# Assignment 6 6/3/2025

Pedro Ramirez

Implement the last release of your term project. You will incorporate a database manipulated from your application. Note the second evaluation criterion—your application of database should fit your project well.

The same instructions as for Assignment 3 apply, including leveraging AI, showing your value added in red font and by means of explanations, inserting comments (in red) in figures that describe clearly your value added, commenting on what you consider significant prompt contributions, and submitting.

# 1 SUMMARY DESCRIPTION

One- or two-paragraph overall description of your proposed term project. Color red the parts changed from Assignment 5.

MEDTRACK is a Java-based program that helps clinics manage tasks like signing up patients, booking appointments, and keeping records. It follows object-oriented design, uses files to save data, checks user input, and supports multiple users at the same time. It also includes strong error handling, a simple interface (facade), and Java streams to handle data easily.

🟥 **In this final version, MEDTRACK no longer saves appointments in a basic file—it now uses an SQLite database to store them permanently. This means the data stays safe even after the program is closed and reopened.**

🟥 **I added a lock to make sure appointments are booked safely when many users use the system at once, and we added a background worker (AppointmentSaverWorker) to save appointments without slowing down the program.**

🟥 **At the beginning, the user is asked if they want to load or reset the database, and when closing the app, the system saves all remaining data before it stops.**

🟥 **These updates make the program faster, more reliable, and ready for future upgrades like a web or desktop interface.**

# 2 KEY REQUIREMENTS (FEATURES) IMPLEMENTED IN THIS RELEASE

Title and one or two sentences per requirement. Repeat requirements implemented for prior assignments if they are necessary to provide context. Make it clear which requirements are new vs. old. Your application should prompt the user at runtime, not pre-populate the database.

#### 

#### 2.1 SQLite Database Integration (NEW)

Appointments are now saved in an SQLite database instead of a text file. The user is asked at startup whether they want to reset or load the database.

#### 2.2 Safe Multi-User Appointment Booking (modified)

A lock (ReentrantLock) is used to make sure appointments are saved safely, even when multiple users are booking at the same time.

#### 2.3 Background Appointment Saving (NEW)

A background worker (AppointmentSaverWorker) saves appointments to the database while the program keeps running. When the app closes, it finishes saving all data before shutting down.

# 3 I/O SUPPORTING THE NEW REQUIREMENTS LISTED ABOVE

Provide an example of input / output showing the new features of your application. Include validation of stored data.

#### Input File(s)

**File:** data/doctors.csv  
Used to load doctor records into the system at startup.

**File:** data/patients.csv  
Contains patient records that are also loaded at startup (when reinitializing the database).

**File:** data/appointments.txt  
Contains past appointment data imported into the database for continuity.

#### Console I/O:

**At Startup:**

🗃️ Reinitialize the database from scratch? (yes/no): yes

✅ Tables ensured.

✅ Patients seeded from CSV.

✅ Doctors seeded from CSV.

✅ Appointments seeded from TXT.

🟢 Appointment saver started...

✅ Loaded patients from SQLite

✅ Loaded doctors from SQLite

💡 Autosave is enabled. Your data will be saved automatically in the background.

**Main Menu and Registration:**

==== MEDTRACK MENU ====

0. Clean db and reload from CSV files (WARNING: this will delete all existing data!)

1. Register as new patient

2. Book an appointment

3. View my appointments

4. Exit

Choose an option: 1

Enter your name: Pedro Ramirez

Enter your insurance provider: MedCare

✅ Patient saved to database.

✅ Registered successfully! Your patient ID: P0862

**Appointment Booking with Error Handling and Success:**

Choose an option: 2

Enter your patient ID: D8000

❌ Invalid patient ID.

