Technical Interview Exercise: Mining and Structuring Drug Indications from Labels

Objective

Develop a **microservice-based application** that extracts drug indications from **DailyMed** drug labels, maps them to standardized medical vocabulary (ICD-10 codes), and provides a **queryable API**. The implementation must be in **Python, Node.js, or .NET** and follow **enterprise-grade software principles**, including:

- **Test-Driven Development (TDD)**
- **Clean Architecture** (separation of concerns, layered design)
- **High Code Quality** (readability, maintainability, modularity)
- **Scalability & Performance Considerations**
- **Dockerized Deployment** (with `docker-compose` for execution)

Requirements

1. Core Features

Data Extraction

- Scrape or parse **DailyMed** drug labels for **Dupixent**.
    - Extract relevant sections describing **indications**.

Indication Processing & Mapping

- Map extracted indications to **ICD-10** codes using an open-source dataset.
- Handle edge cases like:
    - **Synonyms** (e.g., "Hypertension" vs. "High Blood Pressure").
    - **Drugs with multiple indications**.
    - **Unmappable conditions**.
- This should use AI/LLM to complete this step.

Structured Data Output

- Store structured drug-indication mappings in a **database or NoSQL store**.
- Make mappings queryable via an API.

2. Enterprise-Grade API

- **Develop a Web API** using **.NET (C#), Python (FastAPI/Flask), or Node.js (Express/NestJS)**.
- Implement **CRUD operations**:
    - Create, read, update, and delete drug-indication mappings.
- **Authentication & Authorization**
    - Users should be able to register and log in.
    - Implement role-based access control.
- Include **Swagger or Postman workspace** for API testing.
- Ensure consistent data types (e.g., `true/false`, numbers as strings).
- Implement validation rules for missing or ambiguous data.
- Provides an endpoint (`/programs/<program_id>`) returning structured JSON.
- Supports querying program details dynamically.

3. Data & Storage Layer

- Use a **database** (SQL or NoSQL) to store:
    - Drug-indication mappings
    - User authentication data

4. Business Logic Layer

- Keep **business rules** independent of the API and data layers.
- Implement validation logic for incoming data.

5. Testing & Quality

- Follow **TDD**: write unit tests **before** implementation.
- Cover:
    - Data extraction and processing logic.
    - API endpoints.
    - Business rules.
    - Authentication flows.
- Ensure **high test coverage**.

Deliverables

1. **GitHub Repository** containing:
    - **Source code** for the full project.
    - **Unit tests** for API, business logic, and data handling.
    - **README.md** with detailed setup and execution instructions.
2. **README.md must include:**

    - **Step-by-step setup** for running the project.
    - **API documentation**.
    - **Sample output** of the system.
    - **Scalability considerations**.
    - **Potential improvements & production challenges**.
3. **Dockerized Deployment**
    - Project must be runnable using `docker-compose up` as the only setup step.
4. **Answer this short prompt**:
    - How would you lead an engineering team to implement and maintain this project?

Evaluation Criteria

Category	Description
Clean Architecture	Separation of concerns, modularity, maintainability.
Test-Driven Development	Unit tests for API, business logic, data layer.
Code Quality	Readability, documentation, adherence to best practices.
Functionality	API correctness, data extraction accuracy, ICD-10 mapping.
Scalability & Design	Consideration for large-scale use, error handling.
Dockerization	Ability to launch project using only `docker-compose up`.
Presentation (Interview)	Clear walkthrough of code, choices, and trade-offs.

Bonus Points

- **Data Enrichment**:
  - Implement additional **rules-based logic** for missing fields.
- **Performance Optimization**:
  - Preprocess and **cache structured data** for API efficiency.
  - Allow filtering results dynamically based on parameters.
  - Implement **rate-limiting and security best practices**.
- **Gap Analysis**:
  - What did we not think of?
  - What edge cases did you find?
  - What improvements would you recommend?
  - For each gap:
    - Would you:
      - Adapt the data model?
      - Abstract it at the service or API layer?
    - Why?
    - What are the tradeoffs in performance, maintainability, or team clarity?

Submission & Interview

1. **Submit your GitHub repo link**.
2. **Prepare a Zoom presentation**:
   - Walk through your **user story**, **architecture**, and **technical decisions**.
   - Demo API functionality.
   - Answer code review questions.

Good luck!