PATIENTS	UPDATE	2023-11-23 17:44:21
7	admin	BEDS	DELETE	2023-11-23 17:46:54
8	admin	CLINICAL_INFORMATION	UPDATE	2023-11-23 17:59:08
9	admin	CLINICAL_INFORMATION	UPDATE	2023-11-23 17:59:56
10	admin	CLINICAL_INFORMATION	UPDATE	2023-11-23 18:01:48
11	admin	BILLING	UPDATE	2023-11-23 18:10:03

	Time	Action	Response	Du
15	18:52:04	SELECT * FROM emergencyehr2.log_details LIMIT 0, 1000	11 row(s) returned	0.0

Indexes :

Indexes in a database system are used to improve query performance by allowing faster retrieval of data. They work by creating a separate data structure that contains a subset of the data from the indexed column(s), along with a reference to the original data. This data structure enables the database system to locate specific rows quickly when querying based on the indexed column(s).

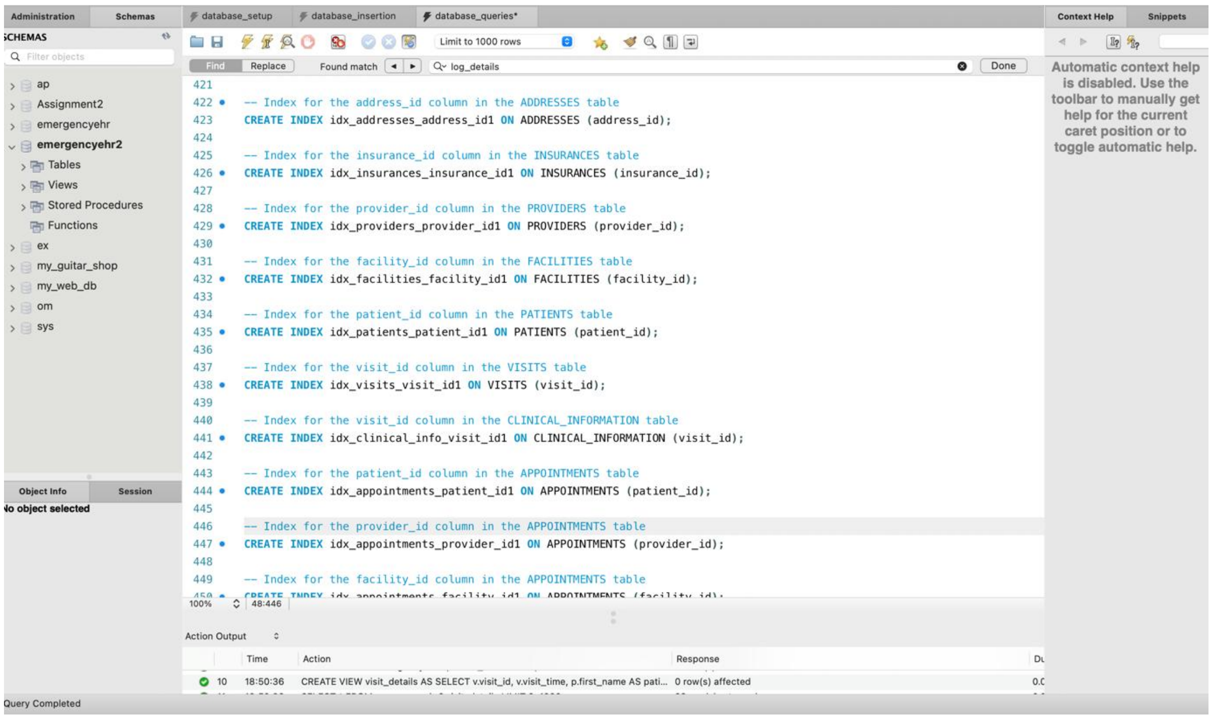
idx_addresses_address_id: This index is created on the address_id column in the ADDRESSES table. It improves the performance of queries that involve searching or filtering based on the address_id.

idx_insurances_insurance_id: This index is created on the insurance_id column in the INSURANCES table. It enhances the retrieval speed of data when querying using the insurance_id.

- **idx_providers_provider_id:** This index is created on the provider_id column in the PROVIDERS table. It facilitates faster data access for queries that involve searching or joining based on the provider_id.
- **idx_facilities_facility_id:** This index is created on the facility_id column in the FACILITIES table. It speeds up queries that involve filtering or joining on the facility_id.
- **idx_patients_patient_id:** This index is created on the patient_id column in the PATIENTS table. It improves the performance of queries that involve searching, joining, or filtering based on the patient_id.
- **idx_visits_visit_id:** This index is created on the visit_id column in the VISITS table. It enhances the retrieval speed of data when querying using the visit_id.
- **idx_clinical_info_visit_id:** This index is created on the visit_id column in the CLINICAL_INFORMATION table. It facilitates faster data access for queries that involve searching or joining based on the visit_id.
- **idx_appointments_patient_id:** This index is created on the patient_id column in the APPOINTMENTS table. It speeds up queries that involve filtering or joining on the patient_id.
- **idx_appointments_provider_id:** This index is created on the provider_id column in the APPOINTMENTS table. It improves the performance of queries that involve searching, joining, or filtering based on the provider_id.
- **idx_appointments_facility_id:** This index is created on the facility_id column in the APPOINTMENTS table. It enhances the retrieval speed of data when querying using the facility_id.

- **idx_orders_visit_id:** This index is created on the visit_id column in the ORDERS table. It facilitates faster data access for queries that involve searching or joining based on the visit_id.
- **idx_beds_facility_id:** This index is created on the facility_id column in the BEDS table. It speeds up queries that involve filtering or joining on the facility_id.
- **idx_supplies_facility_id:** This index is created on the facility_id column in the SUPPLIES table. It improves the performance of queries that involve searching, joining, or filtering based on the facility_id.
- **idx_billing_visit_id:** This index is created on the visit_id column in the BILLING table. It enhances the retrieval speed of data when querying using the visit_id.
- **idx_logt_user_id:** This index is created on the user_id column in the LOGT table. It facilitates faster data access for queries that involve searching or joining based on the user_id.

Overall, these indexes help optimize query performance by providing quicker access to data based on the indexed columns, resulting in improved overall system performance.



```sql

-- Index for the address_id column in the ADDRESSES table
CREATE INDEX idx_addresses_address_id1 ON ADDRESSES (address_id);

-- Index for the insurance_id column in the INSURANCES table
CREATE INDEX idx_insurances_insurance_id1 ON INSURANCES (insurance_id);

-- Index for the provider_id column in the PROVIDERS table
CREATE INDEX idx_providers_provider_id1 ON PROVIDERS (provider_id);

-- Index for the facility_id column in the FACILITIES table
CREATE INDEX idx_facilities_facility_id1 ON FACILITIES (facility_id);

-- Index for the patient_id column in the PATIENTS table
CREATE INDEX idx_patients_patient_id1 ON PATIENTS (patient_id);

-- Index for the visit_id column in the VISITS table
CREATE INDEX idx_visits_visit_id1 ON VISITS (visit_id);

-- Index for the visit_id column in the CLINICAL_INFORMATION table
CREATE INDEX idx_clinical_info_visit_id1 ON CLINICAL_INFORMATION (visit_id);

-- Index for the patient_id column in the APPOINTMENTS table
CREATE INDEX idx_appointments_patient_id1 ON APPOINTMENTS (patient_id);

-- Index for the provider_id column in the APPOINTMENTS table
CREATE INDEX idx_appointments_provider_id1 ON APPOINTMENTS (provider_id);

-- Index for the facility_id column in the APPOINTMENTS table
CREATE INDEX idx_appointments_facility_id1 ON APPOINTMENTS (facility_id);
```

```
   Time      Action                                                                Response            Du
10 18:50:36  CREATE VIEW visit_details AS SELECT v.visit_id, v.visit_time, p.first_name AS pati...  0 row(s) affected   0.0
```

Role Based Access

Role-based access control serves as a method to limit access to system resources based on the assigned roles of individual users within an organization. Under this framework, users are designated specific roles, and access privileges are granted to these roles rather than to individual users. These roles outline the permissions and actions that users are permitted to undertake within the system.

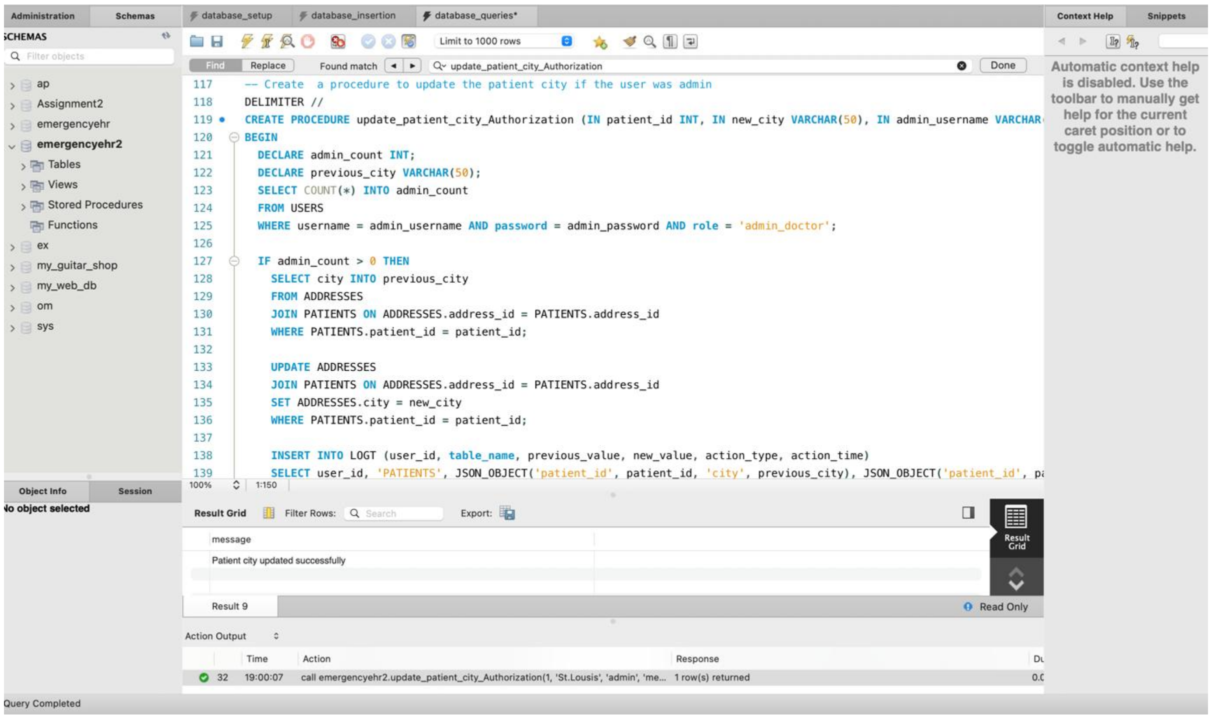
update_patient_city_Authorization:

