# Experiment No 4

**Aim :** REST API Design with MongoDB + Mongoose Integration,

**Theory :**

## 1. Introduction

This report details the design and implementation of a RESTful API using the Node.js runtime environment. The backend architecture leverages the Express.js framework for routing, MongoDB as the NoSQL database, and Mongoose as the Object Data Modeling (ODM) library. The primary objective is to create a secure and scalable API with core functionalities such as user registration, authentication, and access to protected resources.

## 2. Core Components and Technologies

### 2.1 Node.js

Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside a web browser. It is built on the Chrome V8 JavaScript engine and is highly performant due to its event-driven, non-blocking I/O model, which makes it ideal for building data-intensive and real-time applications. The project relies on Node.js to run the server-side logic.

### 2.2 Express.js

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for building web and mobile applications. It simplifies the process of creating routes, handling HTTP requests and responses, and managing middleware. The project uses Express.js to define the API endpoints and manage the request-response cycle.

### 2.3 MongoDB & Mongoose

MongoDB is a document-oriented NoSQL database that stores data in flexible, JSON-like documents. It is highly scalable and allows for a more fluid and less structured data model compared to traditional relational databases.

Mongoose is an elegant ODM library for MongoDB and Node.js. It provides a schema-based solution to model your application data, enforcing a consistent structure while retaining the flexibility of MongoDB. Mongoose handles the connection to the database and provides a clear API for interacting with data collections through schemas and models. In this project, it is used to define the `User` schema and handle all database operations.

## 3. Middleware

Middleware functions are a crucial part of the Express.js framework. They are functions that have access to the request (`req`), response (`res`), and the next middleware function in the application's request-response cycle. They can execute any code, make changes to the request and response objects, and end the request-response cycle or call `next()` to pass control to the next function.

### 3.1 Body Parsers (`express.json`, `express.urlencoded`)

These are built-in Express middleware functions used to parse incoming request bodies.

- `express.json()`: Parses requests with JSON payloads and makes the data available in `req.body`. This is essential for handling `POST` and `PUT` requests that send data in JSON format, like during user login and registration.
- `express.urlencoded({ extended: true })`: Parses URL-encoded payloads, which is useful for handling traditional form submissions.

### 3.2 Security Middleware

- **Helmet:** A collection of 14 middleware functions that set various HTTP headers to protect the application from well-known web vulnerabilities like Cross-Site Scripting (XSS), Clickjacking, and more.
- **CORS (`cors`):** A middleware that enables Cross-Origin Resource Sharing. It allows a web browser on a different domain to make requests to the API, which is necessary for a separate frontend application to interact with the backend.
- **Rate Limiting (`express-rate-limit`):** This middleware protects against brute-force attacks and denial-of-service (DoS) attacks by limiting the number of requests from a single IP address within a specified time window.

### 3.3 Authentication Middleware (`auth.js`)

This is a custom middleware function that verifies a JSON Web Token (JWT) provided in the `Authorization` header of a request.

- It checks for the presence of a token and validates its format (`Bearer <token>`).
- It uses `jwt.verify()` to check if the token is valid, has not expired, and has not been tampered with.
- If the token is valid, it decodes the user's ID and fetches the user from the database. It then attaches the user object to the request (`req.user`) so that subsequent route handlers can access user data.

## 4. User Authentication and Authorization

### 4.1 JSON Web Tokens (JWT)

JWT is a compact, URL-safe means of representing claims to be transferred between two parties. It consists of three parts separated by dots:

- **Header:** Contains the token type (JWT) and the signing algorithm (e.g., HMAC SHA256).
- **Payload:** Contains the claims, which are statements about an entity (typically the user). The project uses the user's ID as a claim.
- **Signature:** A cryptographic signature used to verify that the sender of the JWT is who it claims to be and that the message has not been changed.

When a user logs in, the API generates a JWT and sends it back to the client. The client then stores this token and includes it in the `Authorization` header of every subsequent request to protected routes.

### 4.2 Password Hashing (`bcryptjs`)

Storing passwords in plain text is a major security risk. `bcryptjs` is a library used to hash passwords. Hashing is a one-way process that converts a password into a scrambled string that cannot be reversed. When a user logs in, the provided password is hashed and compared against the stored hash. This ensures that the original password is never stored or exposed.

## 5. Routes and Controllers

### 5.1 Route Separation

The project uses a clean and scalable file structure, separating routes and controllers.

- **Routes (`routes/auth.js`, `routes/protected.js`):** These files define the URL endpoints and link them to the appropriate controller functions.
- **Controllers (`controllers/authController.js`):** These files contain the core business logic for each route, such as creating a new user, logging in, or fetching user data. This separation of concerns makes the codebase more organized and easier to maintain.

### 5.2 API Endpoints

- `POST /api/auth/register`: Public endpoint for creating a new user account.
- `POST /api/auth/login`: Public endpoint for user authentication, which returns a JWT upon successful login.
- `GET /api/auth/me`: A private endpoint protected by the `auth` middleware, which returns the profile of the currently logged-in user.
- `GET /api/protected`: An example of a private endpoint that requires a valid JWT to access.
- `GET /api/admin/users`: A private endpoint that requires both a valid JWT and an `admin` role, demonstrating role-based access control.

## 6. Advanced Concepts and Security (30% Extra)

### 6.1 Cross-Origin Resource Sharing (CORS)

CORS is an HTTP-based security mechanism implemented by web browsers to protect against requests made from a different origin (domain, protocol, or port). By default, browsers enforce the **Same-Origin Policy (SOP)**, which prevents web pages from making requests to a different domain. CORS provides a flexible way to relax this policy.

When a client makes a cross-origin request, the browser sends a **preflight request** to the server first. This preflight request uses the `OPTIONS` method and asks the server for permission to send the actual request. The server's response includes `Access-Control-*` headers (e.g., `Access-Control-Allow-Origin`) to inform the browser whether the request is allowed. The `cors` middleware in Express automates this entire process, ensuring that your API can be securely consumed by a frontend application hosted on a different domain.

