

**FACULTY OF INFORMATION TECHNLOGY AND COMMUNICATIONS**

**BITP - DISTRIBUTED APPLICATION**

**2 BITS (S1G1)**

**GROUP PROJECT**

**DR. HARIZ**

|  |  |
| --- | --- |
| **STUDENT NAME** | **MATRIKS NO** |
| JONATHAN MICHAEL LEONG EUGENE | B032310513 |
| WONG CHOON KIAT | B032310534 |
| KESSIGAN A/L THIRUNAVUKKARASU | B032310876 |
| ISAAC RYAN KOIRIN | B032310337 |
| AMIRUL HAFIZ BIN ANUAR | B032310657 |
| MUHAMMAD ROHAIZAD BIN ROSLI | B032310485 |

**INTRODUCTION**

The Clinic Management System (CMS) is developed to address the operational challenges faced by small to medium-sized clinics. Traditionally, clinics rely on manual processes for patient registration, appointment booking, medical recordkeeping, and billing. This often results in inefficiencies, human errors, and increased workload on clinic staff.

The CMS automates and streamlines these processes by providing a centralized digital platform. It improves data accuracy, reduces paperwork, enhances patient service, and allows easier data access for authorized personnel. The system is designed with scalability and user-friendliness in mind, ensuring it can adapt to the growing needs of healthcare facilities.

The Clinic Management System (CMS) is an end-to-end digital healthcare platform designed to modernize medical practice operations. Built with Java Swing for the frontend and a RESTful Java backend with MySQL/JSON data persistence, this system solves critical inefficiencies in traditional clinic workflows.

**PROBLEM STATEMENTS**

1. **Manual Appointment Scheduling**

Traditional paper-based appointment booking systems create numerous operational challenges for medical clinics. Using physical appointment books or basic computer spreadsheets often leads to scheduling conflicts when multiple staff members try to book patients simultaneously. Without a centralized system, double bookings frequently occur where two patients are accidentally scheduled for the same time slot. This creates frustration for both patients and staff when appointment times need to be adjusted at the last minute.

Paper systems also make it difficult to track appointment attendance. Clinics have no effective way to follow up with patients who miss their appointments, as there's no automated reminder system. Staff must manually call each patient beforehand, which consumes considerable administrative time. Rescheduling appointments becomes a tedious process of erasing and rewriting entries in the appointment book, often leading to messy, hard-to-read schedules.

1. **Fragmented Patient Records**

Physical paper records create significant inefficiencies in clinical workflows. When a patient arrives for an appointment, staff must locate their paper chart from storage, which may be filed among thousands of other records. During busy periods, this chart retrieval process can cause delays in seeing patients. Important health information is often scattered across different forms, sticky notes, and test result attachments within the paper file, making it difficult for providers to quickly find critical information.

Paper records are vulnerable to damage from spills, tears, or normal wear-and-tear. They also present security risks, as paper files can be lost, stolen, or viewed by unauthorized personnel. When patients see multiple providers within a practice, their information may be recorded differently in separate charts, leading to inconsistent medical histories.

1. **Inefficient Clinic Workflows**

Traditional clinic operations require excessive manual coordination between different staff members and departments. Front desk personnel spend considerable time physically routing paper charts to the appropriate providers and treatment rooms. Nurses and medical assistants waste time searching for available equipment or preparing paperwork between patient visits.

The prescription process is particularly cumbersome, requiring providers to hand-write medication orders that front desk staff must then call or fax to pharmacies. Generating reports for practice analysis or patient referrals involves manually compiling data from multiple sources, often resulting in incomplete or inconsistent information.

**OBJECTIVES**

1. **To develop a real-time digital scheduling system with automated conflict detection and multi-channel patient notifications.** If a patient tries to book a slot that's already taken, the system immediately suggests the next available time. Once confirmed, the system sends automated reminders through SMS, email, or app notifications.
2. **To establish secure electronic health records with encrypted data storage and role-based access controls.** A doctor can view full medical histories and see appointment-related information.
3. **To create structured clinical documentation tools with temporal visualization of treatment histories.** A visual timeline showing all the patient’s visits, treatments administered, and lab results over months, aiding better clinical decision-making.
4. **To create a secure and safe to use medical platform.** All users will need proper authentication to access the application.

**COMMERCIAL VALUE & THIRD-PARTY INTEGRATION**

**1. Market Potential**

Target :

Private Clinics: Small-to-medium practices seeking affordable digital transformation.

Multi-Specialty Hospitals: Large facilities needing centralized patient management.

Telehealth Providers: Platforms requiring appointment scheduling and EHR integration.

**2. Third-party integration**

**1) Database System: MySQL (Relational Database)**

**Justification:**

ACID Compliance: MySQL ensures Atomicity, Consistency, Isolation, and Durability, which are essential when dealing with medical data (e.g., patient records, appointments, prescriptions) to prevent data corruption or loss.

**Implementation:**

**DBConnection.java**

This Java class is responsible for establishing a connection to the MySQL database.

