Advanced International Journal for Research (AIJFR)
E-ISSN: 3048-7641 ● Website: www.aijfr.com ● Email: editor@aijfr.com

# Secure Share a Scalable and Secure File Sharing Platform Using Node.js and MongoDB

**Chandan G N[1], Jayashri M[2], Dr Krishna Kumar P R[3]**

[1]MTech Student, Department of Computer Science & Engineering, SEA College of Engineering & Technology,Bangalore,India
[2,3]Faculty, Department of Computer Science & Engineering, SEA College of Engineering & Technology,Bangalore,India

**Abstract**

File sizes keep growing. Because of that, moving information online needs better tools. Old methods struggle. Big transfers slow down easily. Weak spots in safety show up more often. A new approach builds on modern tech instead. It uses Node.js for speed under pressure. Requests come in fast. They get handled without waiting. MongoDB stores details about users and files differently than old databases do. Structure shifts as needed. Security runs deep inside every step. People log in first before doing anything else. Only allowed actions go through. Files split into parts during sending. That makes heavy loads easier to handle. Access tokens act like keys. Each one opens just one door. Nothing extra gets shared by mistake. When tested, the system runs better than older methods, handling growth and staying stable under load. Built to work outside labs, it fits tasks like storing data online, team document access, or shared digital spaces.

## 1. Introduction

Today's online world creates a data flowing through networks, mainly because more people use cloud tools, apps on phones, yet also depend on web-based services. Sharing files now feels essential, whether at home or in big companies, since it helps individuals save, reach, or send digital items without hassle. Tools like online storage spots, team project areas, even systems that push out media - each leans hard on strong ways to move files back and forth.

Even with many ways to share files today, older methods still struggle in key areas. When too many people connect at once, standard server setups slow down, dragging out response times. Because everything runs through one main hub, if that fails, especially under pressure. Hackers looking to sneak in, leaks of private information, or harmful files being shared - all these risks make building safe networks tougher than it seems.

More people want apps that work fast and grow easily. That is why new web tools are spreading so quickly across tech setups today. Because it runs tasks wait for each to finish, Node.js stands out at managing many actions at once. Regular server designs use several threads at the same time yet often slow down under load. Instead, Node.js uses just one thread where jobs line up neatly like cars at a toll booth. Heavy reading or writing operations see smoother flow here than elsewhere. File transfer

platforms benefit because they push data constantly without pause. Its lean structure avoids bulky resource drains seen in older models.

Meanwhile, MongoDB skips fixed structures, letting teams build faster without rigid rules in place. Because it manages messy or incomplete data well, it works nicely for saving document labels, account profiles, and who gets permission. On top of that, spreading data across servers via sharding helps keep pace when storage needs climb steadily.

Built around Node.js and MongoDB, this effort sets up a dependable way to share files while sidestepping flaws found in older methods. Layered like stacked functions, each part handles either display, processing, or saving data - kept apart on purpose. When pieces work solo yet together, updating becomes smoother, growth feels natural, speed stays high.

Built right into the setup are core features like signing users in, moving files up and down, also letting people share them safely. Signed-in status sticks around thanks to JSON Web Tokens, keeping things light and locked down without server memory load. When big files move through, they split into pieces first - that way each part sends faster, less likely to break mid-transfer. Spotty connections handle it better since smaller chunks mean fewer hiccups and smoother overall speed.

One thing matters most: keeping things safe inside this setup. Built right into the core, layers like checking what gets entered, limiting which files are allowed, also who gets in at all. When use tokens - so just the right people open it, nothing slips through. Files vanish later if needed, thanks to timed limits that guard information tighter.

What helps this system grow smoothly? It uses Node.js to handle tasks separately, so things stay fast even when busy. Instead of slowing down under pressure, it spreads work across independently. MongoDB allows data to be stored in different places at once, which keeps everything moving during heavy use. When big files come into play, outside tools like cloud storage step in - connected neatly behind the scenes. Performance stays steady, quietly managing more people and heavier loads over time.

This file sharing system works well across many practical uses - cloud storage, company document handling, education materials exchange, even team collaboration spaces. Modern tools blend smoothly with smart structural choices here, pushing forward systems that scale reliably while staying safe and fast. Performance stays strong without sacrificing protection. Solutions grow naturally with demand, built on thoughtful architecture instead of complexity.