Choose an option: 2

Enter your patient ID: P0862

Available Doctors:

- ID: D8000 | Dr. Lee (Pediatrics)

- ID: D8001 | Dr. Sarah Brown (Cardiology)

- ID: D8002 | Dr. Thomas Wayne (Pediatrics)

- ID: D8003 | Dr. Jor-El (Orthopedics)

- ID: D8004 | Dr. Sophia Park (Dermatology)

- ID: D8005 | Dr. Lee (Pediatrics)

Enter doctor ID to book with: D8000

Enter date (YYYY-MM-DD): 2025-12-12

Enter time (HH:MM): 12:30

✅ Appointment booked! Confirmation code: APT-P0862-D8000-20251212-1230

**Asynchronous Save Confirmation:**

💾 Saved to SQLite: APT-P0862-D8000-20251212-1230

#### Output File(s)

**File:** medtrack.db (SQLite database)

To confirm that the appointment was saved, the following SQL command can be run:

SELECT \* FROM appointments WHERE id = 'APT-P0862-D8000-20251212-1230';

Expected result:

| id | patient\_id | doctor\_id | date | time |

|--------------------------------|------------|-----------|------------|-------|

| APT-P0862-D8000-20251212-1230 | P0862 | D8000 | 2025-12-12 | 12:30 |

This verifies:

* The appointment was stored successfully.
* It was saved by the background worker without interrupting user flow.
* Validation worked as expected (invalid patient ID was rejected

# 4 YOUR DIRECTORY

Show a screenshot of your directory. Include all relevant files. This should include JUnit tests.

# A screenshot of a computer AI-generated content may be incorrect.

# 5 DESIGN

### 5.1 Class Model, Use Case, and Sequence Diagram

## Supply a main use case, the class model, and the sequence diagram corresponding to the use case. These should be consistent. Indicate in red where you applied the features listed below.

### ✅ Use Case: Book an Appointment

**Actor:** Patient  
**Precondition:** Patient must be registered in the system  
**Main Flow:**

1. The patient chooses to book an appointment.
2. The system prompts for patient ID.
3. The system displays available doctors.
4. The patient selects a doctor and provides a date/time.
5. The system checks for conflicts.
6. If available, the system confirms the appointment.
7. The appointment is queued for background saving.

🟥 New functionality appears in steps 5–7 (thread-safe check, background save).

### ✅ Sequence Diagram: Book an Appointment

Patient → Main

Main → FacadeService

FacadeService → UserRegistry: findPatientById()

FacadeService → AppointmentManager: bookAppointment(patient, doctor, date, time)

AppointmentManager → lock.lock()

AppointmentManager → AppointmentSaverWorker: saveLater()

AppointmentSaverWorker → DatabaseManager: getConnection()

AppointmentSaverWorker → SQLite DB: INSERT INTO appointments

AppointmentManager → lock.unlock()

FacadeService → Main: return confirmation

🟥 New methods: *saveLater()*, *getConnection()*, *flushAndStop()*

* AppointmentManager: bookAppointment(), loadAppointmentsFromDatabase(), shutdown()
* AppointmentSaverWorker: saveLater(), flushAndStop()
* UserRegistry: loadUsersFromDatabase(), loadPatientsFromDatabase(), loadDoctorsFromDatabase()
* DatabaseManager: getConnection()
* DatabaseInitializer: initialize(), seedAppointmentsFromTxt()
* FacadeService: now handles DB-based logic and startup loading

### ✅ Highlighted Changes (marked in red in diagram)

📌 [Diagram is referenced here](https://docs.google.com/spreadsheets/d/1Srxutu-G_5bswR1xzbCi4zFVPmfwhTKby1di8F7EdTw/edit?gid=0#gid=0)

A screenshot of a document

AI-generated content may be incorrect.

# 6 CODE SNIPPETS AND EXPLANATIONS

### 6.1 Code of any kind showing table creation

Code is located in the DatabaseInitializer.java class:

// Drop tables for a clean slate

stmt.executeUpdate("DROP TABLE IF EXISTS appointments");

stmt.executeUpdate("DROP TABLE IF EXISTS doctors");

stmt.executeUpdate("DROP TABLE IF EXISTS patients");

// Create tables

stmt.executeUpdate("""

CREATE TABLE patients (

id TEXT PRIMARY KEY,

name TEXT NOT NULL,

insurance TEXT

)

""");

stmt.executeUpdate("""

CREATE TABLE doctors (

id TEXT PRIMARY KEY,

name TEXT NOT NULL,

specialty TEXT

)

""");

stmt.executeUpdate("""

CREATE TABLE appointments (

id TEXT PRIMARY KEY,

patient\_id TEXT NOT NULL,

doctor\_id TEXT NOT NULL,

date TEXT NOT NULL,

time TEXT NOT NULL,

FOREIGN KEY (patient\_id) REFERENCES patients(id),

FOREIGN KEY (doctor\_id) REFERENCES doctors(id)

)

""");

This code ensures that the database is reinitialized each time the user chooses to reset it, and that all three core entities—patients, doctors, and appointments—have their own structured tables.