It likely includes methods for:

* **Connecting to the DB using JDBC**
* **Executing SQL queries ( SELECT, INSERT, UPDATE, DELETE)**
* **Managing prepared statements and result sets**
* **Ensures reusability and separation of concerns by centralizing DB access logic in one place**

**2) JSON Data Sync: org.json Library**

**Justification:**

JSON (JavaScript Object Notation) is a lightweight format that is easy to parse and generate.

It is useful for offline storage, data exchange, or syncing settings/configurations.

Great for mobile apps or browser-based apps that need to sync with server data.

**Implementation:**

**JSONHandler.java**

* **Manages reading from and writing to JSON files.**
* **Handles two-way synchronization, such as:**
* **Exporting DB data into a JSON file (backup or sync to client)**
* **Importing JSON data into the database (restore or sync from client)**
* **Useful in distributed systems, remote backups, or interoperability with non-Java systems.**

**3) HTTP Server: com.sun.net.httpserver**

**Justification:**

A native Java HTTP server (no need for external web server like Apache or Tomcat).

Ideal for lightweight RESTful APIs, internal tools, or simple services.

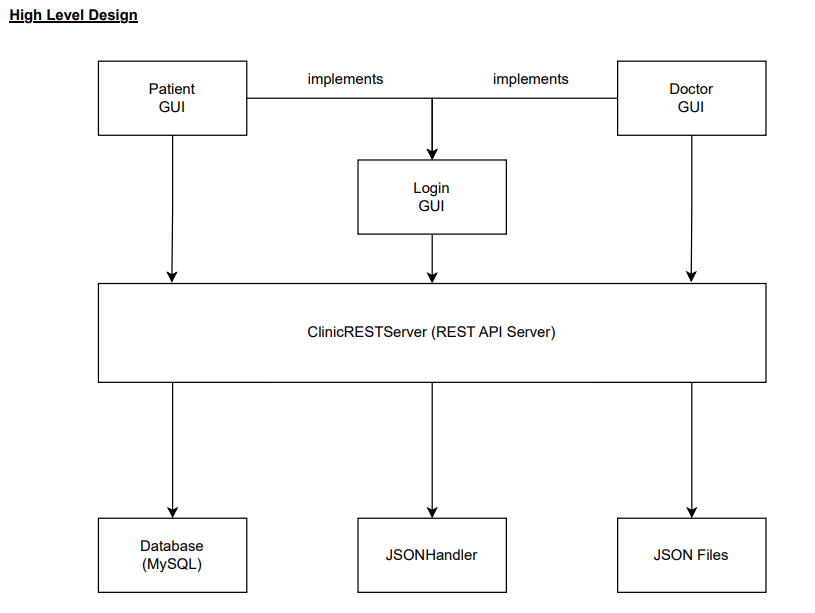
**Implementation:**

**ClinicRestServer.java**

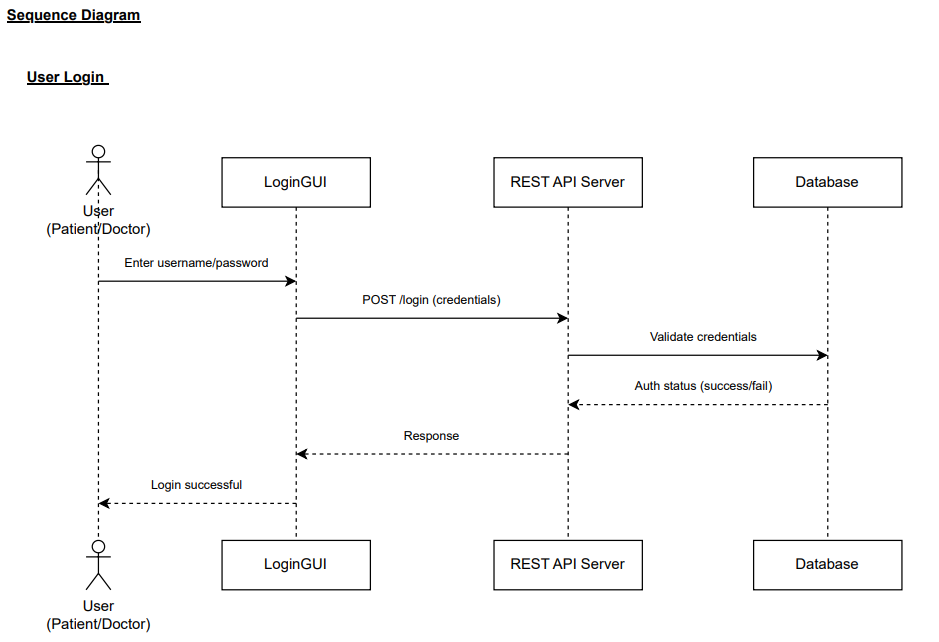
* **Defines the REST API endpoints such as:**
* **GET /patients → fetch all patients**
* **POST /appointments → add an appointment**
* **PUT /prescriptions/{id} → update prescription**
* **Handles routing and response formatting ( JSON responses).**
* **Useful for client-server communication (a front-end app or mobile app consuming this API).**

