Albanian Skills
November 2025

Software Development
Project Statement

Heart+: Building the Core Engine

Overview

91Life, a leader in digital health innovation, is dedicated to building technology that saves lives by streamlining complex medical data. Their latest initiative, Heart+, invites talented engineers to build the core engine for a next-generation cardiology platform.

This contest challenges participants to tackle the complex, high-stakes world of health-tech engineering. You will have the opportunity to design a secure, role-based authentication system,

architect a patient database from the ground up, and build a powerful data ingestion engine to parse complex medical files like HL7 and PDF reports—all within a robust command-line interface.

This event is a true test of engineering skills. Participants will create a working, scalable system from scratch, mastering fundamental concepts of secure system design, data architecture, and real-world parsing logic. By the end of the competition, you will have developed a functional prototype of the Heart+ data engine, laying out the groundwork for a tool that will one day help doctors manage critical cardiac patient data more efficiently than ever before.

What We're Looking For

- **Functional, tested software**: A working system is paramount. It's better to deliver a project that functions correctly, even if simple, than a complex system with issues. The results matter.
- **Object-Oriented Design**: Using OOP principles to reflect the real world inside your code. Clean design should mirror logical systems and domain models.
- **Code readability and re-usability**: Your code is going to be read by others (and perhaps you months later!). Maintain clarity and simplicity to make future improvements and collaborations easier.
- **Efficient solutions**: Your system should run smoothly, with efficient algorithms and data structures. Fast performance and smooth interaction are key for a good user experience.
- **Adaptability to new concepts**: Not every participant may be familiar with every programming tool or technique. Adaptability and willingness to learn new concepts is essential in the ever-evolving tech world.

Mission 1: Authentication & Role-Based Access Control

A medical platform's most critical feature isn't a feature at all—it's security. Before a single piece of patient data is touched, we must build "The Gate" and post its "Gatekeepers." We must be able to answer three questions:

1. **Who are you?** (Authentication)
2. **What are you allowed to do?** (Role)
3. **Can we prove it?** (Logging & Security)

Your first mission is to build the foundation of this trust. You will design the user database from scratch and implement the core Role-Based Access Control (RBAC) that protects all future actions.

The Base Challenge

Your goal is to build a system that allows doctors and nurses to register and log in.

Goal 1: Architect the User Database

You must design a persistent, file-based database for user accounts. We recommend using JSON (`.json`) or plain text (`.txt`) files for simplicity, though you're free to choose the format that best fits your implementation.

Your database schema must support storing:

- User identification (like a username)
- A securely hashed password
- A user role (e.g., "doctor" or "nurse")

Example database structure (you may design yours differently):

```
{
  "users": [
    {
      "username": "dr.smith",
      "passwordHash": "$2b$10$...",
      "role": "doctor"
    }
  ]
}
```

Goal 2: Implement User Registration

Your CLI must provide a mechanism for new users (doctors and nurses) to create an account. This process must: - Validate input (username format, password strength, valid role selection) - Correctly hash their password using a secure algorithm - Save their details, including their role, to your user database - Handle errors gracefully (e.g., username already exists, invalid role)

Example CLI interaction (you may design yours differently):

```
> register
Enter username: dr.smith
Enter password: ********
Select role (doctor/nurse): doctor
SUCCESS: User 'dr.smith' registered successfully.
```

Goal 3: Implement User Sign-in

Your CLI must provide a mechanism for existing users to authenticate. This must: - Securely compare their provided credentials against the stored, hashed data - Validate that the user exists and credentials are correct - Handle authentication failures with clear error messages

Example CLI interaction (you may design yours differently):

```
> login
Enter username: dr.smith
Enter password: ********
SUCCESS: Logged in as dr.smith (doctor)
```

The Bonus Challenge: The "High-Security" Model

For extra points: Real-world compliant systems don't allow open registration. Accounts are provided by technical staff. Your goal is to implement this more secure model.

Goal 1: Evolve the Database & Create Admin

Modify your user database to support a high-privilege admin role. You must manually pre-seed your database with at least one active admin account (e.g., username: "admin", password: "admin123" - you can document this in your README).

Goal 2: Implement Secure User Provisioning

Your CLI must provide a secure mechanism, accessible only by an authenticated admin, to create new doctor and nurse accounts. This command must: - Verify the current user is authenticated and has admin role - Reject requests from non-admin users with a clear error message - Create new accounts with the same validation and security as registration

Example CLI interaction (you may design yours differently):

```
> login
Enter username: admin
Enter password: ********
SUCCESS: Logged in as admin (admin)

> create-user
Enter username: nurse.jones
Enter password: ********
Select role (doctor/nurse): nurse
SUCCESS: User 'nurse.jones' created successfully.
```

Goal 3: Disable Public Registration

The public registration mechanism from the Base Challenge must be disabled. The only way new doctor or nurse accounts can be created is via the admin-only provisioning function.

