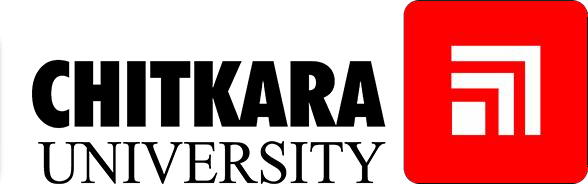
Backend Engineering Project Based Evaluation



Project Report Semester-V (Batch-2023)

## Supervised By: Submitted By:

Dr. Pummy Dhiman Sushobhit Jain (2310992539)

Assistant Professor Soni (2310992534)

Tejasv (2310992547)

Sohini (2310992533)

**Department of Computer Science and Engineering Chitkara University Institute of Engineering & Technology**

**Chitkara University, Punjab**

# Abstract

In today’s digital era, the need for secure, scalable, and efficient file storage and sharing solutions has become increasingly critical. While major platforms like Google Drive and Dropbox offer comprehensive cloud storage, they are often complex, resource-intensive, and not easily customizable for small-scale or educational projects. This project addresses this gap by developing a **lightweight backend system** that allows users to upload, store, share, and manage files securely, effectively serving as a mini version of Google Drive.

The backend is built using **Node.js and Express.js**, with **MongoDB** as the database and **Mongoose ODM** for object modeling. File uploads are handled via **Multer**, with **GridFS** used for large file storage, and optionally, cloud storage integration through **AWS S3 or Cloudinary** for production-ready deployments. Security is a key focus: user authentication is implemented using **JWT tokens**, while **bcrypt** ensures that passwords are safely hashed. This ensures that only authorized users can access and manage files.

A central feature of the system is the **role-based file sharing mechanism**. Users can share files with other registered users using their email addresses, and access permissions are strictly enforced. File owners retain full control, including editing, deleting, and sharing, while shared users are typically restricted to viewing and downloading files. This mimics real-world **role-based access control (RBAC)**, highlighting practical security practices in backend systems.

The project also includes **search and filter functionalities**, allowing users to locate files by name, type, or upload date, enhancing usability for large collections. Metadata such as file size, type, owner, and creation date is stored in MongoDB to support efficient querying and management. Comprehensive **error handling and validation** ensures that only permitted file types and sizes are uploaded, and unauthorized access is prevented.

Deployment considerations are addressed through **MongoDB Atlas** for database hosting and cloud platforms like **Render or Heroku** for backend deployment. The RESTful API design allows for easy integration with frontend frameworks such as React or Angular, making it suitable for full-stack expansion.

Overall, this project demonstrates a **practical implementation of secure file management, sharing, and access control**, providing hands-on experience with key backend concepts. While not as feature-rich as enterprise solutions, it effectively bridges the gap between basic file upload systems and full-scale cloud storage, making it highly relevant for small teams, educational purposes, and portfolio development.

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Topic Name** | **Page No.** |
| 1 | Introduction   * 1.1 Project Background * 1.2 Problem Statement * 1.3 Objectives and Goals * 1.4 Scope of the Project | 4 - 5 |
| 2 | Requirement Analysis  -2.1 Functional Requirements  -2.2 Non- Functional Requirements | 6 - 7 |
| 3 | System Architecture  -3.1 Client - Server Architecture  -3.2 Technology Stack Overview  - 3.3 System Components | 8 |
| 4 | Implementation   * 4.1 Project Structure * 4.2 Backend Implementation * 4.3 Frontend Implementation | 9 |
| 5 | Database Design | 10 |
| 6 | Key Features | 11 |
| 7 | Project Glimpse | 12 |
| 8 | Conclusion | 13 |
| 9 | References | 14 |

**1. Introduction**

**1.1 Project Background**

In the digital era, the demand for secure and efficient file storage and sharing solutions has increased significantly. Individuals, small teams, and organizations frequently rely on cloud-based platforms to manage documents, images, PDFs, videos, and other digital resources. While established services like Google Drive and Dropbox provide comprehensive solutions, they are often complex, resource-intensive, and not easily customizable for small-scale projects or educational purposes. This creates a need for a **lightweight, flexible, and secure backend system** that can provide essential file management and sharing functionalities. The File Storage and Sharing Service project is designed as a **mini version of Google Drive**, focusing on core backend functionalities such as authentication, file handling, sharing, and access control.

