

INI8 LABS Assignment

Name: Aman Joharapurkar

Degree: B.Tech in Computer Sciecne and Engineering (CGPA: 9.49)

Year of Passing: 2025

Github Repository URL:

Frontend: <https://github.com/AmanJ10/pdf-upload-application-frontend>

Backend: <https://github.com/AmanJ10/pdf-upload-application-backend>

1. Tech Stack Choices

Q1. What frontend framework did you use and why?

I used **React** for the frontend. React's component-based architecture made it easy to build reusable UI components and maintain a clean structure. It efficiently manages UI state, handles routing effectively with React Router, and integrates smoothly with RESTful APIs.

Q2. What backend framework did you choose and why?

I chose **Express.js** for the backend because it is lightweight, fast which gives me full control over how I structure the API. Express integrates seamlessly with Node.js, providing a clean way to build RESTful APIs. It also has a large ecosystem of middleware (like **Multer** for file uploads and CORS for security), which made it easy to implement features such as document upload and routing.

Also, both the frontend and backend are written in a single language (JavaScript).

Q3. What database did you choose and why?

I chose **PostgreSQL** as the database for this project because it is a powerful, reliable, and production-grade relational database. It supports strong data consistency, ACID transactions.

Prisma is a next-generation Object-Relational Mapper (ORM) that simplifies database interactions by providing a type-safe query builder and an intuitive API for Node.js and TypeScript applications. Instead of writing raw SQL, developers use JavaScript functions to

interact with the database, and Prisma handles the conversion to SQL, improving developer productivity and preventing errors like SQL injections.

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

To support 1,000+ users, I would focus on improving scalability, performance, and reliability across the stack:

1. Backend Scaling

- Move the Express server behind a **reverse proxy** like Nginx.
- Enable **horizontal scaling** by running multiple Node.js instances.
- Add **rate limiting** and **request validation** to protect the API from overload.
- Add **load balancing** so requests are evenly distributed. For this, we can use Nginx too, and use an appropriate algorithm of load balancing like Round Robin or FCFS (First Come First Served).

2. Database Optimisation

- To support 1,000+ users, I would add indexes on frequently queried columns (e.g., `id`, `createdAt`, `userId` etc. in future) to speed up lookups. This reduces full table scans, improves query performance, and lowers database load.
- The N+1 problem occurs when the system performs one main query and then executes an additional query for every result returned. For example, fetching 100 documents and then executing 100 separate queries to fetch each document's user. This results in 101 queries instead of a single optimized JOIN. Avoiding N+1 queries improves performance and scalability.

3. File Storage Optimisation (for PDFs/images)

- Move uploaded files from local storage to a cloud service like:
 - **AWS S3**,
 - Firebase Storage, or
 - Google Cloud Storage.
- This reduces server load and improves global access speed.

