YENEPOYA
(DEEMED TO BE UNIVERSITY)
Recognized under Sec 3(A) of the UGC Act 1956
Accredited by NAAC with 'A' Grade

YENEPOYA INSTITUTE OF ARTS, SCIENCE, COMMERCE AND MANAGEMENT

YENEPOYA (DEEMED TO BE UNIVERSITY)

BALMATTA, MANGALORE

AN PROJECT REPORT ON

"BUILD A SCALABLE FILE PROCESSING SYSTEM"

SUBMITTED BY
Mohammed Ayad
III BCA (Big Data, Cloud Computing and Cyber Security)
22BDACC158

UNDER THE GUIDANCE OF

Ms. Manjula
DEPARTMENT OF COMPUTER SCIENCE

IN PARTIAL FULLFILLMENT OF THE REQUIREMENT FOR THE AWARD OF THE DEGREE OF

BACHELORS IN COMPUTER APPLICATION

April 2025

CERTIFICATE

This is to certify that the project report entitled **"Build a Scalable File Processing System"** submitted by **Mohammed Ayad**, bearing Reg. No. **22BDACC158**, towards the partial fulfilment of the requirements for the award of the Bachelor of Computer Applications degree at Yenepoya (Deemed to be University) is a record of bonafide work carried out by him under my guidance and supervision.

Internal Guide: Ms. Manjula
Head of Department: Dr. Rathnakara Shetty
Principal and Dean: Prof. (Dr.) Arun Bhagawath
Vice Principals: Dr. Shareena P, Dr. Jeevan Raj, Mr. Narayana Sukumar
Place: Mangalore
Date:22-04-2025

YENEPOYA
(DEEMED TO BE UNIVERSITY)
Recognized under Sec 3(A) of the UGC Act 1956
Accredited by NAAC with 'A' Grade

DECLARATION

I **Mohammed Ayad** bearing Reg. No. **22BDACC158** hereby declare that this project report entitled "**Build a Scalable File Processing System**" had been prepared by me towards the partial fulfilment of the requirement for the award of the Bachelor of Computer Application at Yenepoya (Deemed to be University) under the guidance of **Ms. Manjula**, Department of Computer Science, Yenepoya Institute of Arts, Science, Commerce and Management.

I also declare that this field study report is the result of my own effort and that it has not been submitted to any university for the award of any degree or diploma.

Place: Mangalore
Date: 22-04-2025

Mohammed Ayad
III BCA (BIG DATA)
22BDACC158

ACKNOWLEDGEMET

I express my sincere thanks to **Prof. (Dr.) Arun Bhagawath**, Principal and Dean of Science, for providing the required facilities.
I extend my heartfelt gratitude to **Dr. Shareena P**, **Dr. Jeevan Raj**, and **Mr. Narayana Sukumar**, Vice Principals, for their continuous encouragement.
I am deeply grateful to **Dr. Rathnakara Shetty**, Head of the Department of Computer Science, for his valuable guidance.
I owe a special thanks to my internal guide **Ms. Manjula** for her support and mentoring during the project.
I would also like to acknowledge the support of my peers, friends, and family.
Place: Mangalore
Date:
Name: Mohammed Ayad
Class: III BCA (BDACC)

Place: Mangalore
Date:22-04-2025

Mohammed Ayad
III BCA (BIG DATA)
22BDACC158

SUMMARY

Project Title: Build a Scalable File Processing System

Domain: Cloud Computing & Virtualization

Objective:

The objective of this project is to design and implement a **serverless, scalable, and efficient file processing and management system** using **Amazon Web Services (AWS)**.
The system ensures secure user authentication, automatic file handling via AWS Lambda, and metadata storage in DynamoDB, eliminating the need for traditional server management.

Description:

The project utilizes **AWS S3** for file storage, **AWS Lambda** for serverless file processing, and **DynamoDB** for fast and scalable metadata storage.
A responsive **React.js** frontend allows users to upload, download, view, and manage their files.
The backend is developed using **Node.js** and **Express.js**, ensuring secure authentication and seamless integration with AWS services.
Files uploaded by users automatically trigger Lambda functions to process and store metadata, creating an **event-driven, serverless architecture**.

Technologies Used:

- **Frontend**: React.js, Tailwind CSS, Axios
- **Backend**: Node.js, Express.js
- **Cloud Services**: AWS Lambda, AWS S3, AWS DynamoDB
- **Database**: MongoDB Atlas (for user management)
- **Authentication**: JWT Tokens, HTTP-only Cookies
- **APIs Testing Tools**: Postman

Learning Outcomes:

- Mastery over **Serverless Computing and Event-Driven Architectures**.
- Deep understanding of **Cloud Storage** and **Serverless Functions**.
- Integration of **Secure Authentication Mechanisms**.
- Deployment of a fully functional **cloud-native application**.

Keywords: Cloud Computing, Serverless Architecture, AWS Lambda, AWS S3, DynamoDB, React.js, Node.js, Scalable File Management, Event-Driven Systems

Table of Contents

CHAPTER 1: INTRODUCTION

1.1 Overview of the Project

In today's digital era, data and files are generated in huge volumes every second. Managing, processing, and storing these files securely and efficiently is a primary need for almost every modern organization. Traditional on-premise storage solutions suffer from scalability, high cost, and maintenance overhead. Therefore, there is a growing demand for cloud-native, serverless solutions that are scalable, secure, and cost-effective.

This project focuses on building a scalable, serverless file processing system using Amazon Web Services (AWS). It uses AWS S3 for storage, AWS Lambda for processing, and DynamoDB for storing metadata. The system can handle large numbers of file uploads and retrievals efficiently without any manual intervention or server maintenance.

The frontend is built using React.js along with Tailwind CSS to provide a modern, responsive user interface. The backend API is developed using Node.js and Express.js, ensuring fast and reliable communication with the AWS services.

1.2 Objective of the Project

The main objectives of the project are:

- To design and develop a serverless system for file management.
- To integrate AWS services such as Lambda, S3, and DynamoDB.
- To build a secure authentication system using JWT tokens.
- To develop a user-friendly, responsive frontend application.
- To understand and implement event-driven architectures on the cloud.

1.3 Project Category

The project falls under the category of **Cloud Computing and Virtualization**, specifically focusing on serverless architecture and event-driven systems.

1.4 Tools and Platform to be Used

- **Frontend:** React.js, Tailwind CSS
- **Backend:** Node.js, Express.js
- **Cloud Services:** AWS Lambda, S3, DynamoDB
- **Authentication:** JWT tokens, HTTP-only cookies
- **Database:** MongoDB Atlas
- **Hosting:** AWS Cloud

1.5 Overview of the Technologies Used

AWS Lambda: AWS Lambda is a serverless compute service that lets you run code without provisioning or managing servers.
AWS S3 (Simple Storage Service): AWS S3 provides scalable object storage for file uploading and downloading.
AWS DynamoDB: DynamoDB is a fast and flexible NoSQL database service designed for single-digit millisecond performance at any scale.
React.js: A popular frontend JavaScript library for building user interfaces.
Tailwind CSS: A utility-first CSS framework that provides a modern, responsive design system.
Node.js and Express.js: Used to create backend APIs and server-side logic.
MongoDB: MongoDB is a flexible, document-oriented NoSQL database designed for scalability and developer-friendly data modeling.
TypeScript: TypeScript is a strongly typed programming language that builds on JavaScript, providing optional static typing for improved code quality and maintainability.

1.6 Organization Profile

This project was undertaken as an academic project under Yenepoya Institute of Arts, Science, Commerce and Management, Mangalore, affiliated with Yenepoya (Deemed to be University).

1.7 Structure of the Program

Frontend React Router Structure:

- `/` - Home Page
- `/login` - Login Page
- `/register` - Sign Up Page
- `/dashboard` - User Dashboard
- `/preview/*` - File Preview
- `/starred` - Starred Files

Backend Node.js Routes:

- `/login` - Login
- `/signup` - Sign Up
- `/file/upload` - Upload File

- **`/file/download/*`** - Download File
- **`/file/delete`** - Delete File
- **`/list`** - List Files
- **`/file/signed-url/*`** - Signed URL for Preview
- **`/starred`** - Manage Starred Files
- **`/folder/fetch/*`** – Fetch folder and list all files and subfolders
- **`/folder/*`** – Create Folder
- **`/folder/`** – Delete Folder and files under folder path
- **`/folder/download/*`** – Download Folder and its files
- **`/folder/size/*`** – Get Folder Size (in MB)

1.8 Statement of the Problem

Handling large amounts of file uploads with traditional servers becomes costly and hard to manage. Server maintenance, scaling, security, and performance issues arise. This project solves the problem by building a serverless cloud-native solution where scaling is automatic, maintenance is negligible, and cost is optimized based on usage.

CHAPTER 2: SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

2.1 Introduction

This chapter outlines the complete Software Requirements Specification for the project **"Build a Scalable File Processing System."** It explains the system functionalities, features, performance requirements, and technical environment necessary to successfully develop and deploy the project in the cloud environment. The system focuses on implementing a **serverless, event-driven file processing solution** using **AWS Lambda**, **S3**, and **DynamoDB**.

2.2 Purpose

The purpose of this project is to develop a **highly scalable and efficient file management system** using cloud services without managing any servers.
By utilizing **AWS Lambda** (serverless computing), **Amazon S3** (object storage service), and **DynamoDB** (NoSQL database), the system aims to automate file processing, minimize operational costs, and maximize reliability.
Each uploaded file is processed automatically via **Lambda functions triggered by S3 events**, and its metadata (like filename, size, upload time) is stored in **DynamoDB** for quick retrieval and management.

2.3 Scope

The scope of the project is to provide:

- Secure user authentication and file management.
- Instant file upload through a responsive web interface (React frontend).
- Automatic serverless processing of uploaded files (using AWS Lambda).
- Storage of file metadata (filename, type, upload time) in DynamoDB.
- File listing, download, deletion, and favorite (starred) management.
- Fast and efficient scaling without manual server interventions.
- A fully responsive and mobile-friendly user interface.

This project will cater to users needing reliable, fast, and scalable cloud-based file management.

2.4 Intended Audience

- Students and learners interested in Cloud Computing and Serverless Architectures.
- Organizations planning to move from traditional storage to cloud-native solutions.
- Developers working on building event-driven cloud applications.

2.5 Definitions, Acronyms, and Abbreviations

- **AWS**: Amazon Web Services
- **Lambda**: Serverless Function-as-a-Service (FaaS) by AWS
- **S3**: Simple Storage Service – Scalable object storage provided by AWS
- **DynamoDB**: AWS-managed NoSQL database
- **Serverless**: Computing model where the cloud provider automatically manages server infrastructure
- **Metadata**: Data that describes attributes of a file (such as size, name, path, upload date, etc.)
- **DynamoDB**: A fully managed NoSQL database service provided by AWS that offers fast and predictable performance with seamless scaling. In this project, it is used to store file metadata such as filename, size, upload timestamp, and associated user details.

2.6 References

- **Amazon S3 Developer Guide** - https://docs.aws.amazon.com/s3/
- **AWS Lambda Developer Guide** - https://docs.aws.amazon.com/lambda/
- **AWS DynamoDB Documentation** - https://docs.aws.amazon.com/dynamodb/
- **React.js Official Documentation** - https://react.dev/
- **Node.js Official Documentation** - https://nodejs.org/

2.7 Overview

The system will be composed of three main components:

(1.) **Frontend Interface** (React.js with Tailwind CSS):
- Allows users to sign up, login, upload, and manage files.

(2.) **Backend Server** (Node.js with Express.js):
- Handles authentication, file management APIs, and integration with AWS.

(3.) **Cloud Services** (AWS):
- **S3**: Stores files.
- **Lambda**: Processes file uploads automatically.
- **DynamoDB**: Stores metadata for efficient retrieval.

The entire architecture will be **event-driven**, meaning that actions (like file uploads) will automatically trigger processing workflows without requiring server monitoring or intervention.

CHAPTER 3: SYSTEM ANALYSIS AND DESIGN

3.1 Introduction

System Analysis and Design is a crucial phase that involves understanding user requirements, analyzing system needs, and designing solutions accordingly.
In this project, we focus on analyzing the file processing system's requirements and designing an efficient, scalable cloud architecture using AWS services (Lambda,S3, DynamoDB).

The design ensures that:

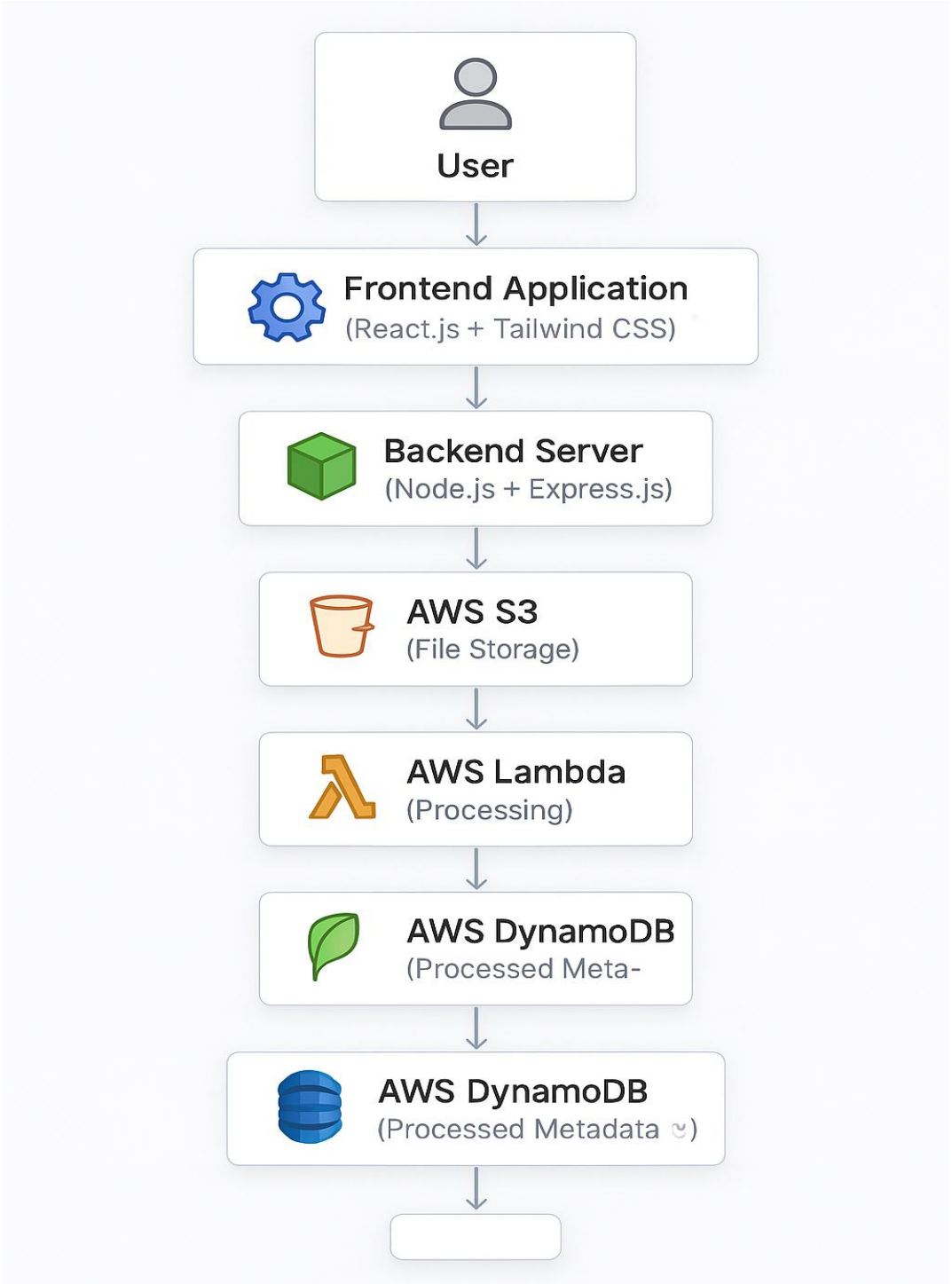
- User interactions are seamless.
- File processing is automatic and reliable.
- System components work together in an event-driven manner without manual intervention.

3.2 Data Flow Diagrams (DFD)

Data Flow Diagrams are used to represent the flow of data through the system, showing how input is transformed into output.

