Group Assessment - Hospital Information System Security

Created	@December 3, 2025 11:45 AM
Assign	DatabaseSecurity
Students	TOMÁS JUAN USÓN, ALBA PRATS BARROSO, ALEJANDRO SERRANO CALVO
Teacher	ANDREA ALVARO MARTIN
Academic Year	2025/2026
DBMS Selected	SQLite (with Python Middleware & Minecraft Integration)

1. Introduction

In the modern healthcare sector, the integrity and confidentiality of patient data are paramount. Hospital Information Systems (HIS) store highly sensitive Personally Identifiable Information (PII), including medical history, social security numbers, and treatment plans. The compromise of such data not only violates privacy laws (such as GDPR) but can also endanger patient safety. The objective of this project is to design, implement, and secure a HIS database that ensures the principles of Confidentiality, Integrity, and Availability (CIA).

For this assessment, our team has selected **SQLite** as the Database Management System. While SQLite is a lightweight, serverless engine often used in embedded systems, it lacks native enterprise security features such as user management, network listening, and built in Role Based Access Control (RBAC).

We selected SQLite specifically to challenge the standard implementation of security. Instead of relying on "out of the box" features found in PostgreSQL, we have architected a **custom security middleware using Python**. This middleware enforces Mandatory Access Control (MAC) and Row-Level Security (RLS) at the application layer. Furthermore, to demonstrate the system's integration capabilities (as per section 3.4.6 of the requirements), we have integrated the database with **Minecraft (**videogame**)**. In our implementation, the Minecraft game world acts as the visual frontend, where physical location and player identity determine data access, simulating a real world physical security environment within a hospital.

2. General Requirements

This project adheres to the general requirements set forth in the course assessment guidelines.

- **Team Composition:** The project was developed by a team of three students:
    - Alejandro Serrano
    - Alba Prats
    - Tomas Usón
- *Degree:* Computer Engineering / Cybersecurity Engineering.
- **Selected DBMS:**
  We have selected **SQLite** as the core Database Management System.
    - *Justification:* SQLite was chosen to demonstrate how security principles (CIA) must be implemented manually in serverless, file-based environments. It allows for high portability and satisfies the requirement to compare a non-standard architecture against PostgreSQL. Also because was randomly selected 🙂.
- **Auxiliary Tools & Environment:**
  To satisfy the "Integration with other systems" requirement and to compensate for SQLite's lack of native network interfaces, we utilized the following stack:
    - **Python (v3.x):** Serves as the security middleware, handling authentication, encryption, and logic enforcement.
    - **Minecraft (Java Edition + Raspberry Juice Plugin):** Acts as the visual front-end interface, simulating physical access controls within a hospital building.
    - **DB Browser for SQLite:** Used for administrative tasks, schema creation, and auditing verification.
- **Language:** All documentation, code comments, and presentation materials are presented in English.

3. Hospital Use Case

3.0 Context and Objectives

The core objective of this project is to secure a **Hospital Information System (HIS)** capable of storing and managing sensitive patient data. In a real-world clinical environment, this data includes Personally Identifiable Information (PII) such as Social Security Numbers (SSN), contact details, and highly sensitive clinical records (diagnoses, treatments, and prescriptions).

The system must ensure the **CIA Triad**:

1. **Confidentiality:** Ensuring that only authorized personnel (e.g., Doctors) can view sensitive medical history, while administrative staff (e.g., Nurses) view only what is necessary for their duties.
2. **Integrity:** Preventing unauthorized modification of patient records or prescriptions.
3. **Availability:** Ensuring the database is accessible to authorized users when needed, while maintaining secure backups.

3.1 Architectural Approach

Unlike traditional Client-Server architectures (e.g., PostgreSQL), our implementation utilizes **SQLite** in an embedded architecture. This presents unique security challenges, as SQLite defaults to a single flat file with no internal user accounts.

To address this, our Hospital Use Case is implemented as follows:

- **The Database (`hospital_mc.db`):** Acts as the passive storage vault. It uses **Foreign Key constraints** to enforce data integrity.
- **The Security Layer (Python Middleware):** Acts as the active "Reference Monitor." It intercepts all user requests, verifies the user's identity against the `Users` table, and enforces **Role Based Access Control (RBAC), MAC and RLS** before any SQL query is executed.
- **The Interface (Minecraft Hospital):** Represents the physical hospital. Access to data is triggered by physical interactions (e.g., a Doctor entering a ward and clicking a terminal), adding a layer of "Physical Security" simulation to the project.

- **The Audit Dashboard and Backup Tool:** Acts as the script to be able to check logs and do schedules incremental backups.

This architecture allows us to demonstrate **Mandatory Access Control (MAC)** and **Row-Level Security (RLS)** programmatically, providing a stark contrast to the declarative security models of enterprise DBMS solutions.

3.2 Database Design

3.2 PART I Database Schema: Description of each table and its attributes

The database has been designed following the **Third Normal Form (3NF)** to minimize redundancy and ensure data integrity. Since SQLite is serverless, we explicitly enforce referential integrity using the command `PRAGMA foreign_keys = ON;` at the start of every connection.

The schema is divided into three logical zones: **Authentication** (Access), **Clinical Assets** (Patient Data) and **Staffing** (HR).

A. Authentication & Authorization Tables

These tables manage the security layer. We implemented a **User-Role-Bridge** architecture to allow flexible Role-Based Access Control (RBAC).

Table: Users

Purpose: Stores credentials and links the physical interface (Minecraft) to the logical database.

Attribute	Constraints	Description / Purpose
`user_id`	**PK**, AUTOINCREMENT	Internal unique identifier for the system user.
`username`	**UNIQUE**, NOT NULL	The **Minecraft Username** (e.g., 'Andrea_MC'). This acts as the external authentication token.
`password_hash`	NOT NULL	SHA-256 hash of the user's password. Plain text passwords are never stored.
`salt`	TEXT	Random string added to the password before hashing to prevent Rainbow Table attacks.

email	**UNIQUE**	**Critical Link:** Used to map the User Account to their professional record (Doctor/Nurse tables).
is_active	DEFAULT 1	Flag to revoke access without deleting audit history.

QUERY:

```
CREATE TABLE Users (
    user_id         INTEGER PRIMARY KEY,
    username        TEXT NOT NULL UNIQUE, -- <<< THE MINECRAFT BRID
GE
    password_hash    TEXT NOT NULL,
    salt            TEXT,
    full_name        TEXT NOT NULL,
    email           TEXT UNIQUE,         -- <<< THE LINK TO DOCTOR/NURSE T
ABLES
    is_active       INTEGER NOT NULL DEFAULT 1,
    created_at       TEXT NOT NULL DEFAULT (datetime('now')),
    last_login_at    TEXT
);
```

Table: Roles

Purpose: Defines the security labels (labels) used for Mandatory Access Control.

Attribute	**Constraints**	**Description / Purpose**
role_id	**PK**	Unique ID for the role.
name	**UNIQUE**, NOT NULL	The role key (e.g., 'doctor', 'nurse', 'admin_db') used by the Python middleware logic.
description	TEXT	Human-readable explanation of privileges.

QUERY:

```
CREATE TABLE Roles (
    role_id      INTEGER PRIMARY KEY,
    name         TEXT NOT NULL UNIQUE,
    description  TEXT
);
```

Table: UserRoles

Purpose: A Many-to-Many bridge table allowing users to hold multiple roles if necessary (scalability).

Attribute	Constraints	Description / Purpose
`user_id`	**PK, FK** (ref Users)	Links to the User entity. `ON DELETE CASCADE`.
`role_id`	**PK, FK** (ref Roles)	Links to the Role entity. `ON DELETE CASCADE`.

QUERY:

```
CREATE TABLE UserRoles (
    user_id   INTEGER NOT NULL,
    role_id   INTEGER NOT NULL,
    PRIMARY KEY (user_id, role_id),
    FOREIGN KEY (user_id) REFERENCES Users(user_id) ON DELETE CASCA
DE,
    FOREIGN KEY (role_id) REFERENCES Roles(role_id) ON DELETE CASCAD
E
);
```

B. Clinical Assets (The Protected Data)

These tables store the sensitive data that requires protection (CIA).

Table: Patients

Purpose: Stores Personally Identifiable Information (PII). **Note:** Patients are data objects, not system users.

Attribute	Constraints	Description / Purpose
`patient_id`	**PK**	Unique identifier for the patient.
`first_name`	**NOT NULL**	Patient given name.
`last_name`	**NOT NULL**	Patient surname.
`dob`	**NOT NULL**, `CHECK(date(dob) IS NOT NULL)`	Date of birth (stored as ISO-8601 text). Validates that the value is a real date.
`gender`	`CHECK (gender IN ('M','F','O'))`	Enforces data integrity for gender values.

Attribute	Constraints	Description / Purpose
`ssn`	**UNIQUE**, `CHECK (ssn GLOB '[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]')`	U.S. Social Security Number in `###-##-####` format. *Sensitive PII – access restricted by application logic.*
`phone`	`CHECK (phone GLOB '[0-9][0-9][0-9] [0-9][0-9][0-9] [0-9][0-9][0-9]')`	Patient phone number in `### ### ###` format.
`email`	**NOT NULL**, `CHECK (email LIKE '%@%.%')`	Patient email address (basic format validation).
`address`	*(none)*	Mailing or residential address of the patient.
`created_at`	**NOT NULL**, `DEFAULT datetime('now')`	Timestamp of when the record was created.
`updated_at`	*(none)*	Timestamp of last update to the record.
`last_modified_by`	*(none)*	ID of the user who last modified the record.

QUERY:

```
CREATE TABLE Patients (
    patient_id   INTEGER PRIMARY KEY,
    first_name   TEXT NOT NULL,
    last_name    TEXT NOT NULL,
    dob          TEXT NOT NULL CHECK (date(dob) IS NOT NULL), -- must be
valid date string
    gender       TEXT CHECK (gender IN ('M','F','O')),
    ssn          TEXT UNIQUE CHECK (ssn GLOB '[0-9][0-9][0-9]-[0-9][0-9]-
[0-9][0-9][0-9][0-9]'), --Checks
    phone        TEXT CHECK (phone GLOB '[0-9][0-9][0-9] [0-9][0-9][0-9]
[0-9][0-9][0-9]'),
    email        TEXT NOT NULL CHECK (email LIKE '%@%.%'), -- simple ema
il check NOT NULL TO IDENTIFIER VIA MIDDLEWARE APPLICATION
    address      TEXT,
    created_at   TEXT NOT NULL DEFAULT (datetime('now')),
    updated_at   TEXT,
    last_modified_by INTEGER
);
```

Table: Treatments

Purpose: Links Patients to Doctors and records medical history.

Attribute	Constraints	Description / Purpose
`treatment_id`	**PK**	Unique ID.
`patient_id`	**FK** (ref Patients)	The patient receiving care.
`doctor_id`	**FK** (ref Doctors)	The doctor responsible.
`status`	CHECK (...)	Enforces workflow states: 'PLANNED', 'ONGOING', 'COMPLETED'.
`description`	NOT NULL	**Sensitive.** Medical diagnosis details.

QUERY:

```
CREATE TABLE Treatments (
    treatment_id INTEGER PRIMARY KEY,
    patient_id   INTEGER NOT NULL,
    doctor_id    INTEGER NOT NULL,
    description  TEXT NOT NULL,
    start_date   TEXT NOT NULL,
    end_date     TEXT,
    status       TEXT NOT NULL CHECK (status IN ('PLANNED','ONGOING','COMPLETED','CANCELLED')),
    created_at   TEXT NOT NULL DEFAULT (datetime('now')),
    updated_at   TEXT,
    last_modified_by INTEGER,
    FOREIGN KEY (patient_id) REFERENCES Patients(patient_id),
    FOREIGN KEY (doctor_id)  REFERENCES Doctors(doctor_id)
);
```

Table: Prescriptions

Purpose: Records medication distribution.