The role-based access control approach is implemented to regulate the update of a patient's city, confining this task exclusively to users holding the "admin_doctor" role. The subsequent section provides an overview of the roles established in the system.

admin_doctor: This role is conferred upon users possessing administrative privileges alongside a doctor's designation. Users with this role enjoy enhanced access rights, including the capability to update a patient's city.

The "update_patient_city_Authorization" procedure verifies whether the provided username and password correspond to a user with the "admin_doctor" role. Upon a match, the procedure retrieves the previous city value, updates the patient's city with the new information, records the modification in the LOGT table, and issues a success message. In case of mismatched credentials or absence of the "admin_doctor" role, an appropriate access denial message is returned.

Employing the role-based access control approach ensures that only authorized users with the requisite roles can execute specific actions or access designated resources within the system. This segregation of permissions into roles streamlines user management and augments security by mitigating the risk of unauthorized access.

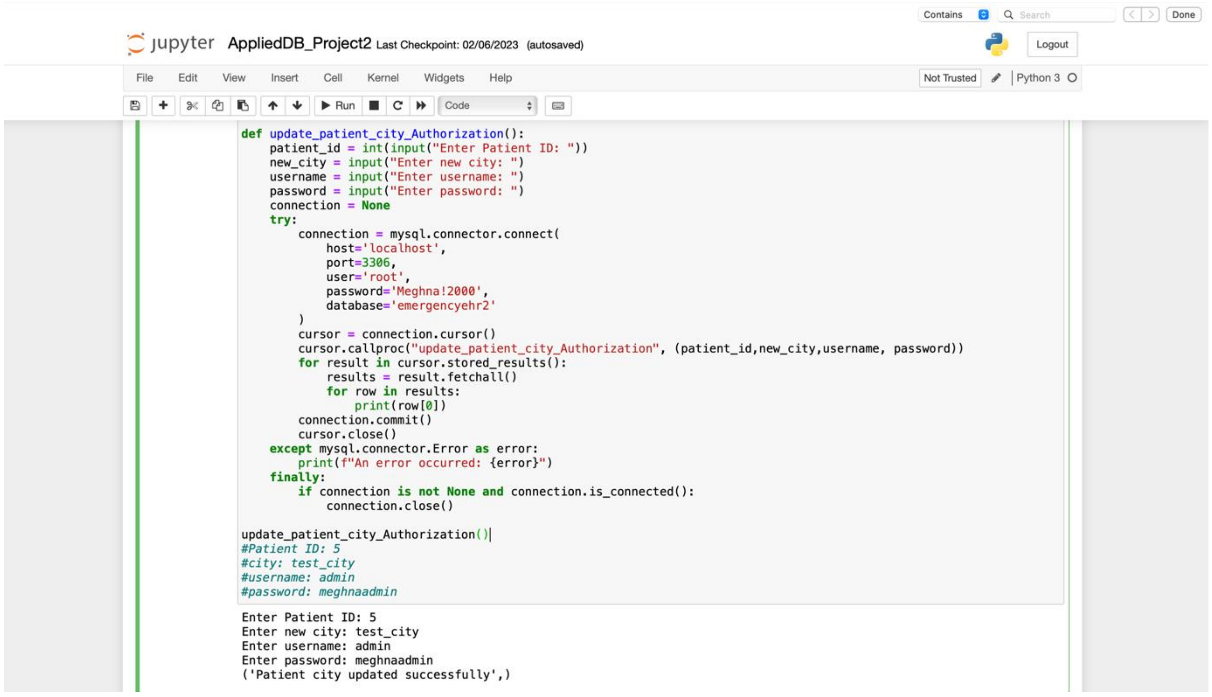


```
-- Create  a procedure to update the patient city if the user was admin
DELIMITER //
CREATE PROCEDURE update_patient_city_Authorization (IN patient_id INT, IN new_city VARCHAR(50), IN admin_username VARCHAR
BEGIN
  DECLARE admin_count INT;
  DECLARE previous_city VARCHAR(50);
  SELECT COUNT(*) INTO admin_count
  FROM USERS
  WHERE username = admin_username AND password = admin_password AND role = 'admin_doctor';

  IF admin_count > 0 THEN
    SELECT city INTO previous_city
    FROM ADDRESSES
    JOIN PATIENTS ON ADDRESSES.address_id = PATIENTS.address_id
    WHERE PATIENTS.patient_id = patient_id;

    UPDATE ADDRESSES
    JOIN PATIENTS ON ADDRESSES.address_id = PATIENTS.address_id
    SET ADDRESSES.city = new_city
    WHERE PATIENTS.patient_id = patient_id;

    INSERT INTO LOGT (user_id, table_name, previous_value, new_value, action_type, action_time)
    SELECT user_id, 'PATIENTS', JSON_OBJECT('patient_id', patient_id, 'city', previous_city), JSON_OBJECT('patient_id', pa
```

```
message
Patient city updated successfully
```

```
     Time      Action                                                                        Response
32   19:00:07  call emergencyehr2.update_patient_city_Authorization(1, 'St.Lousis', 'admin', 'me...  1 row(s) returned
```

Query Completed



```python
def update_patient_city_Authorization():
    patient_id = int(input("Enter Patient ID: "))
    new_city = input("Enter new city: ")
    username = input("Enter username: ")
    password = input("Enter password: ")
    connection = None
    try:
        connection = mysql.connector.connect(
            host='localhost',
            port=3306,
            user='root',
            password='Meghna!2000',
            database='emergencyehr2'
        )
        cursor = connection.cursor()
        cursor.callproc("update_patient_city_Authorization", (patient_id,new_city,username, password))
        for result in cursor.stored_results():
            results = result.fetchall()
            for row in results:
                print(row[0])
        connection.commit()
        cursor.close()
    except mysql.connector.Error as error:
        print(f"An error occurred: {error}")
    finally:
        if connection is not None and connection.is_connected():
            connection.close()

update_patient_city_Authorization()
#Patient ID: 5
#city: test_city
#username: admin
#password: meghnaadmin
```

```
Enter Patient ID: 5
Enter new city: test_city
Enter username: admin
Enter password: meghnaadmin
('Patient city updated successfully',)
```

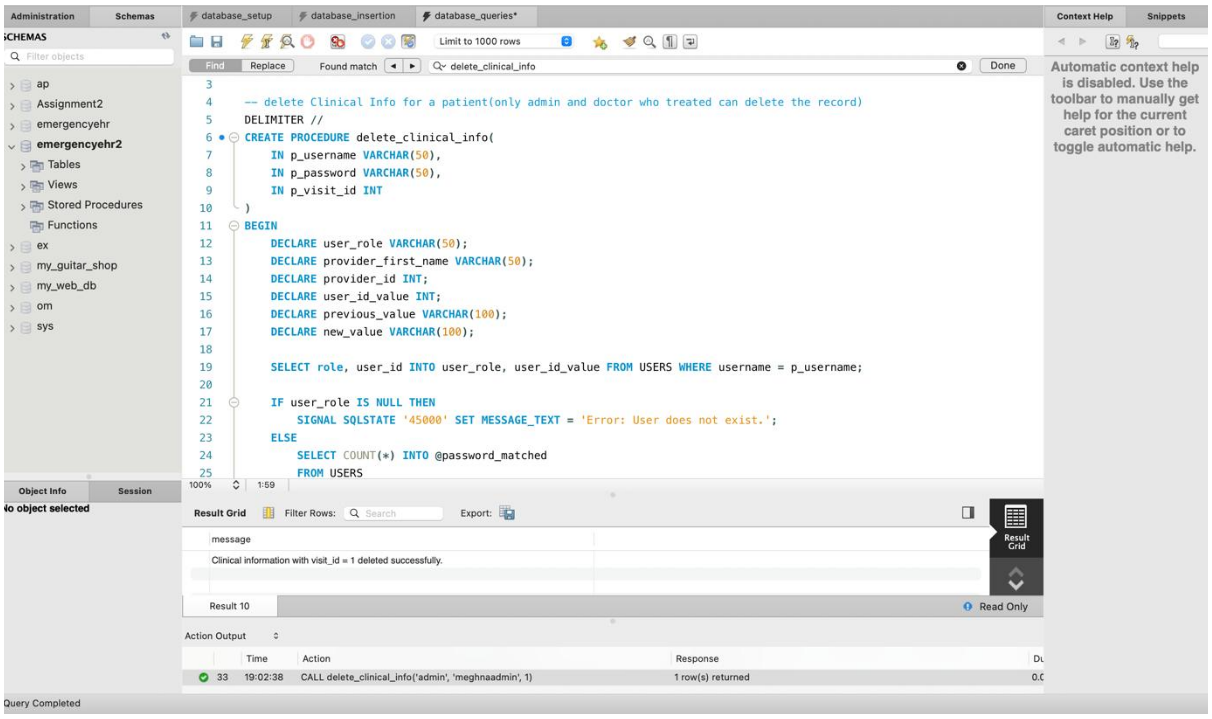
delete_clinical_info: This function requires three input parameters: p_username (VARCHAR), p_password (VARCHAR), and p_visit_id (INT). Its purpose is to eliminate clinical data for a specific visit based on the provided username, password, and visit ID.