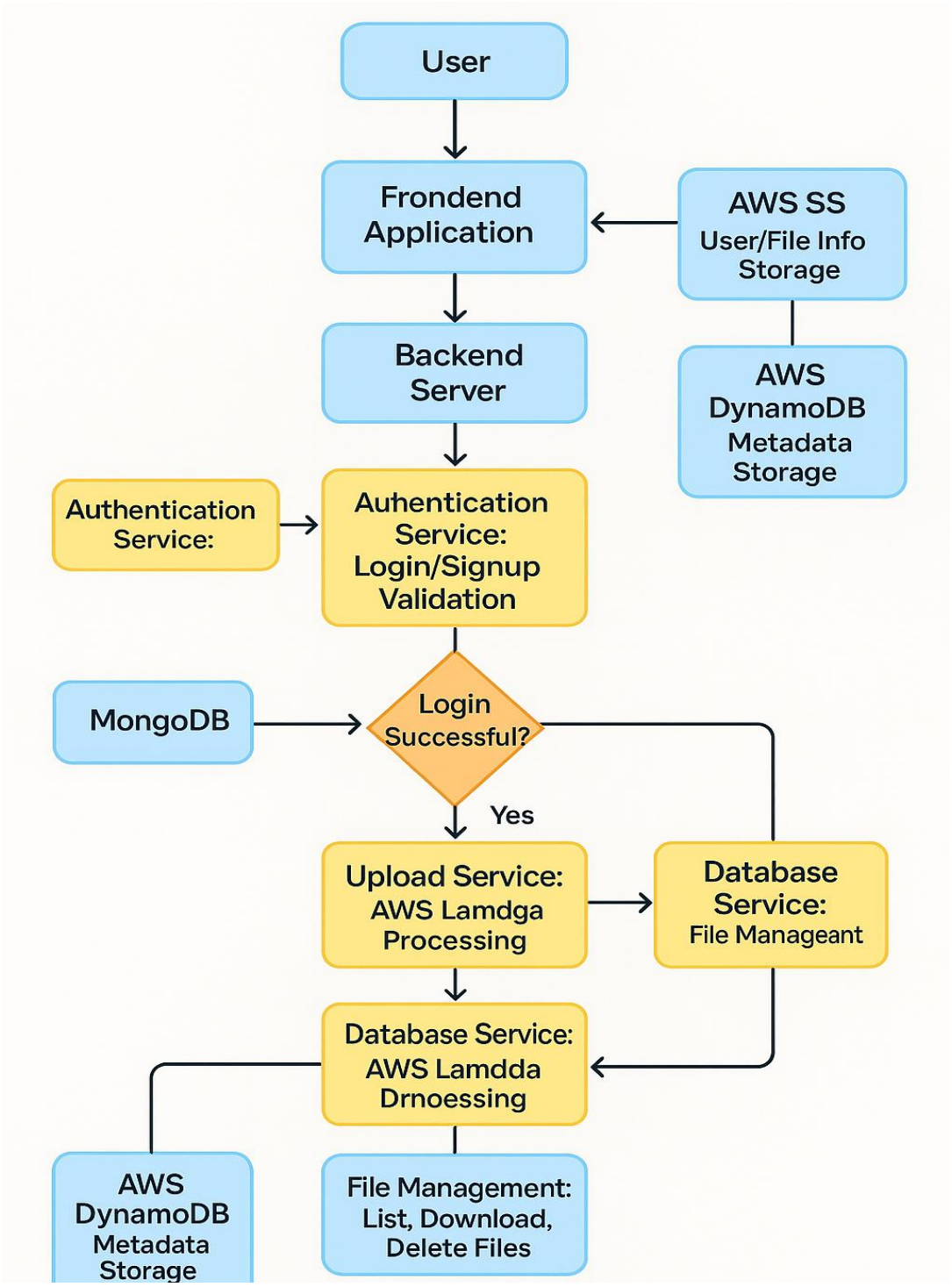
3.2.1 Level 0 DFD (Context Diagram)

- **User** interacts with the **System** via a frontend application.
- Files are **uploaded** to the server, triggering **Lambda processing**.
- Processed data is stored in **S3** and metadata in **DynamoDB**.
- User can **download**, **view**, or **delete** files via the frontend.



3.2.2 Level 1 DFD (Detailed Diagram)

- **Authentication**: Validate User (Login/Sign up).
- **Upload Service**: Upload files to S3.
- **Processing Service**: Lambda automatically processes uploads.
- **Database Service**: DynamoDB and MongoDB saves file metadata.
- **File Management**: List, Download, Delete files.



3.3 Entity-Relationship Diagram (ERD)

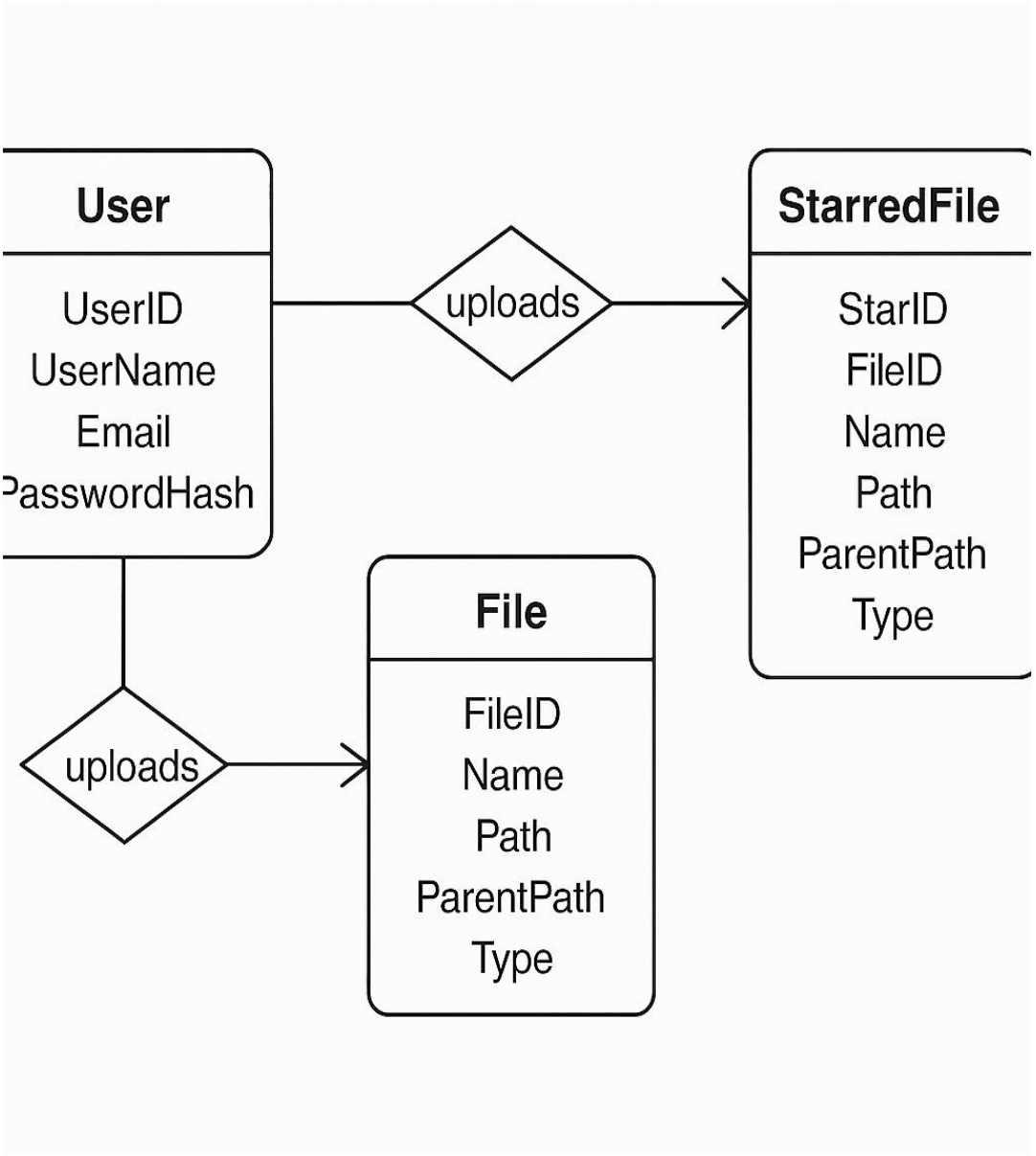
An Entity-Relationship Diagram shows the logical structure of the database.

Entities:

- **User** (UserID, UserName, Email, PasswordHash)
- **File** (FileID, name, path, parentPath, type,)
- **StarredFile** (StarID, FileID, name, path, parentPath, type)

Relationships:

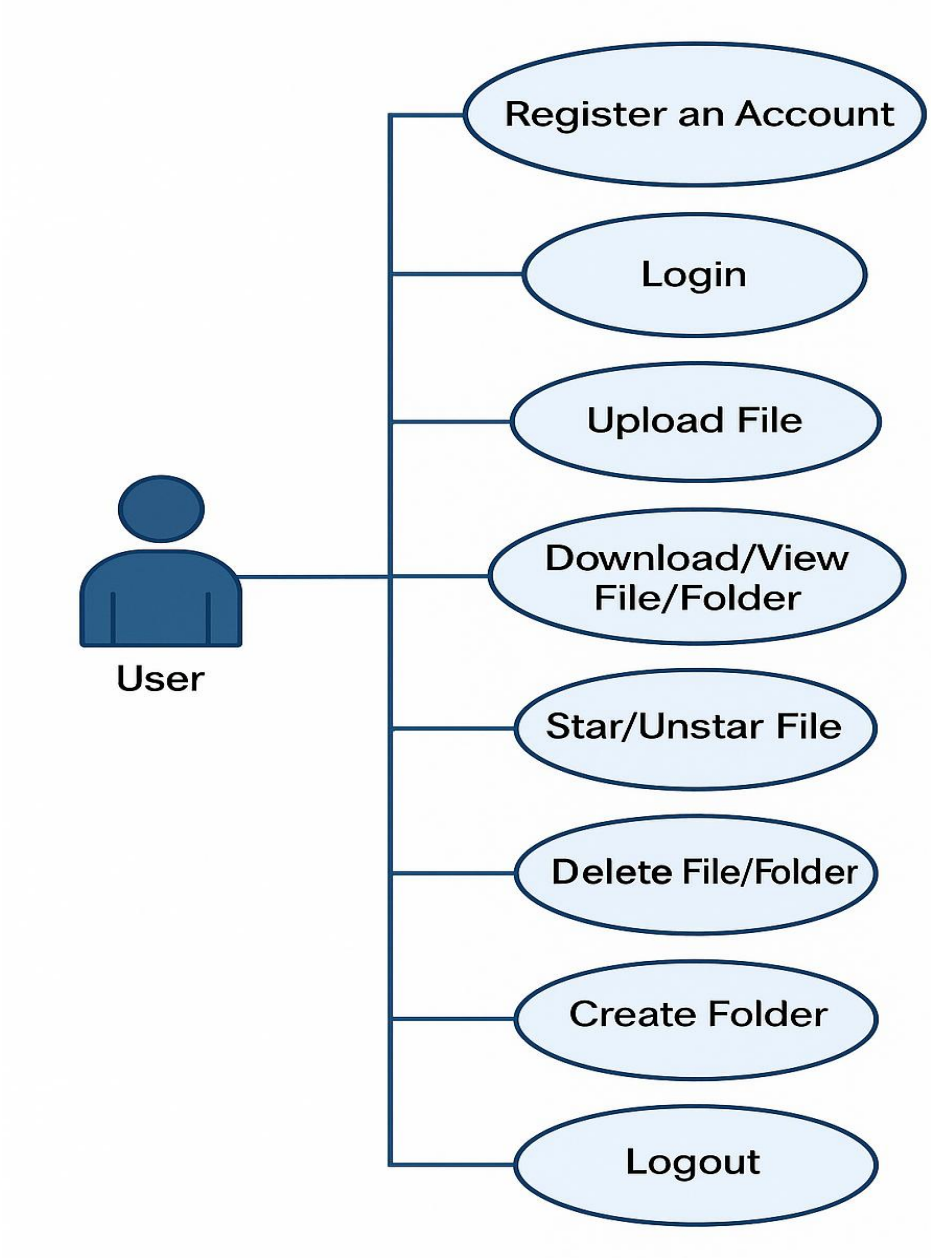
- One **User** uploads one **Files**.
- One **User** can star multiple **Files**.



3.4 Use Case Diagram

Use Case Diagrams describe the interactions between users and the system.
Main Use Cases:

- Register an account
- Login
- Upload file
- Download/View file/folder
- Star/Unstar file
- Delete file/folder
- Create folder
- Logout

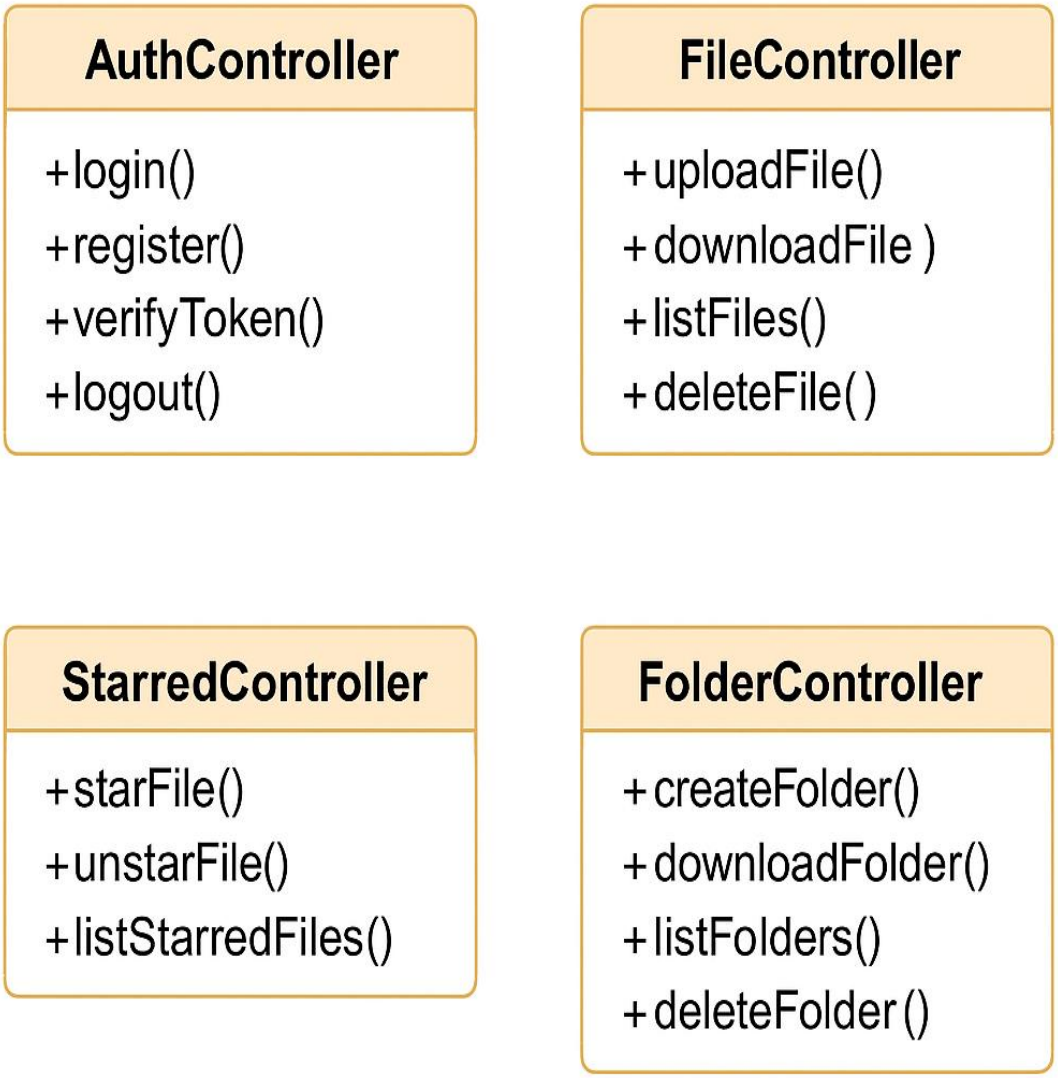


3.5 Class Diagram

The Class Diagram shows how the backend and frontend components are structured.

