COLLEGE CODE : 8203

COLLEGE NAME : AVC COLLEGE OF ENGINEERING

DEPARTMENT : INFORMATION TECHNOLOGY

STUDENT NM-ID : 7F21192CA579AE488C54DD3A90E6D349

ROLL NO : 23IT71

DATE :22.09.2025

Completed the project named as Phase III
technology project name :File Upload Manager

SUBMITTED BY,

NAME :Naveen R

MOBILE NO: 8610226879

Project Setup :

MVP implementation

Project setup:

The first step is to establish the foundation and connect all the required services.

Step	Description	Tools Involved
1. Project Initialization & Dependencies	Create the main project folders and initialize the Node.js backend. Install essential packages for the server, file handling, and MongoDB storage.	**`npm init -y`**, **`npm install express mongoose multer multer-gridfs-storage dotenv`**
2. Environment Configuration	Create a **`.env`** file in the backend directory to securely store the critical connection string: **`MONGO_URI`**.	**`dotenv`**, MongoDB Connection String
3. Server Initialization	Create `server.js` to set up the **Express.js** application, define the server port (e.g., **`PORT=5000`**), and configure middleware and the **MongoDB connection**.	**Node.js, Express.js, Mongoose**
4. GridFS Storage Setup	Implement the configuration logic for **Multer** to use **`multer-gridfs-storage`**, directing all uploads into MongoDB's **GridFS** bucket (e.g., `uploads`).	**`multer-gridfs-storage`**, **Crypto**, **Path**
5. Frontend File Structure	Create the core UI files in the frontend folder: `index.html` (structure), `style.css` (design), and `app.js` (client-side API calls to the backend).	**HTML, CSS, JavaScript**

Data Storage (MongoDB/GridFS)

Unlike systems that use a database for metadata and a separate file system for content, GridFS stores both file **metadata** and **binary content** directly within the MongoDB database.

1. **Files Collection (uploads.files):** This collection stores the **metadata** associated with each uploaded file. Key metadata fields include:
    - The unique file ID (_id).
    - The filename.
    - The size (length).
    - The content type (MIME type).
    - The upload date.
2. **Chunks Collection (uploads.chunks):** This collection holds the **actual binary content** of the file. GridFS splits large files into smaller data chunks (usually 255KB) and stores each chunk as a separate document in this collection.

Role of Multer and GridFS Storage Engine

The core functionality of storing the data is managed by **Multer** and the **multer-gridfs-storage** engine.

- When a user submits a file via the frontend, Multer intercepts the multipart/form-data.
- The multer-gridfs-storage engine takes the incoming file stream and handles the process of automatically chopping the file into chunks and persisting these chunks, along with the file's metadata, into the two GridFS collections in MongoDB.

No Caching Layer

The File Upload Manager **does not** use a separate caching database (like Redis, which was used in the Weather Dashboard). Since the files are stored directly in a performant databas

e (MongoDB/GridFS), a dedicated caching layer is generally unnecessary for the MVP implementation.

Core Features of Implementation

This File Upload Manager involves integrating **Express, Multer, and GridFS** to handle the complete file lifecycle: upload, retrieval, listing, and deletion. This is managed through a set of dedicated REST API endpoints.

1. File Upload (POST)

This feature uses the multer-gridfs-storage engine to stream incoming files directly into MongoDB.

- **Endpoint:** POST /api/files/upload
- **Functionality:**
  - The backend receives the file via multipart/form-data.
  - **Multer** middleware handles the file and directs the stream to the **GridFS storage engine**.
  - The file is stored in MongoDB's **GridFS** (split into files and chunks collections).
- **Response:** Returns a 201 Created status with the file's **metadata** (ID, filename, size, etc.) and its **retrieval URL**.

---

2. File Listing (GET Metadata)

This feature allows clients to retrieve a list of all stored files and their basic information.

- **Endpoint:** GET /api/files
- **Functionality:**
  - The server uses the **GridFS Bucket** instance to query the uploads.files collection.
  - It retrieves the metadata for all files.

    - **Error Handling:** Returns 404 Not Found if no files are present.
- **Response:** Returns a 200 OK status with a **JSON array** of file metadata, including the retrieval URL for each file.

---

3. File Retrieval (GET Stream)

This feature enables downloading or viewing the actual file content.

- **Endpoint:** GET /api/files/:filename
- **Functionality:**
    - The server finds the file in GridFS using the provided **filename**.
    - It sets the correct Content-Type header (e.g., application/pdf, image/jpeg).
    - The **GridFS Bucket** is used to open a download stream (openDownloadStreamByName), piping the file content directly back to the client.
- **Response:** Returns the raw **file content** (a stream).

---

4. File Removal (DELETE)

This feature allows for the permanent removal of a file from storage.

- **Endpoint:** DELETE /api/files/:id
- **Functionality:**
    - The server receives the unique file **ID** (_id) as a URL parameter.
    - The **GridFS Bucket's delete()** method is called using the file ID.
    - GridFS automatically removes both the **metadata** from uploads.files and all associated **chunks** from uploads.chunks.
- **Response:** Returns a 200 OK status with a success message confirming the deletion.

Testing Core Featuers

1. **API Functionality Test (Postman / cURL)**

Ensure all REST endpoints (/upload, /files, /download/:filename, /files/:id) function correctly.

- **Test Case 1: Successful File Upload**
  - **Action**: Send POST /upload with a valid file (form-data: key=file).
  - **Expected Result**:
    - HTTP Status: 200 OK
    - JSON response contains filename, _id, contentType, and uploadDate.
    - File appears in MongoDB (uploads.files & uploads.chunks).

Test Case 2: Invalid File Upload

- **Action**: Upload a file that exceeds size limit or unsupported type.

Test Case 3: Fetch All Files

- **Action**: Send GET /files.

Test Case 4: Fetch Single File Metadata

- **Action**: Send GET /files/:filename.
  - If file does not exist → 404 File not found.

Test Case 5: Download File

- **Action**: Send GET /download/:filename.

Test Case 6: Delete File

- **Action**: Send DELETE /files/:id with a valid MongoDB _id.
- **Expected Result**:
  - HTTP Status: 200 OK
  - Response: { "msg": "File deleted successfully" }.
  - File no longer exists in uploads.files.
  - If invalid ID → 404 File not found.

2. Database Test (MongoDB Compass / Shell)

Goal: Validate that files are stored, retrieved, and deleted properly.

- **Test Case 7: File Storage Validation**
  - After upload, check **MongoDB Compass** → uploads.files contains file metadata, uploads.chunks contains split data.
- **Test Case 8: File Deletion Validation**
  - After DELETE /files/:id, verify both uploads.files and uploads.chunks records are removed.

3. Frontend Test (Browser UI)

Goal: Ensure the HTML + JS interface works with APIs.

- **Test Case 9: Upload via Browser**
  - Select file in index.html form and click upload.
- **Test Case 10: Fetch & Display Files**
  - Click "Get Files" button.
- **Test Case 11: Download from Browser**
  - Click "Download" link.
- **Test Case 12: Delete from Browser**
  - Click "Delete" button next to a file.

Version Control (GitHub)

Repository is maintained for project integrity and collaboration.

- **Commit Strategy**: Separate commits for setup, backend API, frontend integration, testing.
- **Branching**: main for stable releases, dev for development.

https://github.com/Naveen05071/NM-IBM-AVCCE-NAVEEN.git