Tech Stack Choices

Q1. What frontend framework did you use and why?

Ans: For front-end I have chosen **React.js** for its virtual DOM which ensures efficient updates to the file list without reloading the page, and its component-based structure.

Q2. What backend framework did you choose and why?

Ans: I have chosen **Django** (Python) for its built-in security features, ease of database management via ORM, and the speed of setting up REST APIs with DRF. Also comes with preconnected SQLite Database.

Q3. What database did you choose and why?

Ans: SQLite. Chosen because it is file-based, requires no extra database configuration in Django, and is ideal for local development and single-user scenarios.

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

Ans:
1. **Database:** I will Migrate from SQLite to **PostgreSQL** to handle concurrent writes better. Use free alternative (Neon PostgreSQL)
2. **Storage:** Move file upload storage from the local *uploads/* folder to a cloud object storage service like **AWS S3** to ensure scalability and prevent the local server disk from filling up.
3. **Auth:** Implement user authentication (**JWT** or **Session-based**) so users can only see their own files.

Architecture Overview

- **Frontend**: React + Tailwind CSS (*for the user interface*).
- **Backend**: Django + Django REST Framework (*for API and file handling*).
- **Database**: SQLite (*Default Django DB for metadata*).
- **Storage**: Local file system storage (*files saved in an uploads/ folder*).

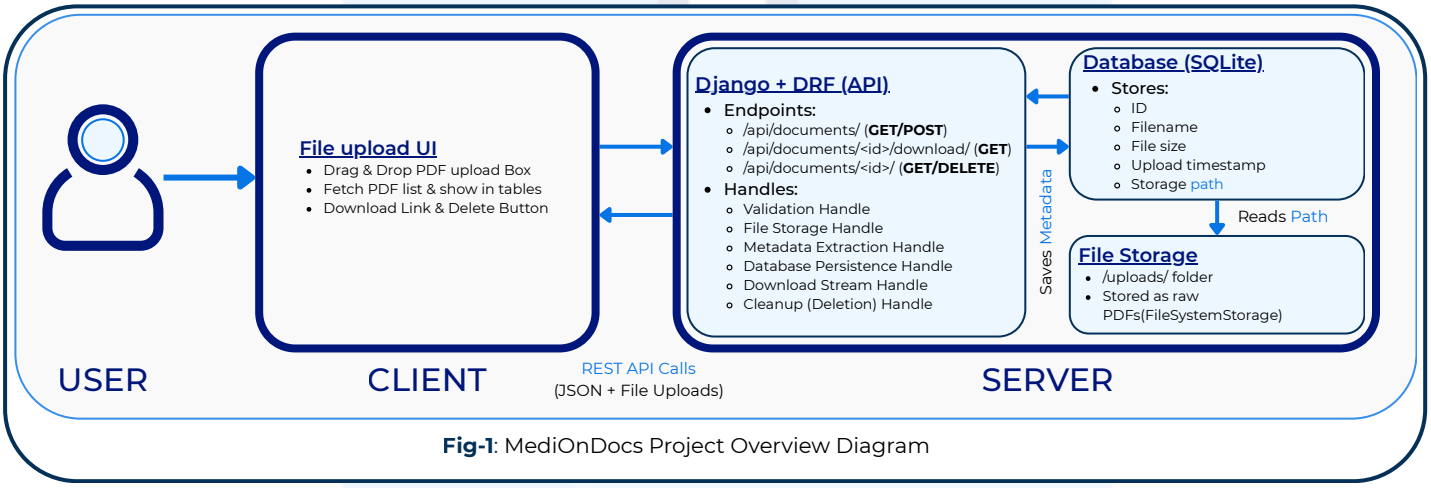

Fig-1: MediOnDocs Project Overview Diagram

API Specification

Method	Endpoint	Description
GET	/	backend server home page with front-end server & GitHub repository link
GET	/api/documents/	**List all files**. Returns a JSON array of all uploaded document metadata. 1
POST	/api/documents/	**Upload a file**. Accepts multipart/form-data with a file key. Validates that the file is a PDF.
GET	/api/documents/<id>/download/	**Download file**. Streams the actual PDF file to the browser with Content-Disposition: attachment.
DELETE	/api/documents/<id>/	**Delete file**. Removes the record from the database and deletes the file from the uploads/ folder.
GET	/api/documents/<id>/	**Retrieve Metadata**. Returns the JSON metadata (filename, size, date) for a specific document ID.