Key Concepts

Password Hashing: Storing plain-text passwords will result in a failed mission. We expect to see secure, one-way hashing (e.g., bcrypt, Argon2). Never store passwords in plain text.

Authentication State: How will your CLI "know" a user is logged in for future missions? You must design a solution. Here are some common approaches (choose what works best for your design):

- **Session File**: Create a temporary file (e.g., `.session` or `session.json`) that stores the current user's information after login. Subsequent commands check this file to verify authentication.
- **Token System**: Generate a simple token after login and require it to be passed with each command.
- **Per-Command Authentication**: Require username/password on every command (simpler but less user-friendly).

Example session file approach:

```
{
  "username": "dr.smith",
  "role": "doctor",
  "loggedInAt": "2024-01-15T10:30:00Z"
}
```

Input Validation: Your system must validate: - Username: non-empty, reasonable length (e.g., 3-50 characters), no special characters that could cause issues - Password: minimum length requirement (e.g., 8 characters), or allow any password but document your policy - Role: must be exactly one of the allowed values ("doctor", "nurse", or "admin" for bonus)

Error Handling: All operations must handle errors gracefully and provide clear, user-friendly error messages. Examples: - "ERROR: Username already exists" - "ERROR: Invalid credentials" - "ERROR: Permission denied. Admin access required." - "ERROR: User not authenticated. Please login first."

Logging (Optional): Consider implementing a simple log file (e.g., `auth.log`) to track critical security events like login attempts (success/failure) and user provisioning actions. This is optional but demonstrates good security practices. Keep it minimal to avoid cluttering your terminal output.

Mission 2: The Digital File Room (Patient Registry)

The Story

"You're in. You are an authenticated clinician. Now, a new patient, John Doe, has just been admitted. Before you can manage their cardiac data, you must be able to create their digital file in the system. This mission is about building the hospital's file room—the foundation where all patient records will be stored."

The Base Challenge: The Essentials

Your goal is to build the core Patient creation functionality, all secured behind your authentication system from Mission 1.

Goal 1: Evolve the Database Schema

You must expand your database design to store patient records. Use the same file format you chose in Mission 1 (JSON or TXT). This new "table" or "collection" must exist alongside your User data.

Your patient schema must, at a minimum, support: - A unique Patient Identifier (you can decide how to generate this) - A Medical Record Number (MRN) - must be unique across all patients - Patient Name - Date of Birth (DOB) - Assigned Doctor (the username or ID of the doctor who created/owns this patient) - A placeholder (like an empty list or array) to store all their future transmissions

Patient ID Generation: You can choose any approach that ensures uniqueness. Common options include: - UUID/GUID (e.g., `550e8400-e29b-41d4-a716-446655440000`) - Auto-incrementing number (e.g., `1`, `2`, `3`) - Custom format (e.g., `PAT-2024-001`)

Medical Record Number (MRN): MRNs are typically alphanumeric identifiers used by healthcare facilities. Common formats include: - Numeric: `12345678` - Alphanumeric with dashes: `MRN-12345-A` - Alphanumeric: `MRN123456`

Your system must ensure MRNs are unique. Validate that the MRN format is consistent (you decide the exact format rules).

Patient Name: You can store the name as: - Full name (single field): `"John Doe"` - Separate fields: `{"firstName": "John", "lastName": "Doe"}`

Choose the approach that works best for your design.

Date of Birth: Accept dates in a consistent format. Common formats include: - `MM/DD/YYYY` (e.g., `01/15/1985`) - `YYYY-MM-DD` (e.g., `1985-01-15`) - `DD/MM/YYYY` (e.g., `15/01/1985`)

Choose one format and document it. Validate that dates are in the correct format.

Patient-Doctor Relationship: In real healthcare systems, each patient is assigned to a specific doctor who is responsible for their care. When a doctor creates a patient record, that patient is automatically assigned to them. Store the doctor's username (or user ID) with each patient record.