Attribute	Constraints	Description / Purpose
`prescription_id`	**PK**	Unique ID.
`patient_id`	**FK** (ref Patients)	The patient.
`doctor_id`	**FK** (ref Doctors)	The prescribing doctor.

medication	NOT NULL	Name of the drug.
dosage	NOT NULL	Dosage instructions.

QUERY:

```
CREATE TABLE Prescriptions (
    prescription_id INTEGER PRIMARY KEY,
    patient_id      INTEGER NOT NULL,
    doctor_id       INTEGER NOT NULL,
    pharmacist_id   INTEGER,
    medication      TEXT NOT NULL,
    dosage          TEXT NOT NULL,
    start_date      TEXT NOT NULL,
    end_date        TEXT,
    created_at      TEXT NOT NULL DEFAULT (datetime('now')),
    dispensed_at    TEXT,
    last_modified_by INTEGER,
    FOREIGN KEY (patient_id)    REFERENCES Patients(patient_id),
    FOREIGN KEY (doctor_id)     REFERENCES Doctors(doctor_id),
    FOREIGN KEY (pharmacist_id) REFERENCES Pharmacists(pharmacist_id)
);
```

Table: LabResults

Purpose: Stores the results of diagnostic tests.

Attribute	Constraints	Description / Purpose
lab_result_id	**PK**	Unique identifier for the test result.
patient_id	**FK** (ref Patients)	The patient who was tested.
lab_tech_id	**FK** (ref LabTechnicians)	The technician who performed the test.
test_name	NOT NULL	Type of test (e.g., 'Blood Count', 'X-Ray').
result_value	NOT NULL	The outcome/value of the test.
unit	TEXT	Measurement unit (e.g., 'mg/dL').
test_date	NOT NULL	When the test was performed.

QUERY:

```
CREATE TABLE LabResults (
    lab_result_id INTEGER PRIMARY KEY,
    patient_id    INTEGER NOT NULL,
    lab_tech_id   INTEGER NOT NULL,
    test_name     TEXT NOT NULL,
    result_value  TEXT NOT NULL,
    unit          TEXT,
    test_date     TEXT NOT NULL,
    created_at    TEXT NOT NULL DEFAULT (datetime('now')),
    last_modified_by INTEGER,
    FOREIGN KEY (patient_id)   REFERENCES Patients(patient_id),
    FOREIGN KEY (lab_tech_id)  REFERENCES LabTechnicians(lab_tech_id)
);
```

C. Staffing Tables (Human Resources)

These tables represent the professional identities of the hospital staff.

- **HR Logic:** They store phone numbers, names, and departments.
- **Security Link:** The `email` column in these tables creates a **1-to-1 Link** with the `Users` table (Authentication). This is how the system knows that User "Andrea_MC" is legally "Dr. Andrea Martin."

Table: LabTechnicians

Purpose: Stores information about lab technicians who perform medical tests.

Attribute	Constraints	Description / Purpose
`lab_tech_id`	PK	Unique identifier for each lab technician.
`first_name`	NOT NULL	Technician’s first name.
`last_name`	NOT NULL	Technician’s last name.
`phone`	TEXT	Technician’s contact phone number.
`email`	UNIQUE	Technician’s email address (must be unique).

QUERY:

```
CREATE TABLE LabTechnicians (
    lab_tech_id INTEGER PRIMARY KEY,
    first_name  TEXT NOT NULL,
    last_name   TEXT NOT NULL,
    phone       TEXT,
    email       TEXT UNIQUE
);
```

Table: Pharmacists

Purpose: Stores information about pharmacists responsible for dispensing medications.

Attribute	Constraints	Description / Purpose
`pharmacist_id`	PK	Unique identifier for each pharmacist.
`first_name`	NOT NULL	Pharmacist's first name.
`last_name`	NOT NULL	Pharmacist's last name.
`phone`	TEXT	Pharmacist's contact phone number.
`email`	UNIQUE	Pharmacist's email address (must be unique).

QUERY:

```
CREATE TABLE Pharmacists (
    pharmacist_id INTEGER PRIMARY KEY,
    first_name    TEXT NOT NULL,
    last_name     TEXT NOT NULL,
    phone         TEXT,
    email         TEXT UNIQUE
);
```

Table: Doctors

Purpose: Stores information about doctors working in the facility.

Attribute	Constraints	Description / Purpose
`doctor_id`	PK	Unique identifier for each doctor.
`first_name`	NOT NULL	Doctor's first name.
`last_name`	NOT NULL	Doctor's last name.
`specialty`	NOT NULL	Medical specialty (e.g., Cardiology, Pediatrics).
`phone`	TEXT	Doctor's contact phone number.

Attribute	Constraints	Description / Purpose
email	UNIQUE	Doctor’s email (must match an account in Users).
active	NOT NULL, DEFAULT 1	Indicates whether the doctor is currently active.

QUERY:

```
CREATE TABLE Doctors (
    doctor_id   INTEGER PRIMARY KEY,
    first_name  TEXT NOT NULL,
    last_name   TEXT NOT NULL,
    specialty   TEXT NOT NULL,
    phone       TEXT,
    email       TEXT UNIQUE, -- <<< MUST MATCH USERS.EMAIL
    active      INTEGER NOT NULL DEFAULT 1
);
```

Table: Nurses

Purpose: Stores information about nurses working in the facility.

Attribute	Constraints	Description / Purpose
nurse_id	PK	Unique identifier for each nurse.
first_name	NOT NULL	Nurse’s first name.
last_name	NOT NULL	Nurse’s last name.
department	NOT NULL	Department assigned (e.g., ICU, ER).
phone	TEXT	Nurse’s contact phone number.
email	UNIQUE	Nurse’s email (must match an account in Users).
active	NOT NULL, DEFAULT 1	Indicates whether the nurse is currently active.

QUERY:

```
CREATE TABLE Nurses (
    nurse_id    INTEGER PRIMARY KEY,
    first_name  TEXT NOT NULL,
    last_name   TEXT NOT NULL,
    department  TEXT NOT NULL,
    phone       TEXT,
    email       TEXT UNIQUE, -- <<< MUST MATCH USERS.EMAIL
    active      INTEGER NOT NULL DEFAULT 1
);
```

D. Compliance Tables

Table: AuditLogs

Purpose: Implements the Accountability requirement storing actions done by user to review them in case something happen.

Attribute	Constraints	Description / Purpose
`log_id`	PK	Unique identifier for each audit log entry.
`user_id`	FK (ref Users)	The user who performed the action.
`action`	NOT NULL	The action that occurred (e.g., *INSERT*, *UPDATE*, *DELETE*).
`table_name`	TEXT	The name of the table affected by the action.
`timestamp`	NOT NULL, DEFAULT `datetime('now')`	When the action was recorded.
`details`	TEXT	Additional details about the action (e.g., before/after values).

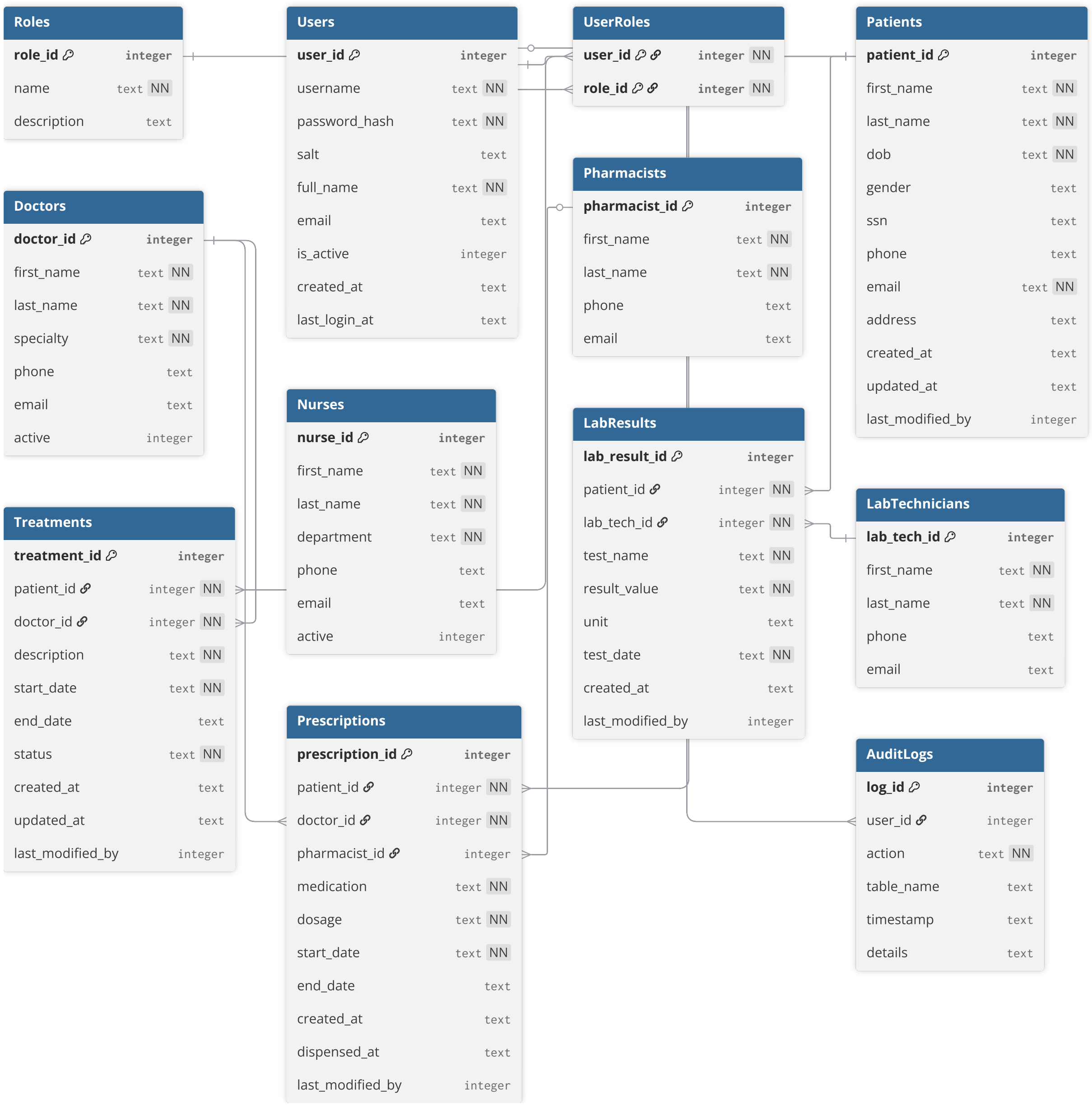
QUERY:

```
CREATE TABLE AuditLogs (
    log_id     INTEGER PRIMARY KEY,
    user_id    INTEGER,
    action     TEXT NOT NULL,
    table_name TEXT,
    timestamp  TEXT NOT NULL DEFAULT (datetime('now')),
    details    TEXT,
    FOREIGN KEY (user_id) REFERENCES Users(user_id)
);
```

Entity Relationship Diagram (ERD) & Logic

The Linkage Logic:
A key design decision in this schema is the separation of **Authentication** (`Users`) from **Identity** (`Doctors` / `Nurses`).