## Literature Review

File sizes keep growing. Because people use more online tools every day, ways to send files must change too. Old methods do not handle today's load well. Different solutions appeared slowly, each fixing certain problems. Some make transfers faster. Others let systems grow without breaking. Security also became stronger over time. Here we look at the main kinds of file transfer setups. Each type uses different tech behind it. The tools built into them shape how they work. Progress did not happen all at once. Small steps made big differences in how data moves now.

One way early networks handled file exchange was through one main computer holding everything. From there, people sent or received information by connecting directly to that machine. Simple setup meant simpler upkeep, true enough. Yet trouble started piling up once too many accessed it at once - speeds dropped sharply. If that core machine stopped working? Everything froze entirely. Big jobs often stumbled because of such weak points.

One way to tackle those problems was launching peer-to-peer, or P2P, file sharing setups. Files move straight between users in these networks - no main hub needed at all. Because of that setup, handling more people became easier while pressure on any single machine dropped.

Yet problems pop up - like needing steady internet, rising backend expenses, and questions about who really controls personal information once it's stored remotely.

Nowadays, files online have become more popular. Built with standard web tech plus REST APIs, they connect users and servers without hiccups. Working across different devices helps, Trouble shows up when older server methods get used - those usually stop everything while waiting for input or output, slowing down how many tasks can happen at once.

One reason newer tools like Node.js caught on? They tackle old problems in fresh ways. Built around events, this system keeps moving without waiting - handling many links at once without slowing down. When apps shuffle lots of info back and forth, say between users swapping files, that speed matters. Light on resources, built to grow: performance gets a quiet boost, unlike bulkier older methods.

Starting off differently, MongoDB - an example of a NoSQL system - often shows up in today's apps that need fluid data handling. Rather than sticking to rigid table formats, it opts for adaptable structures, making room for messy or partly organized info without slowing down. Because it spreads work across servers, coping with growing amounts of data feels less strained. When files move around digital spaces, this database quietly holds bits like who owns what, permissions involved, and where things are tucked away.

Most file sharing setups need strong safety steps. Different tools protect information, like login checks, scrambling data, or limiting who sees what. Using tokens - say JWTs - keeps users signed in without saving session details server-side. Another layer shows up through checking files before they're shared and locking down entry points.

Even with progress, today's file sharing tools face lingering issues. Large files often slow things down, safety isn't always solid, and growing usage can strain systems. Some platforms speed through tasks yet skimp on protection; others lock data tight but lag under load.

Looking at current setups, it's obvious something better is needed - one that uses up-to-date tools for fast, safe, file exchange that can grow. Built to tackle those exact issues, this work puts together a strong system with Node.js paired with MongoDB, pulling strength from non-blocking operations along with adaptable database handling.

**PROBLEM STATEMENT**

Nowhere is the push for better tools more clear than in how we move files online these days. Fast web connections and smart devices keep raising what people expect from sharing software. People trade big chunks of information every day just to stay on track with work or personal tasks. Because teamwork happens across borders now, sending documents smoothly matters more than ever. Still, current platforms struggle under pressure when things get complex or sensitive. Problems pop up around speed, privacy, even basic reliability sometimes. Not everything works right when you need it most.

When too many people use old-style file sharing at once, things tend to slow down. Heavy traffic overwhelms the main server, which drags out responses and weakens overall speed. If that central machine crashes, everything stops - no backup, no workaround. One break, and the whole network stumbles.

Not built for tight security, peer networks struggle when handling private files. Though they scale well, access rules often fall short. Cloud options bring another set of hurdles - relying too much on outside setups can raise running expenses. Privacy questions linger just beneath the surface there. Heavy dependence on external systems creates new risks alongside cost bumps.

Large files often cause problems in current systems because they are handled poorly. When uploads or downloads lack optimization, transfers tend to break - particularly on shaky connections. Without smart ways to manage files, sorting through stored information becomes a hassle. Getting back what you need takes longer than it should.

Staying safe matters when sending files around. Hacking attempts show up through backdoors, leaks, or harmful attachments slipping through. Weak checks on who gets in often explain why current tools fall short under pressure.

Blocking I/O tends to slow things down when many requests come at once. Old-style backends get stuck handling one task after another, so they struggle under pressure. Performance drops because the system cannot keep up.

One reason stands out: handling big files without glitches takes more than old methods offer. Built strong, it must grow easily when more people jump in. Security cannot slide - every bit stays locked down tight. Instead of patching past flaws, fresh tech reshapes how things move. Smooth performance under pressure defines its core job. Real tasks demand steady results, nothing less.