Transmissions Data Structure: The `transmissions` placeholder will store medical reports (parsed from HL7 files in Mission 3). Each transmission will contain clinical data extracted from reports. Important data that transmissions can include:

- **Report Metadata**:
  - Report ID (unique identifier)
  - Report date/timestamp
  - Message type (e.g., ORU^R01 for observation results)
- **Clinical Observations** (from OBX segments in HL7):
  - Heart Rate (value, unit, reference range)
  - Rhythm (e.g., Normal Sinus Rhythm, Atrial Fibrillation)
  - Blood Pressure (systolic/diastolic)
  - Device information (if present)
  - Other cardiac measurements (ventricular rate, PR interval, QRS duration, etc.)
  - Observation codes and values
  - Abnormal flags
- **Patient Identifiers** (for matching purposes):
  - MRN (from PID segment)
  - Patient name (from PID segment)
  - Date of Birth (from PID segment)

Example transmission structure (what will be stored in the `transmissions` array):

```
{
  "reportId": "550e8400-e29b-41d4-a716-446655440000",
  "reportDate": "20230822143000",
  "messageType": "ORU^R01",
  "observations": [
    {
      "code": "Heart Rate",
      "value": "70",
      "unit": "bpm",
      "referenceRange": "60-100",
      "abnormalFlag": "N"
    },
    {
      "code": "Rhythm",
      "value": "Normal Sinus Rhythm",
      "unit": null,
      "referenceRange": null,
      "abnormalFlag": "N"
    }
  ],
  "patientIdentifiers": {
```

```
    "mrn": "123456789",
    "name": "DOE^JOHN^MIDDLE",
    "dateOfBirth": "19900101"
  }
}
```

Example database structure (you may design yours differently):

```
{
  "patients": [
    {
      "patientId": "550e8400-e29b-41d4-a716-446655440000",
      "mrn": "MRN-12345",
      "name": "John Doe",
      "dateOfBirth": "01/15/1985",
      "assignedDoctor": "dr.smith",
      "transmissions": []
    }
  ]
}
```

Goal 2: Implement Patient Creation

Your CLI must provide a mechanism for an authenticated user (e.g., a doctor or nurse) to register a new patient in the system.

This command must: - Verify the user is properly authenticated using the system you built in Mission 1 - Validate all input fields (MRN format, date format, name not empty, etc.) - Ensure MRN uniqueness (reject if MRN already exists) - Automatically assign the patient to the logged-in doctor (store the doctor's username/ID with the patient record) - Handle errors gracefully with clear error messages - Generate a unique Patient ID - Save the patient record to your database

Example CLI interaction (you may design yours differently):

```
> login
Enter username: dr.smith
Enter password: ********
SUCCESS: Logged in as dr.smith (doctor)

> create-patient
Enter MRN: MRN-12345
Enter patient name: John Doe
```

```
Enter date of birth (MM/DD/YYYY): 01/15/1985
SUCCESS: Patient created successfully. Patient ID: 550e8400-e29b-41d4-a
716-446655440000. Assigned to: dr.smith
```

Error Handling: Your system must handle and clearly report errors such as:
- "ERROR: User not authenticated. Please login first."
- "ERROR: MRN already exists. Please use a different MRN."
- "ERROR: Invalid date format. Expected MM/DD/YYYY."
- "ERROR: MRN cannot be empty."
- "ERROR: Patient name cannot be empty."

The Bonus Challenge: The "Role-Based" Action

For extra points: Not all roles are equal. In a real hospital, a nurse might be able to view patient information, but perhaps only a doctor can register a new patient. Additionally, doctors should only be able to access patients assigned to them.

Goal: Implement Role-Based Permissions

Evolve your logic from Mission 1. Your patient creation command must check the user's role, not just if they are logged in.

- **Patient Creation** (Goal 2) should only be possible by a user with the doctor role. If a nurse tries, it should fail with a "Permission Denied" error.
- **Patient Access Control**: When implementing patient viewing/searching (in future missions), doctors should only be able to access patients assigned to them. This ensures proper data privacy and reflects real-world healthcare practices.

Example error for unauthorized role:

```
> login
Enter username: nurse.jones
Enter password: ********
SUCCESS: Logged in as nurse.jones (nurse)

> create-patient
ERROR: Permission denied. Only doctors can create new patients.
```

Mission 3: The Data Extractor (Report Upload)

The Story

"The first step of our pipeline. In a real clinic, both doctors and nurses handle file uploads—doctors are often busy with patients, so nurses frequently upload transmission files. An

authenticated user (doctor or nurse) needs to upload a new transmission file. We don't care which patient it belongs to yet—that matching happens later. For now, we just need to validate the user, 'crack open' the file, and extract all the data into a standardized format. This mission is about building the 'decoder ring' that turns messy files into clean, usable data."