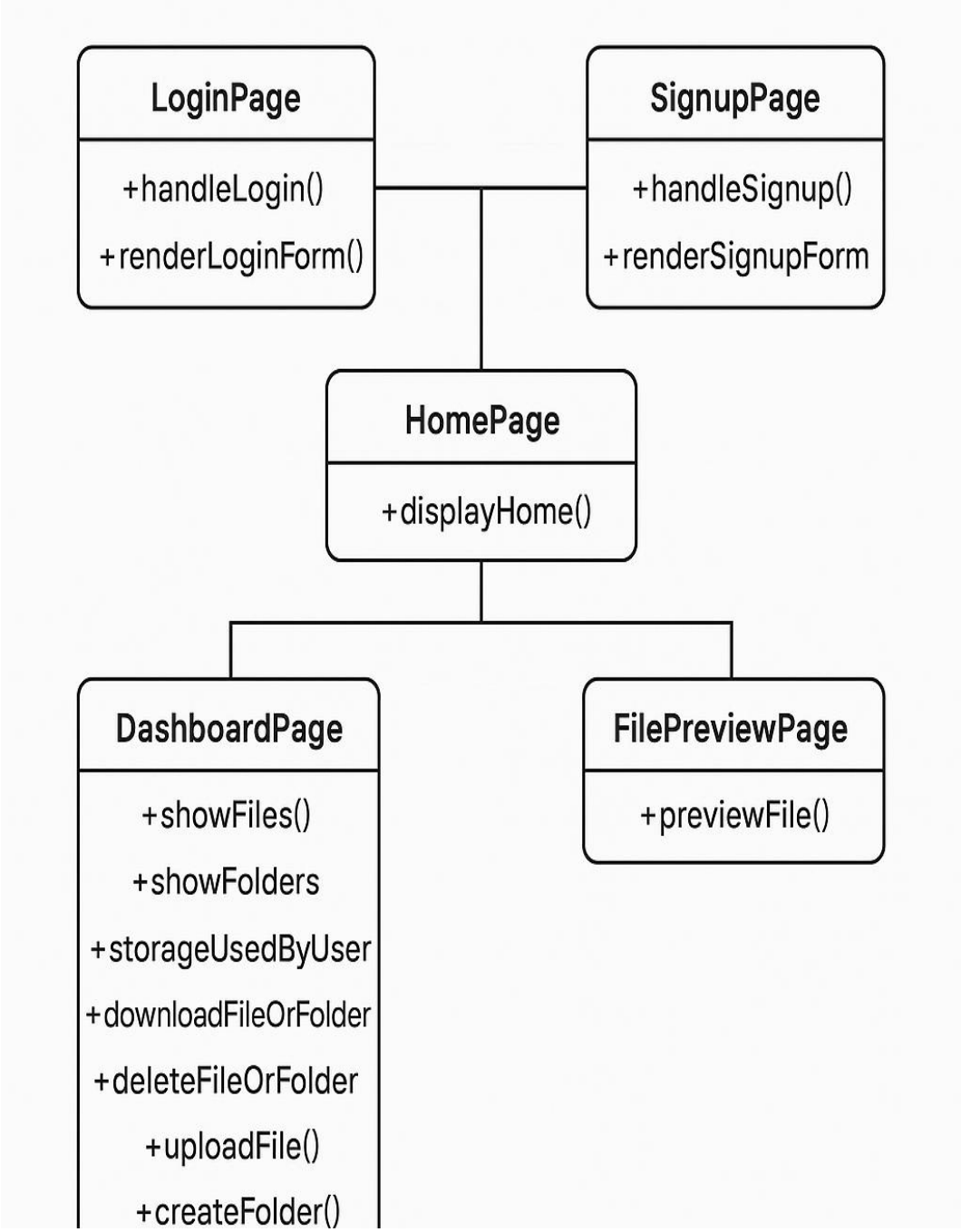
Backend Classes (Node.js):

- **AuthController**: Manages login, registration, token verification.
- **FileController**: Handles file upload, download, list, delete.
- **FolderController**: Handles folder creation, download, list, delete.
- **StarredController**: Manages starred files.



Frontend Components (React.js):

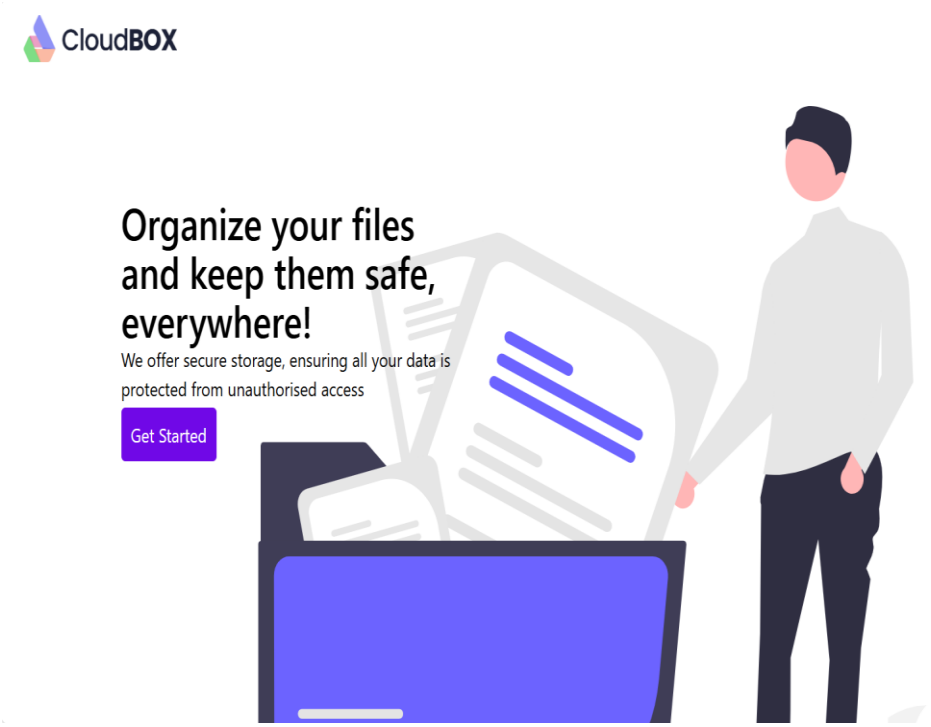
- LoginPage
- SignupPage
- HomePage
- DashboardPage
- FilePreviewPage
- StarredFilesPage



3.6 User Interface Design

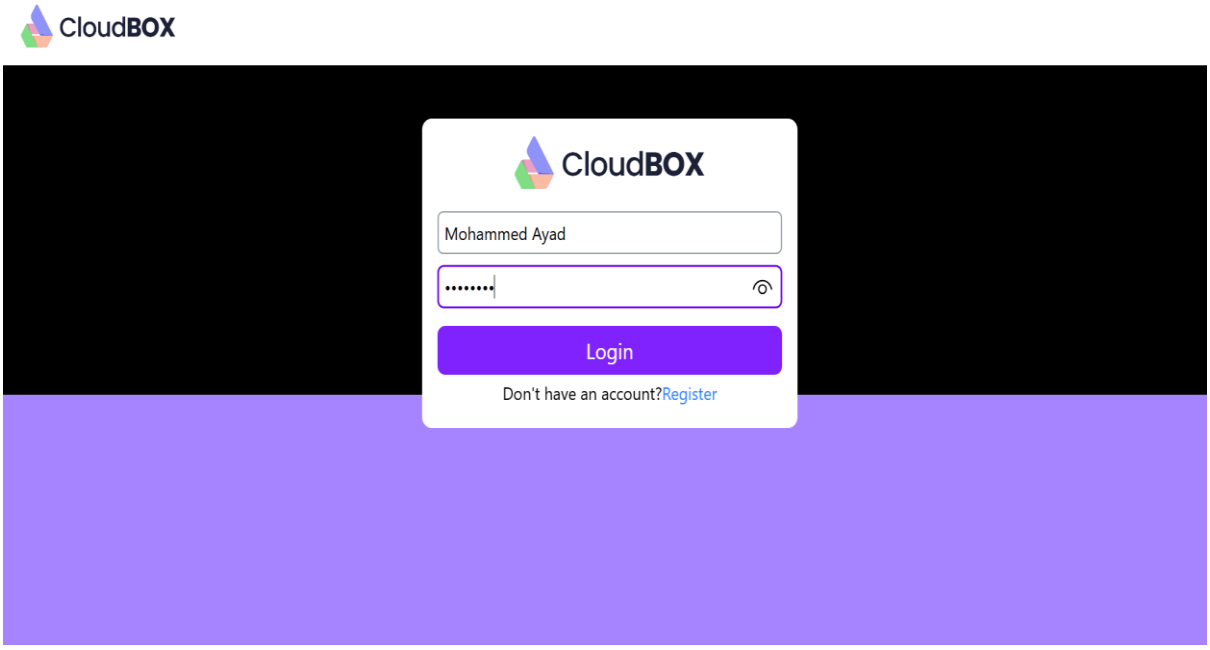
The User Interface is designed to be modern, clean, and mobile-responsive using **React.js** and **Tailwind CSS**.

- **Home Page**: Landing page for the app

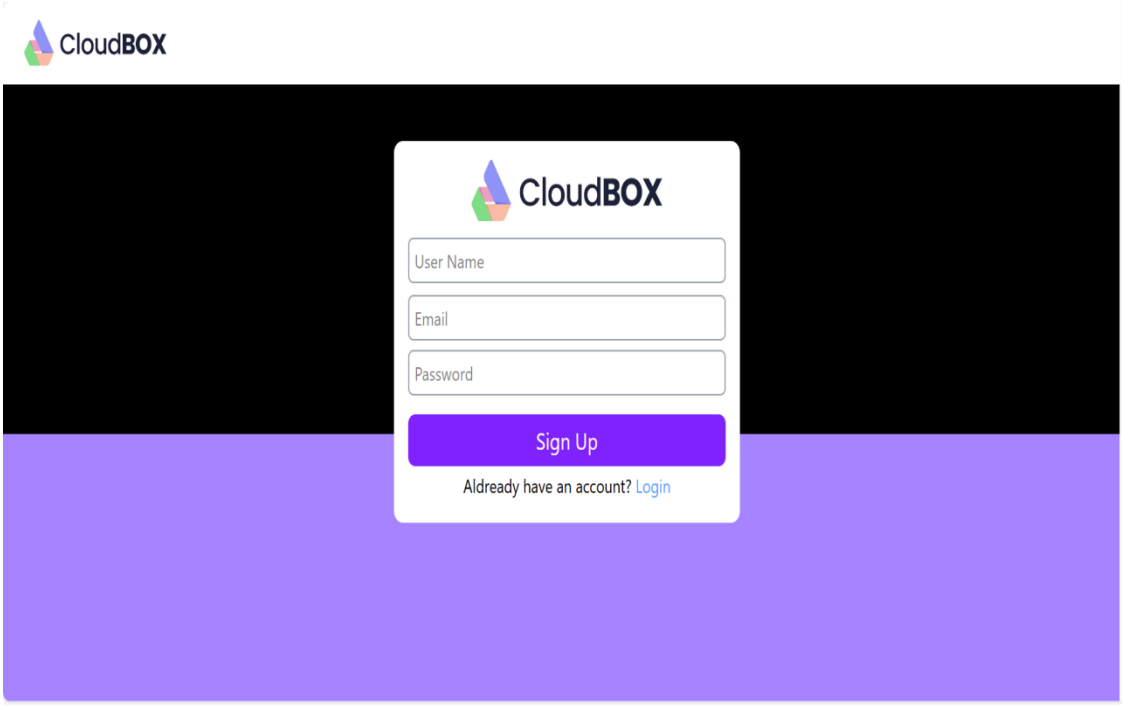


.

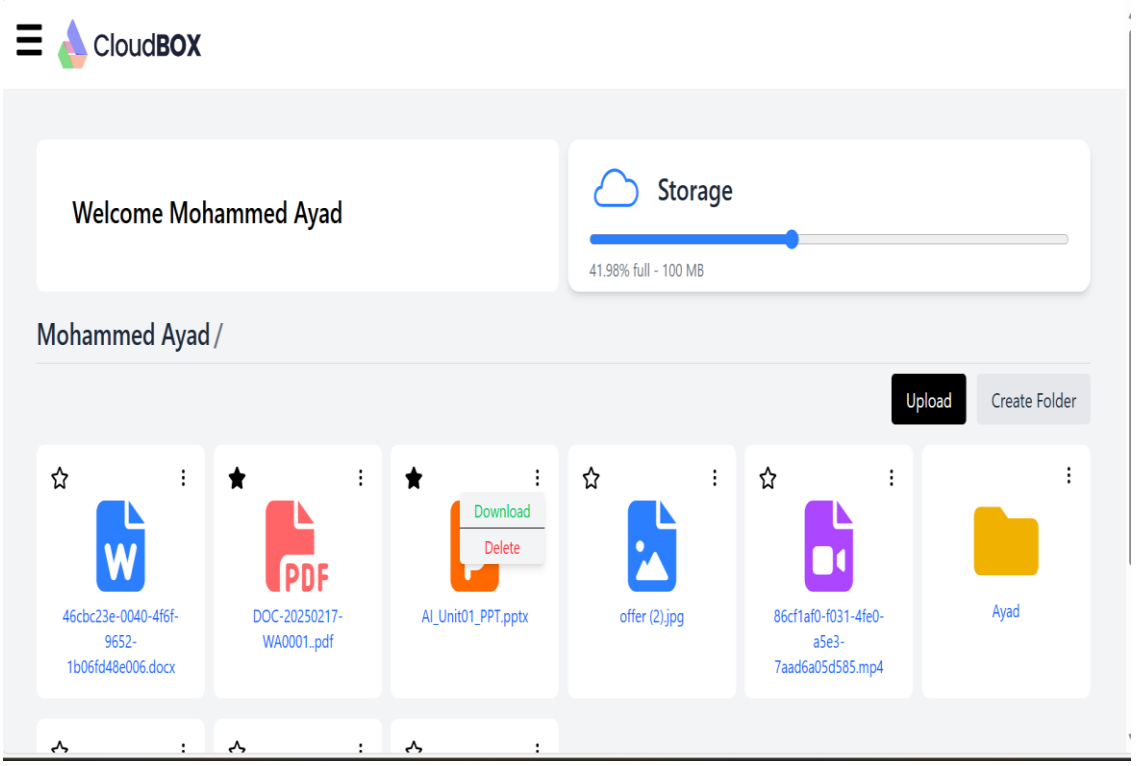
- **Login Page**: User login form.



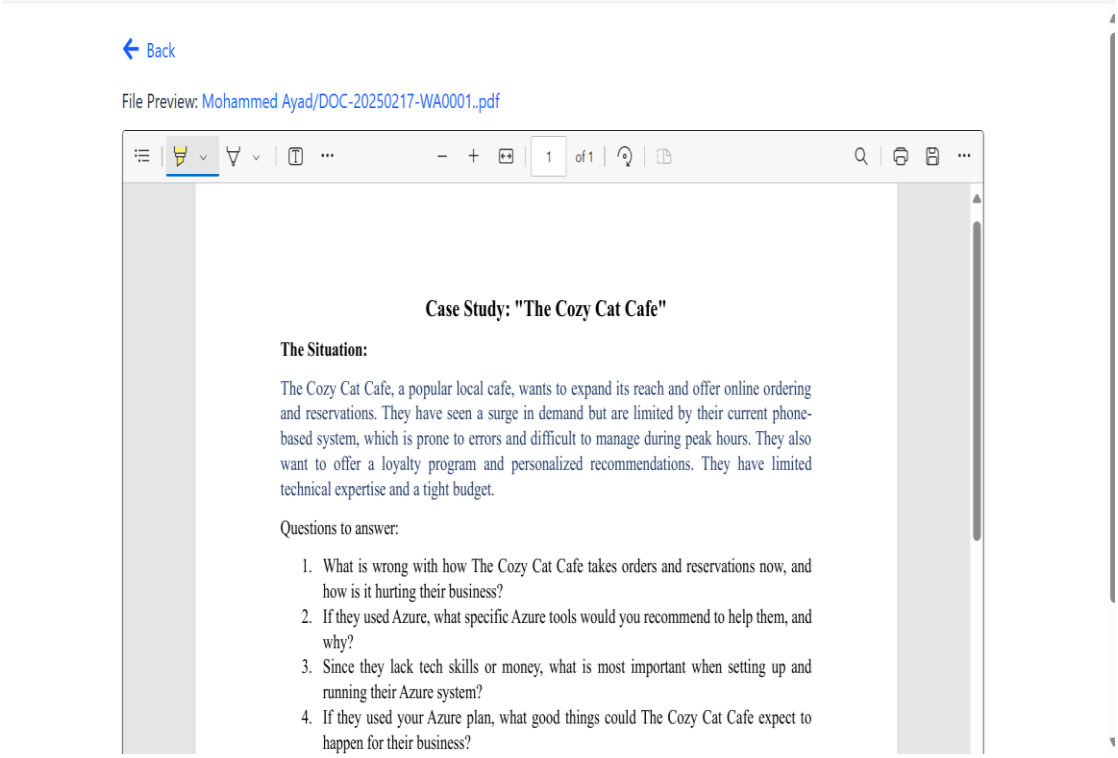
- **Registration Page**: New user signup.