The stored procedure operates as follows:

- It declares various variables to store user information, provider details, as well as previous and new values for logging purposes.
- The procedure checks the existence of the user and retrieves their role from the USERS table.
- If the user does not exist, an error is triggered.
- Upon confirming the user's existence, the password undergoes verification.
- If the password is incorrect, an error is triggered.
- In the case of an admin user, the clinical information linked with the provided visit ID is deleted, with the previous value documented for logging purposes.
- If the user is not an admin, it verifies whether they are the provider associated with the visit.
- If the user is the provider, the clinical information is removed, and the previous value is recorded for logging.
- Unauthorized users prompt an error.
- The stored procedure concludes with appropriate messages based on the success or failure of the operation.

Additionally, the stored procedure incorporates logging functionality utilizing the LOGT table. For each successful clinical information deletion, it adds a new log entry, recording the action type, action time, previous value, and new value.

This stored procedure ensures stringent authentication and authorization procedures before clinical information deletion, thereby enhancing security measures and auditability within the system.

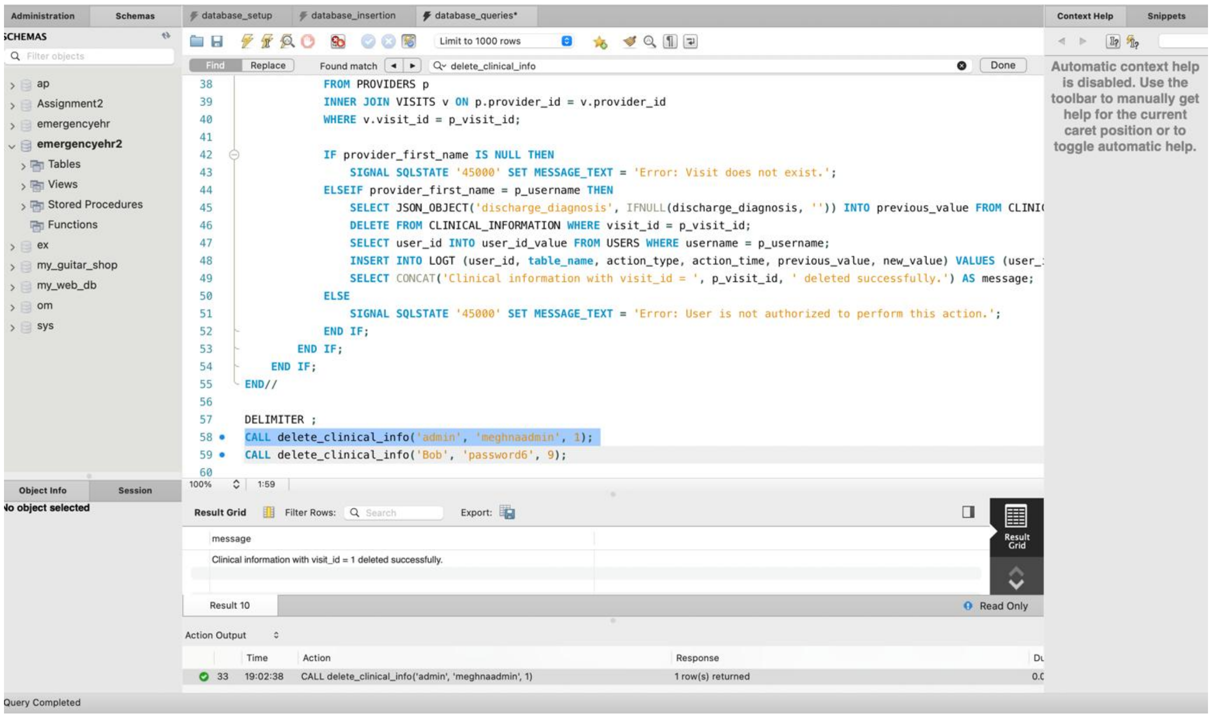


```sql
-- delete Clinical Info for a patient(only admin and doctor who treated can delete the record)
DELIMITER //
CREATE PROCEDURE delete_clinical_info(
    IN p_username VARCHAR(50),
    IN p_password VARCHAR(50),
    IN p_visit_id INT
)
BEGIN
    DECLARE user_role VARCHAR(50);
    DECLARE provider_first_name VARCHAR(50);
    DECLARE provider_id INT;
    DECLARE user_id_value INT;
    DECLARE previous_value VARCHAR(100);
    DECLARE new_value VARCHAR(100);

    SELECT role, user_id INTO user_role, user_id_value FROM USERS WHERE username = p_username;

    IF user_role IS NULL THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error: User does not exist.';
    ELSE
        SELECT COUNT(*) INTO @password_matched
        FROM USERS
```

message
Clinical information with visit_id = 1 deleted successfully.

```
    Time      Action                                              Response           Du
✓ 33  19:02:38  CALL delete_clinical_info('admin', 'meghnaadmin', 1)  1 row(s) returned  0.0
```

Query Completed

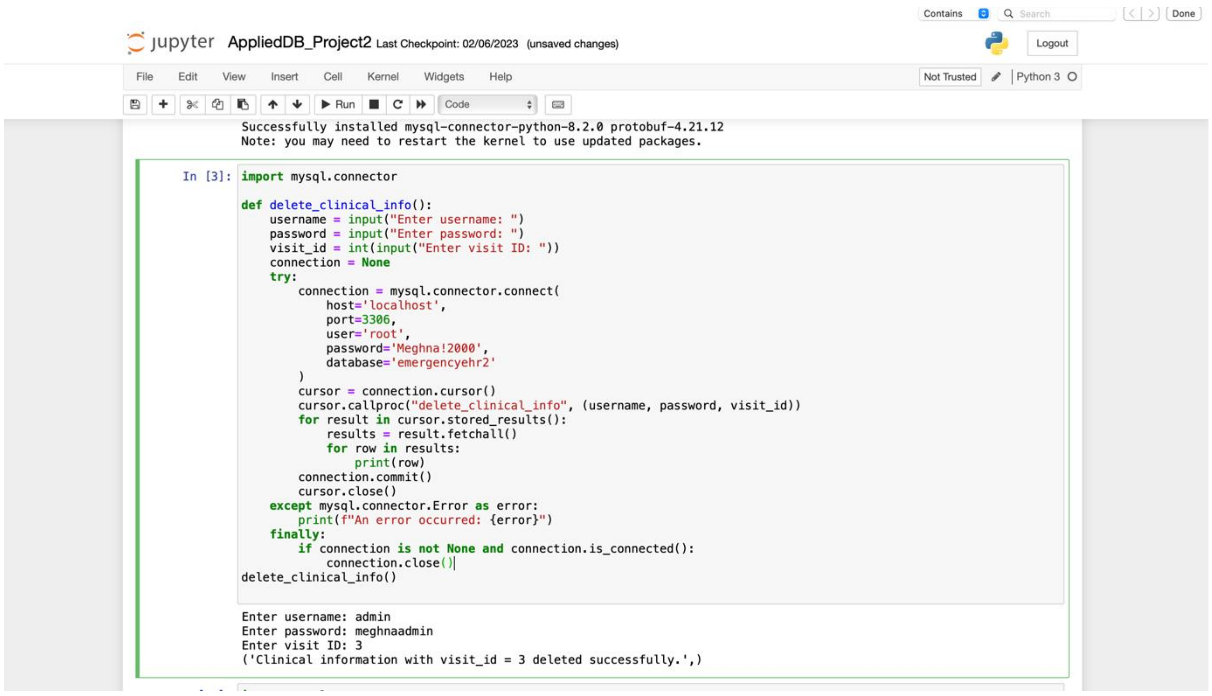


```sql
            FROM PROVIDERS p
            INNER JOIN VISITS v ON p.provider_id = v.provider_id
            WHERE v.visit_id = p_visit_id;

            IF provider_first_name IS NULL THEN
                SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error: Visit does not exist.';
            ELSEIF provider_first_name = p_username THEN
                SELECT JSON_OBJECT('discharge_diagnosis', IFNULL(discharge_diagnosis, '')) INTO previous_value FROM CLINI
                DELETE FROM CLINICAL_INFORMATION WHERE visit_id = p_visit_id;
                SELECT user_id INTO user_id_value FROM USERS WHERE username = p_username;
                INSERT INTO LOGT (user_id, table_name, action_type, action_time, previous_value, new_value) VALUES (user_
                SELECT CONCAT('Clinical information with visit_id = ', p_visit_id, ' deleted successfully.') AS message;
            ELSE
                SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Error: User is not authorized to perform this action.';
            END IF;
        END IF;
    END IF;
END//

DELIMITER ;
CALL delete_clinical_info('admin', 'meghnaadmin', 1);
CALL delete_clinical_info('Bob', 'password6', 9);
```

message
Clinical information with visit_id = 1 deleted successfully.

```
    Time      Action                                              Response           Du
✓ 33  19:02:38  CALL delete_clinical_info('admin', 'meghnaadmin', 1)  1 row(s) returned  0.0
```

Query Completed


Contains
Search
Done
jupyter AppliedDB_Project2 Last Checkpoint: 02/06/2023 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3
Run
Code

```
Successfully installed mysql-connector-python-8.2.0 protobuf-4.21.12
Note: you may need to restart the kernel to use updated packages.
```

```
In [3]: import mysql.connector

        def delete_clinical_info():
            username = input("Enter username: ")
            password = input("Enter password: ")
            visit_id = int(input("Enter visit ID: "))
            connection = None
            try:
                connection = mysql.connector.connect(
                    host='localhost',
                    port=3306,
                    user='root',
                    password='Meghna!2000',
                    database='emergencyehr2'
                )
                cursor = connection.cursor()
                cursor.callproc("delete_clinical_info", (username, password, visit_id))
                for result in cursor.stored_results():
                    results = result.fetchall()
                    for row in results:
                        print(row)
                connection.commit()
                cursor.close()
            except mysql.connector.Error as error:
                print(f"An error occurred: {error}")
            finally:
                if connection is not None and connection.is_connected():
                    connection.close()
        delete_clinical_info()
```

```
Enter username: admin
Enter password: meghnaadmin
Enter visit ID: 3
('Clinical information with visit_id = 3 deleted successfully.',)
```

delete_bed procedure demonstrates role-based access control. role-based access control is a method of granting access permissions based on the roles assigned to users.