This effort tackles those issues through building a system for sharing files with Node.js along with MongoDB, placing weight on speed, growth capacity, safety. A different path emerges when code meets storage in smart ways. Performance matters here more than usual. Security shapes decisions at every turn. Scalability sneaks into design choices early. Tools fit together like puzzle pieces without forcing them. The outcome works whether few or many use it

**PROPOSED SYSTEM**

A solution built on Node.js and MongoDB takes form here, aiming to move beyond typical file transfer constraints through resilience, adaptability, safe handling. Structure supports growth while maintaining integrity under load - security weaves into each layer rather than appearing as an afterthought. Design choices Favor long-term operation across unpredictable conditions instead of momentary performance peaks. Operation remains steady even when demand shifts unexpectedly, thanks to underlying database flexibility paired with efficient server logic. This approach avoids common weaknesses seen in older models relying on rigid architectures.

The design uses separate modules stacked in layers, isolating user interaction from processing tasks as well as data handling to improve long-term upkeep alongside growth potential.

Because Node.js operates on an event-based system, it manages numerous simultaneous requests without blocking tasks. Efficiency in processing comes from how the platform handles input and output operations asynchronously. This particular trait makes scalability possible under heavy demand. Performance remains steady when dealing with many users at once. Such behaviour stems directly from its core design structure.

A document-based system handles storage of personal information along with details about files. This approach allows adaptation to changing structures while enabling quick access when needed at high volumes. Performance remains stable even as usage grows beyond initial expectations.

The system handles access control through JSON Web Tokens (JWT), delivering secure session handling without stored states. With JWT in place, verification occurs reliably each time a user interacts. Security remains intact as tokens manage identity checks independently. Each request carries verified credentials, removing need for server-side tracking. Stateless design supports consistent performance under variable loads.

Secure storage of passwords happens through encrypted methods, specifically crypt hashing, ensuring protection for login details. Encrypted formats like crypt prevent unauthorized access to account information.

External storage units hold the actual files - be it on a local drive or within cloud environments. Metadata alone resides inside MongoDB: details like filename, dimensions, directory location, user identifier, timestamp of submission. Instead of housing content directly, the database references attributes linked to each document. Ownership records, alongside temporal data points, serve tracking purposes across systems. Storage paths get preserved for access continuity despite physical separation between data layers.

Performance sees gains when file storage splits from metadata, since database strain then drops. System efficiency rises because data handling shifts more smoothly across components.

Splitting big files into pieces happens first, so transfer reliability improves when connections fluctuate. Transmission occurs in segments, allowing recovery without restarting from zero if interrupted. Each portion travels separately, reducing strain on bandwidth at any moment. This method maintains progress across disruptions naturally, needing no extra steps afterward.

When a transfer stops, the system keeps track of progress. From where it left off, uploading begins again automatically. Without beginning over, files finish sending through stored data.

Secure file exchange occurs through token-issued links on the system. Access is granted without complexity when recipients receive generated URLs. Sharing proceeds automatically once authentication tokens are validated. Links remain active under defined conditions only. User interaction stays minimal during transfer initiation.

When sharing links, an expiration period might apply. This means file access ends automatically after a set duration. Time-limited entry helps maintain control. Once the clock runs out, viewing stops without further steps. Security improves when availability is temporary by design.

## SYSTEM ARCHITECTURE

### Client Layer

The Client Layer is where users interact with the system. It’s built using common web technologies like HTML, CSS, and JavaScript, creating a simple and user-friendly interface.

Through this layer, users can:

- Register and log in to their accounts
- Upload and download files
- Organize and manage their files
- Share files using secure links

The client communicates with the server using HTTP/HTTPS requests, ensuring that data is transferred safely and efficiently. It’s also designed to work smoothly across different devices, whether it’s a laptop, tablet, or mobile phone.

### Application Layer (Server Layer)

The Application Layer is the backbone of the system. It’s developed using Node.js and Express.js and is responsible for handling all the core functionality.

This layer takes care of:

- Processing user requests
- Managing authentication and authorization using JSON Web Tokens (JWT)
- Handling file uploads and downloads
- Enforcing access control
- Validating user inputs and file data

One of the biggest advantages of Node.js is its event-driven, non-blocking architecture. In simple terms, it can handle many users at the same time without slowing down, making the system fast and efficient even under heavy usage.

**Database Layer (MongoDB)**

The Database Layer manages all the structured data in the system. MongoDB, a NoSQL database, is used because of its flexibility and scalability.