The Base Challenge: The HL7 Parser

Goal 1: The Secured 'Upload' Command

Your CLI must provide a mechanism for an authenticated user (doctor or nurse from Mission 1) to upload a single report file (`.hl7` for base challenge, or `.pdf` for bonus challenge). Both doctors and nurses should be able to upload files—in real clinics, nurses often handle administrative tasks like file uploads while doctors focus on patient care.

The command must: - Verify the user is properly authenticated using the system you built in Mission 1 - Accept a file path to the report file (as a command argument or through an interactive prompt—you decide) - Validate that the file exists and has a supported extension (`.hl7` for base challenge) - Handle errors gracefully (file not found, invalid format, etc.)

Example CLI interaction (you may design yours differently):

```
> login
Enter username: nurse.jones
Enter password: ********
SUCCESS: Logged in as nurse.jones (nurse)

> upload-report reports/patient_report.hl7
Processing file: reports/patient_report.hl7
...
SUCCESS: Report parsed successfully. Report ID: 550e8400-e29b-41d4-a716
-446655440000. Data extracted and ready for processing.
```

Or with an interactive prompt:

```
> upload-report
Enter file path: reports/patient_report.hl7
Processing file: reports/patient_report.hl7
...
SUCCESS: Report parsed successfully. Report ID: 550e8400-e29b-41d4-a716
-446655440000. Data extracted and ready for processing.
```

Goal 2: The Full-Extract Parser

Understanding HL7 Format: HL7 files are plain text files with a specific structure. They use: - **Segments**: Each line represents a segment (MSH, PID, OBX, etc.) - **Pipe-delimited fields**: Fields within segments are separated by | - **Component separators**: Fields can contain components separated by ^

Example HL7 structure:

```
MSH|^~\&|SENDING_APP|SENDING_FACILITY|RECEIVING_APP|RECEIVING_FACILITY|
20230822143000||ORU^R01|MSG00001|P|2.5
PID|1||123456789^^^MRN^MR||DOE^JOHN^MIDDLE||19900101|M
OBX|1|TX|Heart Rate^Heart Rate^LN|70|bpm|60-100|N
OBX|2|TX|Rhythm^Rhythm^LN|Normal Sinus Rhythm|||N
```

Since HL7 files are plain text, you can parse them using basic string manipulation (splitting newlines, pipes, etc.). No special libraries are required, though you're free to use them if you prefer. **Part of this challenge is learning about HL7 structure and how to extract the data you need**—feel free to research HL7 format documentation and segment specifications.

Your command must:

1. Open and parse the `.hl7` file
2. Extract ALL key data:
   - **Patient Identifiers** from PID segment:
     - PID Field 3: Patient Identifier (MRN)
     - PID Field 5: Patient Name (may contain components separated by ^, e.g., `DOE^JOHN^MIDDLE` = Last^First^Middle)
     - PID Field 7: Date of Birth
   - **Clinical Observations** from OBX segments (there can be multiple OBX segments—extract all of them):
     - OBX Field 3: Observation Code/Name (may contain components separated by ^)
     - OBX Field 5: Observation Value
     - OBX Field 6: Units (if present)
     - OBX Field 7: Reference Range (if present)
     - OBX Field 8: Abnormal Flag (if present)
3. Create a single JSON object containing all this extracted data, plus a unique `reportId` (or `report_id`—choose a consistent naming convention)
4. Output the JSON data to the console (you may optionally save it to a file, but console output is sufficient for this mission)

Report ID Generation: Generate a unique identifier for each parsed report. Common approaches include: - UUID/GUID (e.g., `550e8400-e29b-41d4-a716-446655440000`) - Timestamp-based (e.g., `20230822143000-001`) - Auto-incrementing number (e.g., `1`, `2`, `3`)

Component Parsing Hints: When fields contain components separated by ^, you'll need to split them. For example: - `DOE^JOHN^MIDDLE` → split by ^ → `["DOE", "JOHN", "MIDDLE"]` (Last, First, Middle) - `Heart Rate^Heart Rate^LN` → split by ^ → `["Heart Rate", "Heart Rate", "LN"]` (Display Name, Code, Code System)

Handling Multiple OBX Segments: HL7 files often contain multiple OBX segments (one per observation). Extract all OBX segments and include them in your observations array.

Handling Missing Segments: If required segments (MSH, PID, OBX) are missing, handle this gracefully with an appropriate error message. At minimum, you should have a PID segment (for patient identifiers) and at least one OBX segment (for observations).

Handling Empty Fields: HL7 files often contain empty fields (shown as consecutive | characters or empty strings). Your parser should handle these gracefully—you can set them to `null` in JSON or omit them entirely.