### 6.2 Rate Limiting

Rate limiting is a technique used to control the number of requests a client can make to a server over a given period. Its primary purpose is to prevent malicious activities such as **Denial-of-Service (DoS) attacks**, **brute-force login attempts**, and web scraping.

The `express-rate-limit` middleware uses a simple token-bucket algorithm to manage requests. Each IP address is given a "bucket" with a maximum capacity (`max`). For every request, a "token" is taken from the bucket. Tokens are refilled at a set rate over a window of time (`windowMs`). If the bucket runs out of tokens, no further requests from that IP are allowed until the next window begins, and the client receives a `429 Too Many Requests` status.

### 6.3 Protected Routes

A protected route is an API endpoint that can only be accessed by authenticated and authorized users. These routes are critical for securing sensitive data and functionality. The protection is enforced by placing a custom middleware function (like `auth.js`) between the route's path and its controller. The flow is as follows:

1. A client sends a request to a protected route with a JWT in the `Authorization` header.
2. The `auth` middleware intercepts the request.
3. The middleware validates the JWT.
4. If the token is valid, the middleware calls `next()`, passing the request to the next handler (the controller).
5. If the token is missing, expired, or invalid, the middleware immediately sends a `401 Unauthorized` response, ending the request-response cycle and preventing the request from reaching the controller.

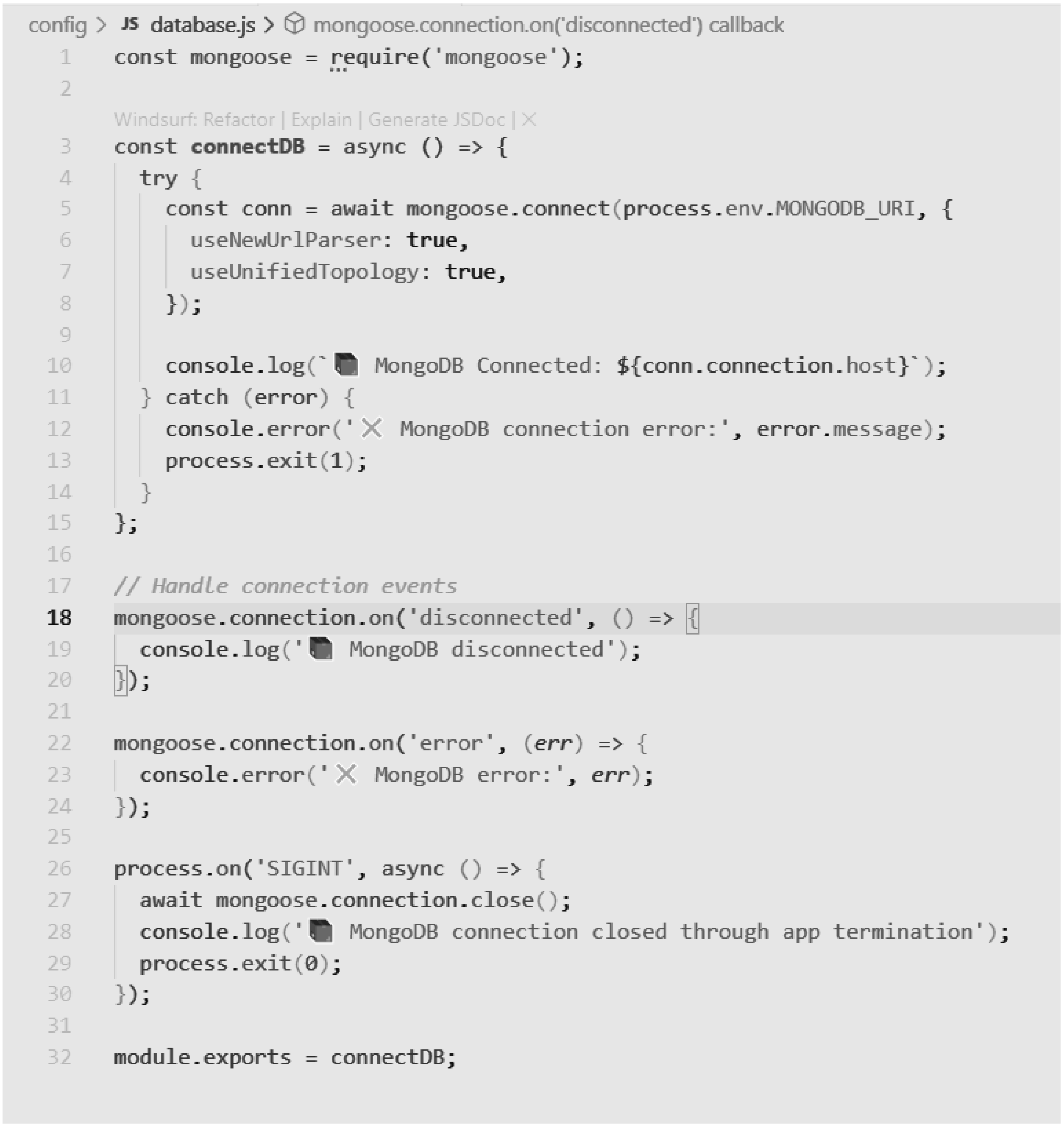
### 6.4 JSON Web Tokens (JWT)

JWTs are a modern and stateless method of authentication. Unlike traditional session-based authentication where the server stores session data, JWTs are self-contained. The token itself holds all the necessary user information (in the payload).

The **stateless** nature of JWTs is a significant advantage. The server does not need to maintain a record of every user session, which makes the application highly scalable and easy to manage across multiple servers. The server only needs to know the secret key to verify the token's signature. As long as the secret key is kept secure on the server, the integrity of the token can be trusted. This enables a distributed and efficient authentication system without the overhead of database lookups on every request.

