**Real-Time Collaborative Document Editor**

1. INTRODUCTION -

In today's interconnected world, seamless collaboration on documents among multiple users is

essential for productivity and efficiency. The Real-Time Collaborative Document Editor is a

cutting-edge application designed to facilitate simultaneous editing of shared documents by

multiple users in real time. This project harnesses the power of modern web technologies,

including NestJS for the backend, WebSocket for real-time communication, and robust

authentication mechanisms to ensure data security and user privacy.

--------------------------------------------------------------------------------------------------

KEY-FEATURES -

a. Real-Time Collaboration: Users can collaborate on documents in real time, seeing each

other's changes instantaneously without the need to refresh the page.

b. Document Management: The application offers robust document management

capabilities, including creating, updating, and deleting documents, ensuring seamless

operations for users.

c. Authentication and Authorization: Secure authentication and authorization mechanisms

guarantee that only authenticated users have access to edit or view documents,

safeguarding sensitive information.

d. Conflict Resolution: Advanced conflict resolution strategies are implemented to handle

concurrent edits by multiple users, ensuring data integrity and preventing conflicts.

e. Persistence: Documents and user data are persisted in a reliable database system (e.g.,

MongoDB), ensuring data durability and recovery in case of server failures.

--------------------------------------------------------------------------------------------------

TECHNOLOGY STACK

a. NestJS: A progressive Node.js framework for building efficient and scalable serverside applications.

b. WebSocket: A communication protocol that enables bidirectional, full-duplex

communication channels over a single TCP connection, crucial for real-time updates.

c. JWT (JSON Web Tokens): A secure and efficient way to handle authentication and

authorization, ensuring secure user sessions.

d. Mongoose: Object-Relational Mapping (ORM) libraries used for database interactions,

simplifying database operations and ensuring data consistency.

--------------------------------------------------------------------------------------------------

2. PROJECT STRUCTURE -

Realtime-collaborative-doc-editor/

│

├── src/

│ ├── app.module.ts // Main module

│ ├── document/ // Document module

│ │ ├── document.controller.ts // Document controller

│ │ ├── document.service.ts // Document service

│ │ ├── document.entity.ts // Document entity (Mongoose)

│ │ └── document.module.ts // Document module definition

│ │

│ ├── auth/ // Authentication module

│ │ ├── auth.controller.ts // Auth controller (login, register)

│ │ ├── auth.service.ts // Auth service (JWT authentication)

│ │ └── auth.module.ts // Auth module definition

│ │ └── jwt.strategy.ts

│ │ └── guards

│ │ ├── jwt-guards.auth.ts // JWT guard for authentication

│ │

│ ├── user/ // User module

│ │ ├── user.controller.ts // User controller

│ │ ├── user.service.ts // User service

│ │ ├── user.entity.ts // User entity (Mongoose)

│ │ └── user.module.ts // User module definition

│ ├── socket.gateway.ts // WebSocket gateway

│ │

│ └── main.ts // Entry point for the application

├── .env // Environment variables

└── package.json // Project dependencies and scripts

--------------------------------------------------------------------------------------------------

3. API Endpoints

Authentication:

1. POST /auth/register - Register a new user

2. POST /auth/login - Login and receive JWT token

Document Management:

1. GET /documents - Fetch all documents

2. POST /documents - Create a new document

3. GET /documents/:id - Fetch a specific document

4. GET /documentsByUserId/:userId - Getting specific user’s documents

4. PATCH /documents/:id - Update a document

5. DELETE /documents/:id - Delete a document

6. DELETE/documents - Delete all the documents

User Management: (For API testing)

5. GET /users - Fetch all users

6. GET /users/:id - Fetch a specific user

7. PATCH /users/:id - Update a user details

8. DELETE /users/:id - Delete a user

9. DELETE/users - Delete all the users

--------------------------------------------------------------------------------------------------

Maintaining DTO’s (Data Transfer Object) -

Using data transfer object (DTO) for various types of http request such as

createDocumentDto, updateDocumentDto, createUserDto, updateUserDto which becomes

helpful to validate http request and response data.

Class-Validator & Class-Transformer -

Making use of various class validators to validate various attributes in DTO’s such as

@IsNotEmpty(), @IsString(), @IsEnum(), @IsArray() etc. This helps a lot while getting

exact request and response model fields.

--------------------------------------------------------------------------------------------------

5. AUTHENTICATION & AUTHORIZATION -

Importance of JWT-Based Authentication:

1. Stateless Authentication:

JWT allows for stateless authentication, meaning the server does not need to store

session state. This reduces the burden on server resources and simplifies scalability.

2. Secure Communication:

JWTs are cryptographically signed, ensuring that the data contained within them is

tamper-proof. This helps in secure communication between clients and servers.

3. Decentralized Authorization:

JWTs can contain authorization claims, allowing the client to access protected

resources without querying the server for permissions repeatedly.

4. Efficient and Portable:

JWTs are compact and can be easily transmitted as URL parameters, in HTTP headers,

or within the request body, making them suitable for web and mobile applications.

Algorithm Used HS256 :

1. Security and Performance:

HS256 (HMAC with SHA-256) is a widely supported and secure cryptographic

algorithm for JWTs.

It provides a good balance between security and performance, using a symmetric key

(shared secret) for both signing and verifying tokens.