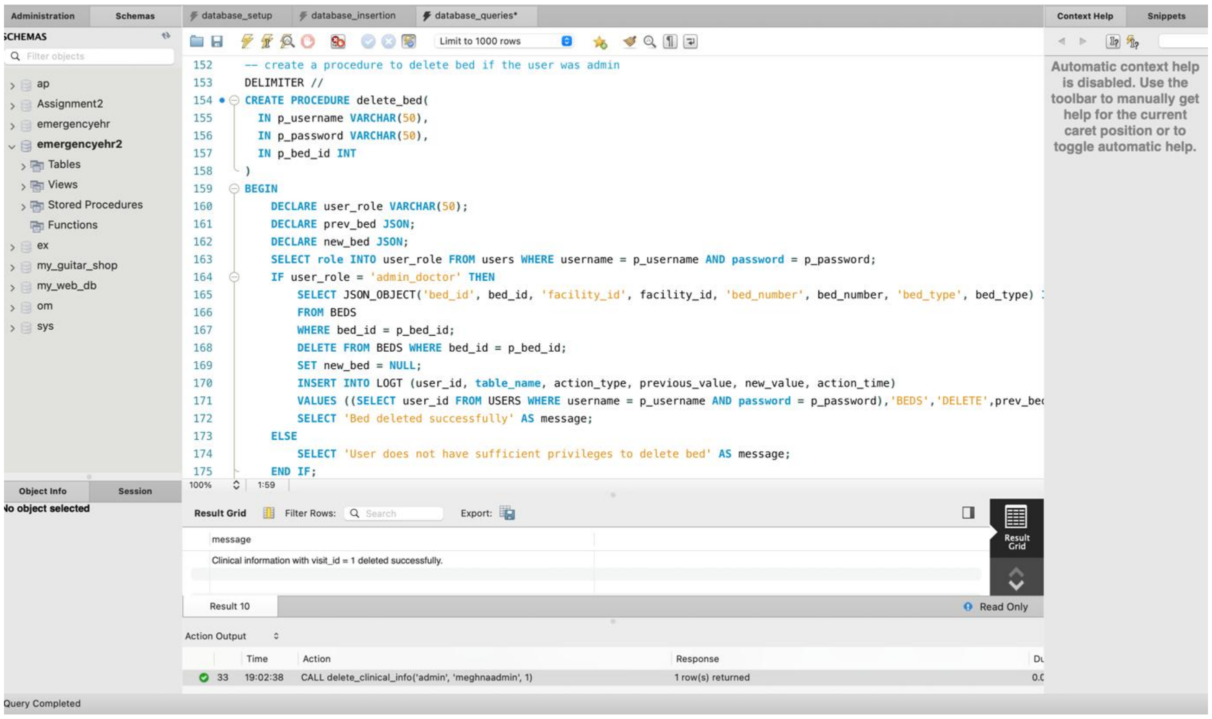
- The delete_bed procedure exemplifies the implementation of role-based access control, a methodology for granting access permissions based on user-assigned roles.
- The procedure necessitates three parameters: the executing user's username, the authentication password, and the bed ID slated for deletion.
- To begin, the procedure checks for the user's existence in the "users" table and authenticates the provided password.
- Upon successful authentication, the user's role is extracted from the "users" table.
- If the user holds the "admin_doctor" role, authorization is granted to proceed with bed deletion; otherwise, an appropriate access denial message is generated.
- Once authorized, the procedure follows these steps:
    - ◆ Retrieves the bed's prior values (bed_id, facility_id, bed_number, bed_type) and stores them as a JSON object in the "prev_bed" variable.
    - ◆ Deletes the specified bed from the "BEDS" table.

- Sets the "new_bed" variable to null, signifying the successful deletion of the bed.

- Inserts a log entry into the "LOGT" table, documenting the bed deletion action. The log entry includes the user ID, table name, action type, previous value (prev_bed), new value (new_bed), and the current timestamp.

Ultimately, the procedure delivers a success message if the bed deletion is successful. Conversely, an access denial message is returned if the user lacks the necessary privileges.

This role-based access control mechanism ensures that only users possessing the "admin_doctor" role can execute the "delete_bed" procedure, safeguarding data integrity and restricting bed deletion capabilities to authorized personnel.

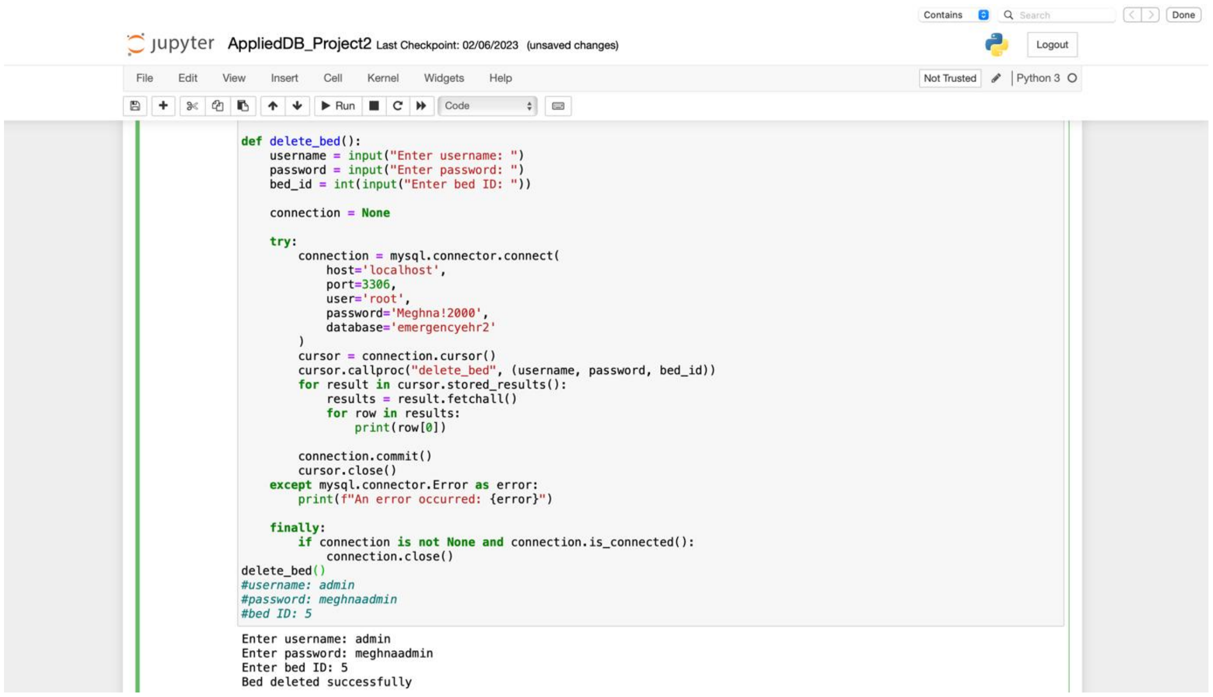


```
-- create a procedure to delete bed if the user was admin
DELIMITER //
CREATE PROCEDURE delete_bed(
  IN p_username VARCHAR(50),
  IN p_password VARCHAR(50),
  IN p_bed_id INT
)
BEGIN
    DECLARE user_role VARCHAR(50);
    DECLARE prev_bed JSON;
    DECLARE new_bed JSON;
    SELECT role INTO user_role FROM users WHERE username = p_username AND password = p_password;
    IF user_role = 'admin_doctor' THEN
        SELECT JSON_OBJECT('bed_id', bed_id, 'facility_id', facility_id, 'bed_number', bed_number, 'bed_type', bed_type) I
        FROM BEDS
        WHERE bed_id = p_bed_id;
        DELETE FROM BEDS WHERE bed_id = p_bed_id;
        SET new_bed = NULL;
        INSERT INTO LOGT (user_id, table_name, action_type, previous_value, new_value, action_time)
        VALUES ((SELECT user_id FROM USERS WHERE username = p_username AND password = p_password),'BEDS','DELETE',prev_bed
        SELECT 'Bed deleted successfully' AS message;
    ELSE
        SELECT 'User does not have sufficient privileges to delete bed' AS message;
    END IF;
```

message
Clinical information with visit_id = 1 deleted successfully.