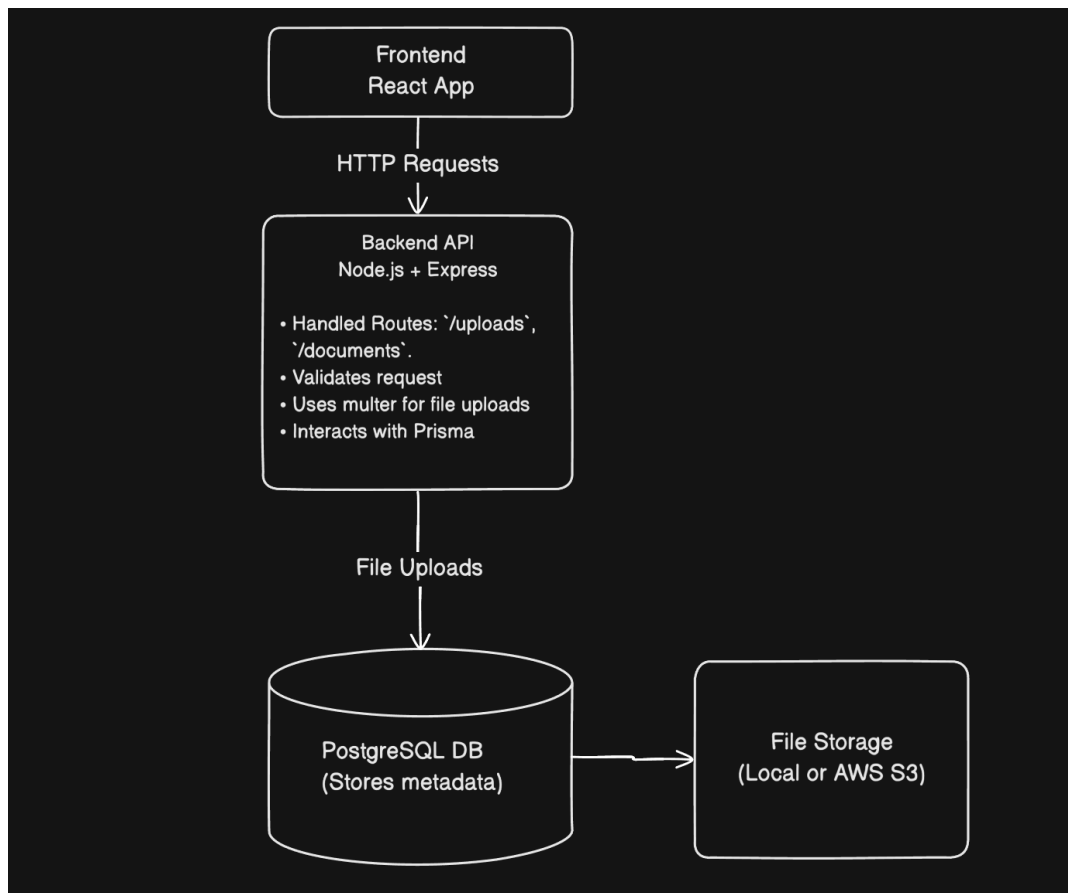
4. Caching

- Introduce **Redis caching** for frequent data retrieval.
- Cache metadata of documents or user data to reduce load on Postgres.
- Also, ensure there is no stale data either in the DB or Redis.

5. Frontend Improvements

- Use **code-splitting** and **lazy loading** to speed up initial load.
 - Optimise API calls and cache responses with React Query/RTK Query.
-