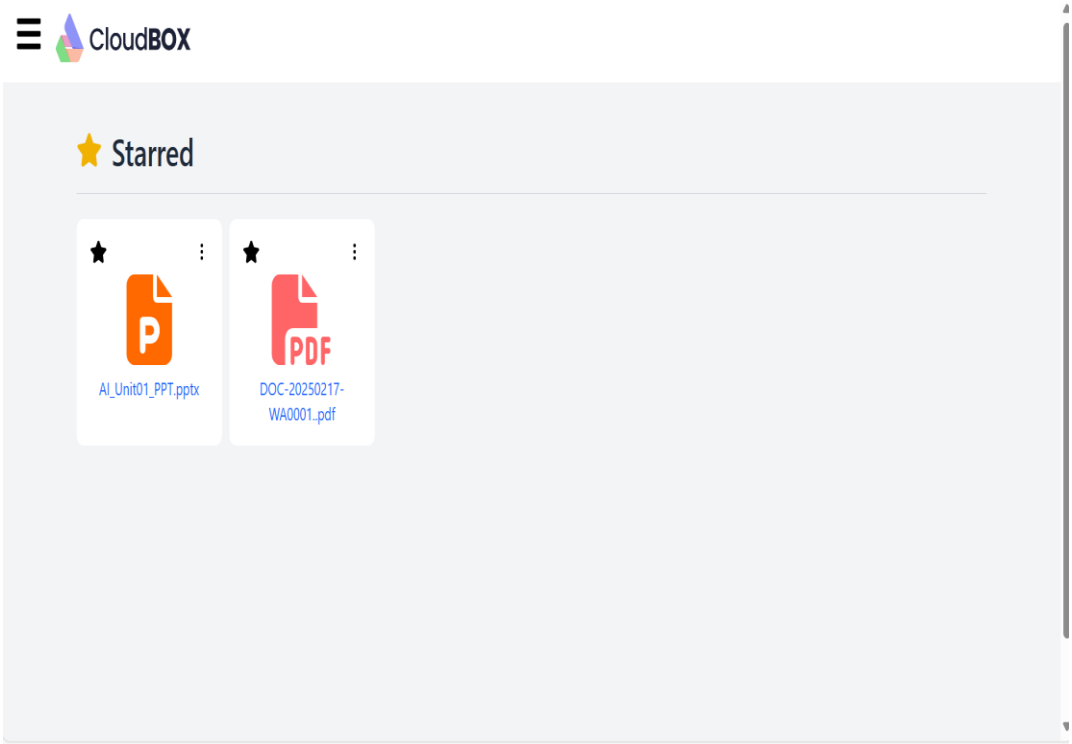
- **Dashboard**: Shows uploaded files, allows uploads, downloads, deletions.



- **Preview Page**: Preview individual file details.



- **Starred Files Page**: Manage favourite files.



CHAPTER 4: TESTING

4.1 Introduction

Testing is a critical part of the software development life cycle.
It ensures that the developed system meets the requirements, is free from major defects, and behaves as expected under various scenarios.
For the **Build a Scalable File Processing System** project, testing was performed both on:

- **Frontend (React Application)**
- **Backend (Node.js APIs)**
- **AWS Services Integration**

The goal of testing was to validate user authentication, file management (upload/download/delete), and serverless event processing reliability.

4.2 Testing Objectives

The main objectives of testing in this project were:

- **Functional Validation**: To ensure that every functionality (Login, Upload, Download, Delete) works correctly.
- **Security Testing**: To verify that user authentication and file access are protected using JWT tokens and HTTP-only cookies.
- **Performance Testing**: To check if file uploads and downloads happen quickly and efficiently.
- **Integration Testing**: To validate that AWS services (Lambda, S3, DynamoDB) work seamlessly together.
- **Usability Testing**: To ensure the user interface is responsive, clear, and easy to navigate.

4.3 Types of Testing Conducted

4.3.1 Unit Testing

- **Frontend**: React components like LoginForm, UploadForm were tested individually.
- **Backend**: Node.js controller functions (Login, FileUpload, FileList) were unit tested.

4.3.2 Integration Testing

- Tested integration between Frontend ↔ Backend ↔ AWS.
- Example: Uploading a file from frontend triggers API → Lambda → S3 → DynamoDB correctly.

4.3.3 System Testing

- End-to-end testing of the full system: login → upload file → view dashboard → download file.

4.3. 4 User Acceptance Testing (UAT)

- Conducted to check that the system is easy to use and performs well from a user's point of view.

4.5 Test Cases

Here are some important test cases:

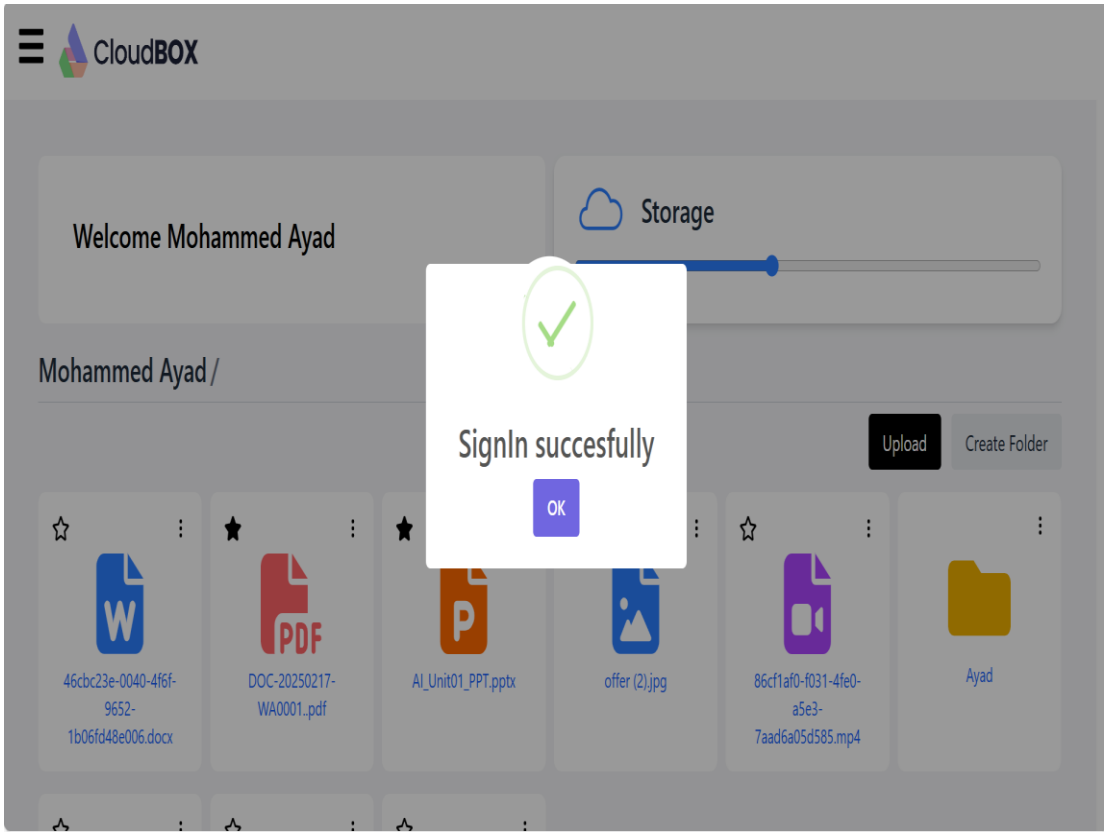
Test Case ID	Description	Input	Expected Output
TC001	User Login	Valid credentia ls	Dashboard page loads
TC002	User Login	Invalid credentials	Show error toast
TC003	Upload File	Select valid file	File saved to S3 metadata saved
TC004	Download File	Click download on file	File downloade d successfully
TC005	Star a File	Click star button	File added to starred list
TC006	Delete File	Click delete button	File removed from S3 and dashboard
TC007	Unauthorized Access	No token provided	Redirect to login page

4.6 Security Testing

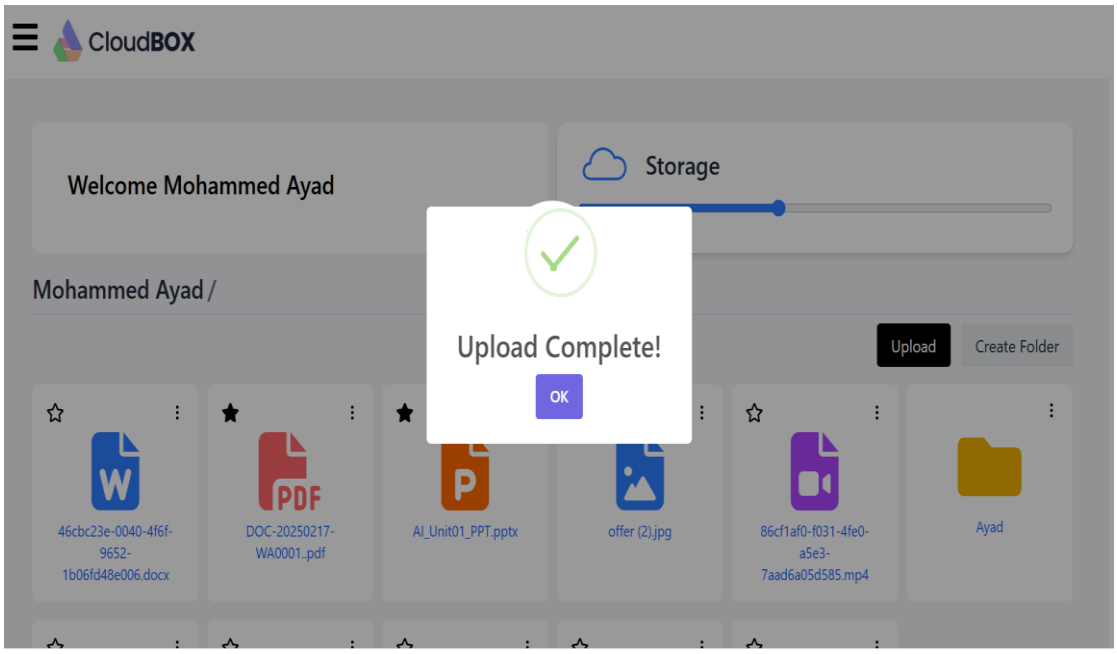
- JWT Token properly set in **HTTP-only** cookies.
- Token verification implemented on **every protected route**.

4.7 Screenshots for Testing Results

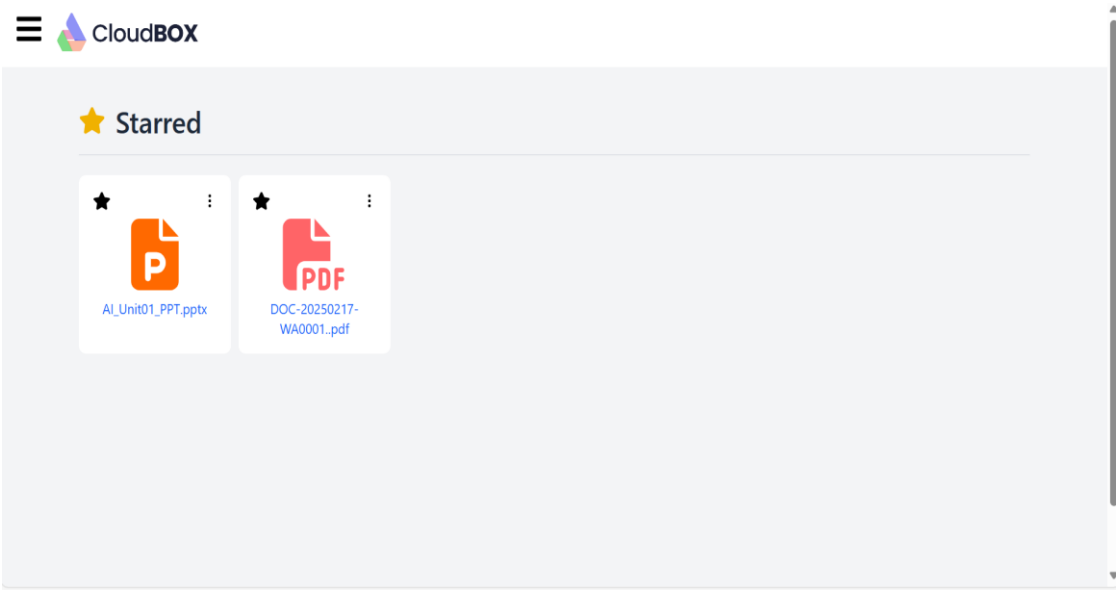
- **Screenshot of Login Success**



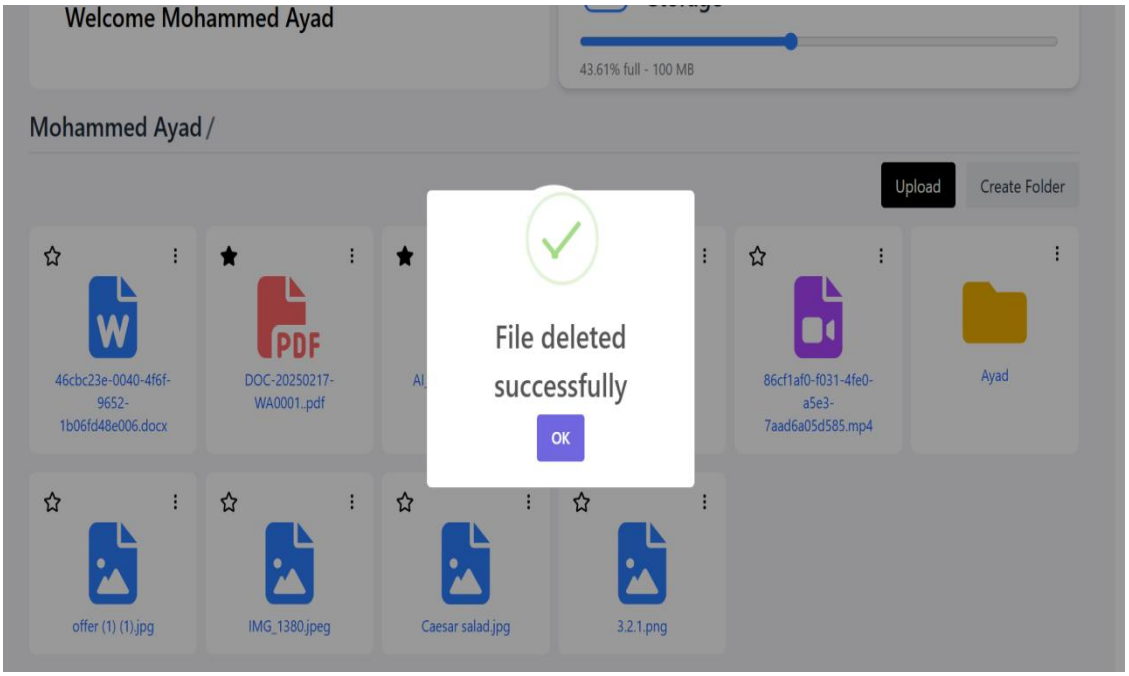
- **Screenshot of Upload Success**



- **Screenshot of Starred File Page**



- **Screenshot of Delete Success**



CHAPTER 5: SYSTEM SECURITY

5.1 Introduction

In any cloud-based system, especially one that involves storing user data and files, security plays a vital role.
Without proper security measures, sensitive user data could be compromised, and unauthorized access to the system could occur.