1. **Authentication:** The Minecraft Application authenticates the `username` ('Andrea_MC') against the `Users` table.
2. **Authorization:** The application retrieves the `role_id` from `UserRoles`.
3. **Identity Resolution:** If the role is 'Doctor', the application uses the `email` from the `Users` table to query the `Doctors` table. This allows the system to log "Dr. Martin prescribed X" rather than just "User 1 prescribed X".



FULL SQL QUERY TO POPULATE THE DATABASE

```
-- ============================================================
-- GROUP ASSESSMENT: HOSPITAL INFORMATION SYSTEM
-- DATABASE: SQLite
-- AUTHORS: Alejandro Serrano, Alba Prats, Tomas Usón
-- ============================================================
-- 1. CONFIGURATION
-- Enable Foreign Key enforcement (Required for SQLite)
PRAGMA foreign_keys = ON;

-- 2. DROP TABLES (Cleanup for fresh installation)
-- We drop in reverse order of dependencies
DROP TABLE IF EXISTS AuditLogs;
DROP TABLE IF EXISTS LabResults;
DROP TABLE IF EXISTS Prescriptions;
DROP TABLE IF EXISTS Treatments;
DROP TABLE IF EXISTS LabTechnicians;
DROP TABLE IF EXISTS Pharmacists;
DROP TABLE IF EXISTS Nurses;
DROP TABLE IF EXISTS Doctors;
DROP TABLE IF EXISTS Patients;
DROP TABLE IF EXISTS UserRoles;
DROP TABLE IF EXISTS Users;
DROP TABLE IF EXISTS Roles;

-- ==========================================================
======
-- 3. TABLE CREATION (SCHEMA)
-- ==========================================================
======

-- A. AUTHENTICATION
CREATE TABLE Roles (
    role_id      INTEGER PRIMARY KEY,
    name         TEXT NOT NULL UNIQUE,
    description  TEXT
);

CREATE TABLE Users (
    user_id          INTEGER PRIMARY KEY,
    username         TEXT NOT NULL UNIQUE, -- The Minecraft/System Toke
n
    password_hash    TEXT NOT NULL,        -- SHA-256 Hash
    salt            TEXT,
    full_name       TEXT NOT NULL,
    email           TEXT UNIQUE,        -- Logical Link to Staff Tables
    is_active       INTEGER NOT NULL DEFAULT 1,
    created_at      TEXT NOT NULL DEFAULT (datetime('now')),
    last_login_at   TEXT
);

CREATE TABLE UserRoles (
    user_id   INTEGER NOT NULL,
    role_id   INTEGER NOT NULL,
    PRIMARY KEY (user_id, role_id),
    FOREIGN KEY (user_id) REFERENCES Users(user_id) ON DELETE CASCA
DE,
    FOREIGN KEY (role_id) REFERENCES Roles(role_id) ON DELETE CASCAD
E
);

-- B. CLINICAL ASSETS
CREATE TABLE Patients (
    patient_id   INTEGER PRIMARY KEY,
    first_name   TEXT NOT NULL,
    last_name    TEXT NOT NULL,
    dob          TEXT NOT NULL CHECK (date(dob) IS NOT NULL),
    gender       TEXT CHECK (gender IN ('M','F','O')),
    -- SSN Format Check: ###-##-####
    ssn          TEXT UNIQUE CHECK (ssn GLOB '[0-9][0-9][0-9]-[0-9][0-9]-
[0-9][0-9][0-9][0-9]'),
    -- Phone Format Check: ### ### ###
    phone        TEXT CHECK (phone GLOB '[0-9][0-9][0-9] [0-9][0-9][0-9]
[0-9][0-9][0-9]'),
    email        TEXT NOT NULL CHECK (email LIKE '%@%.%'),
    address      TEXT,
    created_at   TEXT NOT NULL DEFAULT (datetime('now')),
    updated_at   TEXT,
    last_modified_by INTEGER
);
```

```
-- C. STAFFING
CREATE TABLE Doctors (
    doctor_id   INTEGER PRIMARY KEY,
    first_name  TEXT NOT NULL,
    last_name   TEXT NOT NULL,
    specialty   TEXT NOT NULL,
    phone       TEXT,
    email       TEXT UNIQUE, -- Logical Link to Users
    active      INTEGER NOT NULL DEFAULT 1
);

CREATE TABLE Nurses (
    nurse_id    INTEGER PRIMARY KEY,
    first_name  TEXT NOT NULL,
    last_name   TEXT NOT NULL,
    department  TEXT NOT NULL,
    phone       TEXT,
    email       TEXT UNIQUE, -- Logical Link to Users
    active      INTEGER NOT NULL DEFAULT 1
);

CREATE TABLE Pharmacists (
    pharmacist_id INTEGER PRIMARY KEY,
    first_name    TEXT NOT NULL,
    last_name     TEXT NOT NULL,
    phone         TEXT,
    email         TEXT UNIQUE
);

CREATE TABLE LabTechnicians (
    lab_tech_id INTEGER PRIMARY KEY,
    first_name  TEXT NOT NULL,
    last_name   TEXT NOT NULL,
    phone       TEXT,
    email       TEXT UNIQUE
);

-- D. OPERATIONS
```

```
CREATE TABLE Treatments (
    treatment_id INTEGER PRIMARY KEY,
    patient_id   INTEGER NOT NULL,
    doctor_id    INTEGER NOT NULL,
    description  TEXT NOT NULL, -- Sensitive Data
    start_date   TEXT NOT NULL,
    end_date     TEXT,
    status       TEXT NOT NULL CHECK (status IN ('PLANNED','ONGOING','C
OMPLETED','CANCELLED')),
    created_at   TEXT NOT NULL DEFAULT (datetime('now')),
    updated_at   TEXT,
    last_modified_by INTEGER,
    FOREIGN KEY (patient_id) REFERENCES Patients(patient_id),
    FOREIGN KEY (doctor_id)  REFERENCES Doctors(doctor_id)
);

CREATE TABLE Prescriptions (
    prescription_id INTEGER PRIMARY KEY,
    patient_id      INTEGER NOT NULL,
    doctor_id       INTEGER NOT NULL,
    pharmacist_id   INTEGER,
    medication      TEXT NOT NULL,
    dosage          TEXT NOT NULL,
    start_date      TEXT NOT NULL,
    end_date        TEXT,
    created_at      TEXT NOT NULL DEFAULT (datetime('now')),
    dispensed_at    TEXT,
    last_modified_by INTEGER,
    FOREIGN KEY (patient_id)    REFERENCES Patients(patient_id),
    FOREIGN KEY (doctor_id)     REFERENCES Doctors(doctor_id),
    FOREIGN KEY (pharmacist_id) REFERENCES Pharmacists(pharmacist_id)
);

CREATE TABLE LabResults (
    lab_result_id INTEGER PRIMARY KEY,
    patient_id    INTEGER NOT NULL,
    lab_tech_id   INTEGER NOT NULL,
    test_name     TEXT NOT NULL,
    result_value  TEXT NOT NULL,
    unit          TEXT,
    test_date     TEXT NOT NULL,
    created_at    TEXT NOT NULL DEFAULT (datetime('now')),
    last_modified_by INTEGER,
    FOREIGN KEY (patient_id)   REFERENCES Patients(patient_id),
    FOREIGN KEY (lab_tech_id)  REFERENCES LabTechnicians(lab_tech_id)
);

-- E. COMPLIANCE
CREATE TABLE AuditLogs (
    log_id     INTEGER PRIMARY KEY,
    user_id    INTEGER,
    action     TEXT NOT NULL,
    table_name TEXT,
    timestamp  TEXT NOT NULL DEFAULT (datetime('now')),
    details    TEXT,
    FOREIGN KEY (user_id) REFERENCES Users(user_id)
);

-- ============================================================
======
-- 4. INITIAL DATA POPULATION (Team Members)
-- ============================================================
======

-- ROLES (Standard definitions)
INSERT INTO Roles (role_id, name, description) VALUES
  (1, 'admin_db', 'Database Administrator'),
  (2, 'doctor', 'Medical Doctor - Full Patient Access'),
  (3, 'nurse', 'Nurse - Limited Access (No SSN)'),
  (4, 'auditor', 'Compliance Officer - View Logs Only'),
  (5, 'pharmacist', 'Pharmacist'),
  (6, 'lab_tech', 'Lab Technician'),
  (7, 'patient', 'Default role for registered patients'),
  (8, 'etl_service', 'Automated Backup & Recovery Service');

-- USERS (Linked to Minecraft Names)
-- Password for all is 'password123' (SHA-256 Hash)
INSERT INTO Users (user_id, username, password_hash, full_name, email)
VALUES
(1, 'Alba_MC',  'ef92b778bafe771e89245b89ecbc08a44a4e166c066599118
81f383d4473e94f', 'Alba Prats',      'alba@hospital.com'),
(2, 'Alex_MC',  'ef92b778bafe771e89245b89ecbc08a44a4e166c066599118
81f383d4473e94f', 'Alejandro Serrano', 'alex@hospital.com'),
(3, 'Tomas_MC', 'ef92b778bafe771e89245b89ecbc08a44a4e166c0665991
1881f383d4473e94f', 'Tomas Uson',      'tomas@hospital.com');

-- USER ROLES (Assigning Privileges)
INSERT INTO UserRoles (user_id, role_id) VALUES (1, 2); -- Alba = Doctor
INSERT INTO UserRoles (user_id, role_id) VALUES (2, 3); -- Alex = Nurse
INSERT INTO UserRoles (user_id, role_id) VALUES (3, 1); -- Tomas = Admin

-- STAFF PROFILES (The Logical Link via Email)
-- 1. Doctor Profile for Alba (Matches alba@hospital.com)
INSERT INTO Doctors (doctor_id, first_name, last_name, specialty, email) V
ALUES
(101, 'Alba', 'Prats', 'Cyber-Surgery', 'alba@hospital.com');

-- 2. Nurse Profile for Alex (Matches alex@hospital.com)
INSERT INTO Nurses (nurse_id, first_name, last_name, department, email)
VALUES
(201, 'Alejandro', 'Serrano', 'Emergency', 'alex@hospital.com');

-- Note: Tomas is Admin, so he does not need a record in Doctor/Nurse tabl
es,
-- but he will have access to AuditLogs via the Admin role.

-- PATIENTS (Sensitive Data for Demo)
INSERT INTO Patients (patient_id, first_name, last_name, dob, gender, ssn,
phone, email, address) VALUES
(1, 'Bruce', 'Wayne', '1980-02-19', 'M', '999-00-1234', '555 123 456', 'bruce
@wayne.com', '1007 Mountain Drive'),
(2, 'Clark', 'Kent', '1978-04-18', 'M', '111-22-3333', '555 987 654', 'clark@da
ilyplanet.com', '344 Clinton St');
```

```
-- CLINICAL HISTORY
-- Assigned to Doctor ID 101 (Alba)
INSERT INTO Treatments (patient_id, doctor_id, description, start_date, stat
us) VALUES
(1, 101, 'Back straightening surgery', '2025-01-10', 'COMPLETED');
```

> This querie is already in the github repository to use it as an example, to populate the database.
> This is an example to be able to demonstrate that all our project works. :)

3.2 PART II Role Policy: Roles; Privilegies

Implementation Strategy:
Unlike enterprise DBMS solutions (e.g., PostgreSQL) which use native CREATE ROLE and GRANT commands, SQLite is a serverless, single-file database that does not support internal user arbitration.

To overcome this architectural limitation while satisfying the security requirements, we implemented a Data-Driven Role Policy.