Output Format: Print the extracted JSON to the console so users can see the parsed data. The JSON structure should be consistent with the transmission structure described in Mission 2 (see the "Transmissions Data Structure" section). This ensures compatibility when the data is later matched to patients and stored.

Example JSON Output (based on the HL7 example above):

```json
{
  "reportId": "550e8400-e29b-41d4-a716-446655440000",
  "reportDate": "20230822143000",
  "messageType": "ORU^R01",
  "observations": [
    {
      "code": "Heart Rate",
      "value": "70",
      "unit": "bpm",
      "referenceRange": "60-100",
      "abnormalFlag": "N"
    },
    {
      "code": "Rhythm",
      "value": "Normal Sinus Rhythm",
      "unit": null,
      "referenceRange": null,
      "abnormalFlag": "N"
    }
  ],
  "patientIdentifiers": {
    "mrn": "123456789",
    "name": "DOE^JOHN^MIDDLE",
    "dateOfBirth": "19900101"
  }
}
```

Note: This example structure aligns with the transmission structure described in Mission 2, making it easier to store the data later. You can design your JSON format as you see fit, as long as it contains the extracted patient identifiers and clinical observations. The important part is that the data is clean, structured, and usable for future matching algorithms. When storing in Mission 4, ensure the structure matches Mission 2's transmission format.

Error Handling: Your system must handle and clearly report errors such as:

- "ERROR: User not authenticated". Please login first."
- "ERROR: File not found: reports/patient_report.hl7"
- "ERROR: Invalid file format. Expected .hl7 file."
- "ERROR: Failed to parse HL7 file. Invalid format."
- "ERROR: File is empty or unreadable."
- "ERROR: Missing required segment: PID segment not found."
- "ERROR: Missing required segment: No OBX segments found."
- "ERROR: Corrupted HL7 file. Unable to parse segments."

Important: This mission focuses on parsing and extraction only. The matching of this data to patients will happen in a future mission. Your parser should output clean, structured JSON that can be used by matching algorithms later.

Success Output: `SUCCESS: Report parsed successfully. Report ID: 550e8400-e29b-41d4-a716-446655440000. Data extracted and ready for processing.`

Bonus Challenge: Robust Error Handling

For extra points: Implement robust handling of corrupted or malformed HL7 files. Your parser should: - Detect and report specific parsing errors (e.g., "Invalid segment format at line X") - Attempt to extract as much valid data as possible even if some segments are corrupted - Provide detailed error messages that help identify the issue in the HL7 file - Handle edge cases like files with unusual encoding or special characters

Character Encoding Note: HL7 files are typically encoded in ASCII or UTF-8. Most modern programming languages handle UTF-8 by default, which should cover most cases. If you encounter files with special characters (accents, non-ASCII characters in names), ensure your file reading handles UTF-8 encoding properly. This is usually handled automatically by most programming languages but be aware of it if you encounter parsing issues with international characters.

Bonus Challenge+: PDF Parser

For extra points: Extend your upload system to support PDF report files. In real clinics, cardiac device reports often come as PDF files from manufacturers, containing similar clinical data to HL7 files but in a different format.

Goal: Implement PDF File Parsing

Extend your `upload-report` command to accept both `.hl7` and `.pdf` files. When a PDF file is uploaded:

1. **Detect File Type**: Automatically detect whether the uploaded file is HL7 or PDF based on extension or file content.
2. **Extract Text from PDF**: Use PDF text extraction libraries to extract text content from the PDF file.
3. **Extract Patient Data**: Extract patient identifiers from the PDF text:

- Patient Name
- MRN
- Date of Birth
- Report Date/Timestamp

4. **Extract Clinical Observations**: Extract clinical measurements from the PDF text:
   - Heart Rate
   - Rhythm
   - Other cardiac measurements (Blood Pressure, Device information, etc.)
5. **Output Same JSON Format**: Convert extracted PDF data to the same JSON structure as your HL7 parser, ensuring compatibility with your matching system from Mission 4.

PDF Parsing Approaches: Most programming languages have PDF text extraction libraries (e.g., PyPDF2 for Python, pdf-lib for JavaScript, pdfbox for Java). You'll need to extract text from the PDF and then parse it to find the required patient data and clinical observations. The approach you use for parsing is up to you.

Example CLI interaction:

```
> upload-report reports/patient_report.pdf
Processing file: reports/patient_report.pdf
Detected file type: PDF
Extracting text from PDF...
Parsing PDF content...
Report parsed successfully. Report ID: 550e8400-e29b-41d4-a716-44665544
0000. Data extracted and ready for processing.
```

Error Handling for PDF Parsing: - "ERROR: Failed to extract text from PDF file." - "ERROR: Could not find required patient information in PDF." - "ERROR: PDF file is corrupted or unreadable."