This project, **Build a Scalable File Processing System**, implements strong security practices across the **Frontend**, **Backend**, and **Cloud (AWS)** to ensure **data confidentiality**, **integrity**, and **availability**.

5.2 Importance of Security in Cloud Applications

Cloud-native applications face unique security challenges because they involve:

- Remote data storage.
- Event-driven processing.
- Serverless architectures where the infrastructure is abstracted.

Thus, it becomes crucial to implement **security at every level** — user authentication, API security, cloud resource access, and secure communication channels.

5.3 Security Measures Implemented

The following security measures were incorporated in this project:

5.3.1 User Authentication (JWT Based)

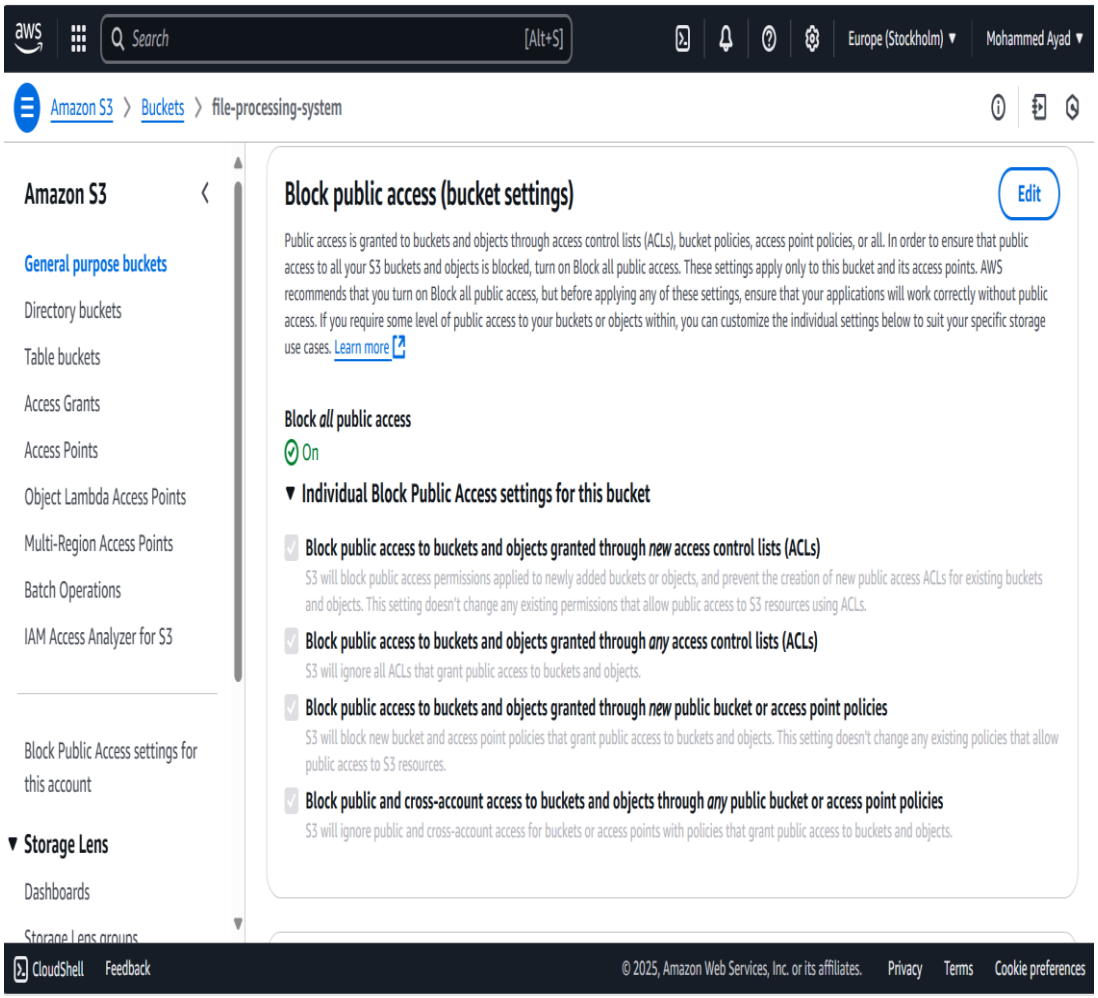
- **JWT Tokens**: JSON Web Tokens are used for user authentication.
- **Secure Storage**: Tokens are stored in **HTTP-only cookies**, which prevents JavaScript access and reduces risk of XSS attacks.
- **Token Verification Middleware**: Every protected API endpoint validates the JWT token before granting access.

5.3.2 HTTPS and Secure Communication

- All communications between frontend and backend are made through **HTTPS** ensuring data-in-transit security.
- Sensitive data like login credentials are securely transmitted over encrypted channels.

5.3.3 AWS IAM Policies and Resource Security

- **AWS IAM Roles**: Minimal permissions are given to Lambda functions and services following the **principle of least privilege**.
- **S3 Bucket Policy**:
  - Public access is disabled on the S3 bucket.
  - Only authenticated users can upload or download files through pre-signed URLs generated via backend APIs.
- **DynamoDB Access**: Only backend Lambda functions and the server can access the database; direct public access is blocked.



Screenshot

5.3.4 Validation and Sanitization

- All user inputs (email, username, password) are **validated** and **sanitized** using libraries like **Joi** on the backend.
- File upload validation (allowed file types, size limits) was also enforced to prevent

malicious file uploads.

5.3.5 Cross-Origin Resource Sharing (CORS)

- **CORS** settings are properly configured to restrict which origins can access the backend APIs.
- Only the allowed frontend domain can make requests to backend APIs.

5.4 System Hardening Practices

Additional best practices used to secure the application include:

- **Environment Variables**: AWS keys, database credentials, and secret keys are stored securely in .env files, not hardcoded.
- **Password Encryption**: User passwords are securely hashed using **bcrypt** before storing in the database.

5.5 Security Testing

Security testing was conducted to verify:

- Authentication and authorization flow.
- Token expiration and invalid token handling.
- Unauthorized access prevention.
- File download links expire after limited time using **S3 signed URLs**.

CHAPTER 6: CONCLUSION

6.1 Introduction

The project titled **"Build a Scalable File Processing System"** was undertaken with the goal of designing a cloud-native, serverless file management solution using AWS technologies.
Throughout the project, an emphasis was placed on building a system that is **scalable**, **secure**, **cost-efficient**, and **highly available**.

6.2 Achievements

The main objectives of the project were successfully achieved:

- Developed a fully **serverless** backend using **AWS Lambda**, **S3**, and **DynamoDB**.
- Implemented a **responsive, user-friendly frontend** with React.js and Tailwind CSS and TypeScript.
- Enabled **secure user authentication** using JWT tokens and HTTP-only cookies.
- Built **REST APIs** for handling file operations such as upload, download, delete, and star.
- Implemented **robust security measures** across all components.

Additionally, a strong understanding of **event-driven architectures**, **cloud-based file management**,
and **serverless computing models** was developed, fulfilling the learning objectives of the project.

6.3 Key Outcomes

- **Efficiency**: Files are processed instantly with event triggers.
- **Scalability**: No need to worry about server load — AWS services scale automatically.
- **Reliability**: S3 provides 99.999999999% (11 nines) durability.
- **Security**: The system enforces authentication, authorization, and secure data storage.

6.4 Conclusion Statement

In conclusion, the project successfully demonstrates how modern cloud services and serverless architectures can be utilized to build powerful, scalable, and cost-effective file
management systems without managing traditional servers.
The skills learned during this project are highly relevant to today's cloud computing industry.

CHAPTER 7: FUTURE ENHANCEMENTS

7.1 Introduction

Although the system in its current form meets the basic requirements,there are several ways it could be extended and enhanced in the future to add more features and scalability.

7.2 Suggested Future Enhancements

7.2.1 Multi-User Role Management

- **Admin**, **Editor**, and **Viewer** roles can be added to provide controlled access based on user types.

7.2. 2 File Sharing Functionality

- Enable secure sharing links so that users can share uploaded files with others externally.

7.2. 3 Version Control

- Implement file versioning using AWS S3 so that users can retrieve older versions of a file.

7.2.4 Multi-File and Folder Upload

- Allow users to upload multiple files and even full folders in a single action, enhancing usability.

7.2.5 File Compression and Optimization

- Integrate functionality where large files are automatically compressed before upload to save storage space.

7.2.6 AI-based File Tagging (Advanced Feature)

- Implement machine learning models to automatically classify and tag uploaded files based on content (e.g., images, documents).

7.2. 7 Mobile App Development

- Build a mobile application using React Native or Flutter to allow users to manage files directly from smartphones.

7.3 Conclusion

Future enhancements can greatly increase the functionality, usability, and value of the system.
Incorporating advanced features like AI tagging, mobile access, and role-based security would further modernize the platform and make it suitable for large-scale deployment in production environments

CHAPTER 9: WEEKLY PROGRESS REPORTS

Build a Scalable File Processing System

Mohammed Ayad

Week	Activities/Work Completed
Week 1	Project planning: selected technologies: AWS Lambda, S3, DynamoDB, React.js, Node.js
Week 2	Frontend design using React.js and Tailwind CSS
Week 3	Backend API development for file handling and user authentication
Week 4	Configured AWS Lambda functions and DynamoDB integration
Week 5	System testing, bug fixing, optimization; project documentation

CHAPTER 9: BIBLIOGRAPHY

Bibliography lists all the references and resources that were used while developing and researching for the project.
Proper citation of materials ensures academic integrity and shows thorough research.

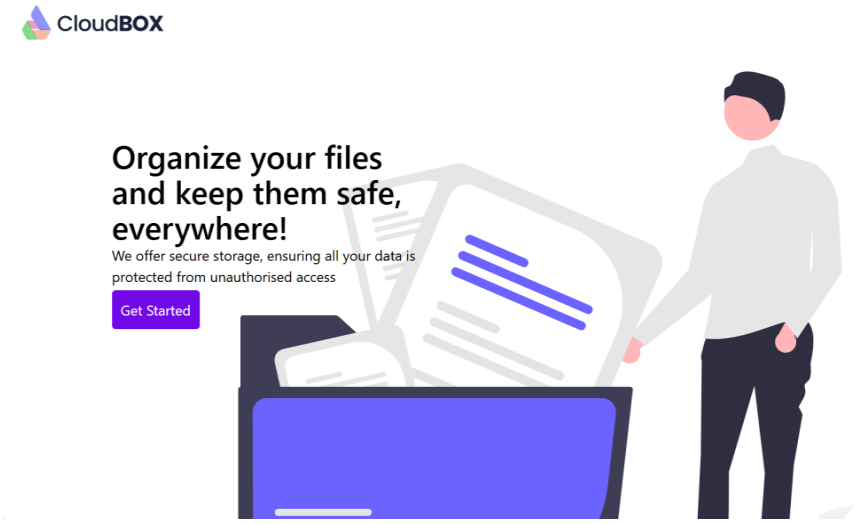
References:

1. **Amazon Web Services (AWS) Documentation**
   https://docs.aws.amazon.com
2. **AWS Lambda Developer Guide**
   https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
3. **AWS S3 Developer Guide**
   https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html
4. **AWS DynamoDB Developer Guide**
   https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
5. **React.js Official Documentation**
   https://react.dev
6. **Amazon Web Services (AWS) Documentation**
   https://docs.aws.amazon.com
7. **AWS Lambda Developer Guide**
   https://docs.aws.amazon.com/lambda/latest/dg/welcome.html
8. **AWS S3 Developer Guide**
   https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html
9. **AWS DynamoDB Developer Guide**
   https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
10. **React.js Official Documentation**
    https://react.dev
11. **Node.js Official Documentation**
    https://nodejs.org/en/docs/
12. **Express.js Guide**
    https://expressjs.com/
13. **MongoDB Official Documentation**
    https://www.mongodb.com/docs/
14. **Tailwind CSS Documentation**
    https://tailwindcss.com/docs
15. **Express.js Guide**
    https://expressjs.com/
16. **MongoDB Official Documentation**
    https://www.mongodb.com/docs/
17. **Tailwind CSS Documentation**
    https://tailwindcss.com/docs