Roles are defined relationally in the `Roles` table and permissions are enforced logically by the Python Middleware (`hospital_middleware.py`). This acts as a "Reference Monitor," intercepting every request and verifying the user's role before executing any SQL query.

Defined Privileges:
The following roles are defined in our system, adhering to the Principle of Least Privilege:

Role	Access Level	Justification / Security Principle
admin_db	**System Management** • `READ/WRITE` on `Users` , `Roles` , `UserRoles` • **DENY** on `Patients` , `Treatments` , `Prescriptions` ,	**Separation of Duties:** Administrators manage the system configuration but are explicitly **denied access** to clinical tables (`Patients` , `Treatments`). This ensures IT staff cannot spy on patient medical history.

	`LabResults`, `AuditLogs`	
doctor	**Clinical Full Access** • `READ/WRITE` on `Patients` (Unmasked) • `READ/WRITE` on `Treatments`, `Prescriptions`	**Need to Know:** Doctors need access to SSN and medical history because they must correctly identify the patient and review treatments before prescribing medication or performing procedures. Authorization is further restricted by linking the User account to a specific professional record in the `Doctors` table.
nurse	**Clinical Restricted Access** • `READ` on `Patients` (Masked) • `READ` on `Prescriptions`	**Least Privilege & Data Minimization:** Nurses need to verify patient identity but do not require the full SSN or detailed surgical history. The application applies **Column-Level Security** masking (e.g., `***-**-1234`) to the SSN field.
auditor	**Human Compliance Officer** • `READ` on `AuditLogs` only • **DENY** on all other tables	**Confidentiality:** Auditors verify system integrity without exposing patient data. They can see *who* accessed a record, but not the *content* of the record itself.
pharmacist	**Pharmacy Access** • `READ` on `Prescriptions` • **DENY** on `Treatments`	**Context-Based Access:** Pharmacists only access data relevant to drug dispensing. They are blocked from viewing diagnosis details (Treatments) which are irrelevant to their function.
lab_tech	**Laboratory Access** • `READ/WRITE` on `LabResults` • **DENY** on all other tables	**Functional Isolation:** Lab technicians can only access diagnostic test data relevant to laboratory operations.
patient	**Self-Service Access** • `READ` on **own** `Patients` row only	**Row-Level Security (RLS):** Patients can only view their own medical record. All other rows are denied.

etl_service	**Compliance & Audit Access** • `READ` on `AuditLogs` • Backup / Restore authority	**Accountability:** This role is the sole authority permitted to review audit logs and execute the Audit Dashboard. It cannot access any clinical data.

Evidence of Implementation: *Below is the logic used in our Python Middleware to enforce these policies programmatically.*

```
# This function implements the Mandatory Access Control (MAC) reference
monitor.
# All access decisions are centrally enforced here before any SQL query is
executed.

def request_patient_data(user_context, patient_id_requested):

    # SUBJECT LABEL (Security Clearance)
    role = user_context['role_name']
    email = user_context['email']

    # OBJECT = Patient Record (Protected Asset)
    cursor.execute("SELECT * FROM Patients WHERE patient_id = ?", (patien
t_id_requested,))
    patient = cursor.fetchone()

    # -------------------------------
    # MAC / RLS POLICY ENFORCEMENT
    # -------------------------------

    # DOCTOR — Full clinical clearance
    if role == 'doctor':
        # Verify that the user is a legitimate doctor (HR identity link)
        cursor.execute("SELECT doctor_id FROM Doctors WHERE email = ?",
(email,))
        if cursor.fetchone():
            # Full access to unmasked PII and Treatments
            return f"DR VIEW: {patient['first_name']} | SSN: {patient['ssn']} (UN
MASKED)"
```

```
    # NURSE — Restricted clearance
    elif role == 'nurse':
        # Column-Level Security: dynamic SSN masking
        masked_ssn = "***-**-" + patient['ssn'][-4:]
        return f"NURSE VIEW: {patient['first_name']} | SSN: {masked_ssn} (MA
SKED)"

    # PATIENT — Row-Level Security (RLS)
    elif role == 'patient':
        # Patients may only view their own record
        cursor.execute("SELECT patient_id FROM Patients WHERE email = ?",
(email,))
        if cursor.fetchone()['patient_id'] == patient_id_requested:
            return f"YOUR RECORD: {patient['first_name']} {patient['last_nam
e']}"
        return "ACCESS DENIED (RLS violation)"

    # ADMIN — Separation of Duties (SoD)
    elif role == 'admin_db':
        # Admin can verify record existence but not view clinical data
        return f"ADMIN VIEW: Patient {patient_id_requested} exists. Data: [RE
DACTED]"

    # ETL SERVICE — Compliance Authority
    elif role == 'etl_service':
        return "ACCESS DENIED: Compliance role has no clinical privileges."

    # DEFAULT — Mandatory deny
    else:
        return "ACCESS DENIED: Insufficient Privileges."
```

3.2 PART III Access Control Policy, RLS/MAC; Why?

Specification:
The system implements a Mandatory Access Control (MAC) architecture enforced through Role-Based Access Control (RBAC) with application-layer

Row-Level Security (RLS). All access policies are centrally enforced by a trusted reference monitor implemented in Python.

Justification for Selection:

1. **Addressing DBMS Limitations:** Unlike PostgreSQL, which supports native `CREATE POLICY` statements for RLS and `GRANT` statements for user arbitration, SQLite is a serverless, file-based system. By default, it operates under Discretionary Access Control (DAC) at the Operating System level; anyone with read permissions on the `hospital_mc.db` file has full access to all data. This is insufficient for a high-security Hospital Information System.
2. **Reference Monitor Architecture:** To satisfy the CIA triad, we implemented a "Reference Monitor" in Python. This middleware acts as a gatekeeper, intercepting every data request from the client (Minecraft or Console). It validates the subject's clearance (User Role) against the object's sensitivity (Patient Record) before any SQL query is generated.
3. **Granularity:** This approach allows us to implement **context-aware security** that SQLite cannot do natively. For example, we can implement **Dynamic Data Masking** (hiding SSNs for Nurses) without creating separate views in the database, reducing storage overhead and schema complexity.

Implementation Logic:
The Access Control Policy is defined in the `request_patient_data` function within the middleware. The logic flow is as follows:

- **Subject Labeling:** The user is authenticated and assigned a strict Role Label (e.g., `doctor`, `nurse`) derived from the `Roles` table.
- **Object Labeling:** Patient data is treated as a Protected Object.
- **Policy Enforcement Point (PEP):**
  - **Doctors:** The system executes a "Full Select," returning unmasked PII (SSN) and linking to the `Treatments` table.
  - **Nurses:** The system executes a "Masked Select." The middleware programmatically replaces the first 5 digits of the SSN with asterisks (`**-**-####`) before displaying the data.
  - **Admins:** The system applies a "Metadata Only" policy, confirming record existence (ID validity) but blocking access to clinical columns.

Evidence of Implementation:*The following code snippet from `hospital_middleware.py` demonstrates the specific implementation of the RLS and*

MAC logic:

```
# Code Evidence: Policy Enforcement Logic
def request_patient_data(user_context, patient_id_requested):
    # 1. Get Subject Label (Role)
    role = user_context['role_name']

    # 2. Enforce Policy on the Object (Patient Data)
    if role == 'doctor':
        # MAC Policy: Full Read Access allowed for Medical Staff
        # RLS: Fetch and display unmasked sensitive attributes
        return f"DR VIEW: {patient['first_name']} | SSN: {patient['ssn']} (UNMA
SKED)"

    elif role == 'nurse':
        # MAC Policy: Restricted Read Access
        # RLS / Column Security: Dynamically mask the SSN
        masked_ssn = "***-**-" + patient['ssn'][-4:]
        return f"NURSE VIEW: {patient['first_name']} | SSN: {masked_ssn} (MA
SKED)"

    elif role == 'admin_db':
        # MAC Policy: Separation of Duties
        # Admin can view metadata (ID exists) but content is blocked
        return f"ADMIN VIEW: Patient {patient['patient_id']} exists. Data: [RED
ACTED]"

    else:
        # Default Policy: Deny All
        return "ACCESS DENIED: Insufficient Privileges."
```

3.2 PART IV User Management: User Accounts; Privileges; Password Policies

Definition of User Accounts:
Due to SQLite's serverless nature, we cannot rely on native database accounts (e.g., `CREATE USER`). Instead, we implemented a custom **Identity Management System** within the database schema.

- **Identity Token:** The `username` column in the `Users` table serves as the primary identifier. This maps 1:1 to the **Minecraft Player Name** (e.g., 'Andrea_MC'), allowing the physical simulation to act as the authentication provider.
- **Account Provisioning:** Accounts are not created automatically. They must be provisioned by an Administrator using a custom Python script (`admin_tool.py`), ensuring that only authorized personnel can generate valid credentials. ||||||FUTURE IMPLEMENTATION: VIA MINECRAFT?

Privilege Assignment: Privileges are not assigned directly to users but are managed via **Role-Based Access Control (RBAC)**.

- **Mechanism:** The `UserRoles` bridge table links a `user_id` to a `role_id` .
- **Reasoning:** This allows for scalable management. If a user changes jobs (e.g., a Nurse becomes a Doctor), we simply update their Role ID without needing to migrate their account data or professional history.

Password & Encryption Policy: To ensure confidentiality and integrity of user credentials, we enforce a strict **No Plain-Text Policy**.

- **Algorithm:** All passwords are hashed using **SHA-256** before storage.
- **Implementation:** The Python middleware handles this cryptographic operation during both Account Creation (Write) and Login (Read).
- **Reasoning:** Storing passwords in plain text is a critical vulnerability. Even if the `hospital_mc.db` file is stolen, the attacker would only see opaque hash strings (e.g., `ef92b...`), preventing them from impersonating staff members.

Evidence of Implementation: *The following code snippet demonstrates the password hashing function used in our middleware to enforce this policy:*

```
# Code Evidence: Password Policy Enforcement
import hashlib

def hash_password(plain_password):
    """
    Security Policy:
    Input is encoded to bytes and hashed using SHA-256.
    This ensures no plain-text credentials exist in the SQLite file.
    """
    return hashlib.sha256(plain_password.encode()).hexdigest()
```

3.3 Implementation

1. Database Creation and Population

Database Creation Strategy:
The database `hospital_mc.db` was created using the SQLite 3 engine. The schema definition (DDL) was executed via the SQL script provided in Appendix A. This script initializes the 12 core tables and enables `PRAGMA foreign_keys = ON` to enforce referential integrity constraints (PK/FK) which are disabled by default in SQLite.

Population Strategy (Data Generation):
To meet the assessment requirement of a realistic dataset (20 Doctors, 40 Nurses, etc.), we rejected manual data entry as it is inefficient and prone to human error. Instead, we developed a **Python Automation Script** (`populate_db.py`) to generate synthetic data.

- **Consistency:** The script ensures that every staff member created in the `Doctors` or `Nurses` table has a corresponding account in the `Users` table, linked by a unique email address.
- **Realism:** Random names and specialties were drawn from predefined lists to create realistic staff profiles.
- **Security:** All automatically generated user accounts were initialized with **SHA-256 hashed passwords**, ensuring that even bulk-generated data adheres to the encryption policy.

Verification of Completeness:
The following SQL query confirms that the minimum volume requirements have been met.