Important Notes: - PDF parsing is more complex than HL7 because PDFs have variable layouts - Different manufacturers format their PDFs differently - You may need to handle multiple PDF formats/styles - Focus on extracting the same data points as HL7 (patient identifiers, clinical observations) - The extracted data should follow the same JSON structure as HL7 parsing for consistency

Mission 4: The Patient Matcher (Linking Reports to Patients)

The Story

"Great! You've successfully parsed the HL7 file and extracted all the clinical data. Now comes the critical step: connecting this report to the right patient. In a real clinic, transmission files arrive without clear patient assignments—they might have typos in names, missing MRNs, or incomplete information. Your job is to build a matching algorithm that automatically links reports to patients after parsing, even when the data isn't perfect."

The Base Challenge: Exact Matching

Your goal is to integrate matching logic into your upload command from Mission 3. After parsing the HL7 file, the system should automatically attempt to match the extracted data to an existing patient in your database, then store the transmission in that patient's `transmissions` array.

Goal 1: Automatic Matching Integration

Extend your `upload-report` command from Mission 3 to automatically perform matching after parsing:

1. After successfully parsing the HL7 file and extracting JSON data
2. Automatically search for matching patients in your database
3. If a match is found, store the transmission in the matched patient's `transmissions` array
4. Display the matching result to the user

Goal 2: Exact Matching Logic

Implement matching logic that handles exact matches:

1. **Primary Match (MRN)**: If the report contains an MRN, search for a patient with an exact MRN match. This is the most reliable match.
2. **Secondary Match (Name + DOB)**: If no MRN match is found, try matching by exact name and date of birth combination.

Example CLI interaction (you may design yours differently):

```
> login
Enter username: nurse.jones
Enter password: ********
SUCCESS: Logged in as nurse.jones (nurse)

> upload-report reports/patient_report.hl7
Processing file: reports/patient_report.hl7
Parsing HL7 file...
Report parsed successfully. Report ID: 550e8400-e29b-41d4-a716-446655440000

Matching to patient...
Found patient: John Doe (MRN: MRN-12345)
Match confidence: 100% (Exact MRN match)
Storing transmission...
SUCCESS: Report matched and stored for patient John Doe (MRN: MRN-12345)
```

Error Handling: Your system must handle and clearly report errors such as:
- "ERROR: User not authenticated". Please login first."
- "ERROR: No matching patient found. MRN: MRN-12345"
- "ERROR: Multiple patients found with same MRN. Please resolve manually."
- "ERROR: Matching failed. Report parsed but could not be stored."

Important: When storing the transmission, ensure it follows the structure defined in Mission 2's "Transmissions Data Structure" section. The transmission should be added to the patient's `transmissions` array in your database.

The Bonus Challenge: Fuzzy Matching

For extra points: Real-world data is messy. Names have typos, MRNs might be missing, and data entry errors are common. Implement fuzzy matching that can handle these scenarios.

Goal: Implement Similarity-Based Matching

Build a matching algorithm that uses similarity scoring when exact matches fail:

1. **Similarity Scoring**: When exact matching fails (no MRN match, name doesn't match exactly), calculate similarity scores for potential matches based on:
   - Name similarity (handle typos like "Jhon" vs "John")
   - Date of Birth similarity (if available)
   - MRN similarity (if partial MRN is available)
2. **Match Threshold**: Define a minimum similarity threshold (e.g., 80% or 0.8) below which matches are rejected.
3. **Match Confidence**: Display match confidence scores to help users understand the quality of the match.

Fuzzy Matching Approaches (you can research and implement any of these):

- **Levenshtein Distance**: Calculate the minimum number of single-character edits needed to change one string into another. Convert to a similarity percentage.
- **Character Overlap**: Compare how many characters match between strings.
- **Weighted Scoring**: Assign different weights to different fields (e.g., MRN = 100%, Name = 80%, DOB = 60%).
- **Phonetic Matching**: Use algorithms like Soundex or Metaphone to match names that sound similar.