	Time	Action	Response
33	19:02:38	CALL delete_clinical_info('admin', 'meghnaadmin', 1)	1 row(s) returned

Query Completed


jupyter AppliedDB_Project2 Last Checkpoint: 02/06/2023 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3
Run
Code

```
def delete_bed():
    username = input("Enter username: ")
    password = input("Enter password: ")
    bed_id = int(input("Enter bed ID: "))

    connection = None

    try:
        connection = mysql.connector.connect(
            host='localhost',
            port=3306,
            user='root',
            password='Meghna!2000',
            database='emergencyehr2'
        )
        cursor = connection.cursor()
        cursor.callproc("delete_bed", (username, password, bed_id))
        for result in cursor.stored_results():
            results = result.fetchall()
            for row in results:
                print(row[0])

        connection.commit()
        cursor.close()
    except mysql.connector.Error as error:
        print(f"An error occurred: {error}")

    finally:
        if connection is not None and connection.is_connected():
            connection.close()
delete_bed()
#username: admin
#password: meghnaadmin
#bed ID: 5
```

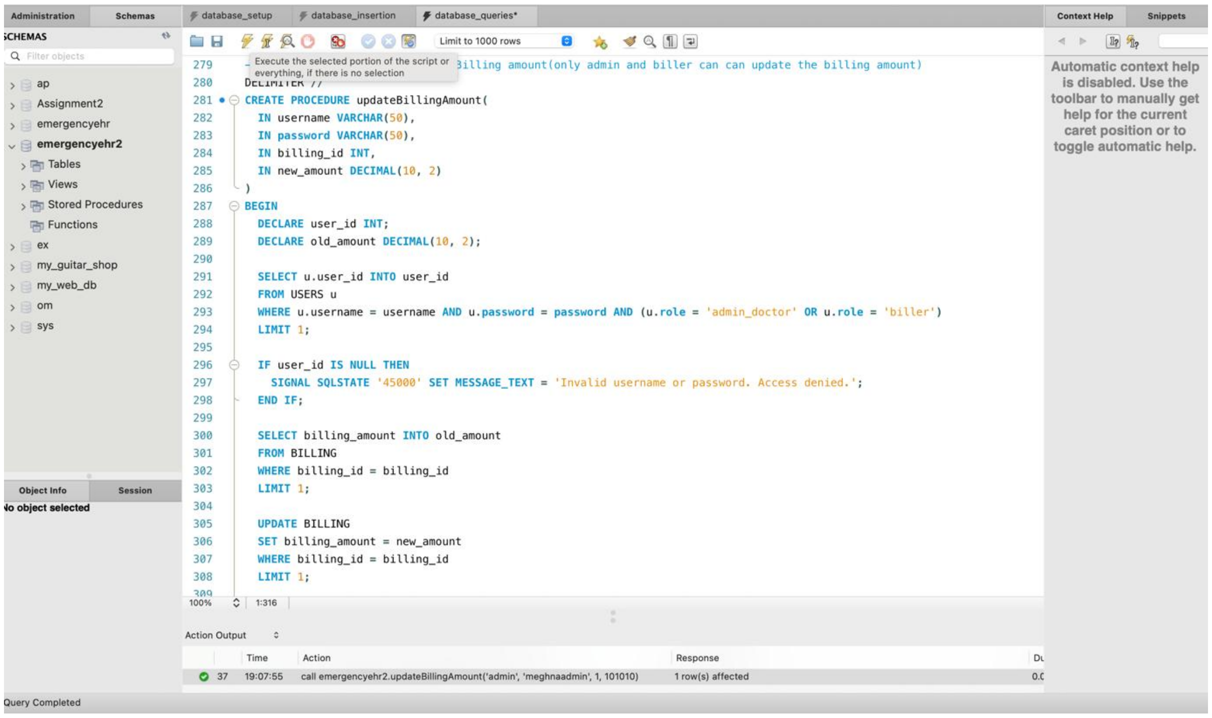
```
Enter username: admin
Enter password: meghnaadmin
Enter bed ID: 5
Bed deleted successfully
```

updateBillingAmount procedure in your system allows authorized users with the roles 'admin_doctor' or 'biller' to update the billing amount for a specific billing record.

- The procedure takes various parameters, including authentication credentials such as username and password, the billing_id of the targeted record, and the new_amount indicating the revised billing amount.
- Execution of the procedure commences with the declaration of variables for user_id and old_amount. It initiates a SELECT query on the "USERS" table to validate the provided username and password against a user with either the 'admin_doctor' or 'biller' role. If a match is identified, the user_id is assigned to the user_id variable.
- In case of no matching user, the procedure triggers an exception using the SIGNAL statement with SQLSTATE '45000,' accompanied by setting the MESSAGE_TEXT to 'Invalid username or password. Access denied.'
- Upon successful authorization, the procedure proceeds with the subsequent steps:
    - ◆ A SELECT query on the "BILLING" table retrieves the current billing_amount value for the specified billing_id, assigning the result to the old_amount variable.

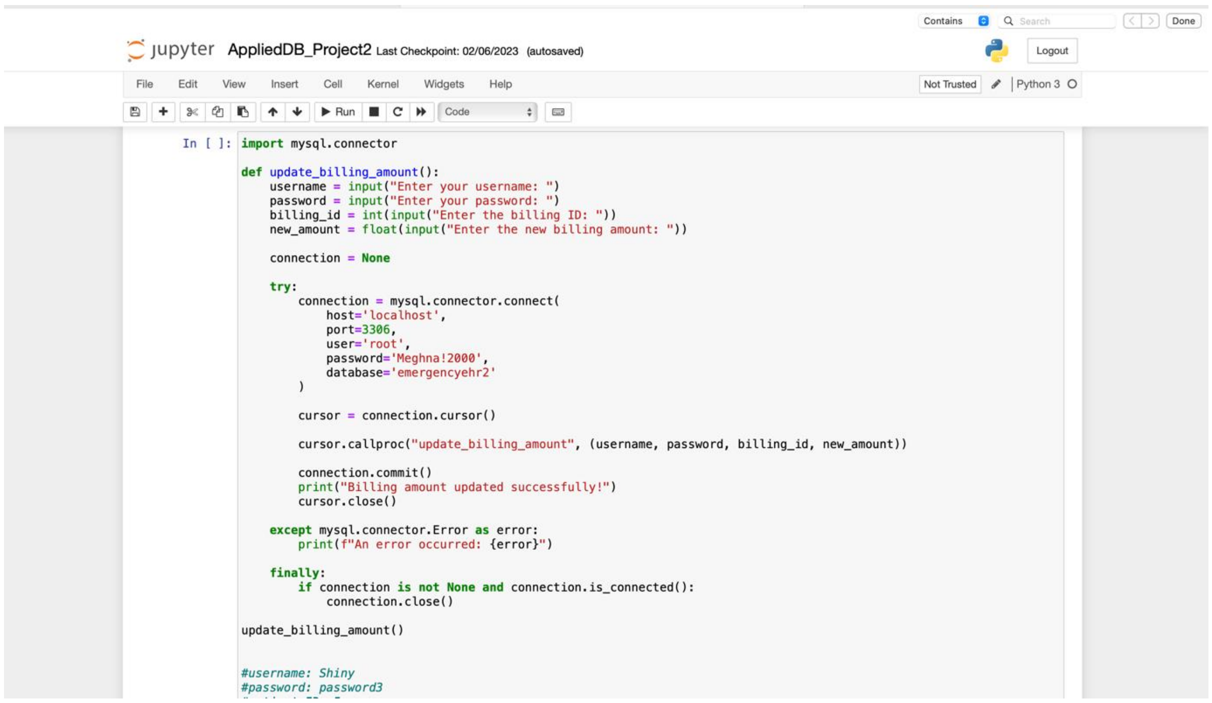
- An UPDATE statement on the "BILLING" table modifies the billing_amount to the new_amount for the specified billing_id.
- A log entry is inserted into the "LOGT" table, documenting the user_id, table name ("BILLING"), action type ("UPDATE"), and the previous and new values of the billing_amount field in JSON format. The action_time is set to the current time using the NOW() function.

The role-based access control mechanism ensures that only users with the 'admin_doctor' or 'biller' roles can execute the "updateBillingAmount" procedure, thereby managing data access and restricting billing amount modification privileges to authorized personnel.



```
-- ...Billing amount(only admin and biller can can update the billing amount)
DELIMITER //
CREATE PROCEDURE updateBillingAmount(
  IN username VARCHAR(50),
  IN password VARCHAR(50),
  IN billing_id INT,
  IN new_amount DECIMAL(10, 2)
)
BEGIN
  DECLARE user_id INT;
  DECLARE old_amount DECIMAL(10, 2);

  SELECT u.user_id INTO user_id
  FROM USERS u
  WHERE u.username = username AND u.password = password AND (u.role = 'admin_doctor' OR u.role = 'biller')
  LIMIT 1;

  IF user_id IS NULL THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Invalid username or password. Access denied.';
  END IF;

  SELECT billing_amount INTO old_amount
  FROM BILLING
  WHERE billing_id = billing_id
  LIMIT 1;

  UPDATE BILLING
  SET billing_amount = new_amount
  WHERE billing_id = billing_id
  LIMIT 1;
```

```
   Time      Action                                                            Response            Du
37 19:07:55  call emergencyehr2.updateBillingAmount('admin', 'meghnaadmin', 1, 101010)  1 row(s) affected  0.0
```


Contains
Search
Done
jupyter AppliedDB_Project2 Last Checkpoint: 02/06/2023 (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3
Run
Code