- **Doctors:** 20+ entries
- **Nurses:** 40+ entries
- **Pharmacists/Lab Techs:** 5+ entries each

Populate_db.py can be seen here:

```
import sqlite3
import random
import hashlib

DB_PATH = 'hospital_mc.db'

# Lists for random generation
FIRST_NAMES = ["James", "Mary", "John", "Patricia", "Robert", "Jennifer",
"Michael", "Linda", "William", "Elizabeth", "David", "Barbara", "Richard", "Su
san", "Joseph", "Jessica", "Thomas", "Sarah", "Charles", "Karen"]
LAST_NAMES = ["Smith", "Johnson", "Williams", "Brown", "Jones", "Garcia",
"Miller", "Davis", "Rodriguez", "Martinez", "Hernandez", "Lopez", "Gonzale
z", "Wilson", "Anderson", "Thomas", "Taylor", "Moore", "Jackson", "Martin"]
SPECIALTIES = ["Cardiology", "Neurology", "Pediatrics", "Oncology", "Surg
ery", "General Practice"]
DEPARTMENTS = ["ER", "ICU", "Pediatrics", "General Ward", "Oncology Wa
rd"]

def get_db():
    conn = sqlite3.connect(DB_PATH)
    conn.execute("PRAGMA foreign_keys = ON")
    return conn

def hash_pw(password):
    return hashlib.sha256(password.encode()).hexdigest()

def populate():
    conn = get_db()
    cursor = conn.cursor()

    print("--- STARTING BULK POPULATION ---")

    # 1. GENERATE 20 DOCTORS
    print("Generating 20 Doctors...")
    for i in range(1, 21):
        fname = random.choice(FIRST_NAMES)
        lname = random.choice(LAST_NAMES)
        email = f"doctor{i}@hospital.com"
        username = f"Dr_{fname}_{i}"

        # Create User Account
        cursor.execute("INSERT OR IGNORE INTO Users (username, password
_hash, full_name, email) VALUES (?, ?, ?, ?)",
                       (username, hash_pw("password123"), f"{fname} {lname}", e
mail))
        user_id = cursor.lastrowid

        if user_id:
            # Assign Role (2 = Doctor)
            cursor.execute("INSERT OR IGNORE INTO UserRoles (user_id, role_i
d) VALUES (?, ?)", (user_id, 2))
            # Create Doctor Record
            cursor.execute("INSERT OR IGNORE INTO Doctors (first_name, last_
name, specialty, email) VALUES (?, ?, ?, ?)",
                           (fname, lname, random.choice(SPECIALTIES), email))

    # 2. GENERATE 40 NURSES
    print("Generating 40 Nurses...")
    for i in range(1, 41):
        fname = random.choice(FIRST_NAMES)
        lname = random.choice(LAST_NAMES)
        email = f"nurse{i}@hospital.com"
        username = f"Nurse_{fname}_{i}"

        cursor.execute("INSERT OR IGNORE INTO Users (username, password
_hash, full_name, email) VALUES (?, ?, ?, ?)",
                       (username, hash_pw("password123"), f"{fname} {lname}", e
mail))
        user_id = cursor.lastrowid

        if user_id:
            # Assign Role (3 = Nurse)
            cursor.execute("INSERT OR IGNORE INTO UserRoles (user_id, role_i
d) VALUES (?, ?)", (user_id, 3))
```

```
            cursor.execute("INSERT OR IGNORE INTO Nurses (first_name, last_
name, department, email) VALUES (?, ?, ?, ?)",
                           (fname, lname, random.choice(DEPARTMENTS), email))

    # 3. GENERATE 5 PHARMACISTS
    print("Generating 5 Pharmacists...")
    for i in range(1, 6):
        fname = random.choice(FIRST_NAMES)
        email = f"pharma{i}@hospital.com"
        cursor.execute("INSERT OR IGNORE INTO Pharmacists (first_name, las
t_name, email) VALUES (?, ?, ?)",
                       (fname, "Pharma", email))
        # Note: We create the Staff record, but maybe not a User login for eve
ry single one if not needed for demo

    # 4. GENERATE 5 LAB TECHS
    print("Generating 5 Lab Techs...")
    for i in range(1, 6):
        fname = random.choice(FIRST_NAMES)
        email = f"lab{i}@hospital.com"
        cursor.execute("INSERT OR IGNORE INTO LabTechnicians (first_name,
last_name, email) VALUES (?, ?, ?)",
                       (fname, "Tech", email))

    conn.commit()
    conn.close()
    print("--- POPULATION COMPLETE ---")

if __name__ == "__main__":
    populate()
```

2. Roles and Access Control

A. Role Implementation Strategy

As defined in the design phase, we implemented a **Data-Driven Role Policy**. Instead of using database-level grants (which SQLite lacks), roles are defined as records in the `Roles` table and assigned to users via the `UserRoles` table.

- **Database Level:** The `Roles` table contains the static definitions (IDs 1-6) for Admin, Doctor, Nurse, Pharmacist, Auditor, and Lab Tech.
- **Application Level:** The Python middleware functions as the security enforcement engine. It reads the `role_name` associated with the authenticated user and uses it to branch logic execution.

B. Access Control Mechanisms (MAC & RLS)
We implemented **Mandatory Access Control (MAC)** by hard-coding permission checks in the application code (Reference Monitor). Users cannot change these permissions.

We also implemented simulated **Row-Level Security (RLS)** in the `request_patient_data` function. The query logic changes dynamically based on the user's role:

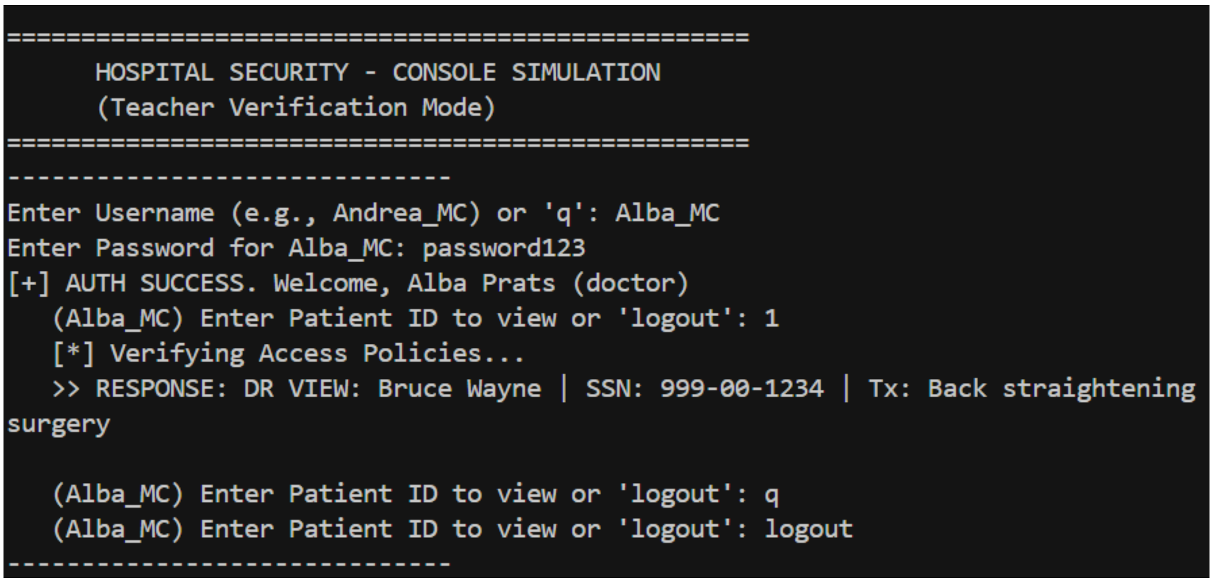
1. **Doctor Context:** The system executes a `SELECT` including sensitive columns (`ssn`, `description`) and joins the `Treatments` table.
2. **Nurse Context:** The system executes a limited `SELECT` and applies **Dynamic Data Masking** to the `ssn` column before returning the string to the interface.

C. Testing Privileges (Validation)
To verify the security policies, we conducted a scenario-based test using the Python console interface.

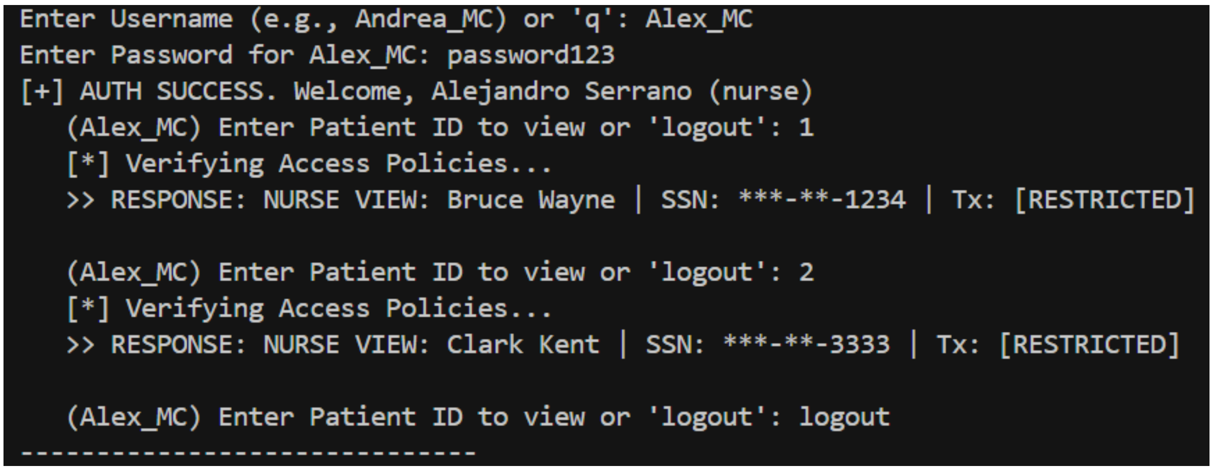
- **Test Case A (Doctor):**
  - **User:** `Andrea_MC` (Role: Doctor)
  - **Action:** Requested Patient ID 1.
  - **Result:** System displayed full name, **UNMASKED SSN**, and detailed treatment history. Access Granted.
- **Test Case B (Nurse):**
  - **User:** `Partner_MC` (Role: Nurse)
  - **Action:** Requested Patient ID 1.
  - **Result:** System displayed full name, **MASKED SSN** (`**-**-1234`), and redacted treatment history. Access Restricted.