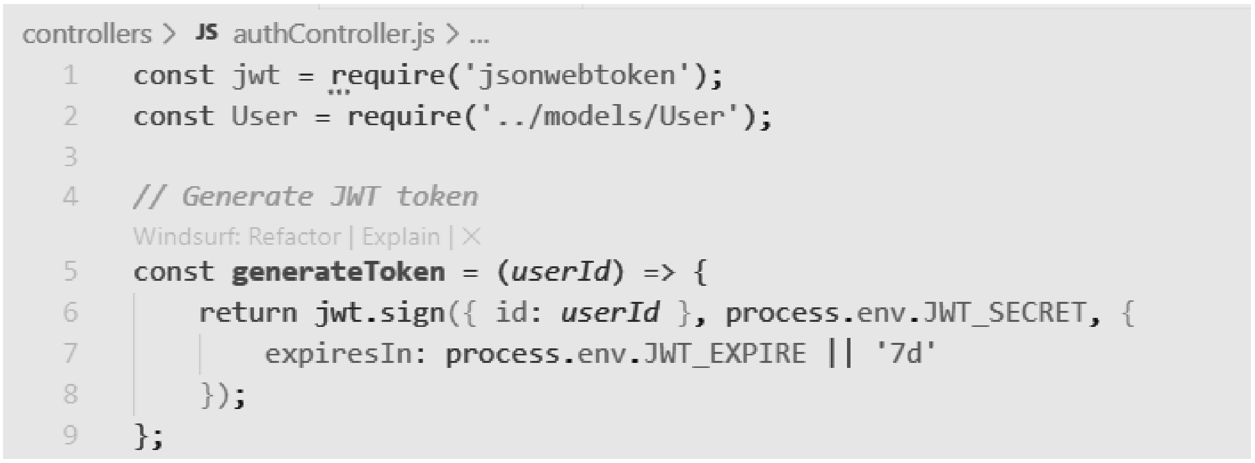
**Code :**

Database.js


config > JS database.js > mongoose.connection.on('disconnected') callback
Windsurf: Refactor | Explain | Generate JSDoc | ×

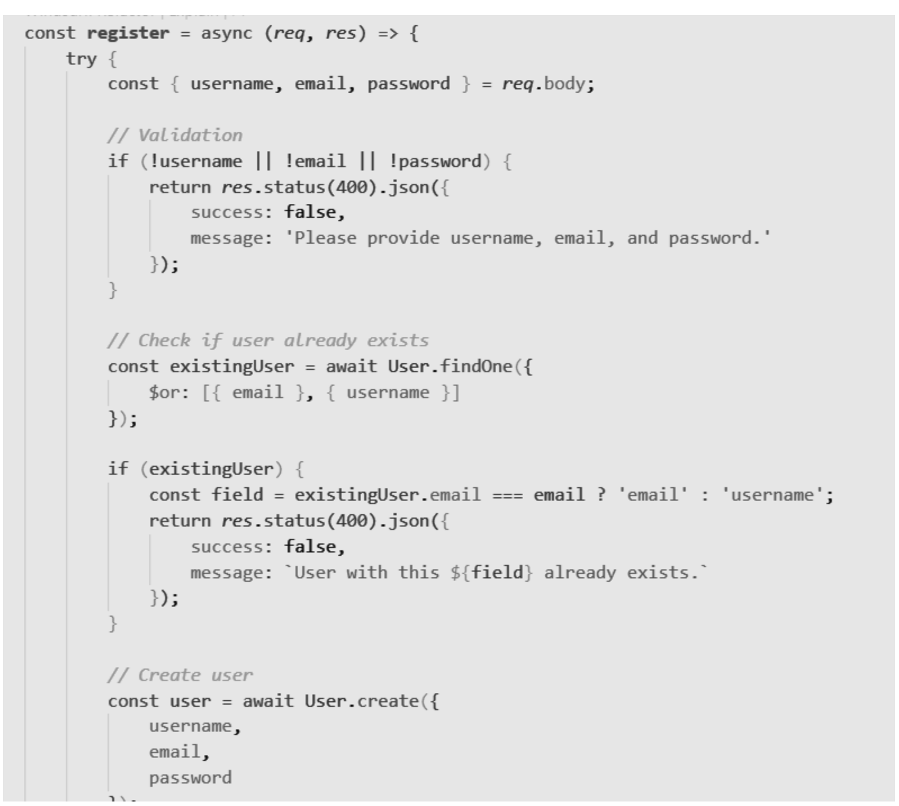
```javascript
const mongoose = require('mongoose');

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGODB_URI, {
      useNewUrlParser: true,
      useUnifiedTopology: true,
    });

    console.log(`📦 MongoDB Connected: ${conn.connection.host}`);
  } catch (error) {
    console.error('❌ MongoDB connection error:', error.message);
    process.exit(1);
  }
};

// Handle connection events
mongoose.connection.on('disconnected', () => {
  console.log('📦 MongoDB disconnected');
});

mongoose.connection.on('error', (err) => {
  console.error('❌ MongoDB error:', err);
});

process.on('SIGINT', async () => {
  await mongoose.connection.close();
  console.log('📦 MongoDB connection closed through app termination');
  process.exit(0);
});

module.exports = connectDB;
```

Create JWT Token :