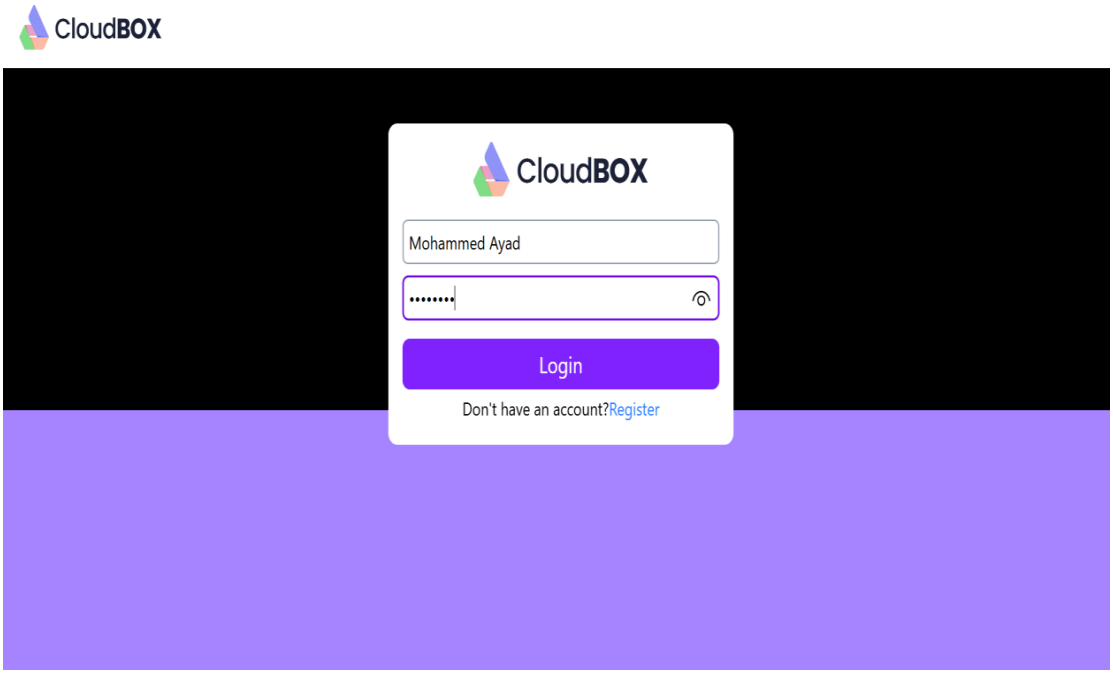
CHAPTER 10: APPENDIX

10.1 User Interface Design

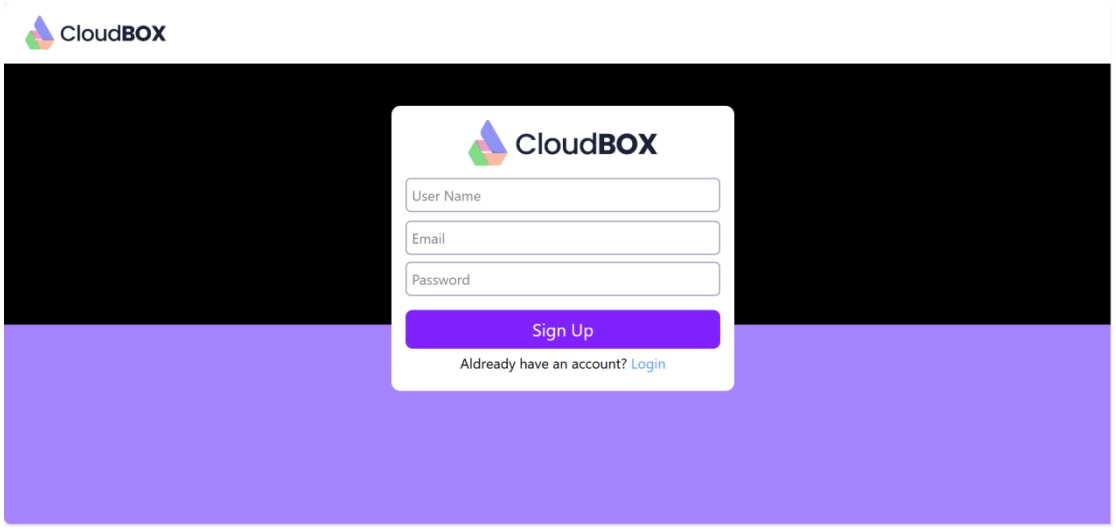
Home Page



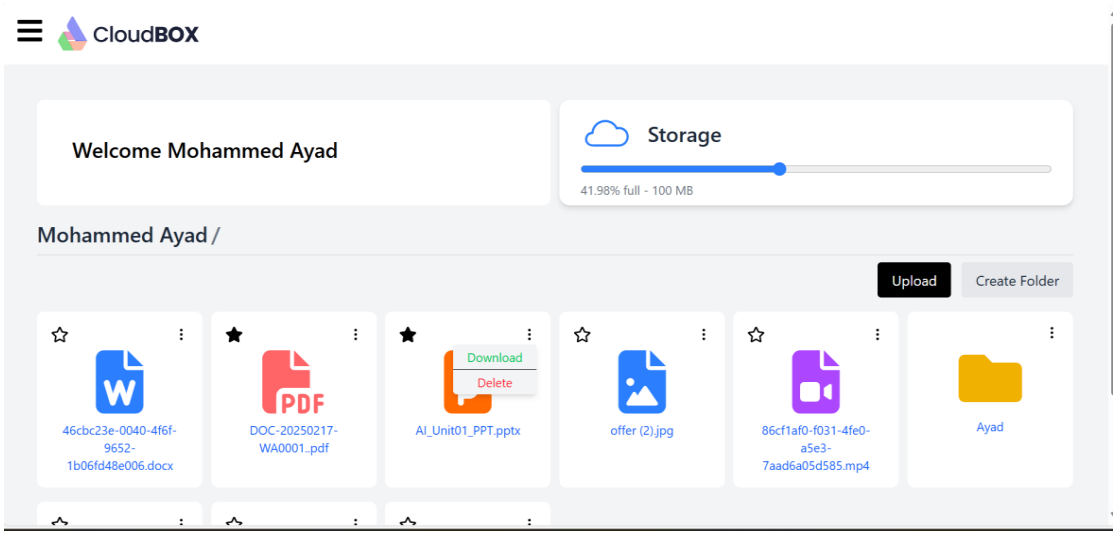
Login Page



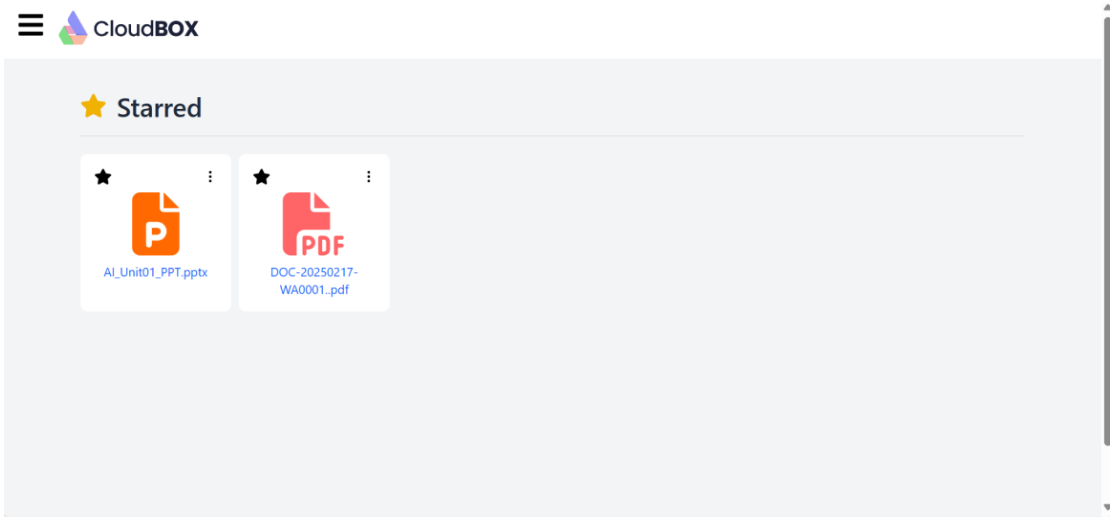
SignUp Page



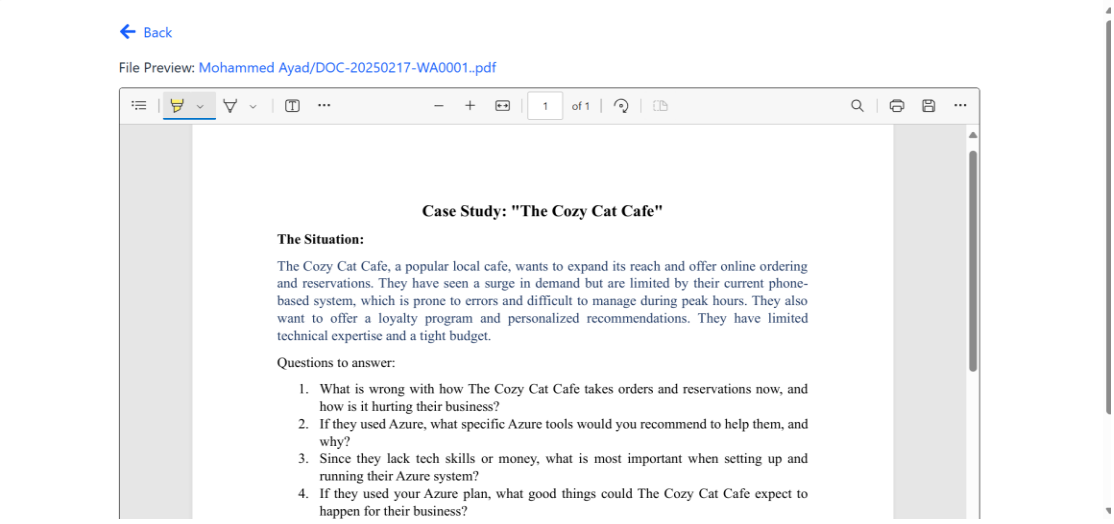
Dashboard Page



Starred File Page

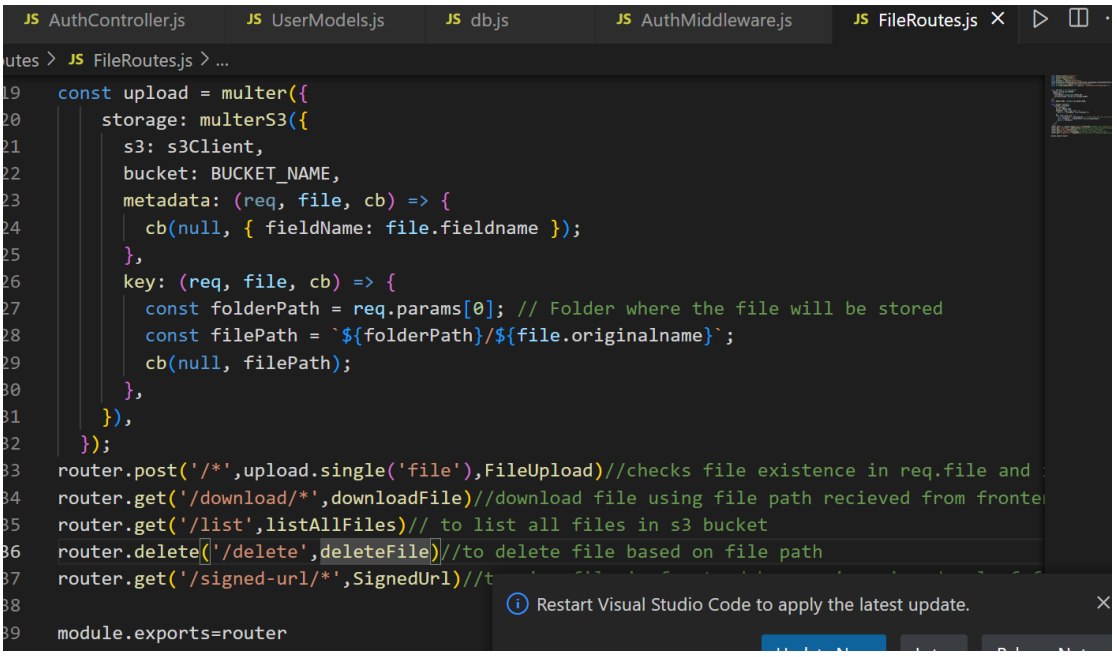


Preview File Page



10.2Backend Interface Design

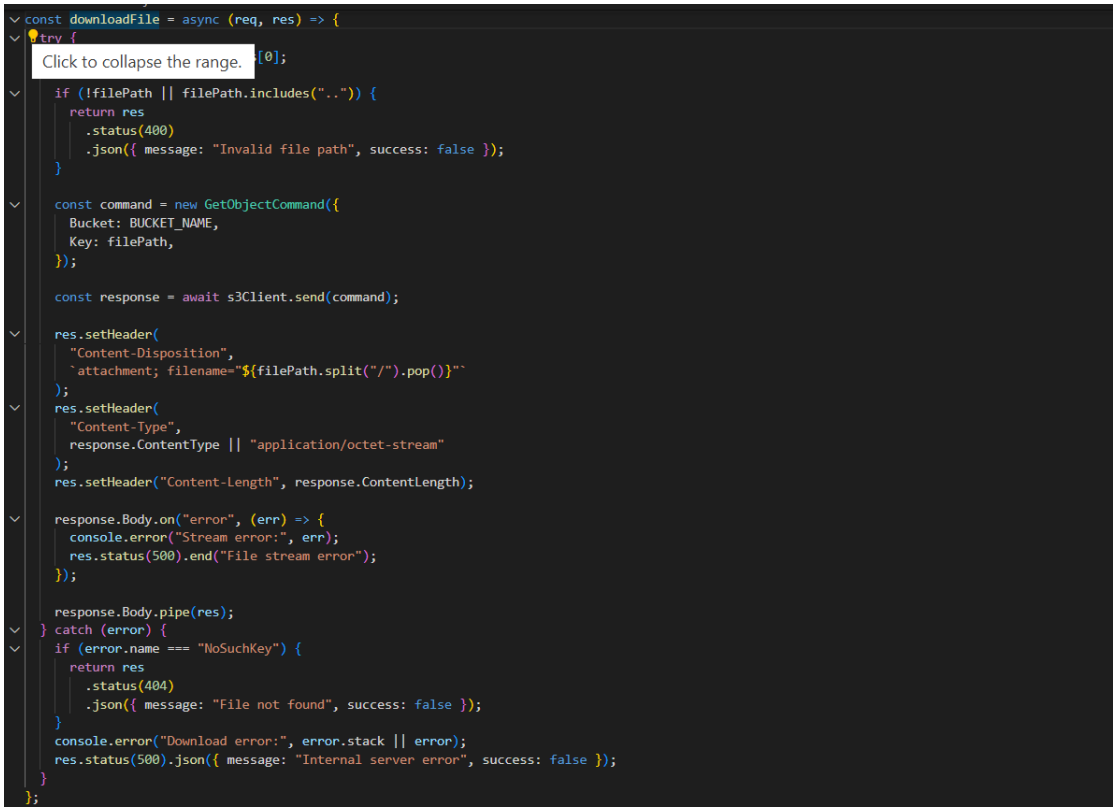
File Routes



File Upload

```
const FileUpload = async (req, res) => {
  try {
    const folderPath = req.params[0];
    const file = req.file;
    const { userName } = req.body;

    // 1. Check if file already exists in DB
    const existingFile = await DirectoryModel.findOne({
      name: file.originalname,
      type: file.mimetype,
      parentPath: folderPath,
    });

    if (existingFile) {
      return res.status(409).json({
        message: "File already exists in this folder",
        success: false,
      });
    }

    const folderSizeMB = await fetchFolderSize(userName); // Ensure you await async call

    const fileSizeMB = file.size / (1024 * 1024); // Convert file size from bytes to MB

    if (folderSizeMB + fileSizeMB >= 100) {
      const deleteCommand = new DeleteObjectCommand({
        Bucket: BUCKET_NAME,
        Key: `${folderPath}/${file.originalname}`,
      });

      await s3Client.send(deleteCommand);
      return res.status(409).json({
        message: "Upload exceeds 100 MB limit.",
        success: false,
      });
    }

    const newFile = new DirectoryModel({
      name: file.originalname,
      path: `${folderPath}/${file.originalname}`,
      parentPath: folderPath,
      type: file.mimetype,
    });

    await newFile.save();

    const updatedFolderData = await DirectoryModel.find({
      parentPath: folderPath,
    });

    return res.status(200).json({
      message: "File uploaded successfully",
      success: true,
      data: updatedFolderData,
    });
  } catch (err) {
    console.error("File upload error:", err);
    return res.status(500).json({
      message: "Internal server error",
      success: false,
    });
  }
};
```

Download File




```
const downloadFile = async (req, res) => {
  try {
    const filePath = req.params[0];

    if (!filePath || filePath.includes("..")) {
      return res
        .status(400)
        .json({ message: "Invalid file path", success: false });
    }

    const command = new GetObjectCommand({
      Bucket: BUCKET_NAME,
      Key: filePath,
    });

    const response = await s3Client.send(command);

    res.setHeader(
      "Content-Disposition",
      `attachment; filename="${filePath.split("/").pop()}"`
    );
    res.setHeader(
      "Content-Type",
      response.ContentType || "application/octet-stream"
    );
    res.setHeader("Content-Length", response.ContentLength);

    response.Body.on("error", (err) => {
      console.error("Stream error:", err);
      res.status(500).end("File stream error");
    });

    response.Body.pipe(res);
  } catch (error) {
    if (error.name === "NoSuchKey") {
      return res
        .status(404)
        .json({ message: "File not found", success: false });
    }
    console.error("Download error:", error.stack || error);
    res.status(500).json({ message: "Internal server error", success: false });
  }
};
```

Delete File

```
const deleteFile = async (req, res) => {
  try {
    const { path, parentPath } = req.body;
    const command = new DeleteObjectCommand({ Bucket: BUCKET_NAME, Key: path });
    await s3Client.send(command);
    await DirectoryModel.findOneAndDelete({ path: path });
    await StarredModel.findOneAndDelete({ path });
    const folderData = await DirectoryModel.find({ parentPath });
    res
      .status(200)
      .json({
        message: "File deleted successfully",
        success: true,
        data: folderData,
      });
  } catch (error) {
    res.status(500).json({ message: "Internal server error", success: false });
  }
};

const SignedUrl = async (req, res) => {
  try {
    const path = req.params[0]; // if your route is router.get("/*", SignedUrl)

    if (!path) {
      return res.status(400).json({
        message: "Missing file path in request",
        success: false,
      });
    }

    const command = new GetObjectCommand({
      Bucket: BUCKET_NAME,
      Key: path,
    });

    const signedUrl = await getSignedUrl(s3Client, command, {
      expiresIn: 3600,
    }); // 1 hour expiry

    res.status(200).json({
      success: true,
      message: "Signed URL generated",
      url: signedUrl,
    });
  } catch (error) {
    console.error("Error generating signed URL:", error);
    return res.status(500).json({
      success: false,
      message: "Internal server error",
      error: error.message,
    });
  }
```

Signed Url

```
const SignedUrl = async (req, res) => {
  try {
    const path = req.params[0]; // if your route is router.get("/*", SignedUrl)

    if (!path) {
      return res.status(400).json({
        message: "Missing file path in request",
        success: false,
      });
    }

    const command = new GetObjectCommand({
      Bucket: BUCKET_NAME,
      Key: path,
    });

    const signedUrl = await getSignedUrl(s3Client, command, {
      expiresIn: 3600,
    }); // 1 hour expiry

    res.status(200).json({
      success: true,
      message: "Signed URL generated",
      url: signedUrl,
    });
  } catch (error) {
    console.error("Error generating signed URL:", error);
    return res.status(500).json({
      success: false,
      message: "Internal server error",
      error: error.message,
    });
  }
};
```

Folder Routes

Routes > JS FolderRoutes.js > [@] <unknown>

```
1   const { DownloadFolder } = require('../Controller/DownloadFolder')
2   const { FetchFolder, CreateFolder, DeleteFolder, getFolderSizeInMB } = require('../Controller/FolderController')
3
4   const router=require('express').Router()
5
6   router.get('/fetch/*',FetchFolder)//to fetch folder and list all files and folder under path
7   router.post('/*',CreateFolder)//to create new folder
8   router.delete('/',DeleteFolder)//delete folder and files under folder path
9   router.get('/download/*',DownloadFolder)//to download folder and files under folder path
10  router.get('/size/*',getFolderSizeInMB)//to get of folder path
11
12
13  module.exports=router
```