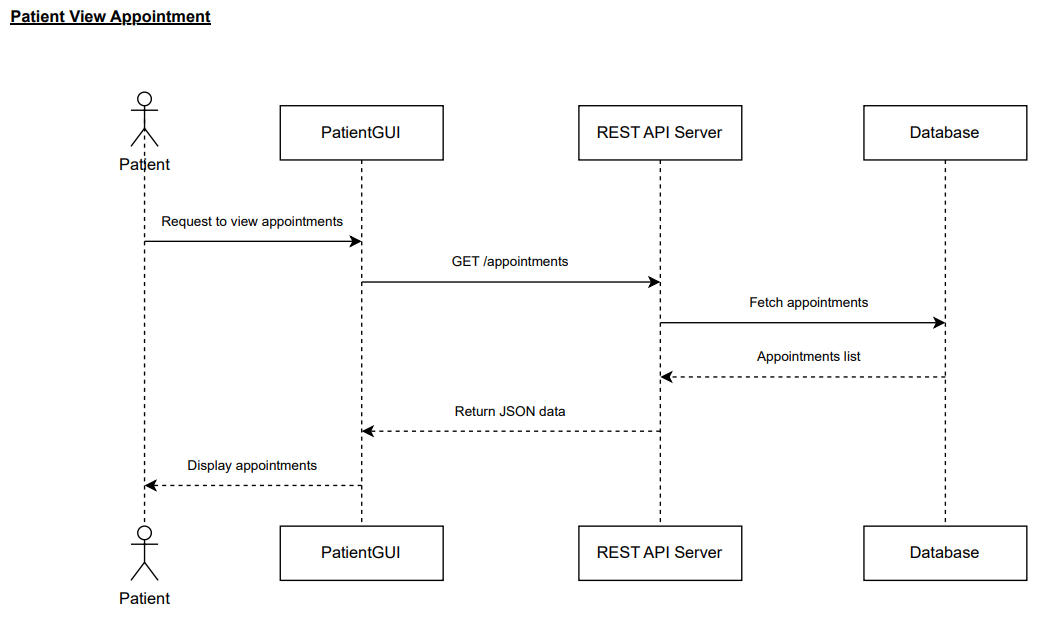
**SYSTEM ARCHITECTURE**

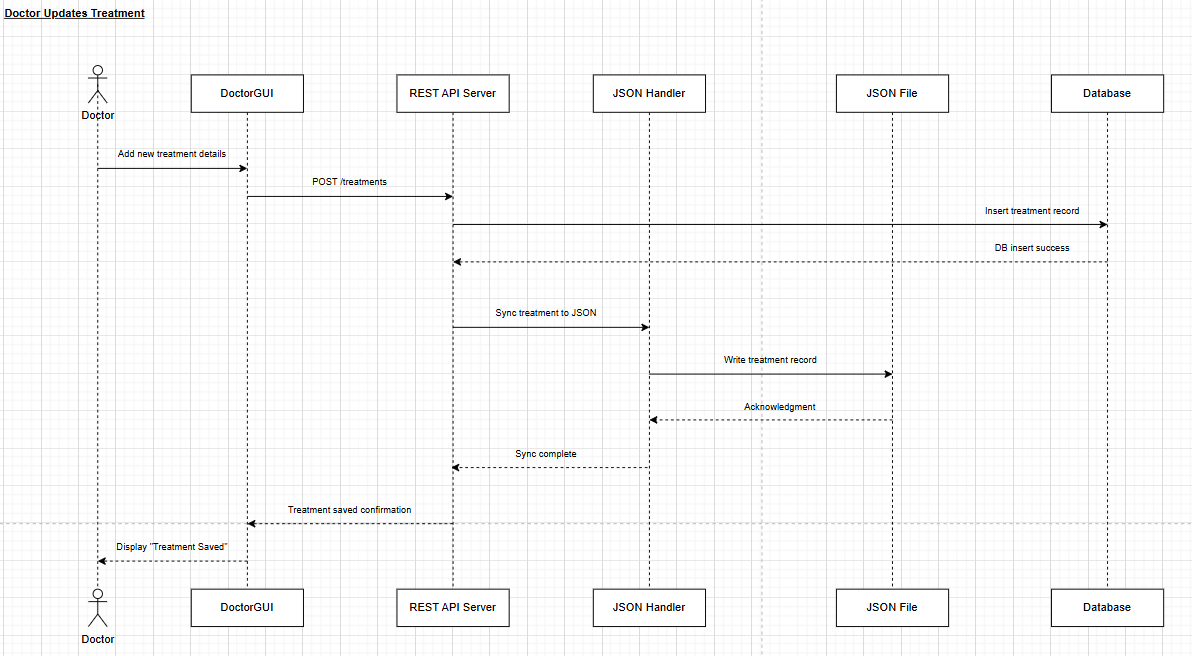
High level design



**Sequence diagram**







**BACKEND APPLICATION**

**Technology stack**

|  |  |  |  |
| --- | --- | --- | --- |
| Component | Technology | Version | Justification |
| Programming language | Java | 11+ | Java offers strong static typing, which reduces runtime errors and ensures greater reliability,critical for handling sensitive medical data. Its mature ecosystem, widespread community support, and built-in security features make it well-suited for developing healthcare applications where data integrity and safety are paramount. Java 11+ also provides performance improvements, long-term support (LTS), and modern APIs that aid in building scalable, maintainable systems. |
| HTTP Server | com.sun.net.httpserver | JDK-native | This built-in lightweight HTTP server eliminates the need for external frameworks, reducing system complexity and improving portability. It is efficient for RESTful service development and is ideal for applications with moderate traffic, such as a university health center. Using a JDK-native component also minimizes deployment overhead and simplifies system updates and maintenance. |
| JSON Processing | org.json | 20231013 | The org.json library provides a simple and effective way to parse and generate JSON, a widely adopted data interchange format. It supports data structures compatible with FHIR (Fast Healthcare Interoperability Resources), enabling easier integration with future medical systems or third-party APIs. Its lightweight footprint makes it an optimal choice for data serialization in resource-constrained environments. |
| Database | MySQL | 8.0+ | MySQL 8.0+ supports full ACID (Atomicity, Consistency, Isolation, Durability) compliance, which is essential for maintaining the integrity and reliability of clinical data. Its support for indexing, foreign keys, and advanced query optimizations ensures fast and secure data access. Additionally, MySQL's widespread adoption and robust documentation make it a stable choice for backend data storage in healthcare systems. |
| Concurrency | java.util.concurrent | JDK-native | The java.util.concurrent package provides high-level concurrency utilities such as thread pools, locks, and atomic variables, which are essential for managing concurrent user interactions like scheduling appointments. It ensures thread safety and responsiveness in a multi-user environment, allowing the system to handle simultaneous operations without race conditions or data corruption. Being JDK-native, it integrates seamlessly with the rest of the Java application. |

**API documentation**

(1) A list of all API endpoints

(2) The HTTP method for each endpoint

(3) Required request parameters, headers, and body formats .

(4) Example success and error responses.

(5) Security: Detail the security measures implemented.