Example CLI interaction with fuzzy matching:

```
> upload-report reports/patient_report.hl7
Processing file: reports/patient_report.hl7
Parsing HL7 file...
Report parsed successfully. Report ID: 550e8400-e29b-41d4-a716-446655440000

Matching to patient...
No exact match found. Attempting fuzzy matching...
Found potential match: John Doe (MRN: MRN-12345)
Match confidence: 85% (Name similarity: 90%, DOB match: 100%)
Reason: Name "Jhon Doe" closely matches "John Doe"
Storing transmission...
SUCCESS: Report matched and stored for patient John Doe (MRN: MRN-12345)
```

Error Handling for Fuzzy Matching:

- "ERROR: No match found above threshold (80%). Closest match: 65%"
- "ERROR: Multiple patients found with similar scores. Please resolve manually."
- "WARNING: Low confidence match (75%). Please verify patient: John Doe"

Success Output: `SUCCESS: Report matched and stored for patient [Patient Name] (MRN: [MRN]). Match confidence: [X]%`

Mission 5: The Doctor's Dashboard (Patient & Transmission Viewer)

The Story

"Perfect! Reports are being parsed, matched, and stored. Now doctors need a way to access all this information. When a doctor logs in, they should see their dashboard—a view of all their patients and the latest transmissions. This is their command center, where they can quickly see what's new and dive into patient details when needed."

The Base Challenge: Dashboard Views

Your goal is to build a dashboard system that allows doctors to view their patients and transmissions. Only doctors should have access to this dashboard (nurses can upload files but don't need dashboard access).

Goal 1: Patient List View

Your CLI must provide a command that displays a list of all patients assigned to the logged-in doctor. The list should show: - Patient Name - MRN - Date of Birth - Number of transmissions (count from the `transmissions` array)

After displaying the list, allow the doctor to select a specific patient to view their full details.

Goal 2: Transmission List View

Your CLI must provide a command that displays a list of all transmissions for patients assigned to the logged-in doctor. The list should be sorted by most recent first (based on report date/timestamp). For each transmission, show: - Report ID - Patient Name (who the transmission belongs to) - Report Date/Timestamp - Number of observations (count from the observations array)

After displaying the list, allow the doctor to select a specific transmission to view its full details.

Goal 3: Patient Details View

When a doctor selects a patient from the patient list, display their complete information: - Patient Name - MRN - Date of Birth - Assigned Doctor - All transmissions (list all transmissions with basic info, or allow drilling down into each)

Goal 4: Transmission Details View

When a doctor selects a transmission from the transmission list, display its complete information: - Report ID - Report Date/Timestamp - Patient Name (who it belongs to) - All observations (full details: code, value, unit, reference range, abnormal flag, etc.) - Patient identifiers used for matching

Example CLI interaction (you may design yours differently):

```
> login
Enter username: dr.smith
Enter password: ********
SUCCESS: Logged in as dr.smith (doctor)

> dashboard
=== Doctor Dashboard ===
Choose view:
1. List my patients
2. List recent transmissions
3. Exit
Enter choice: 1

=== My Patients ===
1. John Doe (MRN: MRN-12345, DOB: 01/15/1985) - 3 transmissions
2. Jane Smith (MRN: MRN-67890, DOB: 03/22/1990) - 1 transmission
3. Back to main menu

Enter patient number to view details: 1

=== Patient Details: John Doe ===
MRN: MRN-12345
Date of Birth: 01/15/1985
Assigned Doctor: dr.smith
Total Transmissions: 3

Transmissions:
1. Report ID: 550e8400-e29b-41d4-a716-446655440000 (2023-08-22 14:30:00)
2. Report ID: 660e8400-e29b-41d4-a716-446655440001 (2023-08-20 10:15:00)
3. Report ID: 770e8400-e29b-41d4-a716-446655440002 (2023-08-18 09:00:00)

Enter transmission number to view details, or 'back' to return: 1

=== Transmission Details ===
Report ID: 550e8400-e29b-41d4-a716-446655440000
Report Date: 2023-08-22 14:30:00
Patient: John Doe (MRN: MRN-12345)

Observations:
1. Heart Rate: 70 bpm (Reference: 60-100, Normal)
2. Rhythm: Normal Sinus Rhythm (Normal)
```

```
Press 'back' to return to patient details.
```

Example for Transmission List View:

```
> dashboard
=== Doctor Dashboard ===
Choose view:
1. List my patients
2. List recent transmissions
3. Exit
Enter choice: 2

=== Recent Transmissions (Most Recent First) ===
1. Report ID: 550e8400-e29b-41d4-a716-446655440000
   Patient: John Doe (MRN: MRN-12345)
   Date: 2023-08-22 14:30:00
   Observations: 2

2. Report ID: 660e8400-e29b-41d4-a716-446655440001
   Patient: John Doe (MRN: MRN-12345)
   Date: 2023-08-20 10:15:00
   Observations: 3

3. Report ID: 880e8400-e29b-41d4-a716-446655440003
   Patient: Jane Smith (MRN: MRN-67890)
   Date: 2023-08-19 16:45:00
   Observations: 1

Enter transmission number to view details, or 'back' to return: 1

=== Transmission Details ===
[Full transmission details as shown above]
```

Error Handling: Your system must handle and clearly report errors such as:

- "ERROR: User not authenticated. Please login first."
- "ERROR: Permission denied. Only doctors can access the dashboard."
- "ERROR: No patients found assigned to you."
- "ERROR: No transmissions found for your patients."
- "ERROR: Invalid selection. Please choose a valid option."