2. Architecture Overview (made using Draw.io)



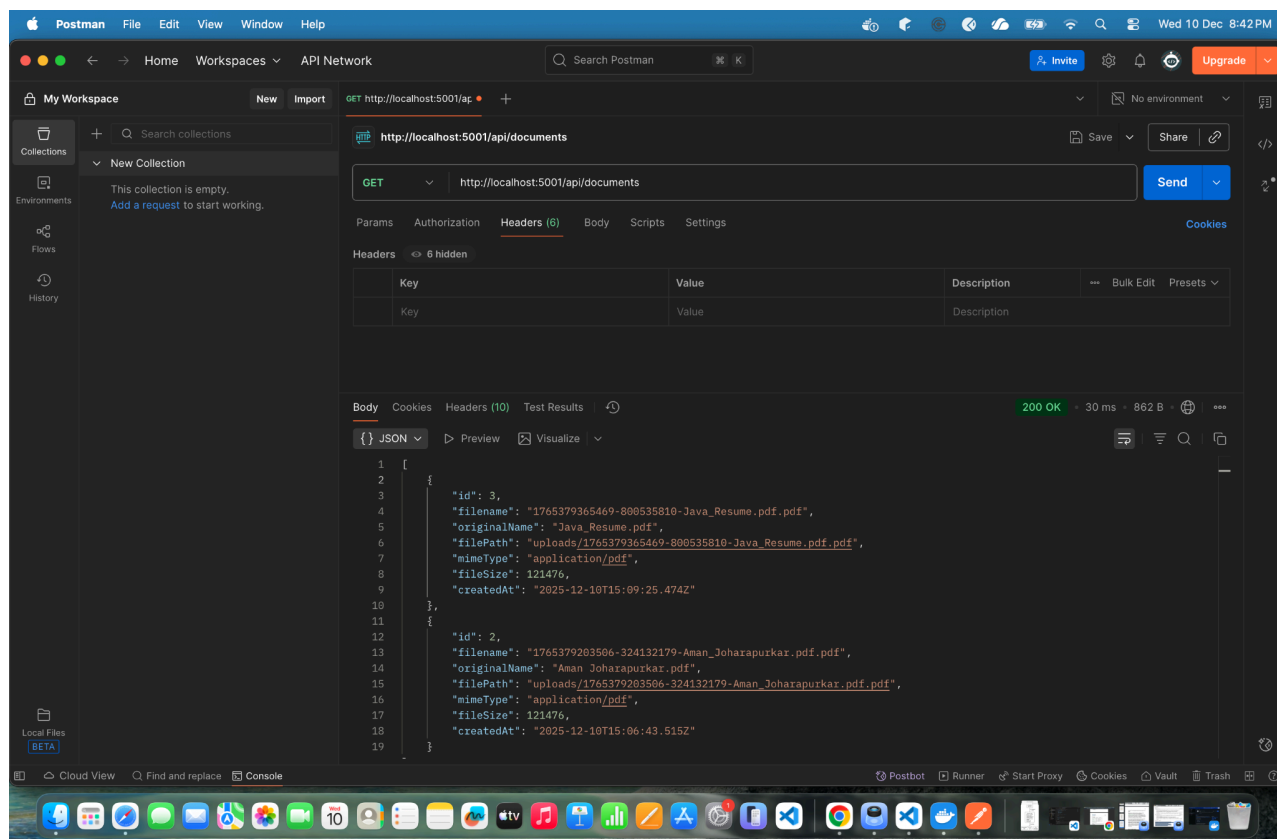
3. API Specification

1.) GET Documents:

API Endpoint: <http://localhost:5001/api/documents>

Method: GET

Sample Request & Response (using Postman):



2.) Upload a Document:

API Endpoint: <http://localhost:5001/api/documents>

Method: POST

Sample Request & Response (using Postman):

The screenshot displays the Postman application interface. The top bar shows the menu (Postman, File, Edit, View, Window, Help) and system status (Wed 10 Dec 8:40 PM). The left sidebar contains 'My Workspace' with a 'New Collection' button and a search bar. The main panel shows a POST request to `http://localhost:5001/api/documents/upload`. The 'Body' tab is selected, showing a 'form-data' type with a single field: 'file' with value 'Java_Resume.pdf' and content-type 'Auto'. The 'Test Results' tab shows a successful response with status '201 Created', time '17 ms', and size '631 B'. The response body is a JSON object:

```
{
  "message": "Uploaded",
  "document": {
    "id": 3,
    "filename": "1765379365469-800535810-Java_Resume.pdf.pdf",
    "originalName": "Java_Resume.pdf",
    "filePath": "uploads/1765379365469-800535810-Java_Resume.pdf.pdf",
    "mimeType": "application/pdf",
    "fileSize": 121476,
    "createdAt": "2025-12-10T15:09:25.474Z"
  }
}
```

The bottom of the screen shows the macOS dock with various application icons.

3.) Get a Particular Document (Download):

API Endpoint: <http://localhost:5001/api/documents/{id}>

Method: GET

Sample Request & Response (using Postman):

The screenshot shows the Postman application interface. The top bar includes the Postman logo and menu items: File, Edit, View, Window, Help. The main workspace is divided into several sections:

- Left Sidebar:** Contains 'My Workspace' with a 'New Collection' button and a search bar. Below it are 'Environments', 'Flows', and 'History'.
- Top Bar:** Includes a search bar, 'API Network' tab, and buttons for 'Invite', 'Upgrade', and 'Send'.
- Request Section:** Shows a GET request to `http://localhost:5001/api/documents/3`. The 'Headers' tab is selected, showing 6 hidden headers.
- Response Section:** The 'Body' tab is selected, showing a 200 OK status with a response time of 17 ms and a size of 119.1 KB. The response body is a resume for Aman Joharapurkar.