**Appointment management**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Endpoint** | **Method** | **Parameter** | **Header** | **Body** | **Success Response** | **Error Responses** |
| **GET /api/appointments** | **GET** | **?doctorId=int** **?patientId=int** **?date=YYYY-MM-DD** | **Authorization: Bearer <token>** | **None** | **200: List of appointments** | **401: Unauthorized** **500: Server error** |
| **POST /api/appointments** | **POST** | **None** | **Content-Type: application/json** | **json<br>{<br> "patientId": int,<br> "doctorId": int,<br> "date": "YYYY-MM-DD",<br> "time": "HH:MM:SS"<br>}** | **201: Created appointment** | **400: Invalid data** **409: Conflict** |
| **PUT /api/appointments/{id}/status** | **PUT** | **id=int** | **Content-Type: application/json** | **json<br>{<br> "status": "confirmed|completed|cancelled"<br>}** | **200 : Status updated** | **404: Not found** **422: Invalid status** |
| **PUT /api/appointments/{id}/reschedule** | **PUT** | **id=int** | **Content-Type: application/json** | **json<br>{<br> "newDate": "YYYY-MM-DD",<br> "newTime": "HH:MM:SS"<br>}** | **200: Rescheduled** | **400: Invalid datetime** **409: Slot occupied** |

**Treatment management**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Endpoint** | **Method** | **Parameter** | **Header** | **Body** | **Success Response** | **Error Responses** |
| **POST /api/treatments** | **POST** | **None** | **Content-Type: application/json** | **json<br>{<br> "appointmentId": int,<br> "diagnosis": string,<br> "treatmentType": string,<br> "medication": string,<br> "notes": string<br>}** | **201: Treatment created** **Location: /api/treatments/{id}** | **400: Missing data** **404: Appointment not found** |
| **GET /api/treatments/{appointmentId}** | **GET** | **appointmentId=int** | **Authorization: Bearer <token>** | **None** | **200: Treatment details** | **403: Forbidden** **404: Not found** |

**Patient data**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Endpoint** | **Method** | **Parameter** | **Header** | **Body** | **Success Response** | **Error Responses** |
| **GET /api/patient/{id}** | **GET** | **id=int** | **Authorization: Bearer <token>** | **None** | **200: Patient profile** | **401: Unauthorized** **404: Not found** |
| **GET /api/patients/{id}/records** | **GET** | **id=int** **?limit=int** | **Authorization: Bearer <token>** | **None** | **200: Medical history** | **403: Access denied** |
| **GET /api/patients/search** | **GET** | **?name=string** **?birthDate=YYYY-MM-DD** | **Authorization: Bearer <token>** | **None** | **200: Patient list** | **400: Invalid query** |

**System operation**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Endpoint** | **Method** | **Parameter** | **Header** | **Body** | **Success Response** | **Error Responses** |
| **GET /api/health** | **GET** | **None** | **None** | **None** | **200: json<br>{<br> "status": "ok",<br> "dbConnected": boolean<br>}** | **503: Service unavailable** |
| **POST /api/sync** | **POST** | **None** | **X-Admin-Token: string** | **None** | **202: Sync initiated** | **401: Unauthorized** **423: Sync in progress** |