It stores:

- User details (such as email and password)
- File metadata (file name, size, type, path, owner)
- Sharing tokens and access-related information

Since MongoDB doesn't require a fixed schema, it's easy to update or modify the data structure as the application evolves. It also supports indexing and scaling, which helps amounts of data.

**Storage Layer**

The Storage Layer is where the actual files are kept. Instead of storing files directly in the database, they are saved either in a local file system or in cloud services like AWS S3.

This separation offers several advantages:

- Reduces the load on the database
- Improves overall performance
- scalable and efficient

Using techniques like chunk-based uploads, where files are uploaded in smaller parts.

**Data Flow**

All layers through clearly defined processes:

**File Upload Process:**
When a user selects a file, the client sends to the server. The server checks if file is valid, stores it in the storage layer, and saves its metadata in the MongoDB database.

**File Download Process:**
the server first verifies their permissions. If access is granted, the file is retrieved from storage and sent back to the user.

**File Sharing Process:**
Users can create secure sharing links. Each link is associated with a unique token, which is validated before access is allowed, users can view the file.

**Advantages of the Architecture**

key benefits:

- **Scalability:** Can handle growth in users and data without major changes
- **Performance:** Efficiently manages multiple requests at once using Node.js
- **Security:** Layered structure ensures controlled access and data protection
- **Flexibility:** Easy to extend or integrate with other systems
- **Maintainability:** Clear separation of layers makes development and debugging simpler