controllers > JS authController.js > ...
Windsurf: Refactor | Explain | ×

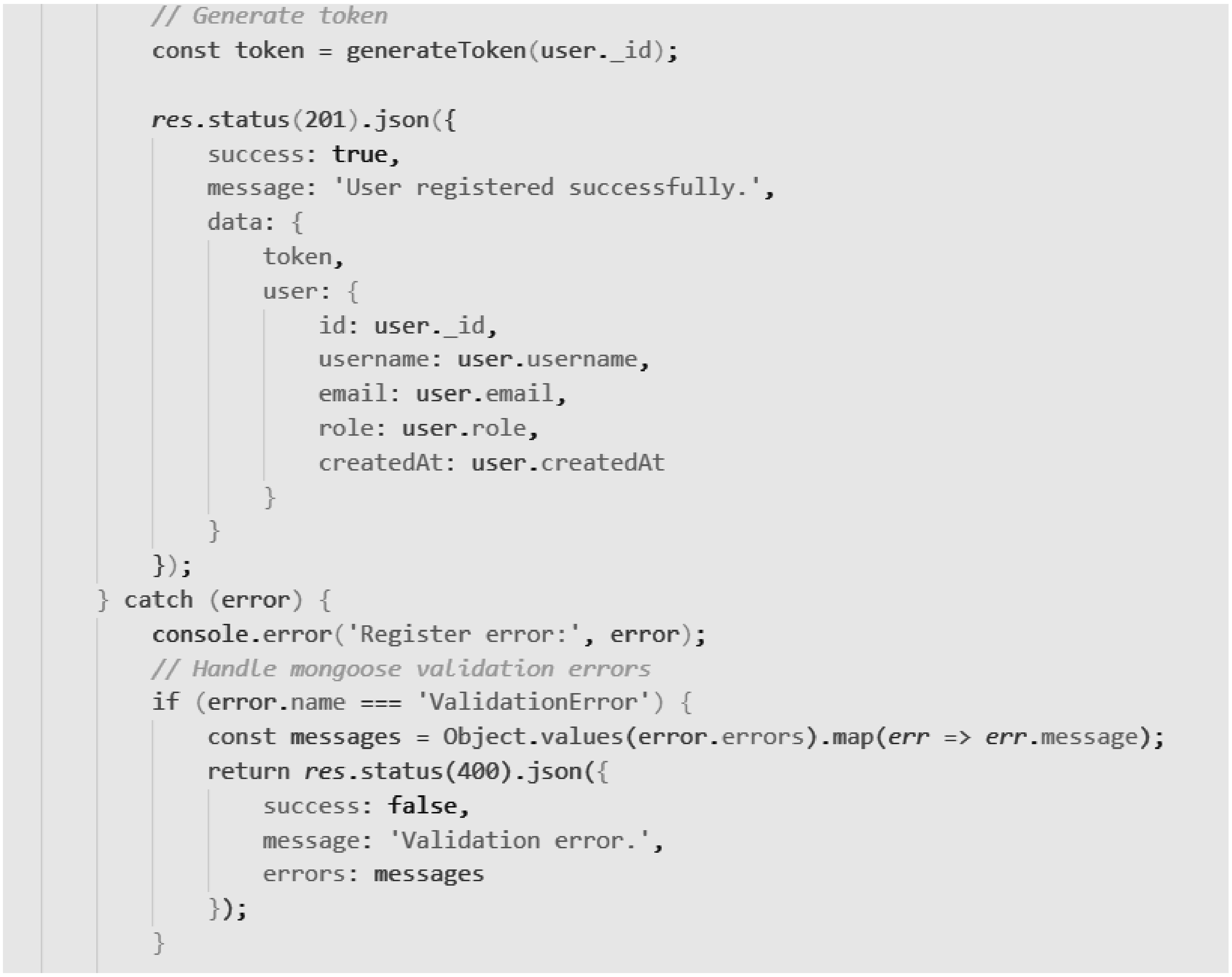
```javascript
const jwt = require('jsonwebtoken');
const User = require('../models/User');

// Generate JWT token
const generateToken = (userId) => {
    return jwt.sign({ id: userId }, process.env.JWT_SECRET, {
        expiresIn: process.env.JWT_EXPIRE || '7d'
    });
};
```

Register a user :



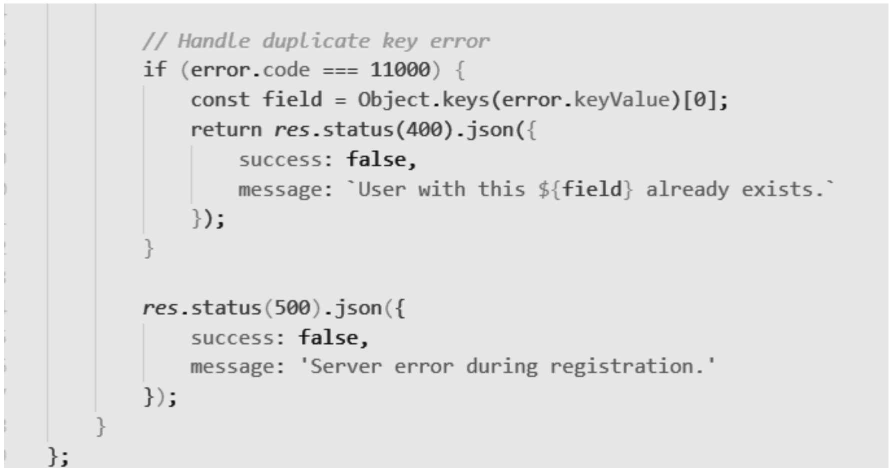
```javascript
const register = async (req, res) => {
    try {
        const { username, email, password } = req.body;

        // Validation
        if (!username || !email || !password) {
            return res.status(400).json({
                success: false,
                message: 'Please provide username, email, and password.'
            });
        }

        // Check if user already exists
        const existingUser = await User.findOne({
            $or: [{ email }, { username }]
        });

        if (existingUser) {
            const field = existingUser.email === email ? 'email' : 'username';
            return res.status(400).json({
                success: false,
                message: `User with this ${field} already exists.`
            });
        }

        // Create user
        const user = await User.create({
            username,
            email,
            password
```

Generate Token :



