Design Document – Healthcare App

1. Tech Stack Choices

Q1. What frontend framework did you use and why?

Answer: I used **React** for the frontend.

- **Reasons:**
    - Component-based architecture makes UI reusable and maintainable.
    - Large ecosystem and community support.
    - Excellent developer tooling (React DevTools, Vite, Create React App, ReactQuery devtools).
    - Works well with modern state/data libraries like React Query and Redux for server state management.

Q2. What backend framework did you choose and why?

Answer: I used **Django (with Django REST Framework)** for the backend.

- **Reasons:**
    - Batteries-included framework with ORM, authentication, and admin panel out of the box.
    - Django REST Framework makes it easy to build robust APIs.
    - DRF framework has built in class views and function views which makes so easy to implement any REST API for CRUD actions.
    - Strong security defaults (CSRF protection, SQL injection prevention).
    - Easy to implement JWT based auth with the help of DRF rest framework.
    - Works well with PostgreSQL and supports scaling with minimal changes.
    - Easy to upload Files as just set the Media URL in settings.py

Q3. What database did you choose and why?

Answer: I chose **PostgreSQL**.

- **Reasons:**
    - Powerful relational database with strong ACID compliance.
    - Good indexing and query optimization for large datasets.

- Easy integration with Django's ORM.
- Wide opensource support.
- In Production can be used with lots of cloud service available with one connection string like Neon DB.
- Can access and mutate database with the help of PgAdmin.

Q4. If you were to support 1,000 users, what changes would you consider?

Answer:
If scaling from local/testing to production for 1,000+ users, I would:

1. **Move to a Managed Cloud Environment:**
   - Host backend on AWS with auto-scaling.
   - Use managed PostgreSQL (e.g., Neon DB) for reliability.
2. **Static Media & File Storage:**
   - Store user-uploaded files (PDFs) in **Amazon S3** instead of local /media.
   - Store Media/files in Special Storages like cloudinary.
   - Use a CDN for faster file delivery.
3. **Optimise Files:**
   - Can use background file compressing technique with help of celery and Redis as a queue.
4. **Optimize Database:**
   - Add proper indexes on frequently queried fields.
   - Enable caching (Redis) for repeated API calls.
   - Use proper Joins and query fetching like prefetch and select related to reduce N+1 queries issues.
5. **Container Orchestration:**
   - Use Docker with Kubernetes for deployment and scaling.
6. **Load Balancing:**
   - Introduce Nginx to distribute traffic across backend instances.
7. **Monitoring & Logging:**
   - Use tools like AWS CloudWatch for metrics and ELK stack for logs.
8. **Security Improvements:**

    - Enforce HTTPS everywhere.
    - Use environment variables for all secrets (never commit .env).

2. Architecture Overview

Components

- **Frontend (ReactJS)**: User interface for uploading, previewing, downloading, and deleting PDFs.
- **Backend (Django REST Framework inside Docker container)**: Handles API requests, authentication, file processing, and database interactions.
- **Database (PostgreSQL inside Docker container)**: Stores metadata about users and documents.
- **File Storage (Django media folder inside Docker container)**: Stores uploaded PDF files on disk (under /media/uploads/).

Data Flow

1. **User Interaction on React Frontend**
    - Upload PDF file → POST request with file to Django API
    - View list of uploaded documents → GET request to Django API
    - Preview/download document → GET request for specific document file
    - Delete document → DELETE request to Django API
2. **Backend API (Django REST Framework)**
    - **ListCreateAPIView**:
        - GET returns list of user's documents metadata from PostgreSQL
        - POST accepts PDF file → validates file type → saves file to media folder → stores metadata in DB
    - **RetrieveAPIView (download)**:
        - Checks user permission → opens file in binary mode → streams file back with appropriate headers
    - **DestroyAPIView**:
        - Deletes metadata record and file from storage
3. **PostgreSQL DB**

- Stores document metadata (title, file path, file size, user, upload timestamp)

4. **File Storage**
    - Files saved under /media/uploads/ in Docker container filesystem
    - Served directly by Django backend (not through separate static server)

3. API Specification

Category	**Method**	**Endpoint**	**Description**
Auth	POST	http://localhost:8000/api/auth/login/	Login & get tokens
Auth	POST	http://localhost:8000/api/auth/sign-up/	Register new user
Documents	GET	http://localhost:8000/api/documents/	List all uploaded documents
Documents	POST	http://localhost:8000/api/documents/	Upload a new document (PDF)
Documents	GET	http://localhost:8000/api/documents/<id>/download/	Download document by ID
Documents	DELETE	http://localhost:8000/api/documents/<id>/delete/	Delete document by ID

4. Data Flow Description

File Upload Process

1. **User selects a PDF file** in the React frontend upload form.
2. **Frontend sends a POST request** to the backend API endpoint /api/documents/ with:
    - The PDF file as multipart/form-data
    - Any other metadata (like title)
3. **Backend receives the request** in DocumentListCreateView (a ListCreateAPIView):
    - MultiPartParser parses the multipart data and extracts the file.
    - The view calls perform_create().

4. **Backend validates the file**:
    - Checks if the file extension is .pdf.
    - If invalid, raises a validation error → returns 400 Bad Request to frontend.
5. **If validation passes**:
    - The file is saved to the Django media folder (e.g., /media/uploads/) using FileField.
    - Metadata (file name, size, user, timestamp, etc.) is saved to the PostgreSQL database in the MedicalDocument table.
6. **Backend returns a success response** (usually 201 Created) with the new document's metadata.
7. **Frontend updates the document list UI** to show the newly uploaded document.

File Download Process

1. **User clicks a download or preview button** on a document in the frontend UI.
2. **Frontend sends a GET request** to the backend API endpoint /api/documents/<pk>/ where <pk> is the document's primary key.
3. **Backend receives the GET request** in DocumentDownloadView (a RetrieveAPIView):
    - It authenticates the user.
    - Fetches the MedicalDocument object belonging to the user with pk.
    - If not found, returns 404 Not Found.
4. **Backend opens the file** on disk in binary read mode (rb).
5. **Backend returns a streaming response (FileResponse)**:
    - The file's binary content is streamed to the client.
    - HTTP headers are set:
        - Content-Type: based on the file type (usually application/pdf for PDFs).
        - Content-Disposition: controls whether the browser should display inline (as_attachment=False) or force download (as_attachment=True).
        - Content-Length: size of the file.
6. **Frontend or browser receives the response**:

   - If inline display (as_attachment=False), browser opens the PDF in a new tab or embedded viewer.
   - If forced download (as_attachment=True), browser opens a “Save as…” dialog.
7. User previews or can click download to save it locally.

5.Assumptions

- Only Authenticated users can see their uploaded documents.
- Users need to be sign-in to upload a document
- Users have not to refresh after adding a new doc so implemented react query so whenever new doc is added it will invalidate and fetch in background without need to fresh
- Check pdf file extensions so that user can only upload pdfs.
- Added accept application/pdf in frontend for pdf check and only pdf file allowed.