Resume Content:

Aman Joharapurkar
+91-8237849163 | joharapurkaraman95@gmail.com | Portfolio | GitHub | LinkedIn | LeetCode

EDUCATION

Institution	Location	Graduation Date
Shri Ramdeobaba College of Engineering and Management	Nagpur	May 2025
B.Tech in Computer Science and Engineering (CGPA: 9.49)		
Centre Point School Katol Road	Nagpur	July 2021
XII: Percentage: 87.60		
Centre Point School Amravati Road	Nagpur	May 2019
X: Percentage: 92.30		

EXPERIENCE

Software Development Intern — Ceinsys Tech Ltd Jan 2025*

Nagpur

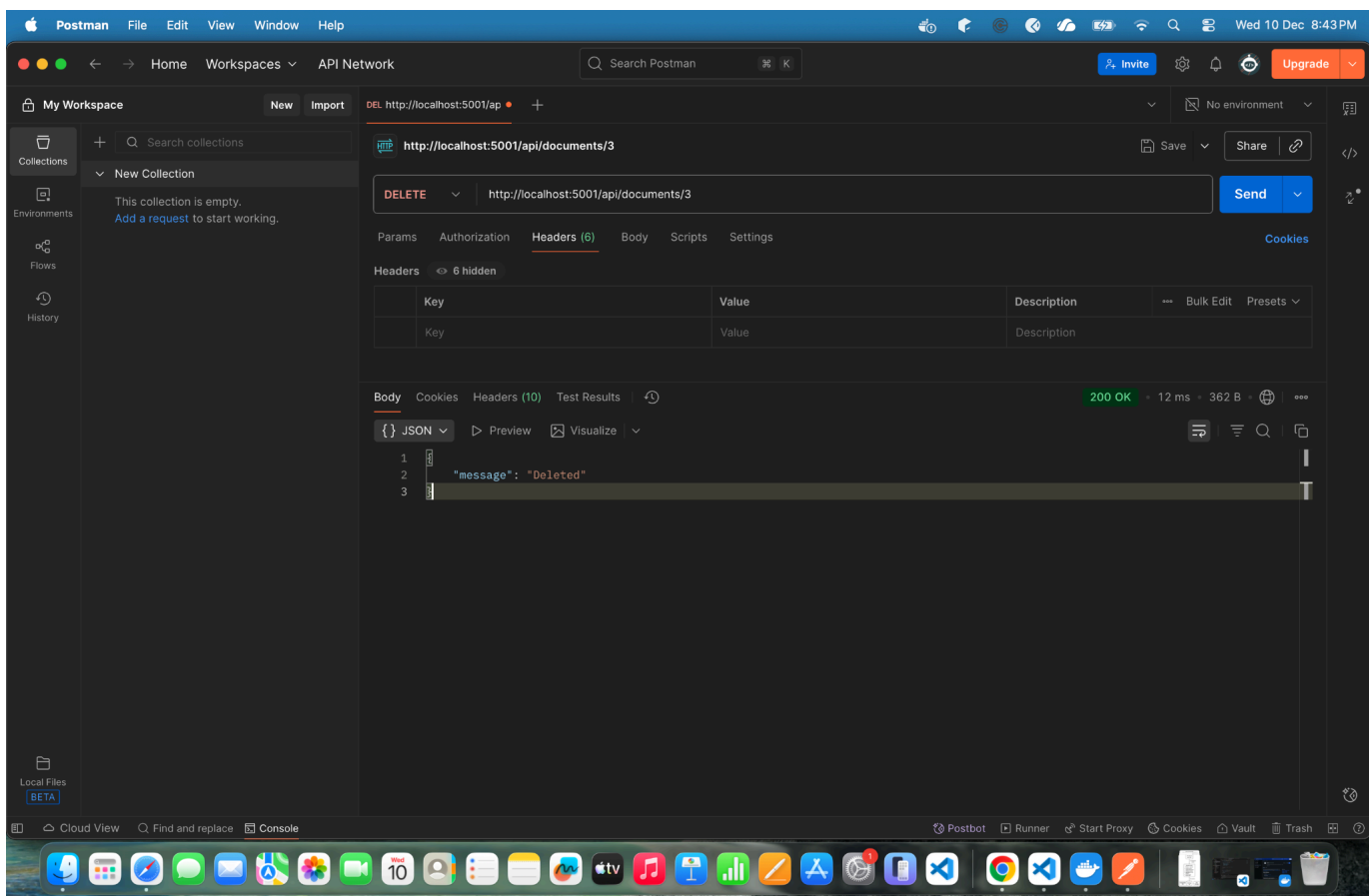
- Contributing to the **IDTS (Integrated Digital Transformation System)** by developing role-based dashboards and integrating **Single Sign-On (SSO)** using Keycloak for MHADA officials.
- Developed the **User Management module** by implementing a Tree Menu structure for assigning applications and menu pages to different user roles.
- Built the **Electricity Bill Payments module** using ReactJS, MUI, and CSS — designed 7-8 role-specific pages with dynamic form components to automate manual workflows.
- Collaborating in an **AWS** environment using Terra and GitLab for task management and version control.