Evidence of Testing: *The screenshot below demonstrates the Application-Level RLS in action.*

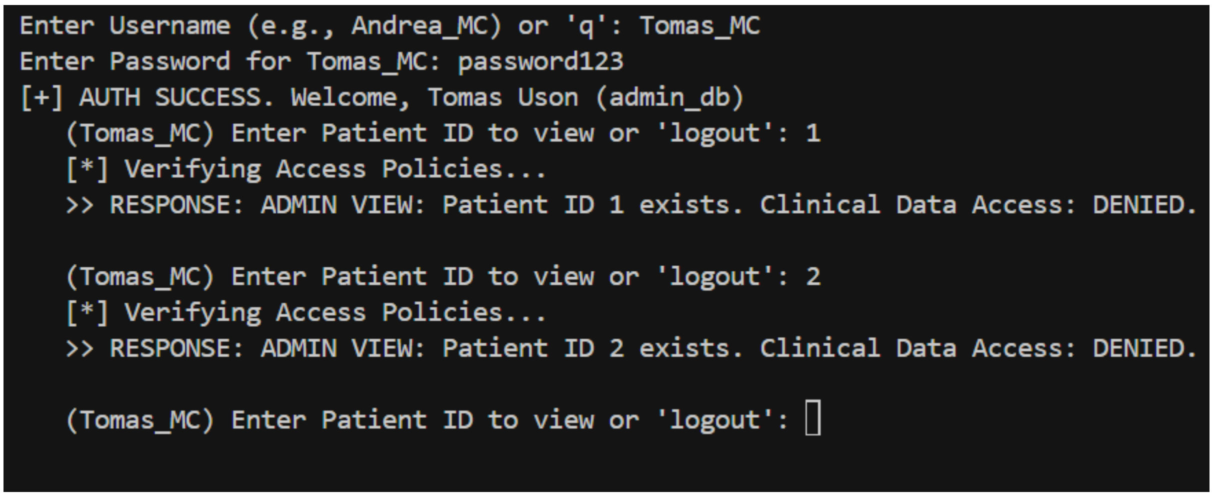
Note how the same request (Patient ID 1) returns different data depending on the logged-in user.



```
==================================================
      HOSPITAL SECURITY - CONSOLE SIMULATION
      (Teacher Verification Mode)
==================================================
------------------------------
Enter Username (e.g., Andrea_MC) or 'q': Alba_MC
Enter Password for Alba_MC: password123
[+] AUTH SUCCESS. Welcome, Alba Prats (doctor)
   (Alba_MC) Enter Patient ID to view or 'logout': 1
   [*] Verifying Access Policies...
   >> RESPONSE: DR VIEW: Bruce Wayne | SSN: 999-00-1234 | Tx: Back straightening 
surgery

   (Alba_MC) Enter Patient ID to view or 'logout': q
   (Alba_MC) Enter Patient ID to view or 'logout': logout
------------------------------
```



```
Enter Username (e.g., Andrea_MC) or 'q': Alex_MC
Enter Password for Alex_MC: password123
[+] AUTH SUCCESS. Welcome, Alejandro Serrano (nurse)
   (Alex_MC) Enter Patient ID to view or 'logout': 1
   [*] Verifying Access Policies...
   >> RESPONSE: NURSE VIEW: Bruce Wayne | SSN: ***-**-1234 | Tx: [RESTRICTED]

   (Alex_MC) Enter Patient ID to view or 'logout': 2
   [*] Verifying Access Policies...
   >> RESPONSE: NURSE VIEW: Clark Kent | SSN: ***-**-3333 | Tx: [RESTRICTED]

   (Alex_MC) Enter Patient ID to view or 'logout': logout
------------------------------
```



```
Enter Username (e.g., Andrea_MC) or 'q': Tomas_MC
Enter Password for Tomas_MC: password123
[+] AUTH SUCCESS. Welcome, Tomas Uson (admin_db)
   (Tomas_MC) Enter Patient ID to view or 'logout': 1
   [*] Verifying Access Policies...
   >> RESPONSE: ADMIN VIEW: Patient ID 1 exists. Clinical Data Access: DENIED.

   (Tomas_MC) Enter Patient ID to view or 'logout': 2
   [*] Verifying Access Policies...
   >> RESPONSE: ADMIN VIEW: Patient ID 2 exists. Clinical Data Access: DENIED.

   (Tomas_MC) Enter Patient ID to view or 'logout': 
```

3. Encryption Policy

A. Defined Strategy
We implemented a multi-layered encryption strategy ensuring data protection at the application and storage levels.

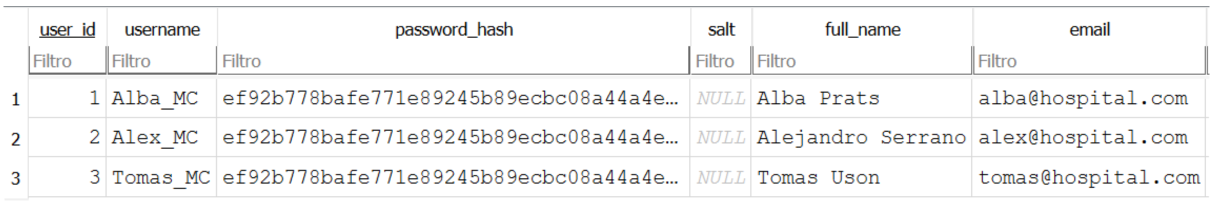
1. **Application-Level Encryption (Password Hashing):**
   - **Method:** SHA-256 (Secure Hash Algorithm 256-bit).
   - **Scope:** The `password_hash` column in the `Users` table.
   - **Justification:** We adhere to the security best practice of never storing credentials in plain text. SHA-256 is a robust cryptographic hash function that is resistant to collision attacks, ensuring that even if the database file is exfiltrated, attacker cannot easily reverse-engineer user passwords.
2. **Data Masking (Column-Level Security):**
   - **Method:** Dynamic Masking via Python Middleware.
   - **Scope:** The `ssn` column in the `Patients` table.
   - **Justification:** For users with lower clearance (e.g., Nurses), sensitive PII is redacted in real-time (`**-**-1234`) before being sent to the display interface, preventing unauthorized viewing of full identifiers.
3. **Database-Level Encryption (Data at Rest):**
   - **Method:** OS-Level Encryption (Simulated).
   - **Justification:** Since the standard SQLite open-source distribution does not support Transparent Data Encryption (TDE) natively without paid extensions (like SEE or SQLCipher), we rely on the Operating System's file system encryption (e.g., BitLocker on Windows or LUKS on Linux) to protect the `hospital_mc.db` file from physical theft.

B. Implementation Evidence

1. Password Hashing Code:
The following Python function demonstrates how we secure credentials before interaction with the database.

```
# Code Snippet: SHA-256 Implementation
import hashlib

def hash_password(plain_password):
    """
    Encrypts input using SHA-256 to match the database storage.
    """
    return hashlib.sha256(plain_password.encode()).hexdigest()
```

2. Database Content Verification:
The screenshot below confirms that passwords stored in the `Users` table are opaque hash strings, not readable text.



	user_id	username	password_hash	salt	full_name	email
	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1	1	Alba_MC	ef92b778bafe771e89245b89ecbc08a44a4e…	NULL	Alba Prats	alba@hospital.com
2	2	Alex_MC	ef92b778bafe771e89245b89ecbc08a44a4e…	NULL	Alejandro Serrano	alex@hospital.com
3	3	Tomas_MC	ef92b778bafe771e89245b89ecbc08a44a4e…	NULL	Tomas Uson	tomas@hospital.com

4. Backup and Recovery Policy

A. Defined Policy

To ensure Business Continuity and Availability (the 'A' in CIA), we have defined the following backup strategy:

1. **Backup Type: Online Hot Backup**.
    - *Justification:* Hospitals operate 24/7. We cannot stop the database service to copy the file. We utilize the native SQLite Backup API to generate a consistent snapshot of `hospital_mc.db` while the system remains active.
2. **Frequency & Schedule:**
    - **Full Backups:** Scheduled daily at 02:00 AM (Low traffic period).
    - **Retention:** 30 Days rolling retention policy.
3. **Storage & Encryption:**
    - Backups are saved to a designated `./backups/` directory.
    - *Security:* In a production environment, this directory would be mounted on an encrypted volume (e.g., LUKS) to ensure that stolen backup files are unreadable.

B. Recovery Procedure
In the event of data corruption or ransomware attack:

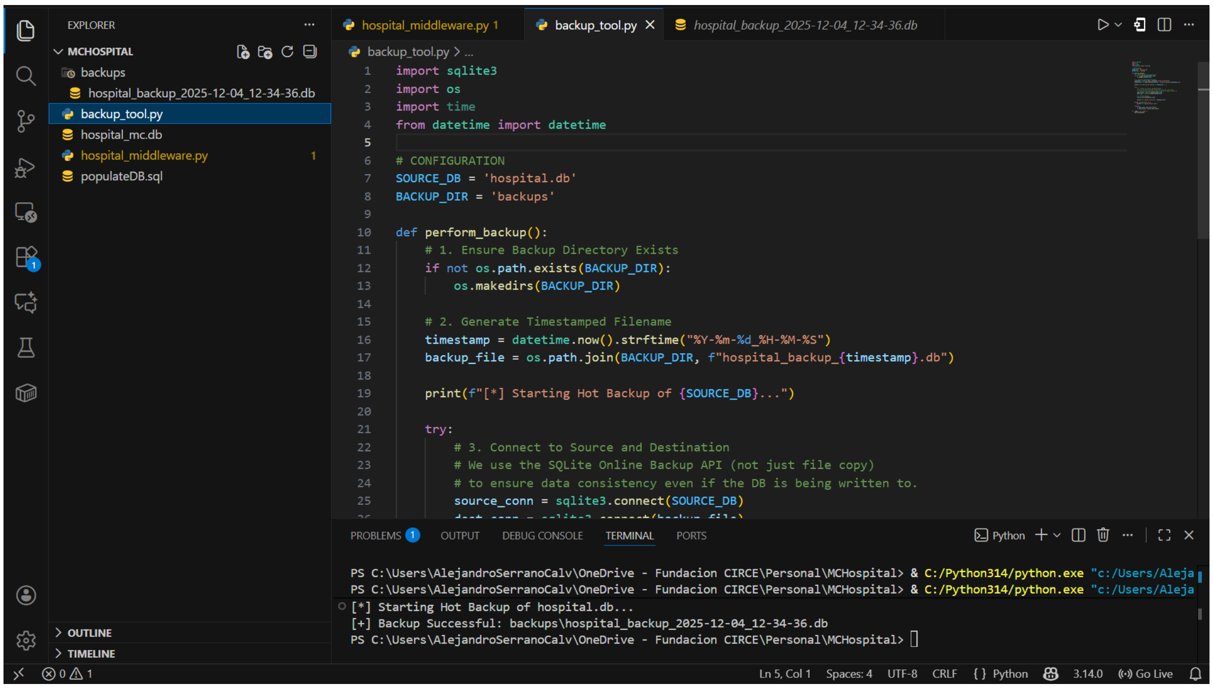
1. Stop the Python Middleware service.
2. Identify the last known good backup (e.g., `backup_2025-01-05.db`).
3. Replace the corrupted `hospital_mc.db` file with the backup copy.
4. Restart the Middleware.

C. Implementation Evidence (The Automation Script)
We developed a custom Python tool (`backup_tool.py`) to automate this process. This script connects to the active database and safely streams the data to a backup file.

backup_tool.py

```
import sqlite3
import os
import time
from datetime import datetime

# CONFIGURATION
SOURCE_DB = 'hospital_mc.db'
BACKUP_DIR = 'backups'

def perform_backup():
    # 1. Ensure Backup Directory Exists
    if not os.path.exists(BACKUP_DIR):
        os.makedirs(BACKUP_DIR)

    # 2. Generate Timestamped Filename
    timestamp = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    backup_file = os.path.join(BACKUP_DIR, f"hospital_backup_{timestamp}.
db")

    print(f"[*] Starting Hot Backup of {SOURCE_DB}...")

    try:
        # 3. Connect to Source and Destination
        # We use the SQLite Online Backup API (not just file copy)
        # to ensure data consistency even if the DB is being written to.
        source_conn = sqlite3.connect(SOURCE_DB)
        dest_conn = sqlite3.connect(backup_file)

        # 4. Perform Backup
        source_conn.backup(dest_conn)

        print(f"[+] Backup Successful: {backup_file}")

    except sqlite3.Error as e:
        print(f"[-] Backup Failed: {e}")

    finally:
        if dest_conn: dest_conn.close()
        if source_conn: source_conn.close()

if __name__ == "__main__":
    perform_backup()
```

Screenshot of one single backup executed without the schedule:



```python
import sqlite3
import os
import time
from datetime import datetime

# CONFIGURATION
SOURCE_DB = 'hospital.db'
BACKUP_DIR = 'backups'

def perform_backup():
    # 1. Ensure Backup Directory Exists
    if not os.path.exists(BACKUP_DIR):
        os.makedirs(BACKUP_DIR)

    # 2. Generate Timestamped Filename
    timestamp = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    backup_file = os.path.join(BACKUP_DIR, f"hospital_backup_{timestamp}.db")

    print(f"[*] Starting Hot Backup of {SOURCE_DB}...")

    try:
        # 3. Connect to Source and Destination
        # We use the SQLite Online Backup API (not just file copy)
        # to ensure data consistency even if the DB is being written to.
        source_conn = sqlite3.connect(SOURCE_DB)
```

```
PS C:\Users\AlejandroSerranoCalv\OneDrive - Fundacion CIRCE\Personal\MCHospital> & C:/Python314/python.exe "c:/Users/Aleja
PS C:\Users\AlejandroSerranoCalv\OneDrive - Fundacion CIRCE\Personal\MCHospital> & C:/Python314/python.exe "c:/Users/Aleja
[*] Starting Hot Backup of hospital.db...
[+] Backup Successful: backups\hospital_backup_2025-12-04_12-34-36.db
PS C:\Users\AlejandroSerranoCalv\OneDrive - Fundacion CIRCE\Personal\MCHospital>
```

It creates a folder if its not already there and inside creates a .db which works as the backup database.