**Json examples**

1. **Appointment management:**

**GET /api/appointments**

**Request:**

GET /api/appointments?doctorId=5&date=2025-07-20  
 Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

**Success Response (200):**

{

"data": [

{

"id": 1023,

"patientId": 101,

"patientName": "John Doe",

"doctorId": 5,

"date": "2025-07-20",

"time": "09:30:00",

"status": "confirmed",

"notes": "Annual checkup"

}

]

}

**Error Response (401):**

{

"error": "Unauthorized",

"message": "Invalid or expired token",

"timestamp": "2025-07-20T08:15:00Z"

}

**POST /api/appointments**

**Request:**

POST /api/appointments

Content-Type: application/json

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

{

"patientId": 101,

"doctorId": 5,

"date": "2025-07-21",

"time": "14:00:00",

"notes": "Follow-up visit"

}

**Success Response (201):**

{

"id": 1024,

"status": "confirmed",

"location": "/api/appointments/1024",

"confirmationNumber": "CLINIC-2025-1024"

}

**Error Response (409):**

{

"error": "Conflict",

"message": "Doctor not available at requested time",

"nextAvailable": "2025-07-21T14:30:00"

}

1. **Treatment management**

**POST /api/treatments**

**Request:**

POST /api/treatments

Content-Type: application/json

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

{

"appointmentId": 1024,

"diagnosis": "Hypertension (I10)",

"treatmentType": "Medication",

"medication": "Lisinopril 10mg",

"dosage": "Once daily",

"notes": "Monitor blood pressure weekly"

}

**Success Response (201):**

{

"id": 789,

"appointmentId": 1024,

"diagnosisCode": "I10",

"medications": [

{

"name": "Lisinopril",

"strength": "10mg",

"refills": 3,

"instructions": "Take once daily in the morning"

}

],

"followUpDate": "2025-08-21"

}

**Error Response (404):**

{

"error": "Not Found",

"message": "Appointment 1024 not found",

"suggestedAppointments": [1023, 1025]

}

1. **Patient data**

**GET /api/patients/101/records**

**Request:**

GET /api/patients/101/records?limit=5

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

**Success Response (200):**

{

"patientId": 101,

"name": "John Doe",

"records": [

{

"date": "2025-07-20",

"doctor": "Dr. Smith",

"diagnosis": "Hypertension",

"treatment": "Prescribed Lisinopril"

},

{

"date": "2025-06-15",

"doctor": "Dr. Johnson",

"diagnosis": "Annual physical",

"treatment": "Lab tests ordered"

}

]

}

**Error Response (403):**

{

"error": "Forbidden",

"message": "You don't have permission to access these records"

}

1. **System operation**

**GET /api/health**

**Request:**

GET /api/health

**Success Response (200):**

{

"status": "healthy",

"components": {

"database": {

"status": "connected",

"latency": "12ms"

},

"memory": {

"used": "45%",

"total": "16GB"

}

},

"uptime": "5d 7h 22m"

}

**Error Response (503):**

{

"status": "unhealthy",

"errors": [

{

"component": "database",

"error": "Connection timeout",

"timestamp": "2025-07-20T09:45:00Z"

}

],

"maintenanceWindow": "00:00-02:00 UTC"

}

**Security implementation**

**1. Authentication**

**Method: Direct database credential validation**  
**Location: LoginGUI.java**

// Plaintext credential verification

String sql = "SELECT id FROM patients WHERE email = ? AND password = ?";

try (PreparedStatement stmt = conn.prepareStatement(sql)) {

stmt.setString(1, email);

stmt.setString(2, password); // Stored and compared in plaintext

ResultSet rs = stmt.executeQuery();

return rs.next() ? rs.getInt("id") : -1;

}

**2. Database Security**

**Location: DBConnection.java**

// SSL-enabled connection

String url = "jdbc:mysql://localhost:3306/clinicdb?useSSL=true";

**Protections:**

* **Forces TLS for MySQL connections**
* **Uses parameterized queries to prevent SQL injection**

**3. Input Sanitization**

**Location: Various handlers**

// Basic string cleaning

String cleanInput = input.replaceAll("[^a-zA-Z0-9]", "");

**4. API Protection**

**Location: ClinicRestServer.java**

// CORS headers

exchange.getResponseHeaders().add("Access-Control-Allow-Origin", "\*");

**5. JSON Data Security**

**Location: JSONHandler.java**

// No encryption for JSON files

Files.write(Paths.get("patients.json"), jsonArray.toString().getBytes());

**FRONTEND APPLICATION**

Basically, there are three GUI in the project, which is LoginGUI, DocterGUI and PatientGUI.

**1. Doctor GUI**

Purpose: Designed for medical professionals to manage clinical operations.

Key functions:

* Viewing/filtering appointments by status (confirmed, pending, completed).
* Recording diagnoses, treatments, and medications.
* Updating appointment statuses.
* Accessing/exporting patient medical records.

Target User:  
Medical professionals (doctors) in a clinic.