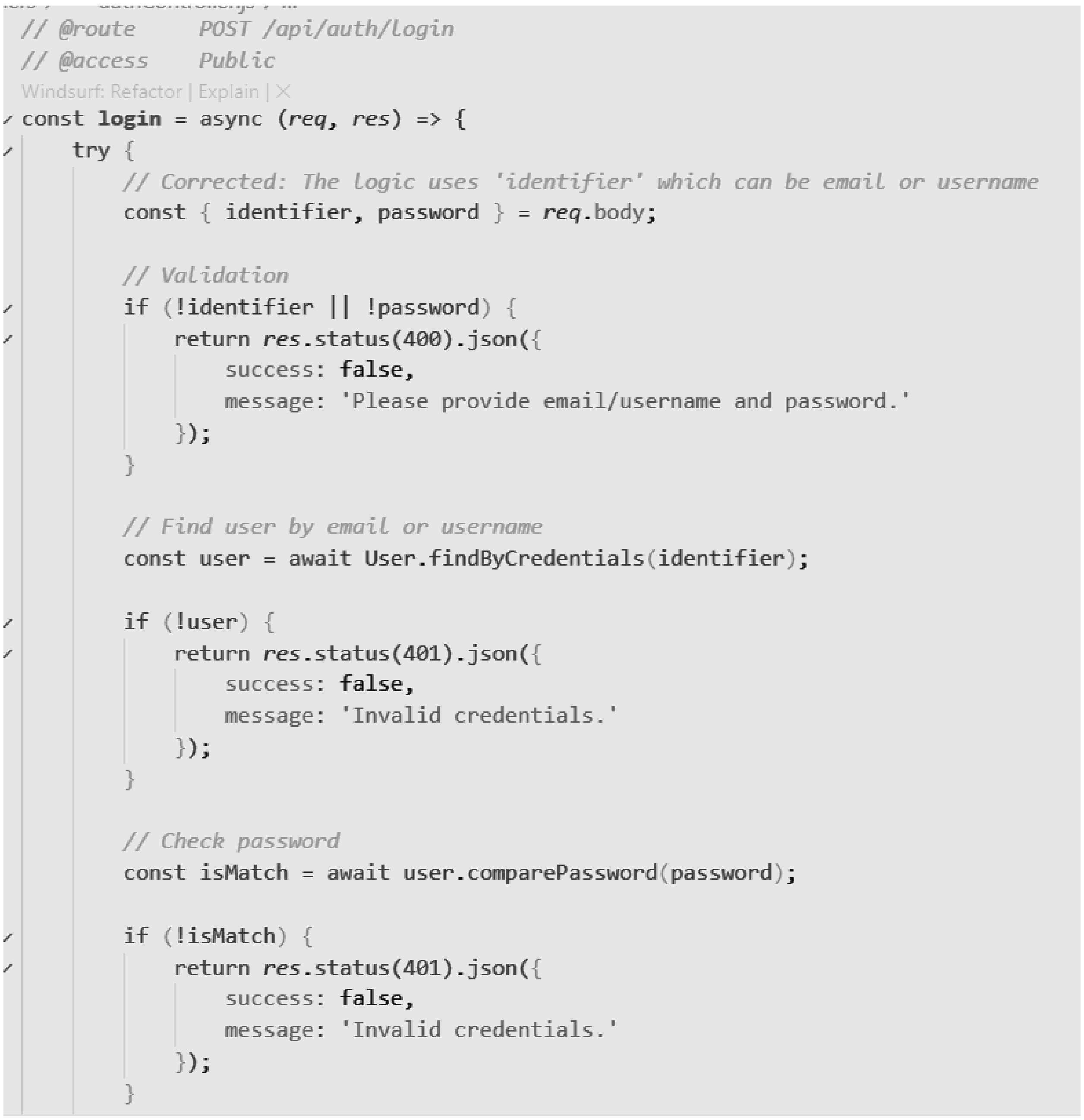
```
        // Generate token
        const token = generateToken(user._id);

        res.status(201).json({
            success: true,
            message: 'User registered successfully.',
            data: {
                token,
                user: {
                    id: user._id,
                    username: user.username,
                    email: user.email,
                    role: user.role,
                    createdAt: user.createdAt
                }
            }
        });
    } catch (error) {
        console.error('Register error:', error);
        // Handle mongoose validation errors
        if (error.name === 'ValidationError') {
            const messages = Object.values(error.errors).map(err => err.message);
            return res.status(400).json({
                success: false,
                message: 'Validation error.',
                errors: messages
            });
        }
```

Handle duplicate key error :



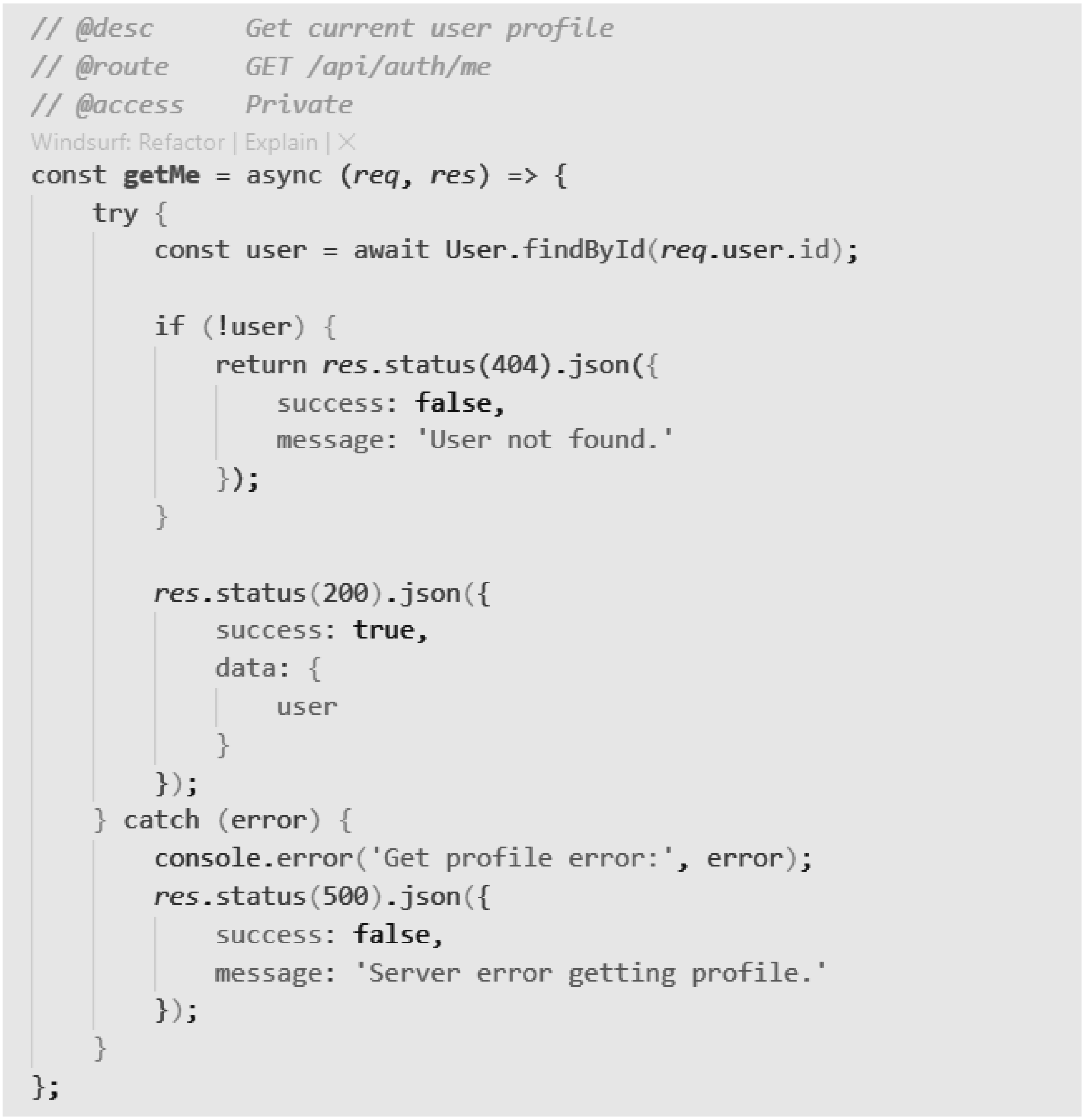
```
        // Handle duplicate key error
        if (error.code === 11000) {
            const field = Object.keys(error.keyValue)[0];
            return res.status(400).json({
                success: false,
                message: `User with this ${field} already exists.`
            });
        }

        res.status(500).json({
            success: false,
            message: 'Server error during registration.'
        });
    }
};
```