**1.2 Problem Statement**

Existing file upload systems for small projects or teams usually provide only basic storage without features like authentication, access control, or file sharing. Users face challenges in ensuring **data security, controlled access, and efficient file management**. There is also a lack of customizable solutions that demonstrate real-world backend concepts such as role-based access control, metadata management, and cloud integration. Therefore, the problem is to design and implement a backend system that enables users to securely upload, store, and share files with controlled permissions, while providing search and filtering functionalities, ensuring usability, security, and scalability.

**1.3 Objectives and Goals**

The main objectives of the project are:

* To build a **secure authentication system** using JWT and password hashing with bcrypt.
* To implement a **robust file upload and storage system** using Multer, GridFS, and optional cloud storage like AWS S3.
* To enable **file sharing** with role-based access control, allowing owners full permissions while shared users have limited access.
* To provide **search and filtering capabilities** for efficient file management.
* To design a **RESTful API** that can be integrated with frontend applications.
* To ensure **data integrity, security, and validation** throughout the system.

**1.4 Scope of the Project**

The project focuses on backend development for file storage and sharing. Key features include secure user registration and login, uploading, downloading, and deleting files, sharing files with other users, and role-based access management. Search and filtering functionalities allow users to manage files efficiently. While the system does not include advanced features like real-time collaboration or file versioning, it serves as a **practical and scalable foundation** for developing a full-fledged cloud storage solution. It is suitable for small teams, educational projects, and portfolio demonstrations, showcasing real-world backend development practices.[1]

**2. Requirement Analysis**

**2.1 Functional Requirements**

Functional requirements describe the **core functionalities** that the File Storage and Sharing Service must provide to users. These requirements define how the system should behave in various scenarios:

* **User Authentication:**
  + Users must be able to **register** with unique email IDs and secure passwords.
  + Users should be able to **log in** and obtain a **JWT token** for secure session management.
  + Passwords must be **hashed using bcrypt** to ensure security.
* **File Upload and Storage:**
  + Users can **upload files** of different types such as PDFs, images, videos, and documents.
  + Files are stored using **MongoDB GridFS** for handling large files, with optional integration to **AWS S3 or Cloudinary** for production-level storage.
  + Each file stores metadata including filename, file type, size, owner, shared users, and timestamps.
* **File Management:**
  + Users can **view file details**, **download files**, and **delete files** (delete permitted only for the file owner).
* **File Sharing:**
  + File owners can **share files with other registered users** using email IDs.
  + Shared users can **view and download files**, but cannot delete or re-share them.
  + A “**Shared With Me**” section displays all files shared with a user.
* **Search and Filtering:**
  + Users can **search files** by filename, file type, or upload date.
  + Filtering allows users to quickly locate files by type (e.g., PDFs, images, videos).
* **Access Control:**
  + Role-based access ensures that only the **owner has full permissions**, while shared users have **restricted access**.
  + File permissions are checked before every action such as download or delete.
* **Error Handling and Validation:**
  + File size and type must be validated before uploading.
  + Unauthorized access or invalid operations return proper error messages.

**2.2 Non-Functional Requirements**

Non-functional requirements define **how the system performs** and other quality attributes:

* **Security:**
  + Secure authentication with JWT and hashed passwords ensures protection against unauthorized access.[1][2]
  + Access control mechanisms prevent users from performing actions beyond their permissions.
* **Performance:**
  + The system should support **efficient file uploads and downloads**, even for large files.
  + API responses must be optimized for minimal latency.
* **Scalability:**
  + The system should support **multiple concurrent users** and large volumes of files.
  + Cloud storage integration ensures the system can scale as data grows.
* **Reliability:**
  + Data should remain **consistent and intact** during file operations.
  + Proper error handling ensures that failures do not corrupt data.
* **Maintainability:**
  + The backend should follow a **modular structure** with RESTful APIs, allowing easy updates and integration with frontend applications.
* **Usability:**
  + APIs should be **well-documented and intuitive**, facilitating integration and usage by developers.