2. Ease of Use:

HS256 is straightforward to implement and configure, making it a popular choice for

many applications.

It requires a single secret key for both token signing and verification.

jwt.strategy.ts & jwt-auth.guard.ts file roles ->

1. jwt.strategy.ts

The jwt.strategy.ts file typically contains the implementation of a custom JWT (JSON Web

Token) strategy using NestJS's @nestjs/passport module. This strategy is responsible for

validating and extracting user information from JWTs provided in HTTP requests.

import { Injectable } from '@nestjs/common';

import { PassportStrategy } from '@nestjs/passport';

import { ExtractJwt, Strategy } from 'passport-jwt';

import { AuthService } from './auth.service';

import \* as dotenv from 'dotenv';

dotenv.config();

@Injectable()

export class JwtStrategy extends PassportStrategy(Strategy) {

constructor(private readonly authService: AuthService) {

super({

jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),

secretOrKey: process.env.JWT\_SECRET, // Change this to your secret key

});

}

async validate(payload: any) {

// `JWT validation successfull - invokes automatically after success `,

console.log(`JWT validated : sharing auth payload :${payload}`);

return { userId: payload.sub, username: payload.username };

}

}

Explanation -

a. JwtStrategy class - Extends PassportStrategy from @nestjs/passport and configures

itself to extract JWTs from the Authorization header using the Bearer scheme

(ExtractJwt.fromAuthHeaderAsBearerToken()).

b. SecretOrKey - Specifies the secret key used to verify the JWT signature

(process.env.JWT\_SECRET).

c. Validate Method - Called by Passport when a request is made with a JWT. It extracts

the payload from the token and passes it to AuthService for user validation

2. jwt-auth.guard.ts

The jwt-auth.guard.ts file implements a custom authentication guard using NestJS's

@nestjs/passport module. This guard is responsible for protecting routes by requiring valid

JWTs for authentication.

Usage in Controllers:

@Controller('documents')

export class DocumentController {

private readonly logger = new Logger(DocumentController.name);

constructor(private readonly documentService: DocumentService) {}

@Get()

@UseGuards(JwtAuthGuard)

async getAllDocuments() {

this.logger.log(`document-controller - getAllDocuments`);

return await this.documentService.getAllDocuments();

}

}

Explanation:

The @UseGuards(JwtAuthGuard) decorator applies the JwtAuthGuard to the

getAllDocuments route, ensuring that only requests with valid JWTs can access this route.

In summary, the jwt.strategy.ts file defines a strategy for validating JWTs and extracting user

information, while the jwt-auth.guard.ts file implements a guard that enforces JWT-based

authentication for protected routes in NestJS controllers.

--------------------------------------------------------------------------------------------------

PERSISTENCE -

Steps ->

a. Setting up Mongoose models,

b. Integrating MongoDB with NestJS,

c. Implementing basic CRUD (Create, Read, Update, Delete) operations for documents

& users

Step 1: Installed Mongoose - npm install --save @nestjs/mongoose mongoose

Step 2: Set Up Mongoose Configuration

Configuring Mongoose to connect to MongoDB using the provided URI from environment

variables (process.env.MONGO\_URI):

// app.module.ts

@Module({

imports: [

MongooseModule.forRoot(process.env.DATABASE\_URL),

AuthModule,

UserModule,

DocumentModule,

MongooseModule.forFeature([

{ name: 'User', schema: UserSchema },

{

name: 'Document',

schema: DocumentSchema,

},

]),

],

controllers: [AppController, UserController, DocumentController],

providers: [AppService, SocketGateway, DocumentService, UserService],

})

export class AppModule {}

Step 3: Defined Mongoose Schema for Document & User -

Document/document.schema.ts

User/user.schema.ts

Step 5: Implemented Document & User Service with CRUD Operations

Step 6: Implemented Document & User Controller

Step 7: Registered Document & User Module and Controller

Step 8: Tested CRUD Operations

--------------------------------------------------------------------------------------------------

6. RUNNING PROJECT -

Backend ->

1. Npm install // To install all dependencies & packages

2. Set Environment Variables: Create a .env file with necessary environment variables

(e.g., JWT\_SECRET, database credentials).

3. Npm start // Start the backend application

Frontend ->

1. Npm install // To install all dependencies

2. Npm start // Start the frontend application

Then I Ensured WebSocket and HTTP server can coexist in NestJS by configuring the

application in such a way that both would be run at same PORT.

I have also implemented error handling, logging, validations as needed throughout the

application.

--------------------------------------------------------------------------------------------------

7. GITHUB REPOSITORY URL -

https://github.com/sudipta274/Collaborative\_tool

--------------------------------------------------------------------------------------------------

8. PROJECT DEMO VIDEO FILE -

Frontend Code Explanation -

Part 1 -

https://www.loom.com/share/db0d9a09a1a64a1196e2fb717db1fea2?sid=df006aac-9ffb416d-9666-a04dc70e5f87

Part 2 -

https://www.loom.com/share/ef97f7ba1e16435baf3506f7e2333ab8?sid=2066ccd3-ddb4-

41d1-8b13-79c8e8da1c48

Backend Code Demonstration -

https://www.loom.com/share/d32ac1f69e454fbab57841eb05fe958a?sid=217e02df-53d2-430b-ac32-c602ed248bb3