Assignment: Full Stack Developer

This assignment is designed to give you an opportunity to showcase your skills across the full technology stack — including frontend development, backend APIs, database modeling, file handling, caching, containerization, and documentation. We are more interested in your **thought process**, **design justifications**, and **quality of implementation** than in having a single "correct" answer.

Please read the problem statement carefully. You will notice some requirements are intentionally open-ended. We expect you to make reasonable assumptions, state them clearly, and justify your architectural and technical decisions.

Problem Statement

A health-tech platform is building a secure and user-friendly portal that allows patients to **upload and manage their medical records** (e.g., prescriptions, diagnostic reports). Clinicians or patients should be able to view, download, or delete previously uploaded files. Each file must be linked to a particular patient and must be retrievable by authorized users only.

Your task is to design and build a **Proof-of-Concept (PoC) full-stack application** to simulate this healthcare document management feature. The system should include a frontend UI, backend API service, file storage mechanism, and a relational database, and should be runnable locally using Docker Compose.

Core Requirements

1. **Frontend Application**
    - Allow a user to upload PDF files (e.g., medical documents)
    - View uploaded records in a list with download & delete options
    - Show progress/loading indicators and success/error messages

2. **Backend API Service**
    - RESTful APIs to upload, download, delete, and list files
    - Each file should be stored and linked with a mock patient ID
    - Generate and store metadata (e.g., upload date, file size, filename)

3. **Database Layer**
    - Use a SQL database to store file metadata (and optionally user info)
    - Structure should support multi-user scenarios
4. **Caching Layer (Optional)**
    - Use Redis or similar to cache recently accessed files/metadata
5. **Security**
    - Implement basic file validation (only accept PDFs)
    - Secure APIs with mock authentication (JWT or session token)
    - Avoid exposing real file paths in APIs

Part 1: System Design and Architecture Document

This is the most critical part of the assignment. Before writing any code, create a concise technical design document.

Your document must include:

1. Tech Stack and Architecture Choice:

- Which frontend and backend frameworks did you choose and why?
- What DB and storage mechanism did you use?
- Provide a high-level architecture diagram
- List the key components and justify your choices

2. Data Flow:

- Describe the full journey from file upload to file download
- Show how metadata and binary files are handled separately

3. API Specification:

- Define at least the following endpoints:
    - `POST /documents/upload`

  - `GET /documents`
  - `GET /documents/:id/download`
  - `DELETE /documents/:id`
- Specify the request/response payloads (JSON + multipart/form-data)

4. Key Considerations (Bulleted List):

- **Scalability**: Can this handle 100k+ uploads?
- **File Storage**: Where are the files stored (e.g., local disk/S3-compatible)?
- **Error Handling**: What if a file is missing or corrupted?
- **Security**: How is the system protected against unauthorized access?

5. CI/CD Pipeline Design:

- Describe the stages of a CI/CD pipeline using GitHub Actions or similar
- Include build, test, lint, and artifact validation steps

6. Infrastructure as Code (IaC) Snippets:

- Provide a `docker-compose.yml` to run all components
- Provide Dockerfiles for frontend and backend
- Show how file storage is mounted

7. Design Justification Questions:

- Why did you choose your file storage approach?
- If this were HIPAA-compliant, what additional changes would you make?
- How would you integrate antivirus scanning into this pipeline?

Part 2: Local Implementation

Implement a simplified, locally runnable version of your design. The goal is to validate full-stack capabilities, including file handling.

Tasks:

1. Application Code:

- **Frontend**:
    - UI to upload a PDF file (max size 10 MB)
    - List all uploaded documents (filename, size, upload date)
    - Option to download or delete individual documents

- **Backend**:
    - REST APIs for document upload, list, download, and deletion
    - Validate file type and size
    - Save file metadata in DB and store files in mounted volume

- **Database**:
    - Store file metadata (filename, storage path, patient_id, created_at)
    - Use SQLite, PostgreSQL, or MySQL

- **Optional**:
    - Cache most recently accessed file metadata using Redis

2. Local Orchestration:

- Provide Dockerfiles for frontend and backend
- Use `docker-compose.yml` to spin up:
    - API service
    - Frontend service
    - Database
    - (Optional) Redis

3. Testing:

- Backend: Write at least 3 unit tests (e.g., file validation, metadata storage)
- Frontend: Optional test for upload or file listing UI

Deliverables

- A link to a private GitHub repository with access granted to the reviewer
- The repository must contain:

1. **Design Document** (from Part 1) in PDF or Markdown format
2. **All source code**, including:
    - `frontend/` directory
    - `backend/` directory
    - Dockerfiles and `docker-compose.yml`
3. **README.md** with:
    - Setup instructions
    - Assumptions made
    - Curl/Postman examples
    - Screenshots (optional)