**System Architecture Diagram**

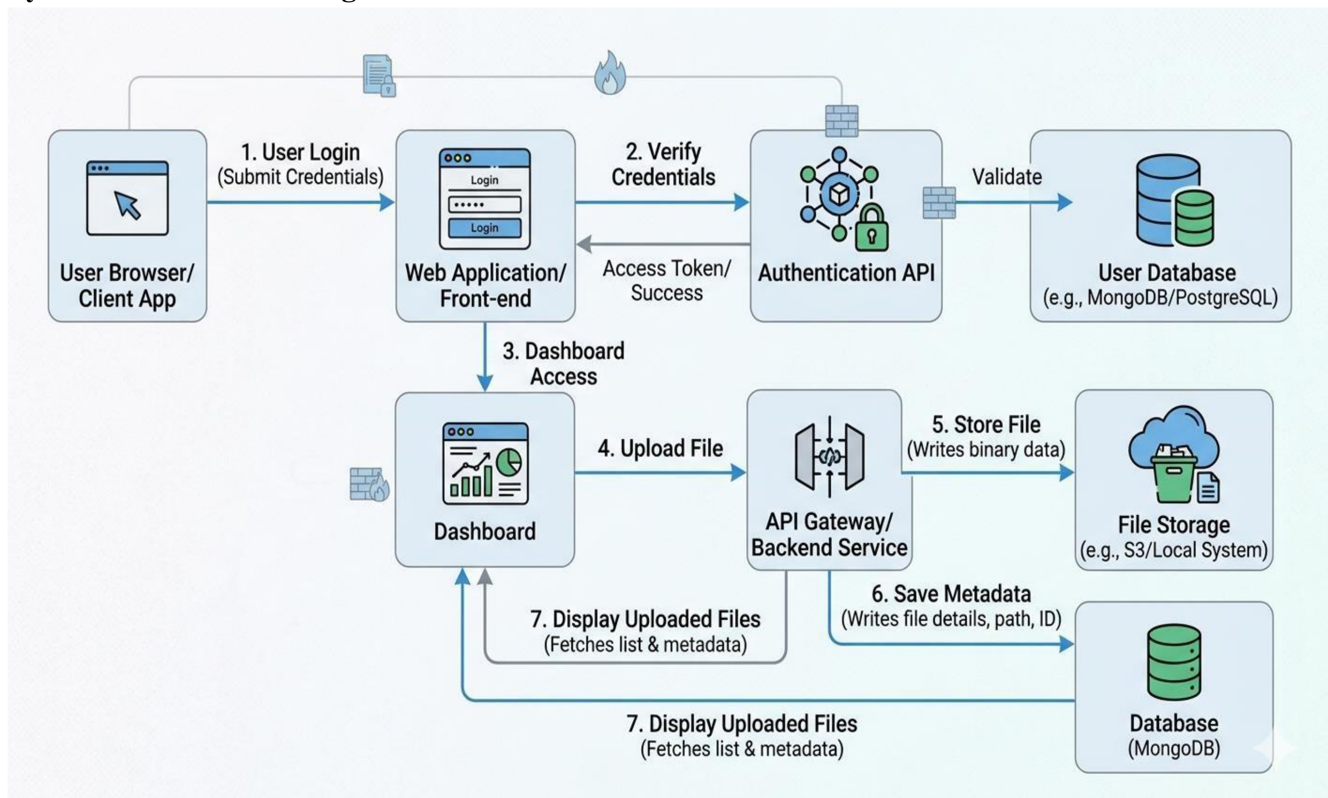


Figure 1. User Login and File Management System Architecture

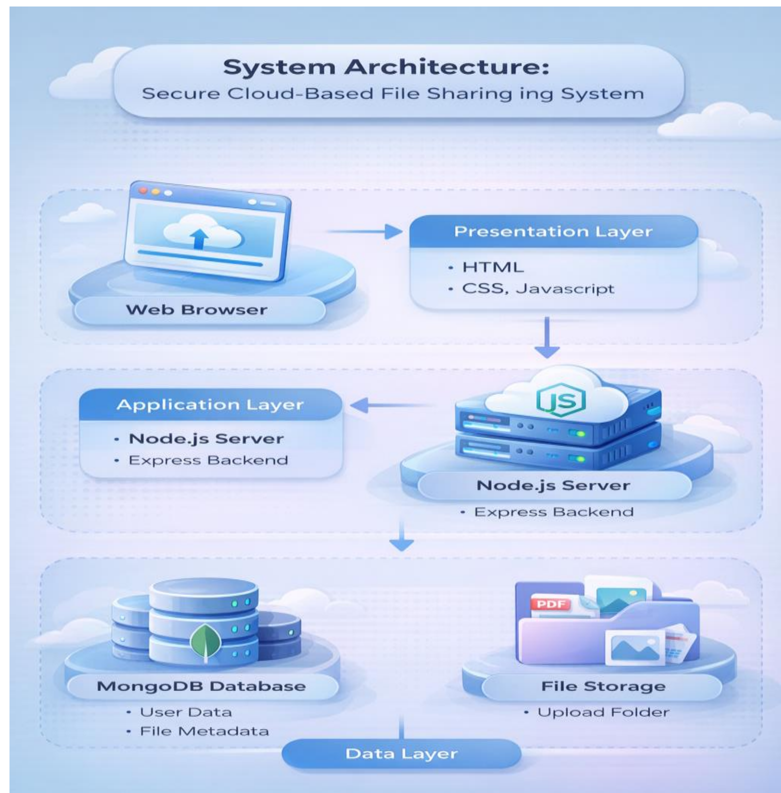


Figure 2. System Architecture for Cloud File Storage Platform

The diagram illustrates how the file-sharing platform works. It shows how different components—like authentication, file upload, storage, and retrieval connect and interact with each other to deliver a seamless user experience.

**Step-by-Step Workflow**

**1. User Login (Client → Frontend)**

Starting at the screen, a person types account information into fields seen online. That data moves through the internet, leaving the device toward visible parts of the site. Once there, it gets handled by tools built to receive such input.

**2. Credential Verification (Frontend → Authentication API)**

The frontend forwards the login details to the Authentication API. This API checks the credentials against the User Database (such as MongoDB or PostgreSQL) to verify whether they are valid.

**3. Access Token Generation**

If the credentials are correct, the system generates an access token, typically using JSON Web Tokens (JWT).
Back at the
frontend, that token arrives for checking later visits. It helps confirm who you are each time you come ba
ck.

After verification finishes, the system logs the person in, then sends them straight to their dashboard.

**4. File Upload Request (Dashboard → Backend Service)**

When the user uploads a file from the dashboard, the request is sent to the backend through an API Gateway or backend service.
This acts as the central point that handles incoming requests.

**5. File Storage (Backend → Storage Layer)**

The backend processes the uploaded file and stores it in the storage layer. This could be a local file system or a cloud-based service like AWS S3. The actual file (binary data) is saved securely at this stage.

**6. Metadata Storage (Backend → Database)**

After storing the file, the backend saves important details about the file known as metadata in the database (MongoDB).

This metadata typically includes:

- File name
- File path
- File size
- User ID (owner of the file)

**7. Display Uploaded Files (Database → Dashboard)**

To show files to the user, the system retrieves metadata from the database. displayed on the user dashboard, allowing users and manage their uploaded files easily.

## SYSTEM REQUIREMENTS

### Technologies Used

A fresh setup powers the file-sharing site, so it operates without hiccups even under heavy traffic. Smooth performance comes from up-to-date coding methods that also guard information tightly. Heavy loads don't slow things down thanks to smart design choices behind the scenes. Security stays strong because layers of protection work quietly at every step.