### 6.2 Explanation of 6.1: why the tables are appropriate for this application

The tables created in the MEDTRACK system directly reflect the core entities and relationships in a real-world medical clinic:

* **patients table:**  
  Stores each patient's unique ID, name, and insurance provider. This allows the system to register new patients and retrieve their data quickly for appointment booking or lookup.
* **doctors table:**  
  Holds the doctor’s ID, name, and area of specialty. This enables the application to list available doctors and filter by specialization if needed.
* **appointments table:**  
  Tracks scheduled appointments using a unique ID along with references to a patient and a doctor, plus the appointment date and time. The use of FOREIGN KEY constraints enforces real-world rules—appointments must be tied to registered patients and doctors—ensuring data integrity.

These tables form a normalized schema:

* Each table handles a distinct entity.
* Relationships (e.g., a patient booking with a doctor) are managed using foreign keys.
* The structure supports validation, efficient lookups, and prevents data duplication.

In short, this schema models the MEDTRACK application's requirements accurately while enabling future scalability, such as filtering by doctor, patient history, or appointment analytics.

### 6.3 Java code involving the insertion of data

Below is the code used to insert new patient and doctor records into the SQLite DB. This logic resides in the FacadeService.java class and is called during user registration:

#### 🔹 Inserting a Patient:

private void insertPatientToDatabase(Patient patient) {

String sql = "INSERT OR IGNORE INTO patients (id, name, insurance) VALUES (?, ?, ?)";

try (Connection conn = DatabaseManager.getConnection();

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setString(1, patient.getId());

pstmt.setString(2, patient.getName());

pstmt.setString(3, patient.getInsuranceProvider());

pstmt.executeUpdate();

System.out.println("✅ Patient saved to database.");

} catch (SQLException e) {

System.err.println("❌ Failed to save patient to DB: " + e.getMessage());

}

}

#### 🔹 Inserting a Doctor:

private void insertDoctorToDatabase(Doctor doctor) {

String sql = "INSERT OR IGNORE INTO doctors (id, name, specialty) VALUES (?, ?, ?)";

try (Connection conn = DatabaseManager.getConnection();

PreparedStatement pstmt = conn.prepareStatement(sql)) {

pstmt.setString(1, doctor.getId());

pstmt.setString(2, doctor.getName());

pstmt.setString(3, doctor.getSpecialty());

pstmt.executeUpdate();

System.out.println("✅ Doctor saved to database.");

} catch (SQLException e) {

System.err.println("❌ Failed to save doctor to DB: " + e.getMessage());

}

}

These methods are triggered from the registerUser(User user) method during runtime and are skipped during test execution when test mode is enabled.

### 6.4 Explanation of 6.3: how the insertion logic ensures data integrity for this application

The insertion logic in FacadeService.java ensures data integrity in the following ways:

1. **Use of INSERT OR IGNORE:**  
   This SQL clause prevents duplicate entries by ignoring insert attempts if a record with the same primary key (id) already exists. This ensures that each patient and doctor has a unique identifier and prevents accidental overwrites or duplicates.
2. **Parameterized Queries with PreparedStatement:**  
   By using ? placeholders and setString() methods, the code avoids SQL injection vulnerabilities and ensures type safety when writing user-provided input to the database.
3. **Transaction Safety via Try-With-Resources:**  
   The connection and statement objects are managed with Java’s try-with-resources, ensuring that database resources are properly closed even if an error occurs. This prevents data corruption and resource leaks.
4. **Validation through Object-Oriented Input:**  
   The data inserted comes from strongly-typed Patient and Doctor objects. Since these are built from validated console inputs or CSV data, it ensures that no malformed records reach the database layer.
5. **Clear Logging and Error Handling:**  
   Each insert operation prints a confirmation or logs an error. This feedback loop helps developers and testers quickly detect and address data-related issues during development or testing.