Technology Stack:

|  |  |
| --- | --- |
| Category | Technologies & Libraries |
| Core Framework | Java Swing (GUI) |
| HTTP Client | Java 11+ HttpClient |
| JSON Handling | org.json library |
| UI Components | Java AWT (graphics), Swing components (tables, comboboxes) |
| Utilities | JDBC (authentication), Java NIO (file export) |

**API Integration:**  
Communicates with backend REST API at http://localhost:8000:

**1) Data Retrieval**

GET /api/appointments?doctorId={id}: Loads doctor's appointments

GET /api/all-records?search={term}: Searches patient records

**2) Data Submission**

POST /api/treatments: Submits diagnosis/treatment details

PUT /api/appointments/{id}/status: Updates appointment status

**3) Data Format**

Requests: JSON payloads

Responses: JSON arrays (appointments) or objects (patient details)

**4) Authentication**

Uses doctorId from login session in all API requests

**2. Patient GUI**

Purpose: Enables patients to manage personal healthcare interactions.

Key functions:

* Booking/rescheduling/canceling appointments
* Viewing medical records and appointment history
* Exporting health data
* Locating pharmacies via Google Maps

Target User:  
Patients registered with the clinic.

Technology Stack:

|  |  |
| --- | --- |
| Category | Technologies & Libraries |
| Core Framework | Java Swing (GUI) |
| HTTP Client | Java 11+ HttpClient |
| JSON Handling | org.json library |
| Maps Integration | java.awt.Desktop (Google Maps links) |
| UI Components | Java AWT layouts, Swing tables/dialogs |

**API Integration:**  
Communicates with backend REST API at http://localhost:8000:

**1) Appointment Management**

POST /api/appointments: Books new appointments

PUT /api/appointments/{id}/reschedule: Changes appointment time

DELETE /api/appointments/{id}: Cancels appointments

**2) Data Access**

GET /api/records?patientId={id}: Retrieves medical records

GET /api/patients?id={id}: Fetches patient profile

**3) Profile Updates**

PUT /api/patients/{id}/profile: Saves profile changes

**4) Authentication**

Uses patientId from login session in all requests

**3. Login GUI**

Purpose: Central authentication gateway for both patients and doctors.

Features:

* Role-based login (Patient/Doctor)
* Secure password field with visibility toggle
* Input validation (email format)
* Redirect to appropriate portal

Target User:  
Patients and medical staff.

Technology Stack:

|  |  |
| --- | --- |
| Category | Technologies & Libraries |
| Core Framework | Java Swing (GUI) |
| Authentication | JDBC (SQL queries) |
| UI Components | Java AWT graphics, Swing text fields/buttons |
| Validation | Regex (email format) |

**API Integration:**  
Direct database access :

**1) Authentication Flow**

Runs SQL queries against patients/doctors tables:

SELECT id FROM patients WHERE email=? AND password=?

SELECT id FROM doctors WHERE email=? AND password=?

**2) Session Initialization**

Launches PatientGUI or DoctorGUI with user ID on success

**3) Security**

Uses parameterized SQL queries to prevent injection

**Key Architectural Notes**

**1) Shared Backend**  
Both portals consume the same REST API (http://localhost:8000) but access role-specific endpoints.

**2) Stateless Design**  
User IDs (doctorId/patientId) are passed in all API requests instead of sessions.

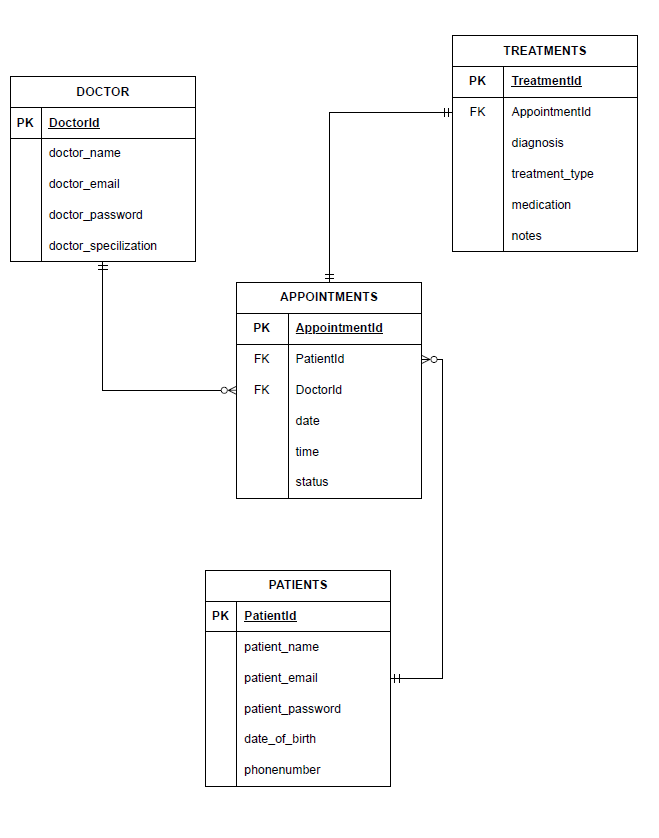
**3) Error Handling**

API errors shown in Swing JOptionPane dialogs

HTTP status codes (200, 201, 400) drive UI feedback

**DATABASE DESIGN**

**ENTITY RELATIONSHIP DIAGRAM (ERD)**



**Schema Justification**

**Normalization:**

* The database schema is designed with a degree of normalization to reduce data redundancy and improve data integrity. Each entity (Doctors, Patients, Appointments, Treatments) has its own table.