Login user through the route POST /api/auth/login



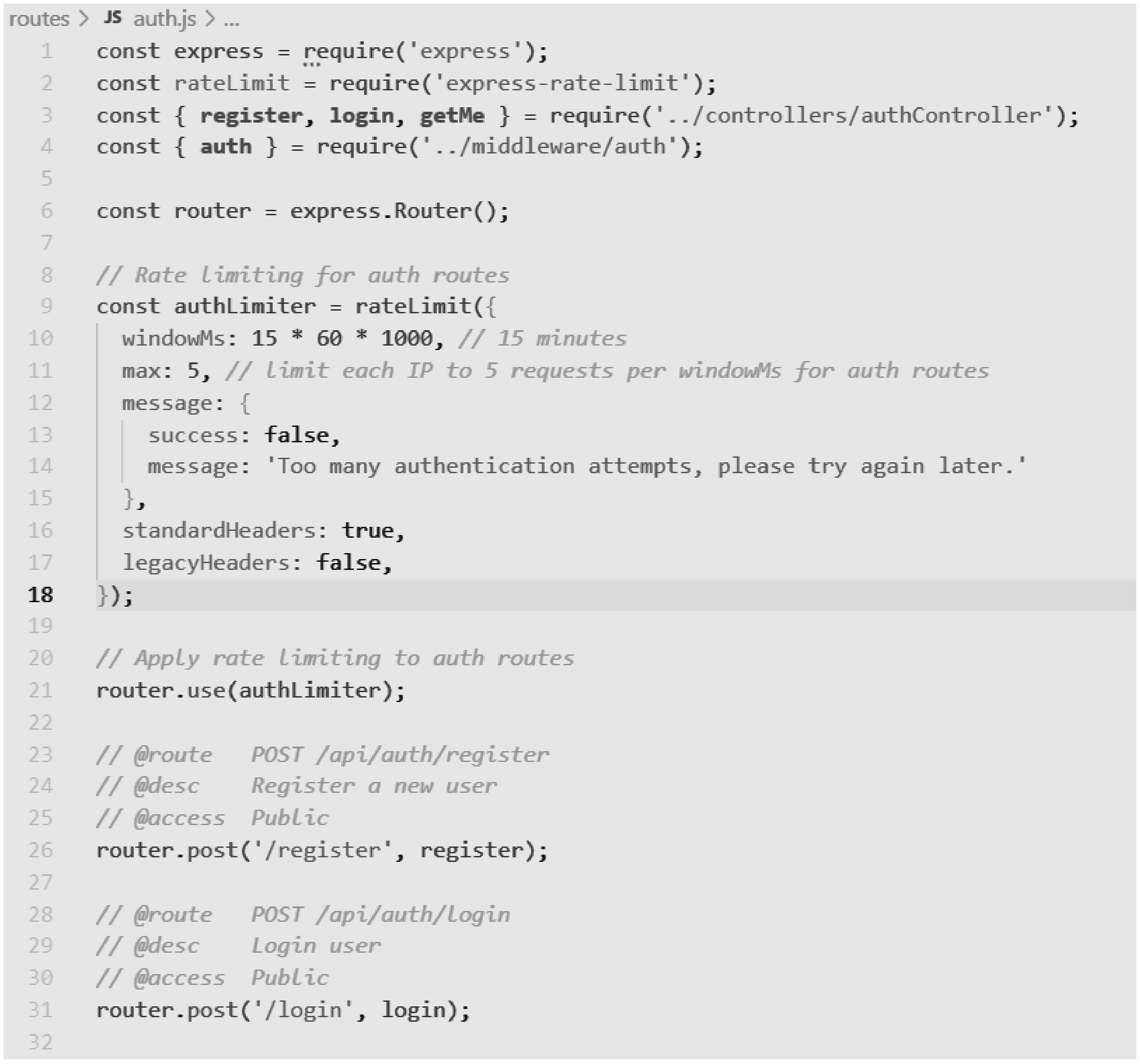
```
// @route     POST /api/auth/login
// @access    Public
const login = async (req, res) => {
    try {
        // Corrected: The logic uses 'identifier' which can be email or username
        const { identifier, password } = req.body;

        // Validation
        if (!identifier || !password) {
            return res.status(400).json({
                success: false,
                message: 'Please provide email/username and password.'
            });
        }

        // Find user by email or username
        const user = await User.findByCredentials(identifier);

        if (!user) {
            return res.status(401).json({
                success: false,
                message: 'Invalid credentials.'
            });
        }

        // Check password
        const isMatch = await user.comparePassword(password);

        if (!isMatch) {
            return res.status(401).json({
                success: false,
                message: 'Invalid credentials.'
            });
        }
```