In short, the logic ensures that:

* No duplicate users are inserted.
* Only valid and complete data is written.
* Errors are caught early.
* The system maintains consistent, reliable records.

### 6.5 Java code involving the selection of records, and ordering with SQL

Although the current MEDTRACK application does not include a built-in SQL query using ORDER BY, this feature would be useful for displaying appointments chronologically. Below is a small addition that could be implemented to retrieve appointments ordered by date and time for better readability:

// Example: Select all appointments ordered by date and time

String sql = """

SELECT id, patient\_id, doctor\_id, date, time

FROM appointments

ORDER BY date ASC, time ASC

""";

try (Connection conn = DatabaseManager.getConnection();

PreparedStatement stmt = conn.prepareStatement(sql);

ResultSet rs = stmt.executeQuery()) {

while (rs.next()) {

System.out.printf("Appointment: %s on %s at %s\n",

rs.getString("id"), rs.getString("date"), rs.getString("time"));

}

} catch (SQLException e) {

System.err.println("Error fetching ordered appointments: " + e.getMessage());

}

This query would allow the application to present upcoming appointments in a logical, time-based order.

### 6.6 Explanation of 6.5: why the correct, ordered result is useful for this application

Displaying appointments in chronological order is essential for usability and workflow in a real-world medical system like MEDTRACK. Here's why:

1. **Improves User Experience:**  
   When appointments are shown in date and time order, users (patients or admins) can easily see what's coming up next, rather than having to search through unordered records.
2. **Reduces Human Error:**  
   A sorted list helps avoid confusion or missed appointments, especially when reviewing schedules or verifying bookings.
3. **Supports Efficient Scheduling:**  
   By seeing upcoming time slots in order, clinic staff or patients can identify gaps or conflicts more easily when making new bookings.
4. **Reflects Real-World Practice:**  
   Clinics and hospitals typically organize their daily or weekly schedule by time, and this ordering mimics that natural workflow.

Although not yet implemented in the codebase, adding this type of query would significantly enhance the readability and reliability of the appointment list.

### 6.7 Java code involving selection involving at least two tables

The current MEDTRACK project does not include any SQL queries that join two or more tables. However, I can provide a realistic example that would be helpful in the system.  
I could join the appointments, patients, and doctors tables to display appointment details with patient and doctor names:

String sql = """

SELECT

a.id AS appointment\_id,

p.name AS patient\_name,

d.name AS doctor\_name,

a.date,

a.time

FROM appointments a

JOIN patients p ON a.patient\_id = p.id

JOIN doctors d ON a.doctor\_id = d.id

ORDER BY a.date, a.time

""";

try (Connection conn = DatabaseManager.getConnection();

PreparedStatement stmt = conn.prepareStatement(sql);

ResultSet rs = stmt.executeQuery()) {

while (rs.next()) {

String id = rs.getString("appointment\_id");

String patient = rs.getString("patient\_name");

String doctor = rs.getString("doctor\_name");

String date = rs.getString("date");

String time = rs.getString("time");

System.out.printf("🗓️ %s | %s with %s at %s\n", date, patient, doctor, time);

}

} catch (SQLException e) {

System.err.println("❌ Error fetching appointment details: " + e.getMessage());

}

This code would show a more complete appointment view and make the output user-friendly, especially for front-desk users or admin dashboards.

### 6.8 Explanation of 6.7: how the query uses real relationships to support this application

The query in Section 6.7 reflects real relationships within the MEDTRACK application and supports its practical goals:

1. **Represents Real-World Links Between Entities:**  
   Each appointment is connected to one patient and one doctor. The query joins the appointments table with the patients and doctors tables using patient\_id and doctor\_id foreign keys. This mirrors how appointments work in a real clinic.
2. **Delivers a Complete View:**  
   By combining data across tables, the result shows not just appointment times, but also who the appointment is for (patient name) and who it's with (doctor name). This is much more useful than viewing IDs alone.
3. **Improves Readability for Users:**  
   Users see clear, human-readable output—e.g., "Pedro Ramirez with Dr. Sophia Park"—rather than raw IDs like "P0862" or "D8000". This is especially helpful for front-desk staff or anyone checking the schedule.
4. **Demonstrates Proper Use of Foreign Keys:**  
   The query respects the database design, using join conditions that depend on defined foreign key relationships. This ensures that the output is accurate and consistent with the stored data.