**1. Backend Technologies**

- **Node.js** is used as the server-side runtime environment. It follows an event-driven, non-blocking model.
- **Express.js** is used to build RESTful APIs, manage routes, and handle server-side logic in a simple and efficient way.

**2. Database Technology**

- **MongoDB** is used to store user data and file-related information.
- flexible structure, high performance, and easy scalability, dynamic applications.

**3. Frontend Technologies**

- **HTML** is used to build the structure of web pages.
- **CSS** is used to design and style the user interface.
- **JavaScript** is used to add interactivity and handle client-side functionality.

**4. Development Tools**

- **Visual Studio Code** is used as a main code editor, lightweight and supports many useful extensions.
- **Git** is used to source code, track changes, and support team collaboration.

**5. Storage Technology**

- Files are stored using a local file system or **Amazon S3**.

**METHODOLOGY**

One piece at a time, beginning with preparation then moving into rollout.

- **Requirement Analysis:**

  Right away, our team outlined the system's purpose. From there, one key step led to another - pinpointing functions such as file uploads, downloads, access sharing, along with strong security measures.

- **System Design:**

  After that came the layout of the whole setup, split into chunks - what users see on one end, the server handling tasks behind, where data lives, plus space to keep files tucked away.

- **Database Design:**

  Storing data happened through MongoDB. User details live there alongside files, while links for sharing stay neatly sorted too.

- **Frontend Development:**

  A fresh look at how people connect with the system began with building the front layer through HTML, shaped further with CSS styling while interactive behaviours came alive via JavaScript code. Each piece fits together behind the scenes allowing smooth navigation without extra steps or confusion for those using it.

- **Backend Development:**

  A fresh start each time shaped how we put the server together - Node.js took the lead, while Express.js stepped in to sort out incoming requests and keep things running. From there, control found its rhythm without extra weight or noise.

- **Authentication Module:**

  Now users can sign in through a system built on encrypted keys that keep credentials safe. Access works by verifying identity each time with tokens made unique per session. Passwords get locked down using methods that scramble data before storage happens. Signing up follows the same path but starts registration instead.

- **File Upload Module:**

  Chunks of large files go up one piece at a time. This way works better than sending everything in one go.

- **File Storage:**

  Files stay on local servers or inside Amazon S3, yet their info gets recorded into the database.

- **File Download Module:**

  Once the system confirms access rights, downloading begins without risk. Permissions verified first - only then does the file transfer start. Safety kicks in when authorization passes. After clearance is granted, retrieval happens securely. With approval confirmed, users receive files free of danger.

- **File Sharing Module:**

  Sharing files happens through protected links. When needed, a time limit applies to keep access in check.

- **API Development:**

  Communication between frontend and backend runs better because we built REST APIs. Smooth data flow happens when these parts connect through structured requests. The system responds more predictably thanks to this setup. Interactions follow clear patterns instead of random guesses. Design choices here support stability over time.

- **Testing:**

  Faults were caught early because we ran checks through Postman. Each piece had to prove it could keep up before moving on.

- **Deployment:**

  At last, the app went live on a server or up in the cloud, letting people reach it whenever they needed. One way or another, access became constant once everything settled into place.

## IMPLEMENTION

The implementation phase focuses on building the file-sharing platform by dividing it into smaller modules. This makes the system easier to develop, manage, and maintain. Technologies like **Node.js**, **Express.js**, and **MongoDB** are used to develop different parts of the system.

### Modules

- divided into key modules for better organization:
  - Authentication Module
  - File Upload Module
  - File Management Module
  - File Sharing Module
  - File Download Module

### Authentication Module

- Handles user registration and login
- Uses JWT (JSON Web Token) for secure authentication
- Encrypts passwords using bcrypt.

### File Upload Module

- Allows users to upload files through the interface
- Supports large file uploads using chunk-based techniques
- Validates file type and size before uploading
- Stores files securely in the storage system

### File Management Module

- Manages file-related details such as:
  - File name
  - File size
  - File path
  - Owner information
- Stores all metadata in **MongoDB**
- Allows users to view and delete their files

**File Sharing Module**

- Generates secure sharing links using tokens
- Provides controlled access to shared files
- Supports expiry-based links for better security

**File Download Module**

- Enables users to download files securely
- Verifies user authorization before allowing download
- Retrieves files efficiently from the storage layer

**Testing and Validation**

- The system is tested to ensure all modules work properly
- Checks file upload, download, and sharing features
- Ensures security, reliability, and performance requirements are met