**3. System Architecture**

**3.1 Client-Server Architecture**

The File Storage and Sharing Service follows a **client-server architecture**, a widely adopted design in modern web applications. In this model, the **client** refers to the frontend application or user interface that interacts with users, while the **server** is responsible for processing requests, managing data, and providing responses. Users interact with the system by sending requests through HTTP methods such as GET, POST, and DELETE. The server, built using **Node.js and Express.js**, handles these requests by performing the necessary operations on the database and file storage system. Responses, including file metadata or download links, are sent back to the client. This separation of concerns ensures **scalability, maintainability, and security**, allowing the backend to focus on business logic and data management while the client handles user interactions.[1][2][3]

**3.2 Technology Stack Overview**

The system leverages a **modern, robust technology stack** to achieve performance, scalability, and security:

* **Backend:** Node.js and Express.js are used to develop RESTful APIs for handling authentication, file management, and sharing operations.
* **Database:** MongoDB serves as the primary database, with Mongoose ODM simplifying schema management and database interactions. **GridFS** is used to store large files efficiently.[1]
* **Authentication & Security:** JWT tokens manage user sessions, while **bcrypt** is used to hash passwords. Role-based access ensures proper authorization for file operations.[2][3]
* **File Upload & Storage:** Multer handles file uploads, while optional **cloud storage services** such as AWS S3 or Cloudinary provide scalable, production-ready storage solutions.
* **Deployment & Hosting:** The backend can be deployed on cloud platforms like **Render, Heroku, or Vercel**, with MongoDB hosted on **MongoDB Atlas**, providing high availability and easy scaling.

**3.3 System Components**

The system consists of several interconnected components:

1. **User Management Module:** Handles registration, login, authentication, and user profile management.
2. **File Management Module:** Responsible for uploading, storing, retrieving, downloading, and deleting files. Metadata such as filename, type, size, owner, and timestamps are maintained in the database.
3. **Sharing & Access Control Module:** Manages file sharing among users, enforcing role-based permissions. The module ensures that owners have full control while shared users have limited access.
4. **Search & Filter Module:** Provides functionalities to search files by name, type, or date and filter them efficiently.
5. **Middleware & Validation Module:** Ensures data integrity, checks permissions before file operations, validates file types and sizes, and handles errors.

This modular architecture ensures **separation of concerns**, allowing each component to function independently while contributing to the overall system. It also facilitates **scalability and maintainability**, making the backend ready for future enhancements such as version control, collaborative editing, or advanced analytics.

**4. Implementation**

**4.1 Project Structure**

The project is organized in a **modular and scalable structure** to facilitate easy development and maintenance. The backend directory typically contains folders for **routes**, **controllers**, **models**, **middleware**, and **configurations**.

* **Routes:** Define API endpoints for authentication, file management, and sharing.
* **Controllers:** Contain the business logic for each API, such as handling file uploads, downloads, sharing, and search operations.
* **Models:** Define MongoDB schemas for users and files, including metadata and relationships.
* **Middleware:** Handles authentication, authorization, file validation, and error management.
* **Config:** Contains environment variables, database connection settings, and cloud storage configurations.

This structure ensures **separation of concerns**, making the codebase clean and maintainable.

**4.2 Backend Implementation**

The backend is built using **Node.js and Express.js**, providing a RESTful API interface for all operations. **JWT authentication** is implemented to secure endpoints, while **bcrypt** ensures password security.  
File uploads are managed using **Multer**, which processes incoming files and stores them in **GridFS** for large files. Optionally, **AWS S3 or Cloudinary** can be used for cloud storage. Metadata for each file, including filename, type, size, owner, and shared users, is stored in **MongoDB**, enabling efficient retrieval and search.

The backend also implements **role-based access control**, where owners have full privileges (edit, delete, share) and shared users have limited access (view/download). Middleware functions validate permissions before any file operation. Additional features such as **search and filter** are implemented using MongoDB queries to allow users to find files based on name, type, or date. Comprehensive **error handling** ensures the system is robust against invalid operations or unauthorized access.

**4.3 Frontend Implementation**

Although the project primarily focuses on the backend, a **basic frontend interface** can be developed using frameworks like **React.js** to interact with APIs. The frontend allows users to:

* Register and log in securely.
* Upload, view, and download files.
* Share files with other users and access shared files.
* Search and filter files by different criteria.

The frontend communicates with the backend using **HTTP requests**, sending JWT tokens for authentication. It provides a **user-friendly interface** for managing files and demonstrates the system’s functionality in a real-world scenario.

This implementation ensures that the project is **end-to-end functional**, secure, and modular, providing a strong foundation for future enhancements like real-time collaboration or file versioning.