In short, this multi-table query enables MEDTRACK to present rich, useful information to users, fully aligned with how appointments are managed in real healthcare settings.

### 6.9 Java code involving selection and aggregation of data

Although the MEDTRACK application does not currently include aggregation queries, here’s a relevant and realistic example that I can implement to count how many appointments each doctor has:

String sql = """

SELECT d.name AS doctor\_name, COUNT(a.id) AS total\_appointments

FROM doctors d

LEFT JOIN appointments a ON d.id = a.doctor\_id

GROUP BY d.name

ORDER BY total\_appointments DESC

""";

try (Connection conn = DatabaseManager.getConnection();

PreparedStatement stmt = conn.prepareStatement(sql);

ResultSet rs = stmt.executeQuery()) {

System.out.println("📊 Appointments per Doctor:");

while (rs.next()) {

String doctor = rs.getString("doctor\_name");

int total = rs.getInt("total\_appointments");

System.out.printf("- %s: %d appointments\n", doctor, total);

}

} catch (SQLException e) {

System.err.println("❌ Error fetching appointment counts: " + e.getMessage());

}

This query performs:

* A **JOIN** between doctors and appointments
* A **COUNT()** to tally the number of appointments per doctor
* A **GROUP BY** to aggregate the results
* An optional **ORDER BY** to sort from most to fewest appointments

### 6.10 Explanation of 6.9: how the aggregation delivers relevant insights for this application

The aggregation shown in Section 6.9 provides useful insights by counting how many appointments each doctor has. This supports the MEDTRACK application in several important ways:

1. **Workload Monitoring:**  
   By displaying how many appointments each doctor has, the system can help clinic managers identify who is overbooked or underbooked. This can inform scheduling decisions and help balance workloads fairly.
2. **Operational Planning:**  
   Knowing appointment volume by doctor supports resource planning—for example, assigning support staff or reserving rooms based on which doctors have heavier schedules.
3. **Decision Support:**  
   The data can highlight patterns, such as consistently low or high patient volumes per doctor, which may suggest a need for marketing, training, or reallocation of duties.
4. **Performance Tracking:**  
   Over time, trends in appointment counts can help evaluate how doctors are performing in terms of patient engagement or service demand.
5. **User-Friendly Reporting:**  
   Aggregated summaries are easier to understand than raw data. Showing "Dr. Lee has 12 appointments this week" is far more actionable than listing 12 individual rows.

Overall, this type of aggregation transforms raw data into meaningful, high-level information that improves scheduling efficiency and supports better decision-making in the clinic.

# 7 Evaluation

**Release:**  Enhanced with SQLite Persistence

### ✅ Modified Components

#### Main.java

**Location:** src/app/Main.java  
**Key Updates:**

* Prompts user to reinitialize the database at launch.
* Loads patients, doctors, and appointments from SQLite.
* Starts the background autosave service.
* Adds “Option 0” to reseed data from CSV/TXT files for quick resets.

#### FacadeService.java

**Location:** src/service/FacadeService.java  
**Key Updates:**

* Saves new users directly into the database upon registration.
* Loads users from the database at startup.
* Adds enableTestMode() for controlled testing scenarios.
* Delegates appointment handling to AppointmentManager.

#### UserRegistry.java

**Location:** src/service/UserRegistry.java  
**Key Updates:**

* Adds loadPatientsFromDatabase() and loadDoctorsFromDatabase() for DB integration.
* Centralizes user restoration via loadUsersFromDatabase().

#### AppointmentManager.java

**Location:** src/service/AppointmentManager.java  
**Key Updates:**

* Uses ReentrantLock to ensure thread-safe appointment booking.
* Sends new appointments to AppointmentSaverWorker for async saving.
* Fetches appointments per patient from the database.
* Implements shutdown logic to flush pending saves gracefully.