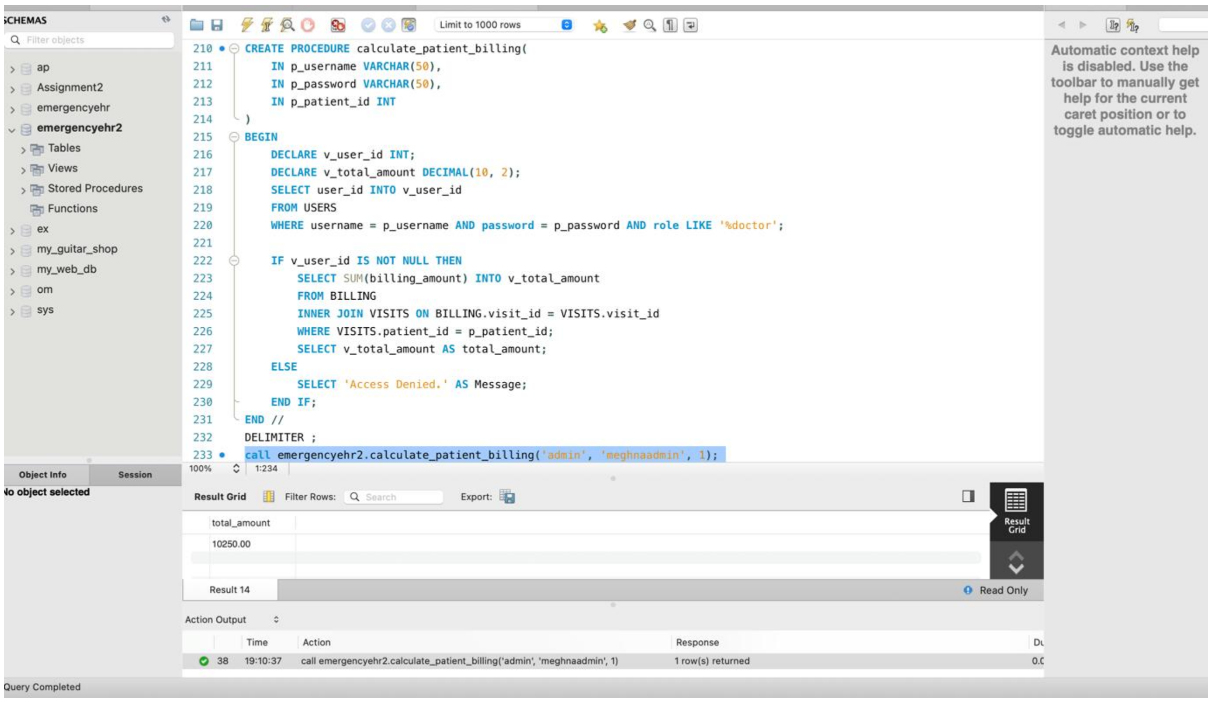
```
In [ ]: import mysql.connector

        def update_billing_amount():
            username = input("Enter your username: ")
            password = input("Enter your password: ")
            billing_id = int(input("Enter the billing ID: "))
            new_amount = float(input("Enter the new billing amount: "))

            connection = None

            try:
                connection = mysql.connector.connect(
                    host='localhost',
                    port=3306,
                    user='root',
                    password='Meghna!2000',
                    database='emergencyehr2'
                )

                cursor = connection.cursor()

                cursor.callproc("update_billing_amount", (username, password, billing_id, new_amount))

                connection.commit()
                print("Billing amount updated successfully!")
                cursor.close()

            except mysql.connector.Error as error:
                print(f"An error occurred: {error}")

            finally:
                if connection is not None and connection.is_connected():
                    connection.close()

        update_billing_amount()


        #username: Shiny
        #password: password3
```

calculate_patient_billing procedure demonstrates role-based access control. Role-based access control is a method of granting access permissions based on the roles assigned to users.

- The "calculate_patient_billing" procedure exemplifies the implementation of role-based access control, a method of providing access permissions based on user-assigned roles.
- The procedure necessitates three parameters: the username of the executing user, the authentication password, and the patient ID for whom the billing amount is to be determined.
- Initially, the procedure checks the existence of the user in the "USERS" table, validates the provided password, and ensures that the user possesses a role containing the term "doctor."
- Upon successful authentication and role verification, the user's ID is retrieved. Subsequently, if the user holds the requisite role, the procedure follows these steps:

  - ◆ Computes the total billing amount for the specified patient by linking the "BILLING" and "VISITS" tables based on visit_id and filtering by patient_id.
  - ◆ Stores the total billing amount in the "v_total_amount" variable.
- In cases where the user lacks authorization, an appropriate access denial message is returned. Conversely, if the user is authorized, the procedure outputs the calculated total amount.

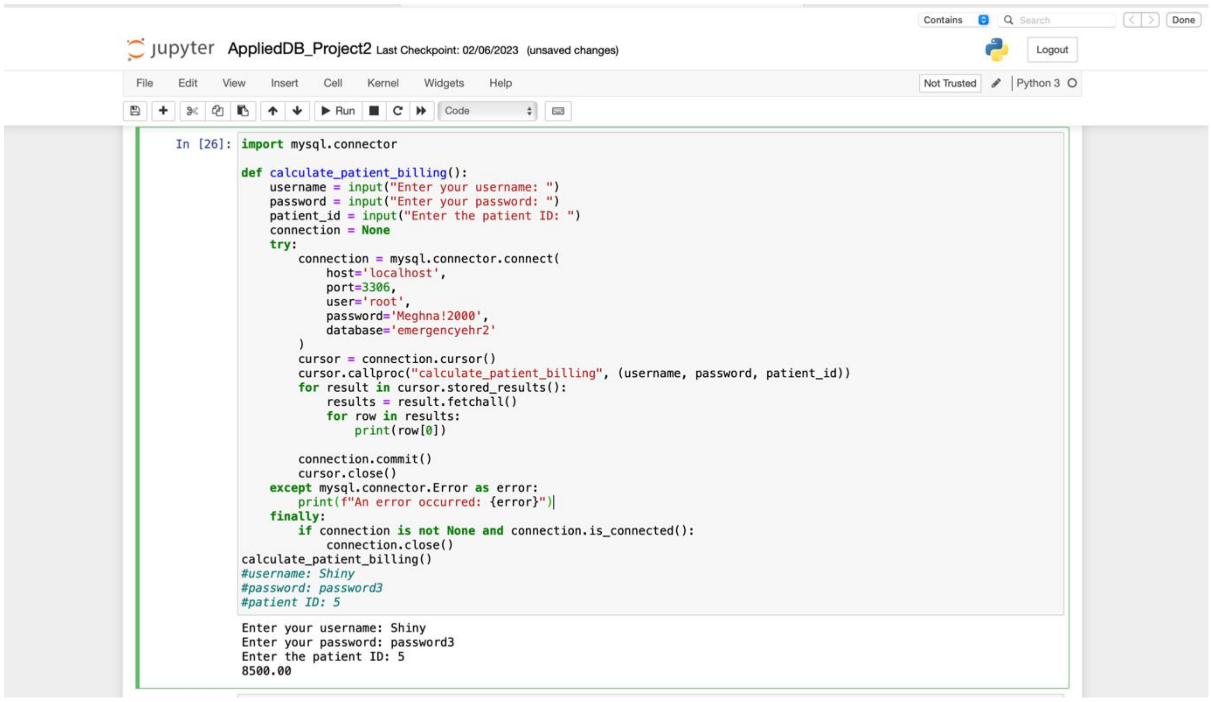
The incorporation of role-based access control guarantees that solely users with a role containing the term "doctor" can execute the "calculate_patient_billing" procedure, safeguarding data privacy and limiting access to billing information to authorized healthcare providers.



```sql
CREATE PROCEDURE calculate_patient_billing(
    IN p_username VARCHAR(50),
    IN p_password VARCHAR(50),
    IN p_patient_id INT
)
BEGIN
    DECLARE v_user_id INT;
    DECLARE v_total_amount DECIMAL(10, 2);
    SELECT user_id INTO v_user_id
    FROM USERS
    WHERE username = p_username AND password = p_password AND role LIKE '%doctor';

    IF v_user_id IS NOT NULL THEN
        SELECT SUM(billing_amount) INTO v_total_amount
        FROM BILLING
        INNER JOIN VISITS ON BILLING.visit_id = VISITS.visit_id
        WHERE VISITS.patient_id = p_patient_id;
        SELECT v_total_amount AS total_amount;
    ELSE
        SELECT 'Access Denied.' AS Message;
    END IF;
END //
DELIMITER ;
call emergencyehr2.calculate_patient_billing('admin', 'meghnaadmin', 1);
```

total_amount
10250.00

		Time	Action	Response	Du
✓	38	19:10:37	call emergencyehr2.calculate_patient_billing('admin', 'meghnaadmin', 1)	1 row(s) returned	0.0


jupyter AppliedDB_Project2 Last Checkpoint: 02/06/2023 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3
Run
Code
Contains
Search
Done

```
In [26]: import mysql.connector

         def calculate_patient_billing():
             username = input("Enter your username: ")
             password = input("Enter your password: ")
             patient_id = input("Enter the patient ID: ")
             connection = None
             try:
                 connection = mysql.connector.connect(
                     host='localhost',
                     port=3306,
                     user='root',
                     password='Meghna!2000',
                     database='emergencyehr2'
                 )
                 cursor = connection.cursor()
                 cursor.callproc("calculate_patient_billing", (username, password, patient_id))
                 for result in cursor.stored_results():
                     results = result.fetchall()
                     for row in results:
                         print(row[0])

                 connection.commit()
                 cursor.close()
             except mysql.connector.Error as error:
                 print(f"An error occurred: {error}")
             finally:
                 if connection is not None and connection.is_connected():
                     connection.close()
         calculate_patient_billing()
         #username: Shiny
         #password: password3
         #patient ID: 5

         Enter your username: Shiny
         Enter your password: password3
         Enter the patient ID: 5
         8500.00
```

addPatient procedure in your system allows authorized users with a role containing the word "doctor" to add a new patient to the database.