# Database Design

The File Storage and Sharing Service is built on **MongoDB**, a document-oriented database that provides flexibility, scalability, and high performance for managing user information, file metadata, and sharing relationships. The system uses **Mongoose ODM** to define schemas and manage database operations efficiently. The **User** collection stores critical information such as a unique email address, full name, securely hashed password using bcrypt, and a registration timestamp, providing a foundation for authentication and secure access control across the system. The **File** collection manages all uploaded files and their associated metadata, including filename, file type (PDF, image, video, or document), file size in bytes, the owner’s user ID, a storage URL or path, an array of shared user IDs, and the timestamp of file creation. This design facilitates **role-based access control**, ensuring that file owners retain full permissions, while shared users have restricted access to view or download files. Relationships are carefully structured to optimize both usability and security: a **one-to-many relationship** exists between users and files, allowing a single user to own multiple files, while the sharedWith array implements a **many-to-many relationship**, enabling multiple users to access the same file without redundancy. GridFS is integrated to store large files efficiently in chunks, overcoming database size limitations and supporting smooth upload and retrieval operations. The metadata structure also enables advanced functionalities such as **search and filtering**, allowing users to query files based on name, type, size, or upload date, enhancing usability for managing large collections. Additionally, validation rules and middleware ensure data integrity, preventing unauthorized access and maintaining consistency between user accounts and file records. This database design prioritizes **security, scalability, and performance**, forming the backbone of the backend system and providing a reliable, robust platform for secure file storage, management, and sharing. By combining MongoDB’s flexible schema, GridFS for large file handling, and Mongoose for structured operations, the system achieves a strong foundation that can be extended to include advanced features such as versioning, collaborative editing, or cloud storage integration, making it suitable for both educational purposes-and-real-world-applications.[1][2][3]

# Key Features

The File Storage and Sharing Service is designed with features that make it a **secure, scalable, and user-friendly backend system** for managing digital files. One of the core features is **user authentication and security**. The system uses **JWT (JSON Web Tokens)** for secure session management and **bcrypt hashing** for password storage, ensuring that only authorized users can access and manage files. This forms the foundation of the role-based access control system, allowing the backend to enforce permissions accurately.

The system provides a **robust file upload and storage mechanism**. Users can upload a wide range of file types including documents, images, PDFs, and videos. The uploaded files are stored using **MongoDB GridFS**, which allows efficient handling of large files by breaking them into manageable chunks. Optionally, **cloud storage services** like AWS S3 or Cloudinary can be integrated to provide scalable, production-ready storage. Each file is associated with **metadata** such as filename, type, size, owner, and timestamps, which supports searching, filtering, and auditing of file operations.

Another key feature is the **file sharing and access control system**. File owners can share their files with other registered users through email-based identification, while maintaining control over permissions. Shared users generally have **read-only access**, meaning they can view or download files but cannot delete or modify them. The system maintains a “**Shared With Me**” section, allowing users to easily view files that have been shared with them. This module demonstrates practical **role-based access control (RBAC)**, which is essential in real-world secure systems.[2]

**Search and filter functionalities** further enhance usability. Users can search files by name, type, or upload date, and filter files according to type (e.g., PDFs, images, or videos). This allows users to efficiently manage large numbers of files without difficulty.