#### AppointmentSaverWorker.java

**Location:** src/service/AppointmentSaverWorker.java  
**Purpose:**

* Runs in the background to save appointments asynchronously.
* Uses a blocking queue and poison pill to handle clean shutdown.
* Writes appointments directly to SQLite without blocking the UI.

#### DatabaseInitializer.java

**Location:** src/service/DatabaseInitializer.java  
**🆕 New Class**  
**Responsibilities:**

* Drops and recreates key tables (patients, doctors, appointments).
* Seeds data from patients.csv and doctors.csv.
* Imports legacy appointments from appointments.txt.

#### DatabaseManager.java

**Location:** src/service/DatabaseManager.java  
**🆕 New Class**  
**Purpose:**

* Provides SQLite connections with thread safety in mind.
* Handles clean connection management and utility access for all DB tasks.

### 📁 External Data Files (used for seeding)

* data/patients.csv – Initial patient records
* data/doctors.csv – Initial doctor records
* data/appointments.txt – Legacy appointment entries

### ✅ Summary of Enhancements

* **SQLite Integration:** Persistent storage for all patients, doctors, and appointments.
* **Thread-Safe Booking:** Prevents race conditions during concurrent access.
* **Background Persistence:** Appointments are saved asynchronously via a worker thread.
* **Clean Startup/Shutdown:** Ensures all data is loaded at launch and flushed at exit.
* **Legacy Support:** Maintains CSV/TXT compatibility for seeding and testing.

### 🧪 Test Coverage

#### AppointmentDatabaseTest.java

**Location:** src/test/model/AppointmentDatabaseTest.java  
**Purpose:** Verifies that the SQLite database is properly seeded and stores valid appointment data.

**Test Cases:**

* ✅ testAppointmentExistsInDatabase()  
  Confirms that at least one appointment exists after initialization. Ensures appointment seeding is working.
* ✅ testAppointmentDateAndTimeAreFormattedCorrectly()  
  Validates that all stored appointment records use correct date (YYYY-MM-DD) and time (HH:MM) formats.

These tests guarantee that:

* The database connection works properly.
* Seeding and format constraints are respected.
* The application’s move to SQLite did not break appointment persistence logic.



Here’s a polished **Prompt Contributions** section you can include in your assignment to clearly

## 🤖 Prompt Contributions (AI Support Summary)

This project was completed with the assistance of AI to enhance clarity, structure, and implementation strategy. Below are the key contributions provided by AI prompts:

### 🧩 Initial Planning Prompt

**Student Prompt:**

“Hi! I'm working on the final release of my CS622 term project. It's a medical appointment system called MEDTRACK. I need to upgrade it from file-based storage to SQLite, support safe multi-user booking, and add background saving. Can you help me break this down and implement it?”

**AI Contribution:**

* Suggested a clear implementation roadmap:
  + Create DatabaseManager and DatabaseInitializer classes
  + Use ReentrantLock for thread-safe booking
  + Add AppointmentSaverWorker with a queue for async saving
  + Prompt the user to reinitialize or load the DB in Main.java

### 🧪 Testing Guidance

**Prompt:**

"Can you help me write a JUnit test that verifies appointments exist in the database and are properly formatted?"

**AI Response:**

* Created AppointmentDatabaseTest with two key methods:
  + testAppointmentExistsInDatabase()
  + testAppointmentDateAndTimeAreFormattedCorrectly()

### 📊 SQL Query Support

**Prompt:**

“What’s an example of a query that joins tables or aggregates appointment data?”

**AI Response:**

* Provided JOIN queries to display patient and doctor names with appointments
* Wrote an aggregation query to count appointments per doctor
* Explained the practical benefits of each query in a clinic setting

### 📋 Documentation and Evaluation Help

**Prompt:**

“Can you help me write sections 6.1 through 6.10 for my assignment using my code?”

**AI Response:**

* Broke down each subsection with accurate code snippets and clear explanations
* Filled in missing queries based on project structure
* Reviewed the entire assignment and suggested wording, formatting, and tone improvements