5. Logging and Auditing Policy

A. Defined Policy

To satisfy the **Accountability** requirement of the CIA triad, we implemented a comprehensive auditing strategy. The goal is to reconstruct the "Who, What, Where, and When" of every transaction, specifically focusing on access to sensitive PII.

- **Scope of Auditing:**
    - **Authentication Events:** Successful logins and failed attempts (Brute force detection).
    - **Access Events:** Reads on `Patients` table (distinguishing between `READ_SENSITIVE` for Doctors and `READ_PARTIAL` for Nurses).
    - **Security Violations:** Any `ACCESS_DENIED` event where a user tries to read data outside their clearance level.

B. Implementation Mechanism

Unlike PostgreSQL, which uses the `logging_collector` background process, our SQLite implementation uses **Application-Level Logging**.
The `log_audit()` function in the middleware is invoked immediately after every permission check. It writes an immutable record to the `AuditLogs` table containing:

- `user_id`: The perpetrator.
- `action`: The type of event (e.g., `READ_SENSITIVE`).
- `table_name`: The target asset.
- `timestamp`: The exact time (UTC).
- `details`: Context (e.g., "Viewed Patient ID 1").

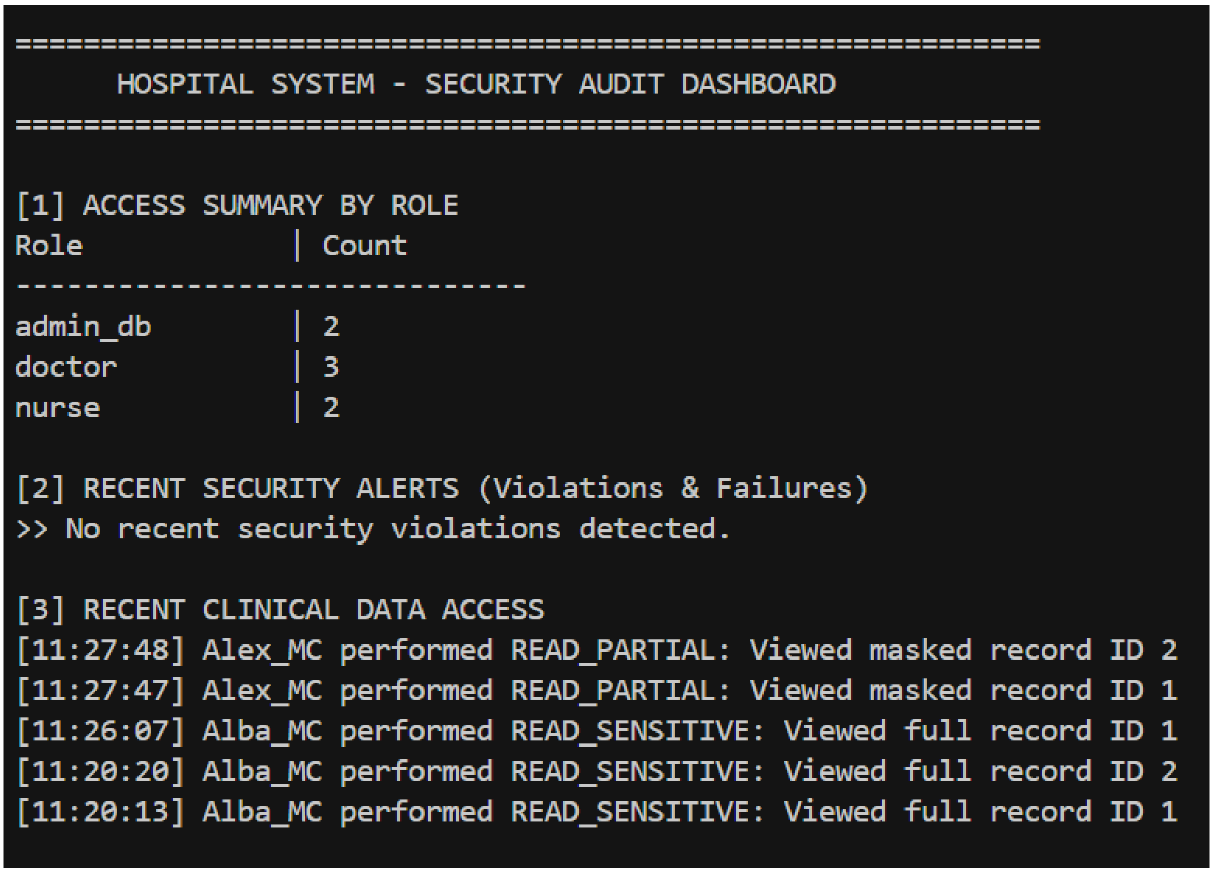
C. Auditing Tools (The Dashboard)

To allow Compliance Officers (ETL_service) and review these logs easily, we developed a Python-based **Audit Dashboard**. This tool aggregates the raw logs into meaningful statistics. IT CAN ONLY BE SEEN BY ETL_SERVICE ROLE.

audit_dashboard.py

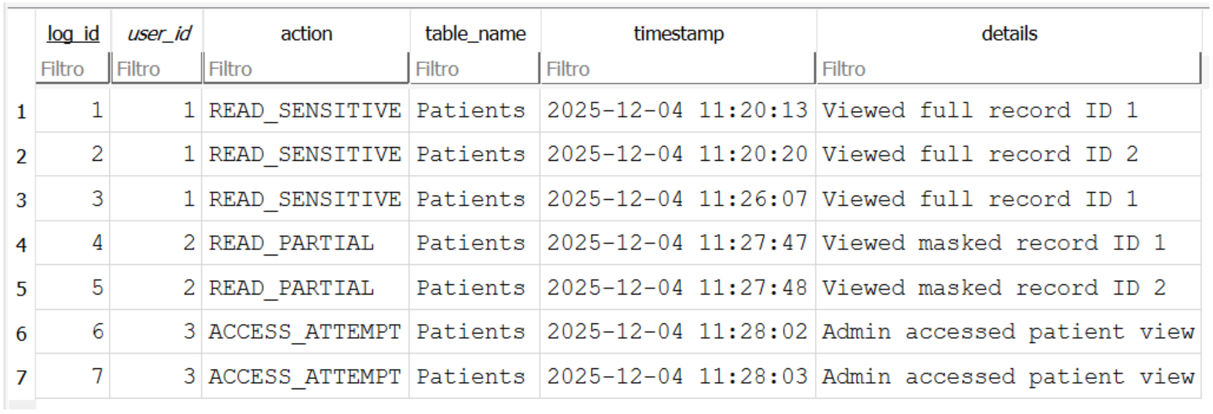
```
import sqlite3

DB_PATH = 'hospital_mc.db'  # Path DB file


def get_db():
    conn = sqlite3.connect(DB_PATH)
    conn.row_factory = sqlite3.Row
    return conn

def authorize_etl(username):
    from hospital_middleware import get_user_credentials
    ctx = get_user_credentials(username)
    return ctx and ctx['role_name'] == 'etl_service'


def run_dashboard():
    conn = get_db()
    cursor = conn.cursor()

    print("\n" + "="*60)
    print("      HOSPITAL SYSTEM - SECURITY AUDIT DASHBOARD      ")
    print("="*60)

    # 1. SUMMARY STATISTICS
    print("\n[1] ACCESS SUMMARY BY ROLE")
    cursor.execute("""
        SELECT r.name as Role, COUNT(l.log_id) as ActionCount
        FROM AuditLogs l
        JOIN Users u ON l.user_id = u.user_id
        JOIN UserRoles ur ON u.user_id = ur.user_id
        JOIN Roles r ON ur.role_id = r.role_id
        GROUP BY r.name
    """)
    rows = cursor.fetchall()

    # Header
    print(f"{'Role':<15} | {'Count':<10}")
    print("-" * 30)
    # Rows
    for r in rows:
        print(f"{r['Role']:<15} | {r['ActionCount']:<10}")

    # 2. SECURITY ALERTS
    print("\n[2] RECENT SECURITY ALERTS (Violations & Failures)")
    cursor.execute("""
        SELECT u.username, l.action, l.details, l.timestamp
        FROM AuditLogs l
        JOIN Users u ON l.user_id = u.user_id
        WHERE l.action IN ('ACCESS_DENIED', 'LOGIN_FAIL')
        ORDER BY l.timestamp DESC
        LIMIT 5
    """)
    rows = cursor.fetchall()

    if not rows:
        print(">> No recent security violations detected.")
    else:
        print(f"{'User':<15} | {'Action':<15} | {'Time':<20} | {'Details'}")
        print("-" * 80)
        for r in rows:
            print(f"{r['username']:<15} | {r['action']:<15} | {r['timestamp'][11:19]:
<20} | {r['details']}")

    # 3. CLINICAL ACCESS LOG
    print("\n[3] RECENT CLINICAL DATA ACCESS")
    cursor.execute("""
        SELECT u.username, l.action, l.details, l.timestamp
        FROM AuditLogs l
        JOIN Users u ON l.user_id = u.user_id
        WHERE l.action LIKE 'READ%'
        ORDER BY l.timestamp DESC
        LIMIT 5
    """)
    rows = cursor.fetchall()
    if not rows:
        print(">> No recent clinical access recorded.")
    else:
        for r in rows:
            # Slicing timestamp [11:19] gives us just the HH:MM:SS time
            print(f"[{r['timestamp'][11:19]}] {r['username']} performed {r['actio
n']}: {r['details']}")

    print("\n" + "="*60)
    conn.close()

if __name__ == "__main__":
    username = input("ETL username: ")
if not authorize_etl(username):
    print("ACCESS DENIED: Only etl_service may run dashboard.")
    exit()

    run_dashboard()
```

Evidence of Implementation:*The screenshot below shows the Auditor's Dashboard displaying a summary of recent access attempts, including a flagged "LOGIN_FAIL" event.*



```
============================================================
      HOSPITAL SYSTEM - SECURITY AUDIT DASHBOARD
============================================================

[1] ACCESS SUMMARY BY ROLE
Role                | Count
------------------------------
admin_db            | 2
doctor              | 3
nurse               | 2

[2] RECENT SECURITY ALERTS (Violations & Failures)
>> No recent security violations detected.

[3] RECENT CLINICAL DATA ACCESS
[11:27:48] Alex_MC performed READ_PARTIAL: Viewed masked record ID 2
[11:27:47] Alex_MC performed READ_PARTIAL: Viewed masked record ID 1
[11:26:07] Alba_MC performed READ_SENSITIVE: Viewed full record ID 1
[11:20:20] Alba_MC performed READ_SENSITIVE: Viewed full record ID 2
[11:20:13] Alba_MC performed READ_SENSITIVE: Viewed full record ID 1
```

We can corroborate this with the actual AuditLogs table in our db using our Databse Workbench   DB Browser:



	log_id	user_id	action	table_name	timestamp	details
	Filtro	Filtro	Filtro	Filtro	Filtro	Filtro
1	1	1	READ_SENSITIVE	Patients	2025-12-04 11:20:13	Viewed full record ID 1
2	2	1	READ_SENSITIVE	Patients	2025-12-04 11:20:20	Viewed full record ID 2
3	3	1	READ_SENSITIVE	Patients	2025-12-04 11:26:07	Viewed full record ID 1
4	4	2	READ_PARTIAL	Patients	2025-12-04 11:27:47	Viewed masked record ID 1
5	5	2	READ_PARTIAL	Patients	2025-12-04 11:27:48	Viewed masked record ID 2
6	6	3	ACCESS_ATTEMPT	Patients	2025-12-04 11:28:02	Admin accessed patient view
7	7	3	ACCESS_ATTEMPT	Patients	2025-12-04 11:28:03	Admin accessed patient view

DISCLAIMER: Compliance auditing is isolated into two independent roles: a human `auditor` role and an automated `etl_service` role. This enforces Separation of Duties by preventing administrators or clinical staff from accessing audit logs while still allowing both manual and automated compliance verification.