- The process involves receiving various parameters representing the patient's details, such as username, password, first name, last name, date of birth, gender, address particulars, insurance company name, and policy number.
- Initiating with the declaration of variables for user_id, address_id, and insurance_id, the procedure executes a SELECT query on the "USERS" table to validate if the entered username and password correspond to a user with a role containing the term "doctor." In case of a match, the user_id is assigned to the respective variable.
- If no matching user is identified, the procedure triggers an exception using the SIGNAL statement with SQLSTATE '45000' and sets the MESSAGE_TEXT to 'Invalid username or password. Access denied.'
- If the user is authorized, the procedure proceeds with the following steps:
    - ◆ It inserts a new address into the "ADDRESSES" table using the provided street address, city, state, and zip code. The address_id of the newly inserted

record is assigned to the address_id variable using the LAST_INSERT_ID() function.

- It adds a new address entry to the "ADDRESSES" table, utilizing the provided street address, city, state, and zip code. The address_id of the newly added record is stored in the address_id variable using the LAST_INSERT_ID() function.
- A new insurance record is inserted into the "INSURANCES" table, incorporating the supplied insurance company name and policy number. The insurance_id of the recently inserted record is assigned to the insurance_id variable using the LAST_INSERT_ID() function.
- A new patient record is created in the "PATIENTS" table, including the provided patient information such as first name, last name, date of birth, gender, address_id, and insurance_id.
- A log entry is added to the "LOGT" table, documenting the user_id, table name ("PATIENTS"), action type ("INSERT"), and the new value of the patient's information in JSON format. The action_time is set to the current time using the NOW() function.

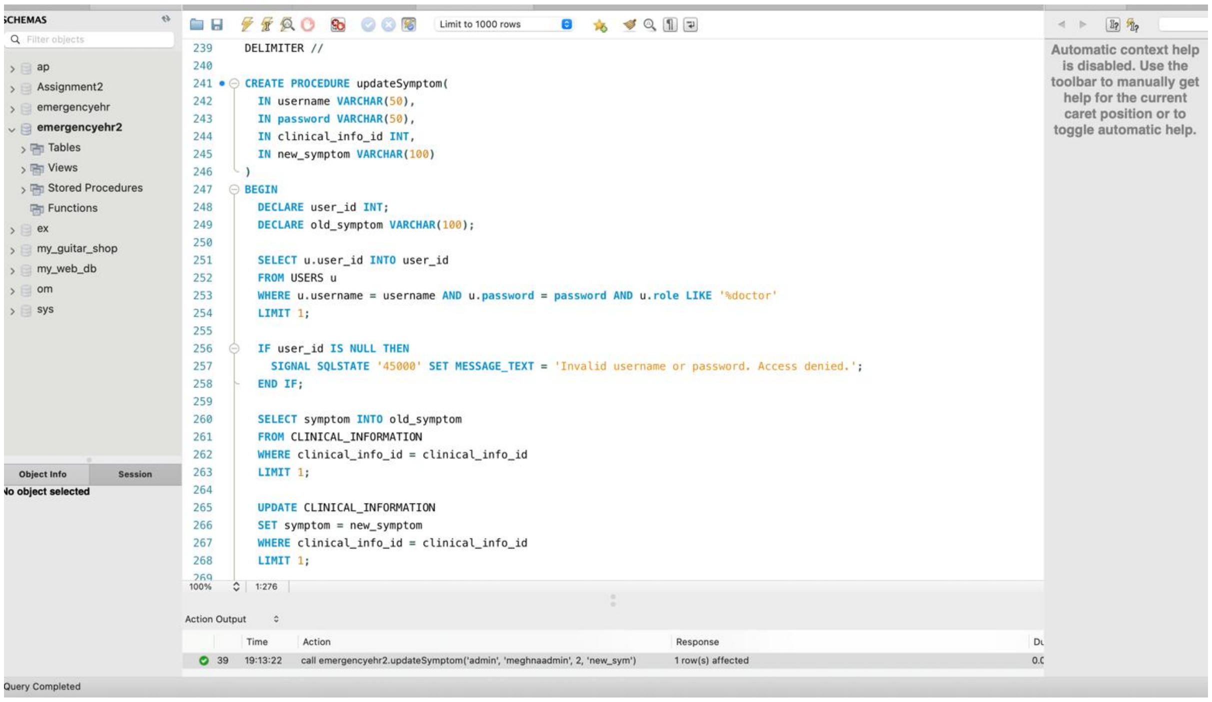
The role-based access control mechanism ensures that only users with a role containing the term "doctor" are permitted to execute the "addPatient" procedure, thereby maintaining strict control over data access and limiting the privilege of adding patients to authorized healthcare providers.

updateSymptom procedure in your system allows authorized users with a role containing the word "doctor" to update the symptom field of a clinical information record.

- The procedure requires various parameters, such as the username and password for authentication, the clinical_info_id of the record to be updated, and the new value for the symptom field.
- Commencing with the declaration of variables for user_id and old_symptom, the procedure progresses through several steps.
- It initiates a SELECT query on the "USERS" table to verify if the provided username and password correspond to a user with a role containing the term "doctor." If a match is identified, the user_id is assigned to the user_id variable.

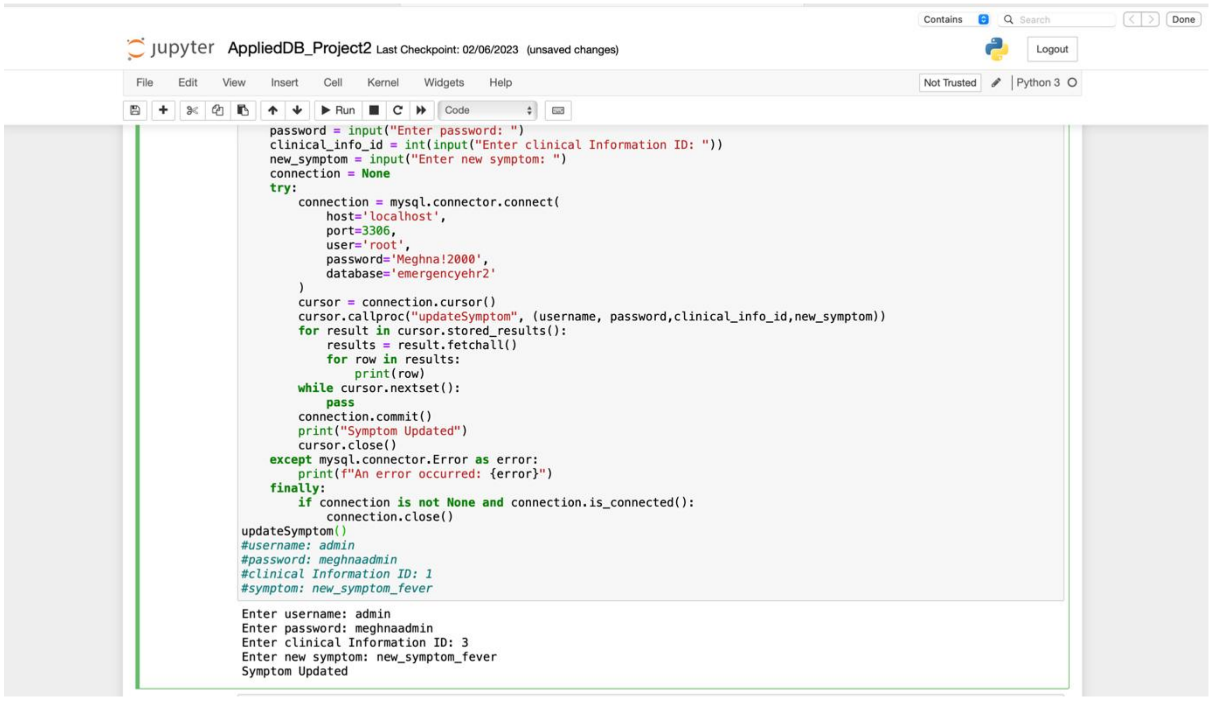
- In the absence of a matching user, the procedure triggers an exception using the SIGNAL statement, with SQLSTATE '45000,' and defines the MESSAGE_TEXT as 'Invalid username or password. Access denied.'
- If the user is granted authorization, the procedure proceeds as follows:
  - Executes a SELECT query on the "CLINICAL_INFORMATION" table to fetch the current value of the symptom field for the specified clinical_info_id. The retrieved old_symptom value is assigned to the old_symptom variable.
  - Performs an UPDATE statement on the "CLINICAL_INFORMATION" table to set the symptom field to the new value for the specified clinical_info_id.
  - Inserts a log entry into the "LOGT" table, capturing user_id, table name ("CLINICAL_INFORMATION"), action type ("UPDATE"), and the previous and new values of the symptom field in JSON format. The action_time is set to the current time using the NOW() function.