Fetch Folder

```
const FetchFolder = async (req, res) => {
  try {
    const folderPath = req.params[0];
    if (!folderPath) {
      return res
        .status(400)
        .json({ message: "Folder path is required", success: false });
    }
    const folderData = await DirectoryModel.find({ parentPath: folderPath });
    res
      .status(200)
      .json({ message: "Folder fetched", success: true, data: folderData });
  } catch (error) {
    res
      .status(500)
      .json({ message: "Internal server error", success: false, error });
  }
};
```

Create Folder

```
const CreateFolder = async (req, res) => {
  try {
    const folderPath = req.params[0];
    const { folderName } = req.body;
    if (!folderName) {
      return res
        .status(409)
        .json({ message: "Folder name required", success: false });
    }
    const filePath = `${folderPath}/${folderName}`;
    const isFolderExist = await DirectoryModel.findOne({
      name: folderName,
      parentPath: folderPath,
      type: "folder",
    });
    if (isFolderExist) {
      return res
        .status(409)
        .json({ message: "Folder already exist", success: false });
    }
    const command = new PutObjectCommand({
      Key: `${filePath}/`,
      Bucket: BUCKET_NAME,
    });
    await s3Client.send(command);
    const newFolder = new DirectoryModel({
      name: folderName,
      parentPath: folderPath,
      path: filePath,
      type: "folder",
    });
    await newFolder.save();
    const folderData = await DirectoryModel.find({ parentPath: folderPath });
    res.status(200).json({ message: "Folder created", success: true ,data:folderData});
  } catch (error) {
    res
      .status(500)
      .json({ message: "Internal server error", success: false, error });
  }
};
```

Delete Folder

```
const DeleteFolder = async (req, res) => {
  try {
    const {path,parentPath}= req.body
    const folderKey = `${path}/`;
    const listCommand = new ListObjectsV2Command({
      Bucket: BUCKET_NAME,
      Prefix: folderKey,
    });
    const listedObjects = await s3Client.send(listCommand);

    if (!listedObjects.Contents || listedObjects.Contents.length === 0) {
      return res
        .status(404)
        .json({ message: "Folder not found or empty folder" });
    }

    const objectsToDelete = listedObjects.Contents.map((item) => ({
      Key: item.Key,
    }));

    const deleteCommand = new DeleteObjectsCommand({
      Bucket: BUCKET_NAME,
      Delete:{
        Objects:objectsToDelete
      }
    });
    await s3Client.send(deleteCommand);
    await DirectoryModel.deleteMany({
      $or: [
        { path: { $regex: `^${path}` } },
        { parentPath: { $regex: `^${path}` } },
      ],
    });
    const folderData=await DirectoryModel.find({parentPath})
     res.status(200).json({message:"Folder deleted",success:true,data:folderData})

  } catch (error) {
    res.status(500).json({message:"Internal server error",success:false,error})
  }
};
```

Download Folder

```
const fs = require("fs-extra");
const path = require("path");
const archiver = require("archiver");
const {
  S3Client,
  ListObjectsV2Command,
  GetObjectCommand,
} = require("@aws-sdk/client-s3");


const s3Client = new S3Client({
    region: process.env.REGION,
    credentials: {
      accessKeyId: process.env.ACCESS_KEY,
      secretAccessKey: process.env.ACCESS_SECRET,
    },
  });
  const BUCKET_NAME = process.env.BUCKET_NAME;

// Download folder as zip
const DownloadFolder= async (req, res) => {
  const folderKey = req.params[0]; // e.g. 'projects/demo'

  const tempFolder = path.join(__dirname, "temp", folderKey); // temp/projects/demo
  const zipPath = path.join(__dirname, "temp", `${path.basename(folderKey)}.zip`);

  try {
    // Step 1: List all S3 objects with the given prefix
    const listCommand = new ListObjectsV2Command({
      Bucket: BUCKET_NAME,
      Prefix: `${folderKey}/`, // trailing slash to limit to the folder
    });
    const listedObjects = await s3Client.send(listCommand);
    if (!listedObjects.Contents || listedObjects.Contents.length === 0) {
      return res.status(404).json({ message: "Folder not found", success: false });
    }
    console.log('step 1');

    // Step 2: Download each file to temp directory
    for (const object of listedObjects.Contents) {
      const key = object.Key;
      console.log(key);

      if (key.endsWith("/")) continue; // skip empty folders
```

```
    await zipFolder(path.join(__dirname, "temp", folderKey), zipPath);

    console.log('step 3');
    let stat;
    try {
      stat = fs.statSync(zipPath);
      res.setHeader('Content-Length', stat.size);
    } catch (err) {
      console.error('Stat error:', err);
      return res.status(500).json({ message: "Error preparing file", success: false });
    }

    // Step 4: Send the zip file
    res.download(zipPath, `${path.basename(folderKey)}.zip`, async (err) => {
      // Optional: cleanup temp files after sending
      await fs.remove(path.join(__dirname, "temp"));
      if (err) {
        console.error("Download error:", err);
      }
    });
    console.log('step 4');

  } catch (error) {
    console.error("Download error:", error);
    res.status(500).json({ message: "Internal server error", success: false });
  }
}

// Helper to stream to buffer
const streamToBuffer = async (stream) => {
  return new Promise((resolve, reject) => {
    const chunks = [];
    stream.on("data", (chunk) => chunks.push(chunk));
    stream.on("end", () => resolve(Buffer.concat(chunks)));
    stream.on("error", reject);
  });
};

// Helper to zip a folder
const zipFolder = (sourceFolder, outPath) => {
  return new Promise((resolve, reject) => {
    const archive = archiver("zip", { zlib: { level: 9 } });
    const output = fs.createWriteStream(outPath);

    output.on("close", resolve);
    archive.on("error", reject);

    archive.pipe(output);
    archive.directory(sourceFolder, false);
    archive.finalize();
  });
};
```

Get Folder Size

```
const getFolderSizeInMB = async (req,res) => {
  try {
    let continuationToken = undefined;
    let totalSizeInBytes = 0;
    const folderPath=req.params[0]

    do {
      const command = new ListObjectsV2Command({
        Bucket: BUCKET_NAME,
        Prefix: `${folderPath}/`,
        ContinuationToken: continuationToken,
      });

      const response = await s3Client.send(command);

      if (response.Contents) {
        response.Contents.forEach((object) => {
          if (!object.Key.endsWith("/")) {
            totalSizeInBytes += object.Size || 0;
          }
        });
      }

      continuationToken = response.IsTruncated ? response.NextContinuationToken : undefined;
    } while (continuationToken);

    const sizeInMB = totalSizeInBytes / (1024 * 1024);
    res.status(200).json({message:'Size fetched',size:sizeInMB.toFixed(2),success:true})
  } catch (error) {
    res.status(500).json({message:'Internal server error',error,success:false})
  }
};
```

Auth Routes

```
routes > JS AuthRoutes.js > ...
1   const { SignUp, Login, verifyToken, SignOut } = require('../Controller/AuthController')
2   const { LoginValidation, SignUpValidation } = require('../Middlewares/AuthMiddleware')
3
4   const router=require('express').Router()
5
6   router.post('/login',LoginValidation,Login)//for login validation middleware and login and passing jwtToken through
7   router.post('/signup',SignUpValidation,SignUp)//for Sign Up validation middleware and login
8   router.get('/verify-token',verifyToken)//to verify user and pass userName
9   router.post('/logout',SignOut)//to logout
10
11
12  module.exports=router
```

Login

```
const Login = async (req, res) => {
  try {
    const { userName, password } = req.body;

    if (!userName || !password) {
      return res.status(400).json({ message: "Please fill all fields" ,success:false});
    }

    const user = await UserModel.findOne({ userName });

    if (!user) {
      return res
        .status(400)
        .json({ message: "User not found", success: false });
    }

    const comparePassword = await bcrypt.compare(password, user.password);

    if (!comparePassword) {
      return res
        .status(400)
        .json({ message: "Incorrect password", success: false });
    }
    const key = process.env.JWT_SECRET_KEY;
    const jwtToken = jwt.sign({ userName }, key, { expiresIn: "5d" });

   res.cookie("token", jwtToken, {
  httpOnly: true,
  secure: false, // Set to true in production with HTTPS
  sameSite: "Lax",
});

    res
      .status(200)
      .json({ message: "SignIn succesfully", success: true, jwtToken });
  } catch (error) {
    res.status(500).json({ message: "Internal error", error, success: false });
  }
};
```

SignUp

```
const SignUp = async (req, res) => {
  try {
    const { email, userName, password } = req.body;

    if (!email || !userName || !password) {
      return res
        .status(404)
        .json({ message: "Please fill all fields", success: false });
    }

    const isEmailExist = await UserModel.findOne({ email }); // it is not showing suggesion for .findOne
    const isUserNameExist = await UserModel.findOne({ userName });

    if (isEmailExist) {
      return res
        .status(400)
        .json({ message: "Email Already Exist", success: false });
    }
    if (isUserNameExist) {
      return res
        .status(400)
        .json({ message: "UserName already Exist", success: false });
    }

    const newUser = new UserModel({ email, userName, password });
    newUser.password = await bcrypt.hash(password, 10);
    await newUser.save();
    const s3Client = new S3Client({
      region: process.env.REGION,
      credentials: {
        accessKeyId: process.env.ACCESS_KEY,
        secretAccessKey: process.env.ACCESS_SECRET,
      },
    });
    const BUCKET_NAME = process.env.BUCKET_NAME;
    const command = new PutObjectCommand({
      Bucket: BUCKET_NAME,
      Key: `${userName}/`,
    });
    const respond = await s3Client.send(command);
    const newDirectory=new DirectoryModel({name:userName,type:"folder",path:`${userName}/`})
    await newDirectory.save()
    res
      .status(201)
      .json({ message: "Signup successfully", success: true });
  } catch (error) {
    res.status(500).json({ message: "Internal server error", success: false });
  }
};
```

Verify Token

```
const verifyToken = async (req, res) => {
  try {
    const token = req.cookies.token;

    if (!token) {
      return res.status(400).json({ message: "Access Denied", success: false });
    }
    const key = process.env.JWT_SECRET_KEY;
    const verify = jwt.verify(token, key);
    if (verify) {
      res.status(200).json({ message: "", success: true, user: verify });
    } else {
      res.status(400).json({ message: "Access denied", success: false });
    }
  } catch (error) {
    res
      .status(500)
      .json({ message: "Internal sever error ", success: false, error });
  }
};
```

Sign Out

```
const verifyToken = async (req, res) => {
  try {
    const token = req.cookies.token;

    if (!token) {
      return res.status(400).json({ message: "Access Denied", success: false });
    }
    const key = process.env.JWT_SECRET_KEY;
    const verify = jwt.verify(token, key);
    if (verify) {
      res.status(200).json({ message: "", success: true, user: verify });
    } else {
      res.status(400).json({ message: "Access denied", success: false });
    }
  } catch (error) {
    res
      .status(500)
      .json({ message: "Internal sever error ", success: false, error });
  }
};
```

Starred Routes

```
const { addToStarred, getStarredFiles, removeFromStarred } = require('../Controller/StarredContoller')

const router=require('express').Router()


router.post('/',addToStarred)//put to starred
router.get('/:userName',getStarredFiles)//fetch starred
router.delete('/:id',removeFromStarred)//remove from starred

module.exports=router
```

Add to Starred

```
const addToStarred = async (req, res) => {
  try {
    const { _id, name, path, parentPath, type, userName } = req.body;
    const newStarredFile = new StarredModel({
      name,
      path,
      parentPath,
      type,
      fileId: _id,
      userName,
    });
    await newStarredFile.save();
    const data = await StarredModel.find({ userName });
    res
      .status(200)
      .json({ message: "File added to starred", success: true, data });
  } catch (error) {
    res
      .status(500)
      .json({ message: "Internal server error", success: false, error });
  }
};
```

Get from Starred

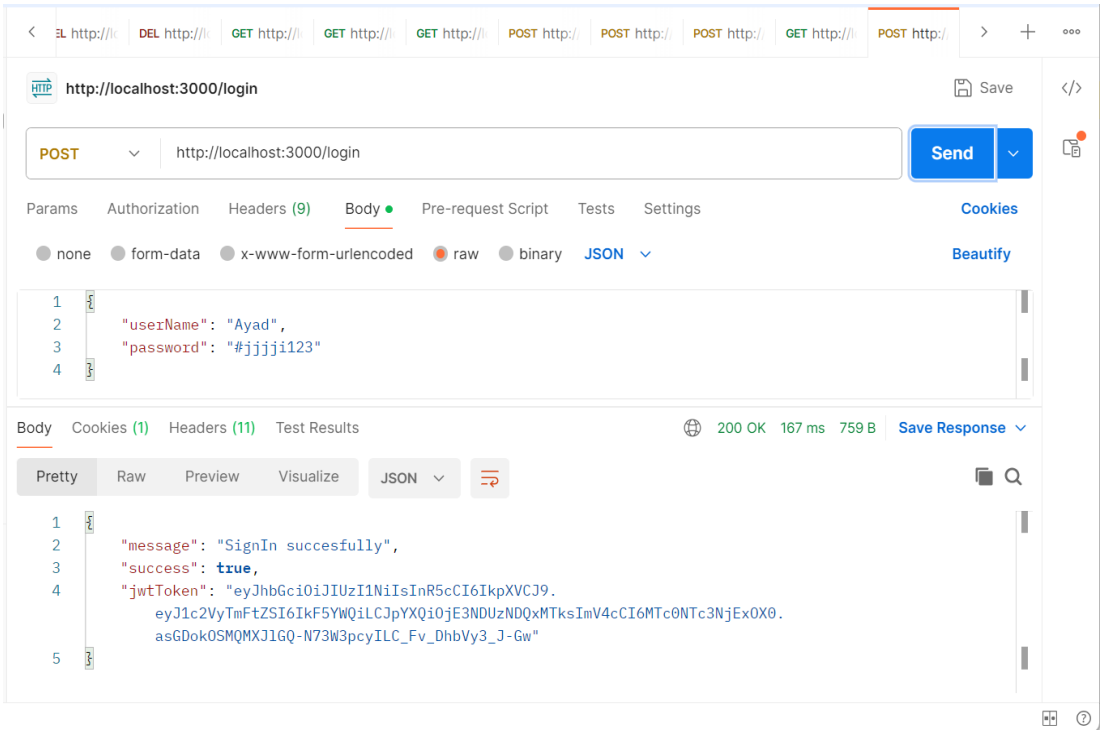
```
const getStarredFiles = async (req, res) => {
  try {
    const { userName } = req.params;
    // If userId is stored in the model, consider using req.user.id
    const data = await StarredModel.find({ userName });
    res
      .status(200)
      .json({ message: "Fetched starred files", success: true, data });
  } catch (error) {
    res
      .status(500)
      .json({ message: "Internal server error", success: false, error });
  }
};
```

Remove from Starred

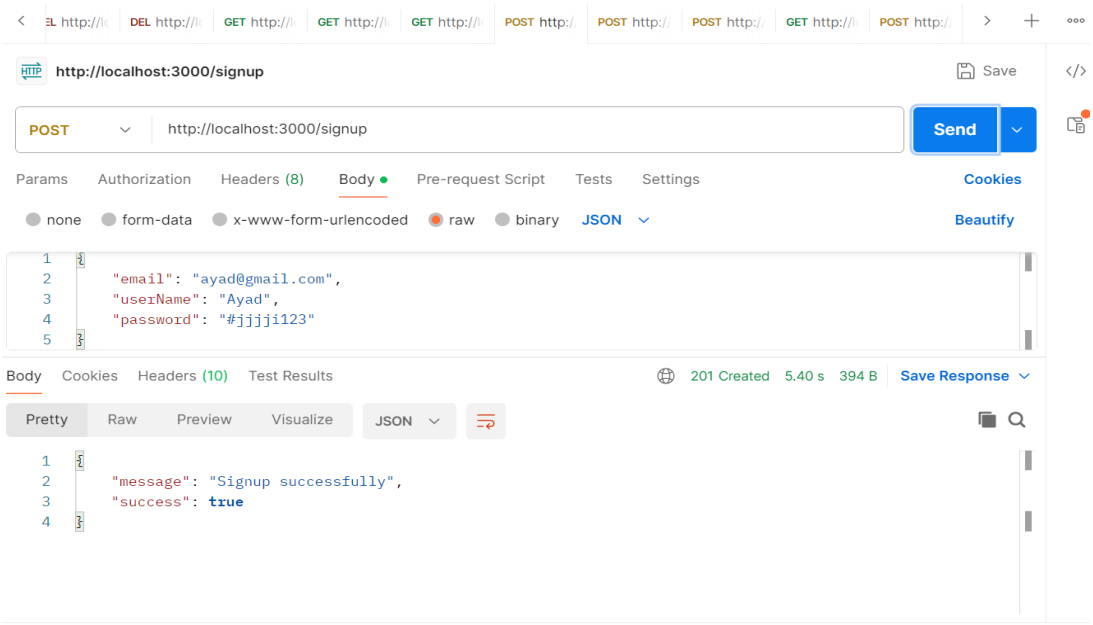
```
const removeFromStarred = async (req, res) => {
  try {
    const { id } = req.params;
    const { userName } = req.query;

    await StarredModel.findOneAndDelete({ fileId: id });

    const data = await StarredModel.find({userName});
    res
      .status(200)
      .json({ message: "File removed from starred", success: true, data });
  } catch (error) {
    res
      .status(500)
      .json({ message: "Internal server error", success: false, error });
  }
};
```