Also to prevent audit log tampering, in a real deployment AuditLogs should be stored in a physically separate write once volume or external SIEM server. SQLite cannot enforce append only audit immutability

3.4 Comparative Analysis

In this section, we evaluate our selected DBMS (**SQLite**) against the industry standard (**PostgreSQL**) for a Hospital Information System context. This comparison highlights the trade-offs between the lightweight, serverless architecture of SQLite and the robust, client-server model of PostgreSQL.

1. Encryption

Feature	PostgreSQL (The Standard)	SQLite (Our Implementation)
Data at Rest	Supports **Transparent Data Encryption (TDE)** natively (via extensions) and column-level encryption using `pgcrypto`.	**No native TDE** in the public domain version. We relied on **OS-Level Encryption** (encrypting the drive(Just for theory we didnt actually do it)) and application-level hashing.
Data in Transit	Native support for **SSL/TLS** encryption between client and server.	**Not Applicable.** Since SQLite is serverless, there is no network traffic to encrypt; the application reads the file directly from the disk.
Credential Storage	Native password management (SCRAM-SHA-256).	**Manual Implementation.** We used Python's `hashlib` to implement SHA-256 hashing before insertion.

Verdict: PostgreSQL is superior for data at rest requirements in a hospital, as it allows granular encryption without relying on the OS.

2. Auditing and Logging

Feature	PostgreSQL	SQLite
Mechanism	**Background Process:** Uses the `logging_collector` and robust extensions like `pgAudit` to automatically capture every query.	**Manual Implementation:** SQLite has no background auditing process. We implemented **Application-Level Logging** where the Python middleware manually inserts records into the `AuditLogs` table.

Tamper Resistance	High. Logs can be sent to a separate syslog server.	Low. If an attacker gains write access to the `.db` file, they can delete the logs.

Verdict: PostgreSQL provides an immutable audit trail required by compliance standards (GDPR/HIPAA). Our SQLite solution relies entirely on the integrity of the application code.

3. Access Control (RLS & MAC)

Feature	PostgreSQL	SQLite
RBAC	**Native:** Uses `CREATE ROLE`, `GRANT`, and `REVOKE` commands managed by the database engine.	**Non-Existent.** We simulated RBAC using a `UserRoles` table and Python logic.
Row-Level Security	**Native:** Supports `CREATE POLICY` to filter rows automatically based on the user context.	**Simulated.** We implemented RLS filters inside the Python `request_patient_data` function (e.g., `if role == 'doctor': query += ...`).

Verdict: PostgreSQL is safer because security policies are enforced by the database kernel. In our SQLite implementation, if a user bypasses the Python middleware and opens the file directly, they see everything.

4. Backup and Recovery

Feature	PostgreSQL	SQLite
Backup Method	**Continuous Archiving:** Supports Point-in-Time Recovery (PITR) using Write-Ahead Logging (WAL). Can roll back to a specific second.	**Snapshot:** Uses the Online Backup API to copy the single file. No native PITR; you can only restore to the last full backup.
Complexity	High. Requires configuring `pg_dump` or archiving commands.	**Zero-Config.** Backing up is as simple as copying the `.db` file.

Verdict: For a critical hospital system where every second of data matters, PostgreSQL's Point-in-Time Recovery is essential. SQLite is only sufficient for small clinics with daily backups.

5. Usability and Management

This evaluates the ease of administration, configuration complexity, and the tooling available for maintaining the database system.

A. Installation and Configuration

- **PostgreSQL (High Complexity):**
    - **Architecture:** Requires setting up a dedicated database server (daemon).
    - **Configuration:** Administrators must configure complex files like `postgresql.conf` (for memory tuning) and `pg_hba.conf` (for network authentication).
    - **Network:** Requires managing ports (default 5432), firewalls, and SSL certificates for secure remote connections.
- **SQLite (Zero-Configuration):**
    - **Architecture:** Serverless. The database engine is integrated directly into the Python application.
    - **Configuration:** There is no "installation." Creating a database is as simple as running a script that generates a single `.db` file.
    - **Advantage for this Project:** This allowed our team to share the database via GitHub and run it on any machine (Windows/Linux/Mac) without environment compatibility issues or Docker containers.

B. User Administration

- **PostgreSQL (Native & Robust):**
    - Provides built-in SQL commands for granular user management (`CREATE USER`, `ALTER GROUP`).
    - Supports enterprise authentication methods including LDAP, Kerberos, and SCRAM-SHA-256.
- **SQLite (Manual & Custom):**
    - **Limitation:** Has no concept of "Users" outside of the operating system file permissions.
    - **Impact:** We were forced to build a custom **Identity Management System** (the `Users` table and `admin_tool.py` script) to provision accounts. While functional for a prototype, this is not scalable for a real hospital as it places the security burden entirely on the application developers rather than the DBMS.

C. Monitoring and Tooling

- **PostgreSQL:**
    - Ecosystem includes powerful monitoring tools like **pgAdmin 4**, **PMM (Percona Monitoring and Management)**, and system views (`pg_stat_activity`) that show real-time query performance, lock contention, and active sessions.
- **SQLite:**
    - **Tooling:** relies on simpler tools like **DB Browser for SQLite**.
    - **Monitoring:** Since there is no persistent server process, there is no "real-time dashboard" of connected users. We addressed this limitation by building a custom Python **Audit Dashboard** (`audit_dashboard.py`) to visualize access logs, but it lacks the depth of enterprise monitoring solutions.

D. General Usability Verdict

- **For the Project: SQLite** was superior. Its portability allowed us to focus on implementing security logic (Python middleware) rather than debugging server connection errors.
- **For a Hospital: PostgreSQL** is mandatory. A hospital administrator requires the ability to kill hung sessions, monitor concurrent usage, and manage storage quotas dynamically features that are standard in PostgreSQL but non-existent in SQLite.

6. Additional Aspects: Concurrency & Scalability

The most significant difference lies in **Concurrency**.

- **PostgreSQL** uses Multi-Version Concurrency Control (MVCC), allowing hundreds of doctors to write prescriptions simultaneously without locking the database.
- **SQLite** typically locks the entire database file during a write operation. In a large hospital with 500+ staff, this would cause significant system delays.

Conclusion and Selection

Which DBMS would we choose for a REAL Hospital?
We would choose **PostgreSQL**. The native support for Row-Level Security, network encryption, and concurrent writes makes it the only viable choice for a mission-critical, multi-user healthcare environment.

Justification for using SQLite in this Project:
We selected **SQLite** for this assessment to demonstrate the **Integration** requirement (Part 3.4.6). Its file based nature allowed us to easily build a custom **Python Security Middleware** and integrate it with **Minecraft**, proving that we can architect security principles (CIA, MAC, RLS) programmatically even when the underlying tool does not support them natively.

ERRORS WHILE WORKING ON IT

SQLite Check Constraints

During schema implementation, several limitations and behavioral differences between **PostgreSQL** and **SQLite** were encountered, especially regarding **CHECK constraints and pattern validation**.

In PostgreSQL, pattern-based validation can be implemented using regular expressions with operators such as `SIMILAR TO`, `~`, or `LIKE` with advanced pattern expressions. This allows complex validation such as:

```
CHECK (ssn SIMILARTO'[0-9]{3}-[0-9]{2}-[0-9]{4}')
```

However, **SQLite does not support full regular expression engines inside CHECK constraints.**

It only supports a simplified pattern system using the `GLOB` operator, which lacks repetition counters (`{3}`, `{4}`), alternation, and grouping.

Because of this limitation, expressions such as:

```
CHECK (ssn GLOB'[0-9]{3}-[0-9]{2}-[0-9]{4}')
```

are **invalid in SQLite** and fail during table creation.

To overcome this limitation, we had to redesign all format constraints using **explicit positional matching**, resulting in more verbose but SQLite-compatible expressions such as:

```
CHECK (ssn GLOB'[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]')
```

The same workaround was required for phone numbers and other formatted identifiers.

This demonstrated a significant **loss of expressive power in SQLite CHECK constraints compared to PostgreSQL**.

Removal of Python Support in Modern Minecraft Versions

Another major technical obstacle was encountered when integrating the Hospital Information System with **Minecraft**.

Earlier Minecraft versions (≤ **1.12.1**) provided native Python compatibility through the **Raspberry Juice API** and the `mcpi` Python library. This allowed Python scripts to directly communicate with the Minecraft game engine.

However, **Mojang removed official Python support after version 1.12.1**, replacing the internal networking model and breaking compatibility with the Raspberry Juice protocol.

As a result:

- Modern Minecraft versions (1.13+) cannot be controlled directly using Python
- The `mcpi` library only works reliably with legacy versions
- Native Python automation is no longer supported by Mojang

To maintain integration functionality, the project was intentionally developed using a **legacy-compatible Minecraft version (≤1.12.1)** combined with the **Raspberry Juice plugin**, enabling Python based real time security enforcement and physical access simulation.

This constraint significantly influenced architectural decisions and limited the use of modern Minecraft features, but was necessary to preserve Python middleware integration.

SQLite Write-Lock & Concurrency Limitation

During multi user testing of the Hospital Information System, we encountered a critical architectural limitation related to **SQLite's locking and concurrency model**.

Unlike PostgreSQL, which uses **Multi Version Concurrency Control (MVCC)** to allow hundreds of concurrent read/write transactions without blocking, SQLite uses a **database level write lock**. This means:

- Multiple users may read simultaneously

- **Only one write transaction can occur at any given time**
- When a write is active, *all other writes are blocked*

In a hospital context, this caused real operational issues. For example:

- While a doctor was creating a prescription

- A nurse attempting to update vitals was blocked
- A lab technician saving test results was delayed

These delays caused visible pauses in the Minecraft hospital interface and demonstrated that SQLite **cannot safely scale to real multi-ward hospital workloads**.

This issue does not appear in PostgreSQL, where MVCC allows concurrent writes without global locking.

To mitigate this limitation during testing, all write operations were serialized by the Python middleware, ensuring consistency but sacrificing performance. This reinforced the conclusion that while SQLite is suitable for educational and small-clinic scenarios, it is not safe for real enterprise hospital deployment.