The role-based access control mechanism guarantees that only users with a role containing the term "doctor" are authorized to execute the "updateSymptom" procedure, thereby maintaining control over data access and restricting the privilege to update symptoms to authorized healthcare providers.



```
DELIMITER //

CREATE PROCEDURE updateSymptom(
  IN username VARCHAR(50),
  IN password VARCHAR(50),
  IN clinical_info_id INT,
  IN new_symptom VARCHAR(100)
)
BEGIN
  DECLARE user_id INT;
  DECLARE old_symptom VARCHAR(100);

  SELECT u.user_id INTO user_id
  FROM USERS u
  WHERE u.username = username AND u.password = password AND u.role LIKE '%doctor'
  LIMIT 1;

  IF user_id IS NULL THEN
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Invalid username or password. Access denied.';
  END IF;

  SELECT symptom INTO old_symptom
  FROM CLINICAL_INFORMATION
  WHERE clinical_info_id = clinical_info_id
  LIMIT 1;

  UPDATE CLINICAL_INFORMATION
  SET symptom = new_symptom
  WHERE clinical_info_id = clinical_info_id
  LIMIT 1;
```

```
Action Output
   Time      Action                                                              Response
39 19:13:22  call emergencyehr2.updateSymptom('admin', 'meghnaadmin', 2, 'new_sym')  1 row(s) affected  0.0
```



jupyter AppliedDB_Project2 Last Checkpoint: 02/06/2023 (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3
Run
Code
Contains
Search
Done

```
    password = input("Enter password: ")
    clinical_info_id = int(input("Enter clinical Information ID: "))
    new_symptom = input("Enter new symptom: ")
    connection = None
    try:
        connection = mysql.connector.connect(
            host='localhost',
            port=3306,
            user='root',
            password='Meghna!2000',
            database='emergencyehr2'
        )
        cursor = connection.cursor()
        cursor.callproc("updateSymptom", (username, password,clinical_info_id,new_symptom))
        for result in cursor.stored_results():
            results = result.fetchall()
            for row in results:
                print(row)
        while cursor.nextset():
            pass
        connection.commit()
        print("Symptom Updated")
        cursor.close()
    except mysql.connector.Error as error:
        print(f"An error occurred: {error}")
    finally:
        if connection is not None and connection.is_connected():
            connection.close()
updateSymptom()
#username: admin
#password: meghnaadmin
#clinical Information ID: 1
#symptom: new_symptom_fever
```

```
Enter username: admin
Enter password: meghnaadmin
Enter clinical Information ID: 3
Enter new symptom: new_symptom_fever
Symptom Updated
```

Audit Trails

Audit Trails in the context of this system involve the meticulous recording and monitoring of all activities and alterations performed on the system or database. These trails serve as a historical documentation of user or process actions, offering accountability, traceability, and the potential for forensic analysis. Within the framework of this project, audit trails are executed through the utilization of the "LOGT" table, designed to collect and retain information pertaining to diverse actions executed within the system.

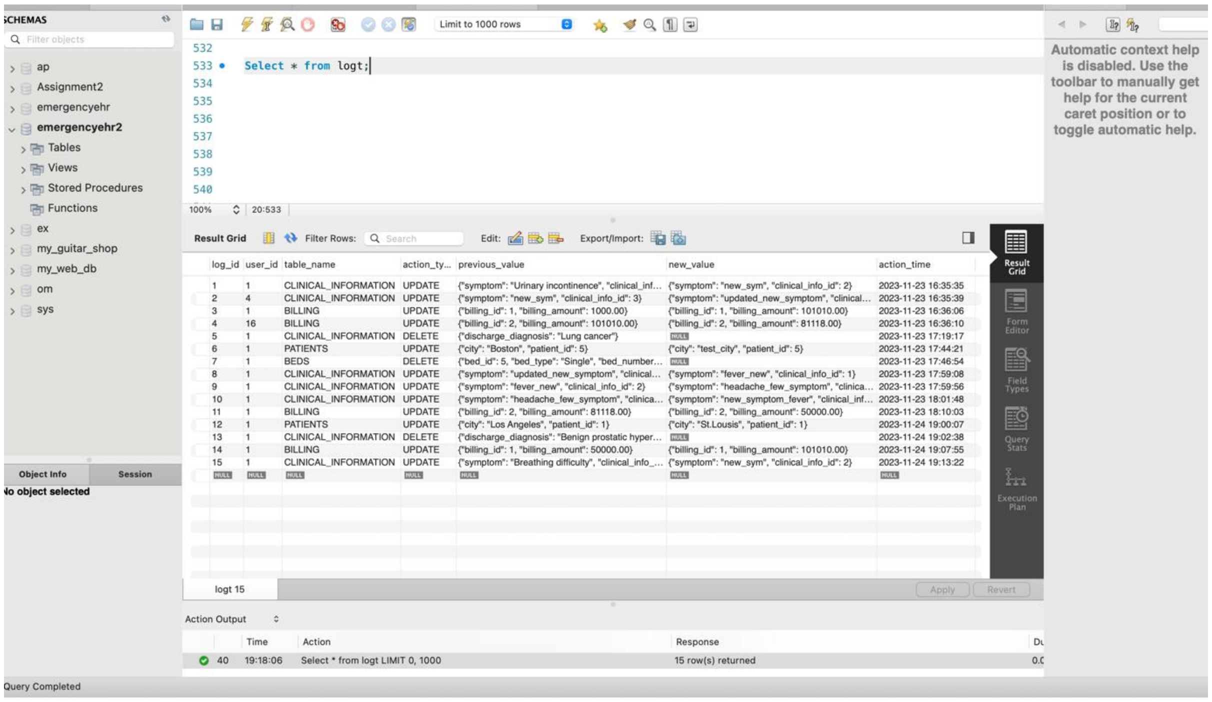
The "LOGT" table, which has been previously established, functions as the designated audit table for cataloging system activities. It encompasses the following columns:

- log_id: An auto-incremented primary key that distinctly identifies each log entry.
- user_id: A foreign key cross-referencing the "user_id" column in the "USERS" table, signifying the user responsible for the action.
- table_name: The designation of the table upon which the action was executed.
- action_type: A descriptor of the type of action conducted (e.g., INSERT, UPDATE, DELETE).
- previous_value: A JSON object that retains the former value of the altered data, facilitating the tracking of modifications during updates or deletions.
- new_value: A JSON object that preserves the updated value of the modified data, aiding in monitoring changes during inserts or updates.
- action_time: A timestamp denoting when the action took place.

Through the incorporation of record entries into the "LOGT" table following specific actions such as inserts, updates, or deletions, the system maintains a comprehensive history of alterations made to the database. This feature empowers administrators and auditors to scrutinize and analyze executed actions, encompassing details such as the executor, the timing of the occurrence, and specifics of the modifications.

The updateBillingAmount procedure, a development of mine, exemplifies the utilization of the "LOGT" table. Post the adjustment of billing amounts in the "BILLING" table, the procedure instigates the insertion of a log entry into the "LOGT" table. This log entry captures essential information, including user_id, table_name ('BILLING'), action_type ('UPDATE'), previous_value (the former billing amount), new_value (the updated billing amount), and action_time (the current timestamp). This approach ensures a thorough audit trail of billing amount adjustments, reinforcing accountability and traceability for financial transactions within the system.

Furthermore, it's worth noting that the "LOGT" table's scope can be broadened to encompass additional pertinent actions within the system, such as inserts, deletes, or updates on other pivotal tables, tailored to meet specific auditing requirements.



```sql
Select * from logt;
```

log_id	user_id	table_name	action_ty...	previous_value	new_value	action_time
1	1	CLINICAL_INFORMATION	UPDATE	{"symptom": "Urinary incontinence", "clinical_inf...	{"symptom": "new_sym", "clinical_info_id": 2}	2023-11-23 16:35:35
2	4	CLINICAL_INFORMATION	UPDATE	{"symptom": "new_sym", "clinical_info_id": 3}	{"symptom": "updated_new_symptom", "clinical...	2023-11-23 16:35:39
3	1	BILLING	UPDATE	{"billing_id": 1, "billing_amount": 1000.00}	{"billing_id": 1, "billing_amount": 101010.00}	2023-11-23 16:36:06
4	16	BILLING	UPDATE	{"billing_id": 2, "billing_amount": 101010.00}	{"billing_id": 2, "billing_amount": 81118.00}	2023-11-23 16:36:10
5	1	CLINICAL_INFORMATION	DELETE	{"discharge_diagnosis": "Lung cancer"}	NULL	2023-11-23 17:19:17
6	1	PATIENTS	UPDATE	{"city": "Boston", "patient_id": 5}	{"city": "test_city", "patient_id": 5}	2023-11-23 17:44:21
7	1	BEDS	DELETE	{"bed_id": 5, "bed_type": "Single", "bed_number...	NULL	2023-11-23 17:46:54
8	1	CLINICAL_INFORMATION	UPDATE	{"symptom": "updated_new_symptom", "clinical...	{"symptom": "fever_new", "clinical_info_id": 1}	2023-11-23 17:59:08
9	1	CLINICAL_INFORMATION	UPDATE	{"symptom": "fever_new", "clinical_info_id": 2}	{"symptom": "headache_few_symptom", "clinica...	2023-11-23 17:59:56
10	1	CLINICAL_INFORMATION	UPDATE	{"symptom": "headache_few_symptom", "clinica...	{"symptom": "new_symptom_fever", "clinical_inf...	2023-11-23 18:01:48
11	1	BILLING	UPDATE	{"billing_id": 2, "billing_amount": 81118.00}	{"billing_id": 2, "billing_amount": 50000.00}	2023-11-23 18:10:03
12	1	PATIENTS	UPDATE	{"city": "Los Angeles", "patient_id": 1}	{"city": "St.Lousis", "patient_id": 1}	2023-11-24 19:00:07
13	1	CLINICAL_INFORMATION	DELETE	{"discharge_diagnosis": "Benign prostatic hyper...	NULL	2023-11-24 19:02:38
14	1	BILLING	UPDATE	{"billing_id": 1, "billing_amount": 50000.00}	{"billing_id": 1, "billing_amount": 101010.00}	2023-11-24 19:07:55
15	1	CLINICAL_INFORMATION	UPDATE	{"symptom": "Breathing difficulty", "clinical_info_...	{"symptom": "new_sym", "clinical_info_id": 2}	2023-11-24 19:13:22
NULL	NULL	NULL	NULL	NULL	NULL	NULL

```
Action Output
     Time       Action                                  Response               Du
✓ 40 19:18:06   Select * from logt LIMIT 0, 1000        15 row(s) returned     0.0
```

Tools Used

- MySql Workbench
- Jupyter Notebook
- Lucid Chart