## RESULTS

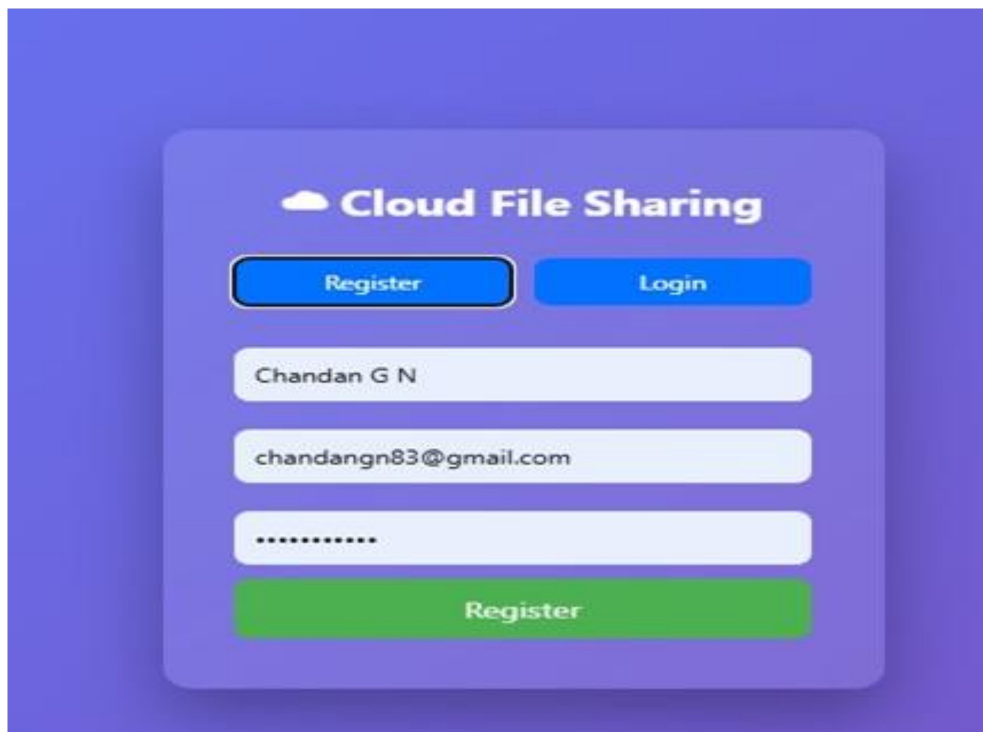


Figure 3 Cloud File Sharing System – Registration Screen

Figure 4 Cloud File Sharing System – Login Screen

Figure 5 File Upload and Management Interface

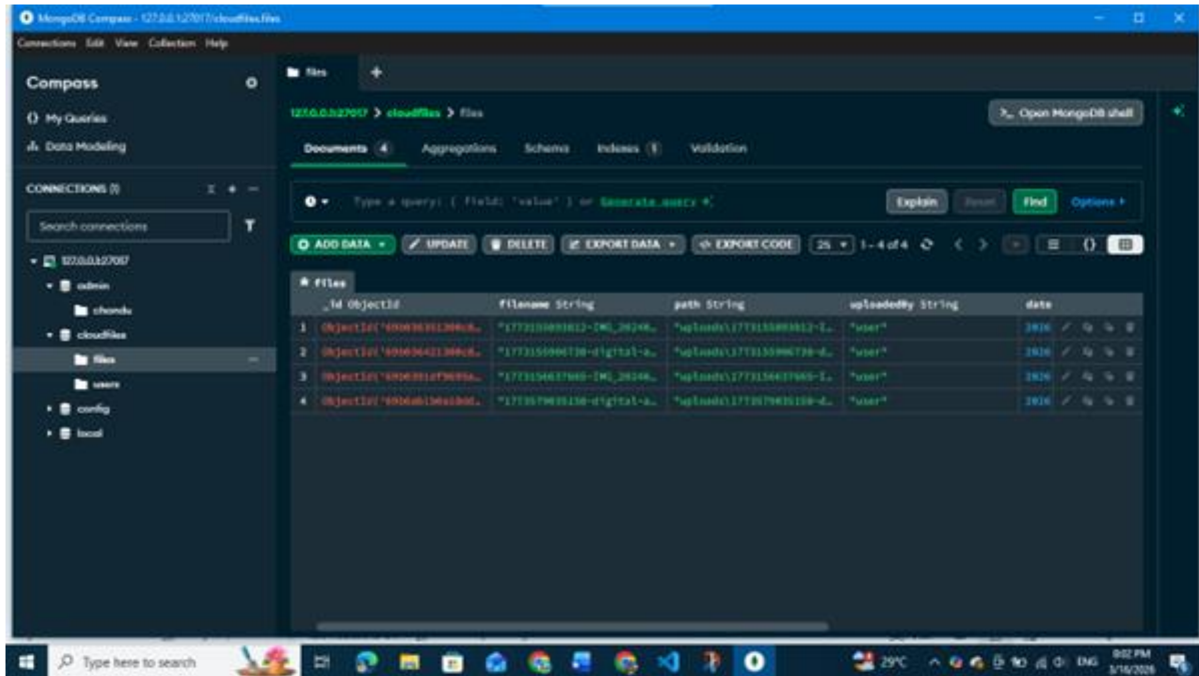

Figure 6 MongoDB Collection View for File Sharing System

**Conclusion**

This project shows how a file-sharing system can be built to handle growth, stay safe, yet run smoothly by using up-to-date tools. Instead of relying on old approaches, it handles core tasks like uploading files, downloading them, organizing content, while also allowing protected sharing. Because it uses JWT for login checks and crypt to lock down data, personal information stays shielded from intruders. Though simple in concept, the outcome tackles weaknesses common in earlier systems.

Heavy on speed, Node.js lifts how the system runs. Built to juggle many users at once, it skips waiting around - responses come through fast. Instead of slowing down, tasks move in bursts, smooth and steady. When it comes to saving data, MongoDB steps in - shape-shifting storage that grows as needed. Information like files and profiles tucks neatly into place, pulled back in a flash when called.

Chunks break up big files so uploads keep working when connections drop. When design stays split into parts, changing things later takes less effort. Future updates that link to online storage feel smoother because layers stay separate now.

From start to finish, the built system hits what it aimed for - offering fast, safe handling of files that grows when needed. Its usefulness shows clearly where it matters most: storing data online, managing documents inside companies, working together across tools, proving itself useful within today’s web software creation.

**FUTURE SCOPE**

- Integrate with cloud storage like **Amazon S3** and **Google Cloud Storage** for better scalability.
- Add advanced security features such as encryption and multi-factor authentication.