**Primary Keys:**

* Each table has a unique id column serving as its primary key, ensuring that each record can be uniquely identified.

**Foreign Keys:**

* patient\_id in appointments links to patients.id, establishing that an appointment must be associated with an existing patient.
* doctor\_id in appointments links to doctors.id, establishing that an appointment must be associated with an existing doctor.
* appointment\_id in treatments links to appointments.id, ensuring that a treatment record corresponds to a specific appointment.

**Data Types:**

* Appropriate data types are used for each column (e.g., INT for IDs, VARCHAR for names and emails, DATE for dates, TIME for times, TEXT for longer descriptions like notes and diagnosis, ENUM for predefined statuses).
* **Constraints:**
* NOT NULL constraints ensure that essential information (like names, emails, and core appointment details) is always present.
* UNIQUE constraints on email in both doctors and patients tables prevent duplicate email addresses, which are crucial for user authentication.
* The status ENUM in appointments provides a controlled set of values for appointment states, ensuring consistency.

**Separation of Concerns:**

**Doctors and Patients:**

* Separate tables for doctors and patients allow for distinct attributes and roles within the system.

**Appointments:**

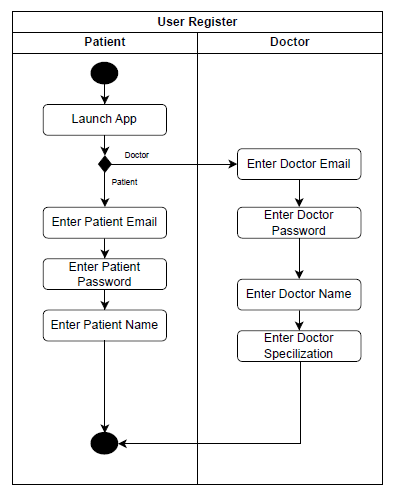
* The appointments table acts as a central linking entity between patients and doctors, capturing the scheduling aspect.

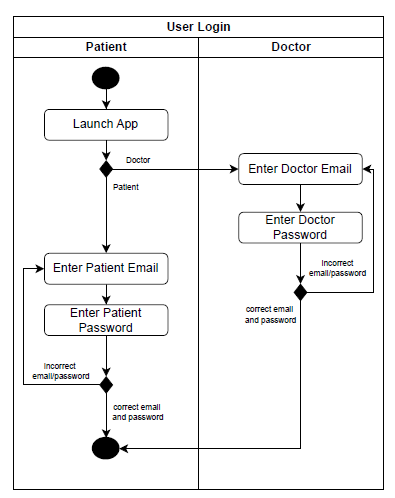
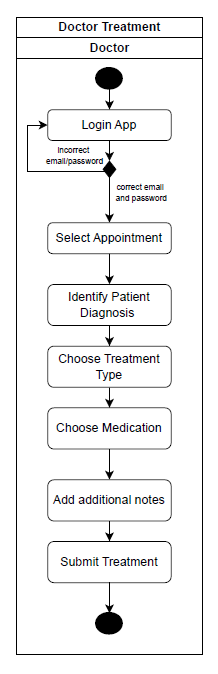
**Treatments:**

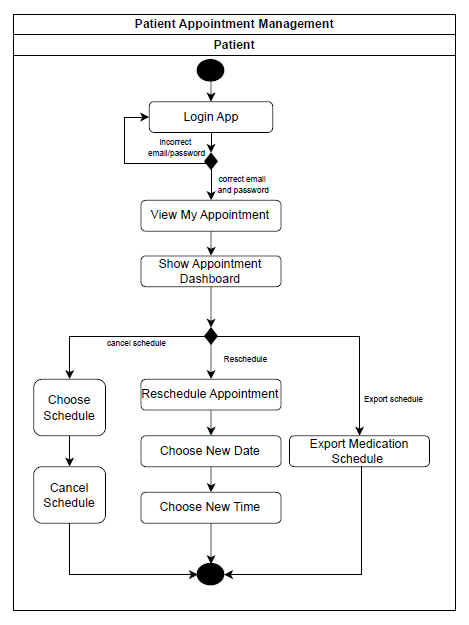
* The treatments table is separate from appointments to store detailed medical information that is only relevant once an appointment has occurred and a diagnosis/treatment has been made. This keeps the appointments table cleaner and focused on scheduling