10.2 Backend API Testing Screenshots

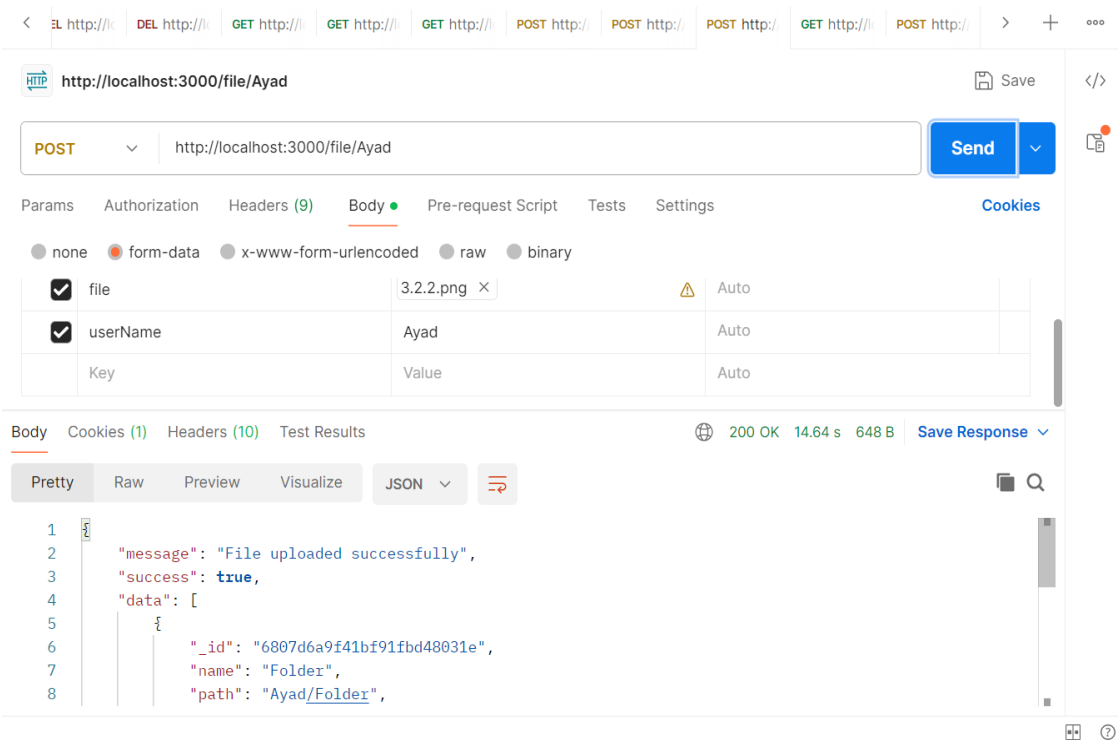
User Login API



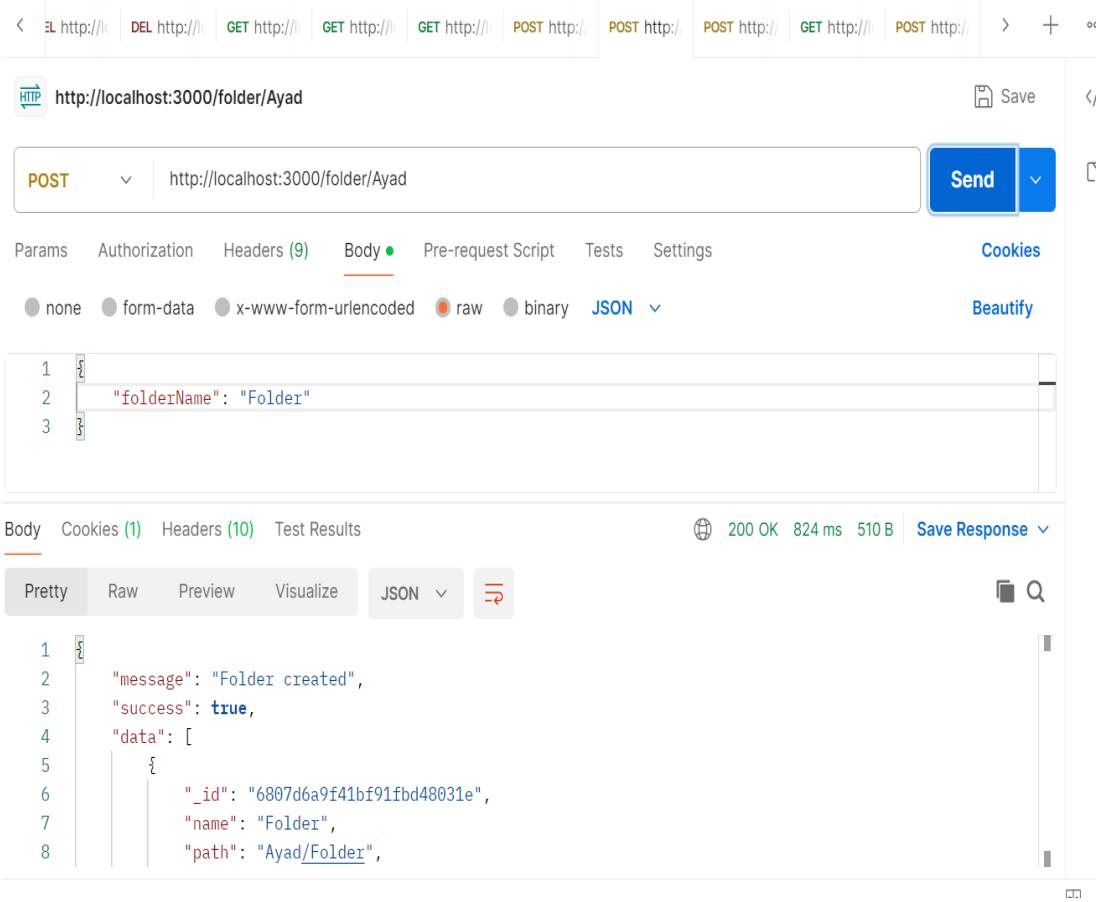
User SignUp API



File Upload API

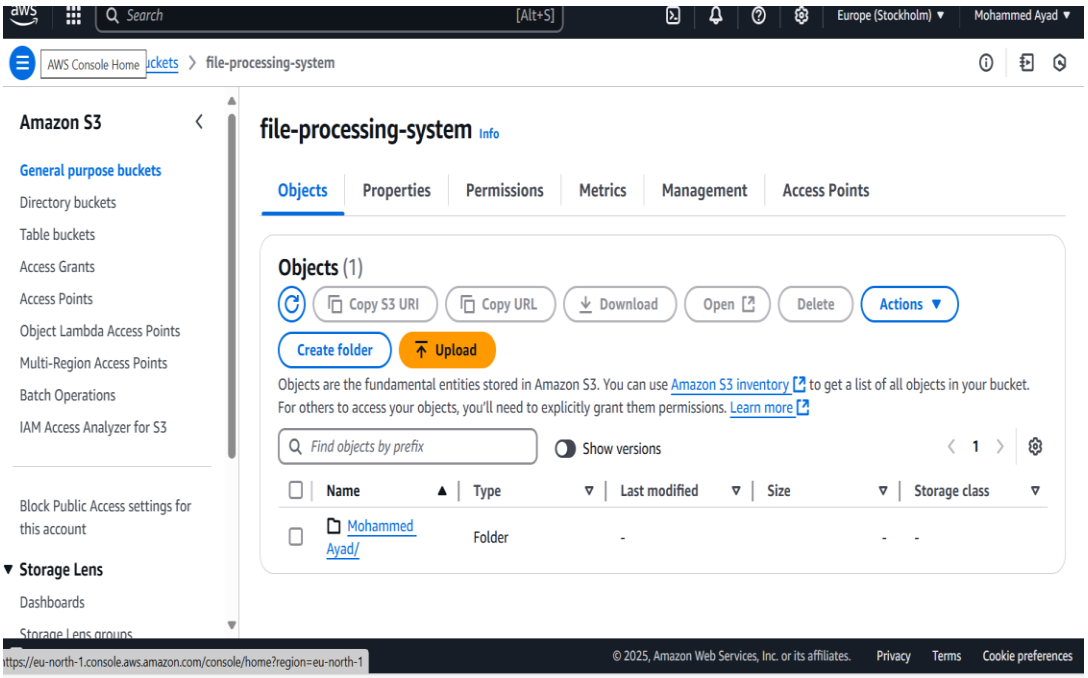


Folder Upload Api

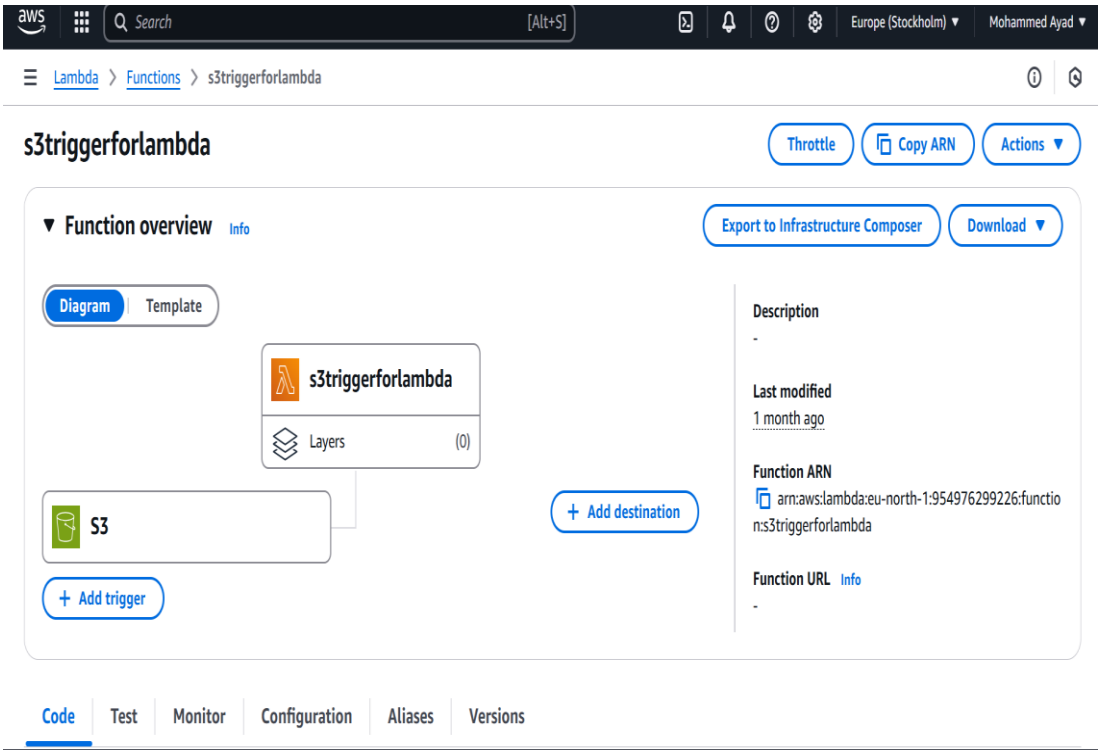


10.1 AWS Console Screenshots

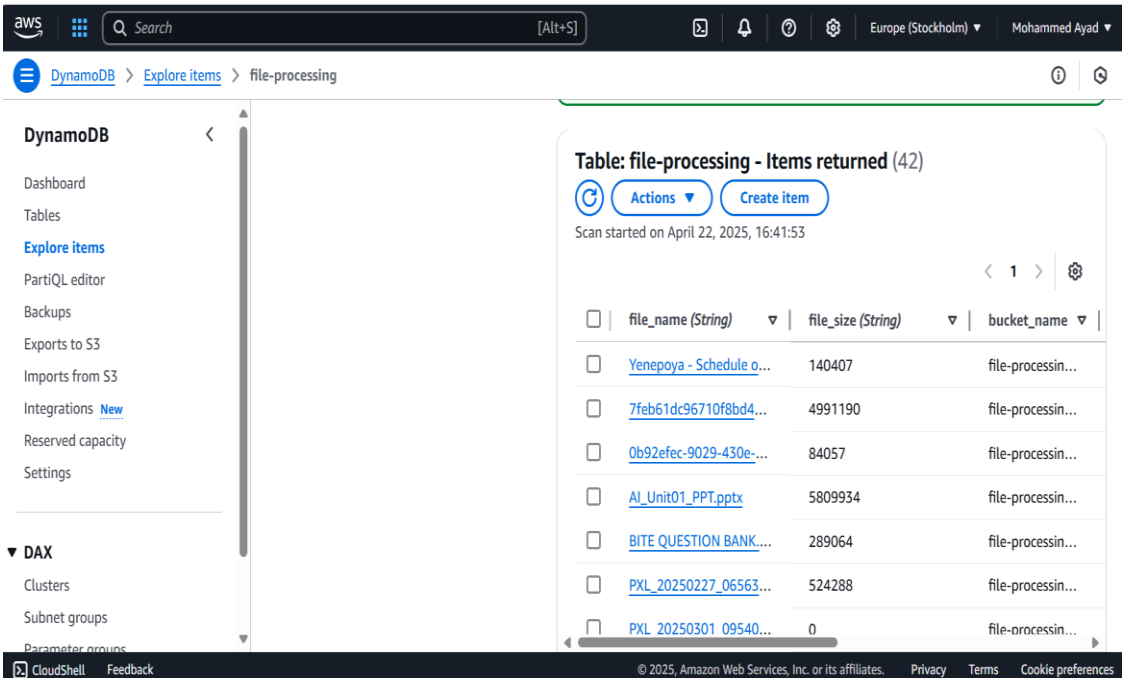
S3 buckets



Lambda triggers



DynamoDB tables.



10.2 MongoDB Model Screenshot

Directory

Models > DirectoryModel.js > DirectorySchema > type

```
1   const mongoose=require('mongoose')
2
3   const DirectorySchema=new mongoose.Schema({
4       name:{
5           required:true,
6           type:String,
7       },
8       path:{
9           required:true,
10          type:String,
11      },
12      parentPath:{
13          required:true,
14          type:String,
15          default:"Users"
16      },
17      type:{
18          required:true,
19          type:String
20      }
21  })
22
23  const DirectoryModel=mongoose.model('Directory',DirectorySchema)
```

Users

Models > JS UserModels.js > UserSchema > userName > unique

```
const mongoose=require('mongoose')

const UserSchema=new mongoose.Schema({
    userName:{
        required:true,
        type:String,
        unique:true
    },
    email:{
        required:true,
        type:String,
        unique:true
    },
    password:{
        required:true,
        type:String,
    },

})

const UserModel=mongoose.model('users',UserSchema)
module.exports=UserModel
```

Starred Files

```
const StarredSchema=new mongoose.Schema({
    fileId:{
        type:mongoose.Schema.Types.ObjectId,
        ref:'Directory',
        required:true
    },
    name:{
        required:true,
        type:String,
    },
    path:{
        required:true,
        type:String,
    },
    parentPath:{
        required:true,
        type:String,
        default:"Users"
    },
    type:{
        required:true,
        type:String
    },
    userName:{
        required:true,
        type:String
    }
})
```