4.) Delete Document:

API Endpoint: <http://localhost:5001/api/documents/{id}>

Method: DELETE

Sample Request & Response (using Postman):



4. Data Flow Description

File Uploading:

- 1.) User selects a file on the React frontend.

The frontend sends a `multipart/form-data` POST request to:

POST /documents/upload

2.) Express receives the request.

The `/upload` route is protected by multer middleware:

`upload.single("file")`

Multer extracts the file from the request and stores it in your configured upload folder (`uploads/`).

3.) Multer attaches file info to `req.file`.

```
{  
  "filename": "abc123.pdf",  
  "originalname": "report.pdf",  
  "path": "uploads/abc123.pdf",  
  "mimetype": "application/pdf",  
  "size": 204800  
}
```

4.) Controller `uploadDocument` runs.

Reads metadata from the `req.file`

Saves metadata to PostgreSQL using Prisma:

```
await prisma.document.create({  
  data: {  
    filename,  
    originalName,  
    mimeType,  
    size,  
    path  
  }  
})
```


}}

5.) The database stores only metadata.

id	originalName	fileName	filePath	contentType	size	path	createdAt
----	--------------	----------	----------	-------------	------	------	-----------

6.) API returns a success response to the frontend:

```
{
  "message": "Uploaded",
  "document": {
    "id": 4,
    "filename":
"1765422247761-573017703-Aman_Joharapurkar-Internship_Report.pdf.pdf",
    "originalName": "Aman Joharapurkar-Internship Report.pdf",
    "filePath":
"uploads/1765422247761-573017703-Aman_Joharapurkar-Internship_Report.pdf.pdf",
    "contentType": "application/pdf",
    "fileSize": 1718877,
    "createdAt": "2025-12-11T03:04:07.786Z"
  }
}
```

7.) Frontend updates UI (refreshes list of documents).

File Download:

1.) The user clicks Download in the frontend.

A GET request hits:

GET /documents/:id

2.) The Express route calls:

prisma.document.findUnique({ where: { id } })

3.) Database returns file metadata, including file path.

4.) The backend checks whether the file exists on disk.

5.) Express sends the actual file using:

res.download(filePath, originalName)

6.) The browser receives the file and triggers a download.

5. Assumptions

While designing and implementing the file upload & document management system, I made the following assumptions:

- 1.) Only one user:
 - a.) No login/logout authentication system.
- 2.) File Types Allowed:
 - a.) Only PDFs are allowed. No JPG, PNG files.
- 3.) Storage:
 - a.) Used Local storage for saving the uploaded images.
 - b.) In production, and to optimise file storage, we can use AWS S3, Firebase or cloud storage.
- 4.) Validation:
 - a.) File size limit is 10 MB. User cannot upload a file which is more than 10MB.
 - b.) User cannot upload multiple files at the same time.