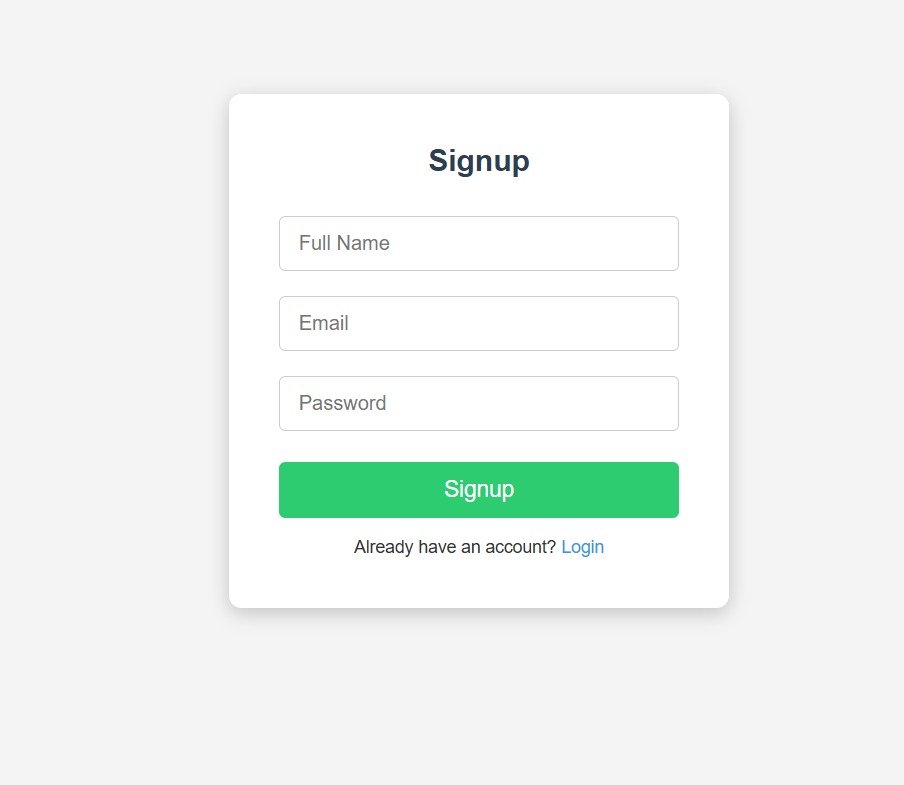
Additional features include **error handling and validation** to ensure only authorized operations are performed, files meet size and type constraints, and users cannot access files they do not have permission for. The system also supports **RESTful APIs**, allowing easy integration with frontend frameworks like React, Angular, or Next.js.

Overall, the project combines **security, efficiency, and usability**, making it a practical mini-version of Google Drive. It highlights real-world backend concepts including authentication, file management, sharing, and role-based access control, providing a strong foundation for further enhancements or production deployment.

# Project Glimpses

# 

**Fig 7.1: Home Page**

****

**Fig 7.2: SignUp page**

# Conclusion

The File Storage and Sharing Service (Mini Google Drive) successfully demonstrates a **secure, scalable, and user-friendly backend system** for managing digital files. The project integrates key functionalities such as **user authentication, file uploads, metadata management, role-based access control, and sharing capabilities**, providing a realistic simulation of cloud storage systems in a simplified form. By using **Node.js and Express.js** for backend development, **MongoDB with GridFS** for efficient file handling, and **JWT with bcrypt** for security, the system ensures both robustness and reliability.

The implementation of **role-based permissions** allows file owners to retain full control over their files, while shared users have restricted access, emphasizing practical security considerations. Additional features such as **search and filter functionalities, error handling, and validation** enhance usability and maintain data integrity, ensuring that users can manage files effectively and safely. Integration with **cloud storage services** like AWS S3 or Cloudinary further demonstrates scalability and production readiness, allowing the system to handle large files and multiple users efficiently.

This project not only provides a functional backend for file storage and sharing but also serves as a **learning platform** for understanding real-world backend concepts, including RESTful API design, authentication, authorization, database schema design, and file management. While it is a simplified version of enterprise systems like Google Drive, it establishes a solid foundation for future enhancements such as collaborative editing, version control, or advanced access policies.

In conclusion, the project bridges the gap between **basic file upload systems and full-scale cloud storage solutions**, combining security, efficiency, and usability in a practical, modular, and maintainable system. It is an excellent example of applying modern backend technologies to solve real-world problems and serves as a strong addition to any developer’s portfolio.

# References

# [1] L. Lukas, "Node.js Express JWT Authentication with MongoDB & Roles," Corbado, Jan. 15, 2025. [Online]. Available: <https://www.corbado.com/blog/nodejs-express-mongodb-jwt-authentication-roles>

# [2] "Authentication in Node.js and MongoDB using JWT and Cookies," Medium, Jun. 2025. [Online]. Available: <https://medium.com/@codemaniac-sahil/authentication-in-nodejs-and-mongodb-using-jwt-and-cookies-d617bd98cdea>

# [3] "Authentication in Node.js with MongoDB, bcrypt, and JWT Web Tokens with Cookies," Dev.to, Aug. 21, 2021. [Online]. Available: <https://dev.to/nyctonio/authentication-in-node-js-with-mongodb-bcrypt-and-jwt-web-tokens-with-cookies-hl3>