Important: - Only show patients assigned to the logged-in doctor (from Mission 2's `assignedDoctor` field) - Only show transmissions that belong to patients assigned to the logged-in doctor - Transmissions should be sorted by most recent first (use the report date/timestamp from the transmission data) - When displaying lists, show only general/summary information - When displaying details, show complete information

---

Mission 6: Patient Search & Discovery

The Story

"Sometimes doctors need to quickly find a specific patient without scrolling through their entire patient list. Whether they're looking for 'John Doe' or searching by MRN 'MRN-12345', a fast and reliable search function is essential for efficient patient care."

The Base Challenge: Patient Search

Your goal is to build a search system that allows doctors to quickly find their patients. The system should automatically search by both name and MRN, without requiring the user to specify what they're searching for.

Goal 1: Implement Search Command

Your CLI must provide a `search-patient` command that allows authenticated doctors to search for patients assigned to them. The search should: - Accept a search query - Search only within patients assigned to the logged-in doctor - Display matching patients with basic information - If only one patient matches, automatically display their full details (skip the selection step) - If multiple patients match, display a list and allow selecting one

Goal 2: Implement Search Logic

Implement search functionality that handles both name and MRN searches:

Search: - Searches patient names/MRN (case-insensitive) - Supports partial matches (e.g., searching "John" finds "John Doe", searching "Doe" finds "John Doe") - Handles cases where user enters only first name or only last name (or partial MRN) - Searches both first and last name if stored separately - Returns all matching patients

Example CLI interaction - Single match (auto-display):

```
> login
Enter username: dr.smith
Enter password: ********
SUCCESS: Logged in as dr.smith (doctor)

> search-patient
Enter search query: MRN-12345
Searching...

Found 1 patient. Displaying details...

=== Patient Details: John Doe ===
MRN: MRN-12345
Date of Birth: 01/15/1985
Assigned Doctor: dr.smith
Total Transmissions: 3
```

```
[Full patient details]
```

Example CLI interaction - Multiple matches (selection required):

```
> search-patient
Enter search query: John
Searching...

=== Search Results ===
Found 2 patients:
1. John Doe (MRN: MRN-12345, DOB: 01/15/1985) - 3 transmissions
2. John Smith (MRN: MRN-67890, DOB: 03/22/1990) - 1 transmission

Enter patient number to view details, or 'back' to return: 1

=== Patient Details: John Doe ===
[Full patient details]
```

Example CLI interaction - Partial name match:

```
> search-patient
Enter search query: Doe
Searching...

Found 1 patient. Displaying details...

=== Patient Details: John Doe ===
[Full patient details]
```

Error Handling: Your system must handle and clearly report errors such as:
- "ERROR: User not authenticated. Please login first."
- "ERROR: Permission denied. Only doctors can search patients."
- "ERROR: No patients found matching 'xyz'."
- "ERROR: Search query cannot be empty."

Important: - Only search within patients assigned to the logged-in doctor - Search should be case-insensitive - The system should automatically determine whether to search by name or MRN (you can optimize this by analyzing the query format, but this is up to your implementation) - If exactly one patient matches, automatically display full details without requiring selection - If multiple patients match, display a numbered list and allow selection - Handle partial name searches (first name only, last name only, or full name)

The Bonus Challenge: Advanced Search Features

For extra points: Implement enhanced search capabilities that make finding patients even easier.

Goal: Enhanced Search Logic

Implement advanced search features:

1. **Optimized Search Order**: Analyze the search query format to determine the most likely match type and search accordingly (e.g., if query looks like an MRN format, try MRN first; if it looks like a name, try name first).

2. **Combined Search**: Search both name and MRN simultaneously and merge results, removing duplicates and prioritizing exact matches.

3. **Search Result Ranking**: Rank search results by relevance:

   - Exact MRN matches first
   - Exact name matches second
   - Partial name matches third

4. **Search History** (Optional): Keep a simple history of recent searches for quick re-searching.

Example CLI interaction with optimized search:

```
> search-patient
Enter search query: 12345
Searching...

Found 1 patient. Displaying details...

=== Patient Details: John Doe ===
MRN: MRN-12345
[Full patient details]
```