Get current user profile using the route GET /api/auth/me



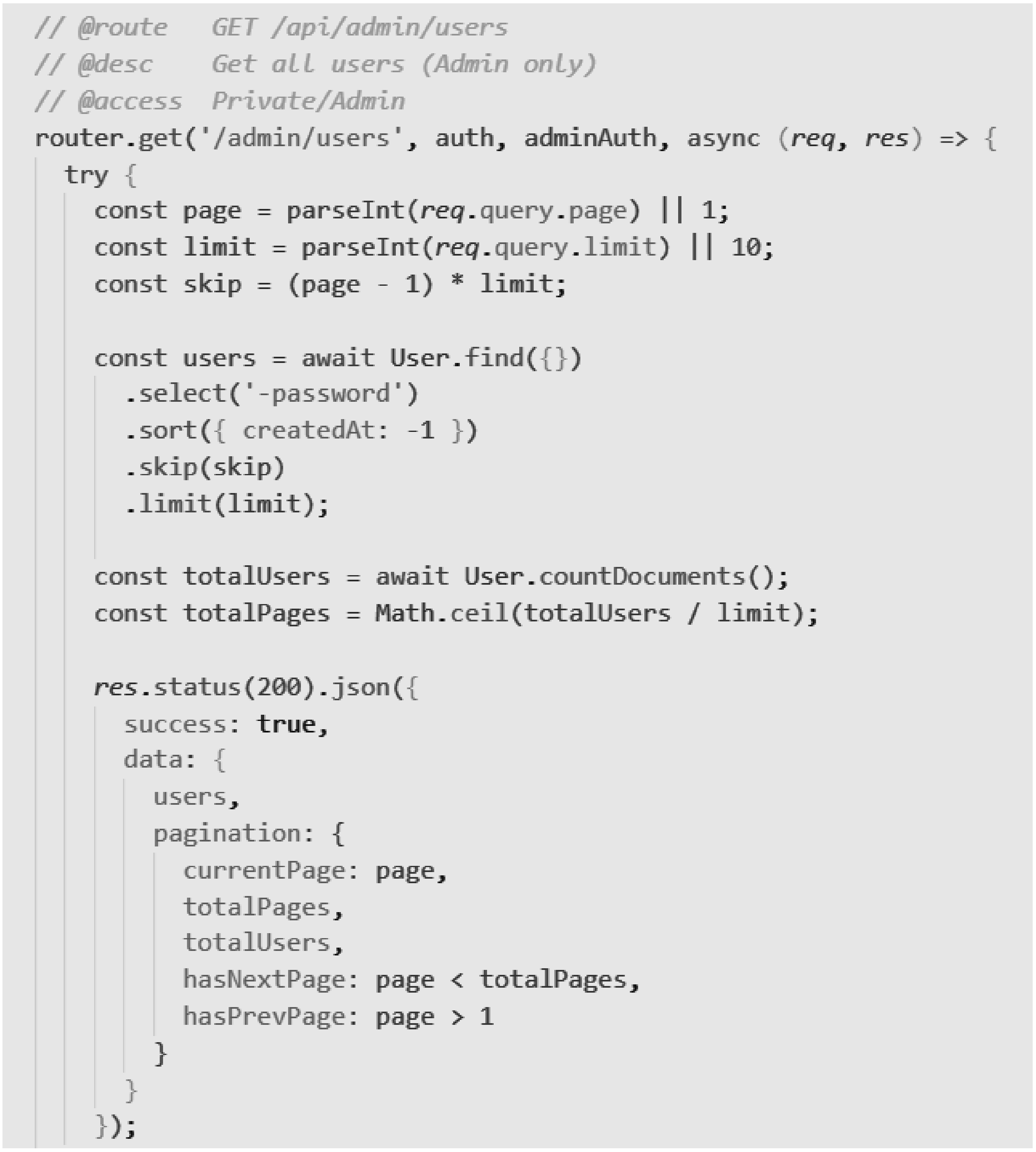
```
// @desc      Get current user profile
// @route     GET /api/auth/me
// @access    Private
Windsurf: Refactor | Explain | ×
const getMe = async (req, res) => {
    try {
        const user = await User.findById(req.user.id);

        if (!user) {
            return res.status(404).json({
                success: false,
                message: 'User not found.'
            });
        }

        res.status(200).json({
            success: true,
            data: {
                user
            }
        });
    } catch (error) {
        console.error('Get profile error:', error);
        res.status(500).json({
            success: false,
            message: 'Server error getting profile.'
        });
    }
};
```

Rate Limiter in Auth (30% Extra )



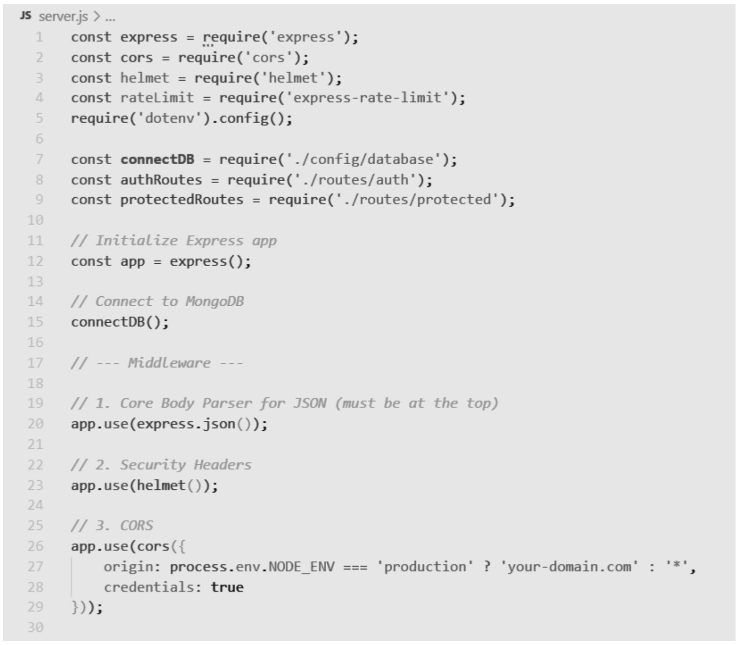
```javascript
const express = require('express');
const rateLimit = require('express-rate-limit');
const { register, login, getMe } = require('../controllers/authController');
const { auth } = require('../middleware/auth');

const router = express.Router();

// Rate limiting for auth routes
const authLimiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 5, // limit each IP to 5 requests per windowMs for auth routes
  message: {
    success: false,
    message: 'Too many authentication attempts, please try again later.'
  },
  standardHeaders: true,
  legacyHeaders: false,
});

// Apply rate limiting to auth routes
router.use(authLimiter);

// @route   POST /api/auth/register
// @desc    Register a new user
// @access  Public
router.post('/register', register);

// @route   POST /api/auth/login
// @desc    Login user
// @access  Public
router.post('/login', login);