1. List All Documents

Sample Request:

GET /api/documents/

Sample Response:

2. Upload Documents

Sample Request:

Headers:

Content-Type: multipart/form-data

Body:

file: <selected PDF file>

Sample Response:

Status Code:

201 created

3. Download a Documents

Sample Request:

GET /api/documents/25/download/

Headers:

Content-Type: application/pdf
Content-Disposition: attachment; filename="assignment.pdf"

Sample Response: (file stream)

Binary PDF file data

4. Delete Documents

Sample Request:

DELETE /api/documents/12/

5. Retrieve Metadata of a Single Document

Sample Request:

GET /api/documents/1/

Sample Response:

Data Flow Description

Q5. Describe the step-by-step process of what happens when a file is uploaded and when it is downloaded.

Ans:

Upload Flow

When a user uploads a document through the MediOnDocs interface, the system follows this complete flow:

1. **Client selects a PDF file**
   The user browses their device or uses the drag-and-drop area on the React frontend to choose a PDF.
   The selected file is temporarily held in the browser before upload.
2. **A POST request is sent to the backend**
   Once the user clicks Upload, the frontend sends the file to the Django API using a multipart/form-data POST request.
   This ensures binary data (PDF) and metadata (file name, size) are transmitted correctly.
3. **Backend validates the file (PDF-only check)**
   Django REST Framework receives the file and first performs validation steps:
   - Ensures a file is actually provided
   - Confirms the file has a .pdf extension
   - If validation fails, the backend returns an appropriate error response.
4. **File is saved to the server's file system**
   After validation passes, Django uses its file storage system to write the file into the designated folder (/media/uploads/).
   The backend automatically handles file naming, collisions, and storage paths.
5. **Metadata is stored in the database**
   Once the physical PDF is stored, the backend extracts details like:
   - Original filename
   - File size (in bytes)
   - Storage path
   - Upload timestamp

   These details are saved as a new entry in the SQLite database.
   This allows the frontend to list, download, and manage the file later.
6. **A success response is returned to the frontend**
   The backend sends a response confirming the upload was successful.
   The React frontend updates the file list and displays the newly uploaded document immediately in the table.

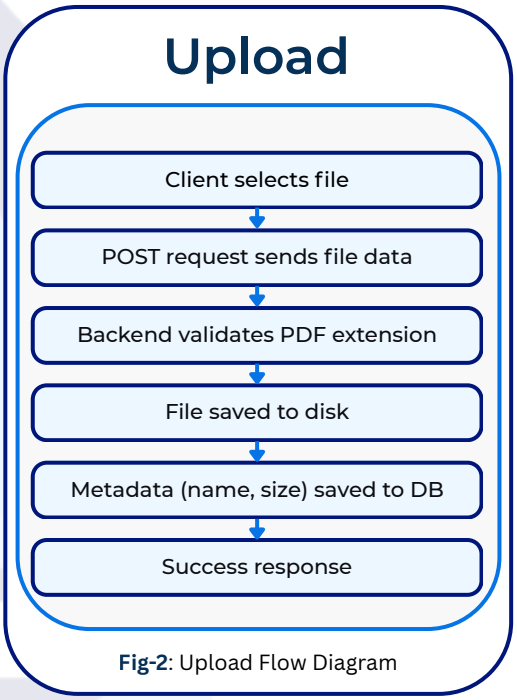

Fig-2: Upload Flow Diagram

Download Flow

When a user wants to download a file in MediOnDocs, the system follows a straightforward process:

1. **The user clicks download / requests the download link**
   From the React frontend, a request is sent to the Django backend. This request contains the ID of the file the user wants to download.
2. **The backend checks the database for that file**
   Django uses the provided ID to look up the file's information in the SQLite database. The database stores details like the file name, upload date, and—most importantly—the exact path where the file is saved on the server.
   If the record isn't found, the backend returns an error message.
3. **The backend locates the actual file on the server**
   Once the database confirms the path, Django reads the PDF from the disk (inside the uploads/ folder). This ensures the correct file is being accessed.
4. **The backend sends the file back to the user as a download**
   Django then streams the file back to the client instead of loading it all at once.
   It sets a special header called Content-Disposition: attachment.
   The browser receives it and automatically starts downloading the PDF with its original name.