- Enable real-time file sharing and collaboration.
- Implement file versioning and backup options.
- Improve performance using caching and load balancing.
- Extend the system to a microservices architecture.
- Use AI for smart file classification and search.
- Develop mobile apps for wider accessibility.
- Enhance the user interface and user experience.
- Integrate email and notification services.

**References**

1. D. Vinaykumar, V. Pranav, V. Yadav, and N. C. Gowda, “Efficient File Sharing System Utilizing MongoDB and Node.js,” *International Journal of Computational Learning & Intelligence*, vol. 3, no. 1, pp. 177–182, 2024.
2. W. Peng, T. Lu, W. Peng, and Z. Wang, “An Efficient Blockchain-Based Framework for File Sharing,” *Scientific Reports*, vol. 14, 2024.
3. M. Rathore and S. S. Bagui, “MongoDB: Meeting the Dynamic Needs of Modern Applications,” *Encyclopaedia*, vol. 4, no. 4, pp. 1433–1453, 2024.
4. M. R. Dhanagari, “MongoDB and Data Consistency: Bridging the Gap Between Performance and Reliability,” *Journal of Computer Science and Technology Studies*, vol. 6, no. 2, pp. 183–198, 2024.
5. K. Hamaji and Y. Nakamoto, “A MongoDB Document Reconstruction Support System Using NLP,” *Software Journal*, vol. 3, no. 2, pp. 206–225, 2024.
6. C. Sravani et al., “Constructing a MERN Stack-Based Web Application Using Node.js and MongoDB,” *Engineering Proceedings*, vol. 66, no. 1, 2024.
7. C. Yadav, R. Dhakad, A. Khan, M. Panchal, and B. Kaur, “Analysis of MERN Stack Integration for Scalable Web Applications,” *Proceedings of ICICC*, 2024.
8. N. Dadkhah, X. Ma, K. Wolter, and G. Wunder, “DBNode: A Decentralized Storage System for Big Data in Blockchain,” *arXiv*, 2024.

9. F. Ahmed, N. K. Jha, and M. Faizan,
"Design and Development of a Resource Sharing Web Application Using Node.js and MongoDB," *arXiv*, 2024.
10. Node.js Foundation,
"Node.js Documentation and Runtime Environment," 2024.