**BUSINESS LOGIC**  
  
**Use Case Diagram**  
  

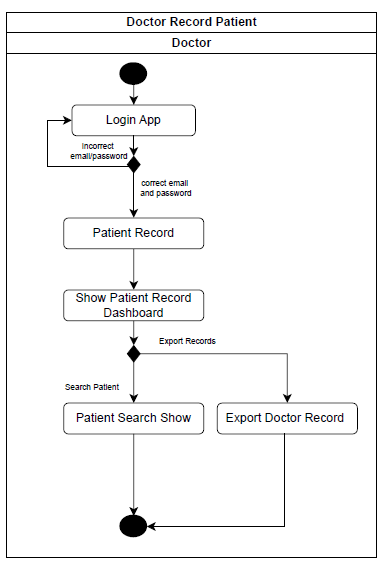

**FLOWCHART**

**User Registration**  
  


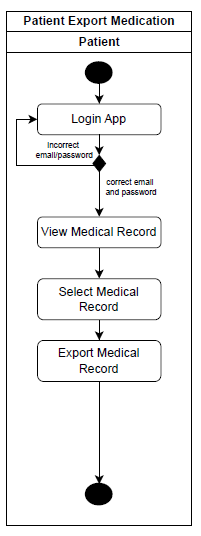
**User Login**  
  
  
**Doctor Treatment**  


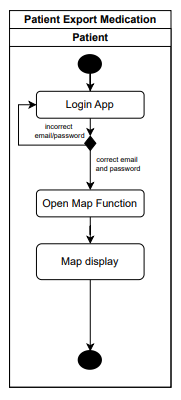
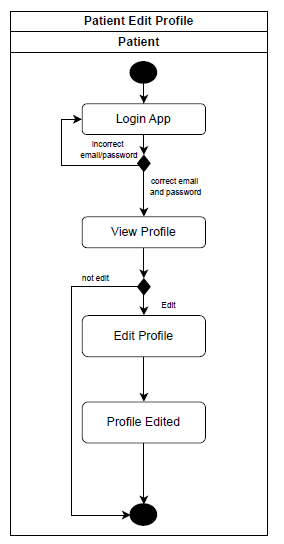
**Patient Appointment Management**  


**Doctor Record Patient**



**Patient Export Medication**



**Patient Open Map**  
  
  
**Patient Edit Profile**  


**DATA VALIDATION**

1. Frontend Validation (GUI Level)

* Empty Field Checks:
* Login: Ensures "Email" and "Password" fields are not empty before attempting authentication.
* Appointment Booking: Ensures a doctor is selected.
* Treatment Submission: Ensures "Diagnosis" field is not empty.

Format Validation:

* Login: Checks if the entered "Email" adheres to a standard email format (e.g., using a regex pattern).
* Logical Checks (Pre-API Call):
* Appointment Reschedule/Cancel: Prevents rescheduling or canceling appointments that are already 'completed' or 'cancelled'.
* Appointment Booking: Displays a message if no doctors are loaded.

2. Backend Validation (REST Server/DBConnection Level)

* Missing Parameter Checks (ClinicRestServer Handlers):
* AppointmentsHandler (GET): Checks for the presence of patientId or doctorId parameters.
* AppointmentsHandler (POST): Validates that patientId, doctorId, date, and time are present in the request body.
* TreatmentsHandler (POST): Validates that appointmentId, diagnosis, treatmentType, and medication are present.
* RescheduleAppointmentHandler: Checks for valid appointmentId in the path and date, time in the request body.
* AppointmentStatusHandler: Checks for valid appointmentId in the path and status in the request body.

Data Integrity (DBConnection):

* Foreign Key Constraints: The database schema enforces that patient\_id, doctor\_id, and appointment\_id must refer to existing records in their respective parent tables. This prevents orphaned records.
* Unique Constraints: The email columns in doctors and patients tables have UNIQUE constraints, preventing the creation of multiple user accounts with the same email address.
* ENUM Type: The status column in appointments only allows predefined values ('pending', 'confirmed', 'completed', 'cancelled', 'rescheduled'), ensuring data consistency.
* ON DUPLICATE KEY UPDATE (during initialization/sync): When populating the database from JSON files, this SQL clause is used to prevent inserting duplicate records based on unique keys (like email for doctors/patients, or a combination of patient/doctor/date/time for appointments), instead updating existing ones.

Business Logic Validation (DBConnection):

* Appointment Conflict Check (checkAppointmentConflict): Before booking a new appointment, the backend explicitly checks if the selected doctor already has an appointment at the exact date and time that is not 'cancelled' or 'completed'. This prevents double-booking.
* Treatment Existence Check (insertTreatment): Before inserting a new treatment, the backend checks if a treatment record already exists for that appointment\_id. If it does, it updates the existing record instead of creating a new one, ensuring a one-to-one relationship between completed appointments and treatments.
* Status Updates: The insertTreatment method automatically updates the appointment status to 'completed' after a treatment is recorded, reflecting the business process.
* Reschedule/Cancel Logic: While some checks are on the frontend, the backend methods (rescheduleAppointment, cancelAppointment, updateAppointmentStatus) are designed to handle the actual database updates, ensuring that the status changes are correctly applied and persisted.
* Error Handling: Both frontend and backend include robust error handling (try-catch blocks, JOptionPane messages, HTTP status codes) to inform users and developers about issues, whether they are network errors, database errors, or validation failures.