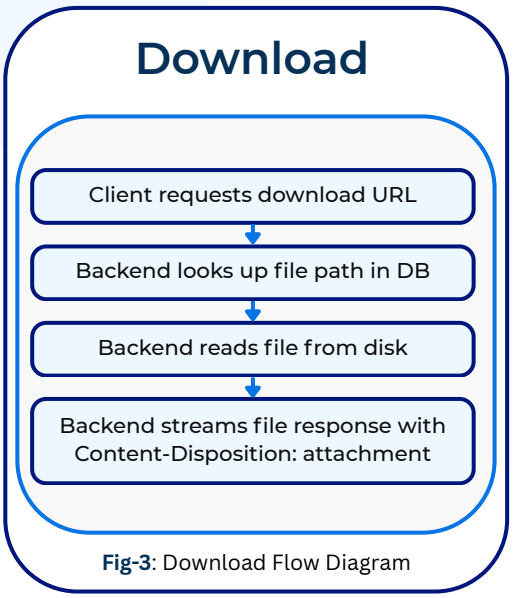

Fig-3: Download Flow Diagram

Assumptions

Q6. What assumptions did you make while building this? (*e.g., file size limits, authentication, concurrency*)

Ans: While building this assessment project, I made a few practical assumptions to keep the implementation simple and focused on the core file upload/download functionality. Since the goal was to demonstrate understanding of React, Django, REST APIs, and basic file handling, I avoided adding heavier production-level features.

No File Size Limit Implemented

I assumed that the files uploaded by the user would be reasonably small (e.g., standard PDF documents). For the assessment, I did not implement:

- Maximum upload size validation
- Large file streaming or chunk uploads

In a production system, I would configure server-side limits (e.g., 10–20 MB) and frontend validation.

No Authentication or User Accounts

To keep the project simple, I assumed an open API where any user can upload, view, download, or delete files. There is no:

- Login / signup system
- Role-based access control
- Token verification (JWT/OAuth)

For a real app, I would integrate OAuth or email-password authentication, encrypted password storage, and JWT-based API access.

Minimal Concurrency & Race Condition Handling

I assumed that the application would be used by a very small number of users, probably one at a time, for testing. So I did not implement:

- File locking
- Concurrent upload handling

In a real cloud deployment, I would add concurrency-safe storage and background workers (e.g., Celery).

Local File Storage Instead of Cloud Storage

For the assessment, I used local disk storage as mention in the requirement (uploads/ folder) is sufficient. No integration with:

- AWS S3
- GCP Storage
- Azure Blob

In a real distributed application, I would use an S3 bucket with proper access policies and signed URLs.

SQLite Used as Lightweight Database

I assumed SQLite is good enough for storing simple metadata during testing. For a production environment, I would migrate to PostgreSQL (e.g., Neon DB) for:

- Scalability
- Better indexing
- Support for concurrent requests

Basic Validation Only (PDF check)

I only implemented a simple file-type validation (checking for .pdf). No extra checks like:

- Malware scanning
- File structure validation
- PDF encryption support

These can be included later if security becomes a requirement.

Basic Error Handling

I kept the error handling simple and readable. I assumed the frontend and backend would operate in a predictable manner during evaluation.

Short Summary

This project is intentionally built in a minimal, clean, assessment-friendly way without complex production features. If I were building this as a real-world application, I would use:

- **Frontend**: React (Vercel)
- **Backend**: Django + DRF (Render/Fly.io)
- **Database**: PostgreSQL (Neon)
- **Storage**: AWS S3
- **Authentication**: JWT + OAuth + Email/Password
- **Security**: File size checks, virus scan, rate limiting
- **DevOps:** CI/CD + environment variables