```

Protected routes (30% extra)

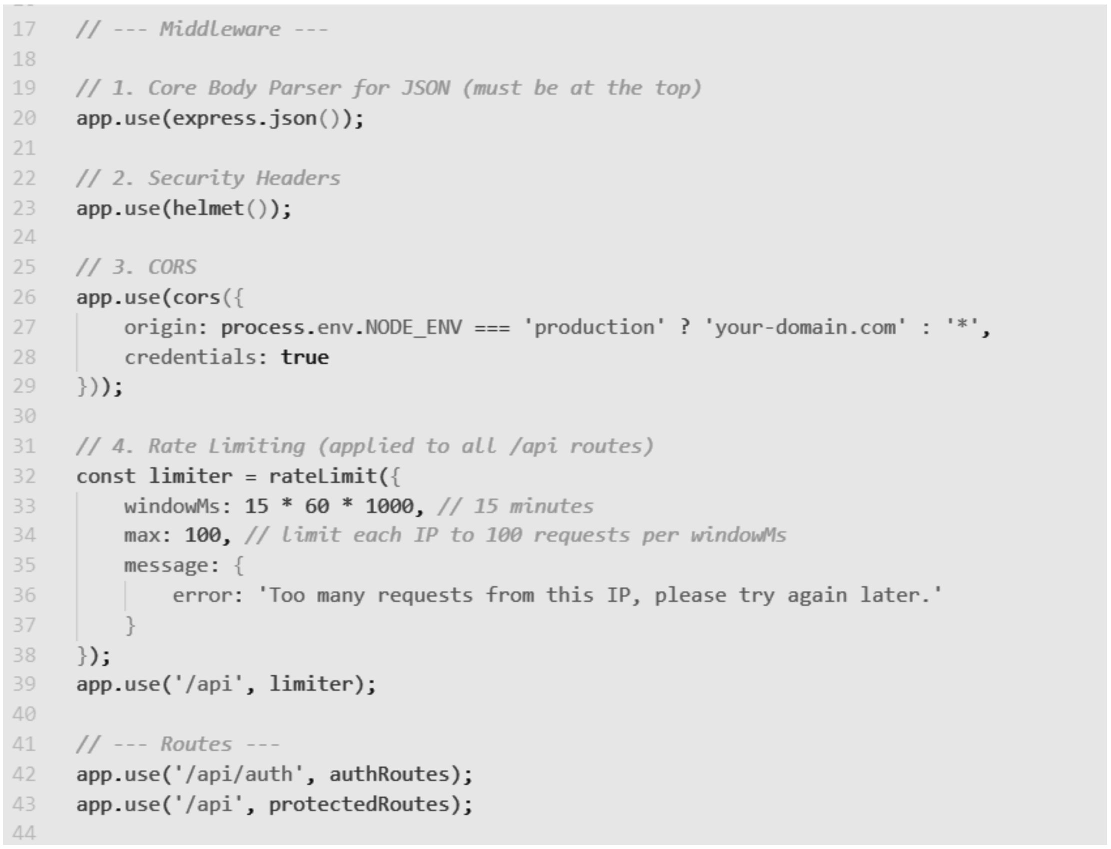


```
// @route   GET /api/admin/users
// @desc    Get all users (Admin only)
// @access  Private/Admin
router.get('/admin/users', auth, adminAuth, async (req, res) => {
  try {
    const page = parseInt(req.query.page) || 1;
    const limit = parseInt(req.query.limit) || 10;
    const skip = (page - 1) * limit;

    const users = await User.find({})
      .select('-password')
      .sort({ createdAt: -1 })
      .skip(skip)
      .limit(limit);

    const totalUsers = await User.countDocuments();
    const totalPages = Math.ceil(totalUsers / limit);

    res.status(200).json({
      success: true,
      data: {
        users,
        pagination: {
          currentPage: page,
          totalPages,
          totalUsers,
          hasNextPage: page < totalPages,
          hasPrevPage: page > 1
        }
      }
    });
```

## Server.js



```javascript
const express = require('express');
const cors = require('cors');
const helmet = require('helmet');
const rateLimit = require('express-rate-limit');
require('dotenv').config();

const connectDB = require('./config/database');
const authRoutes = require('./routes/auth');
const protectedRoutes = require('./routes/protected');

// Initialize Express app
const app = express();

// Connect to MongoDB
connectDB();

// --- Middleware ---

// 1. Core Body Parser for JSON (must be at the top)
app.use(express.json());

// 2. Security Headers
app.use(helmet());

// 3. CORS
app.use(cors({
    origin: process.env.NODE_ENV === 'production' ? 'your-domain.com' : '*',
    credentials: true
}));

```



```javascript
// --- Middleware ---

// 1. Core Body Parser for JSON (must be at the top)
app.use(express.json());

// 2. Security Headers
app.use(helmet());

// 3. CORS
app.use(cors({
    origin: process.env.NODE_ENV === 'production' ? 'your-domain.com' : '*',
    credentials: true
}));

```

```javascript
// 4. Rate Limiting (applied to all /api routes)
const limiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 100, // limit each IP to 100 requests per windowMs
    message: {
        error: 'Too many requests from this IP, please try again later.'
    }
});
app.use('/api', limiter);

// --- Routes ---
app.use('/api/auth', authRoutes);
app.use('/api', protectedRoutes);

```

## Outputs :

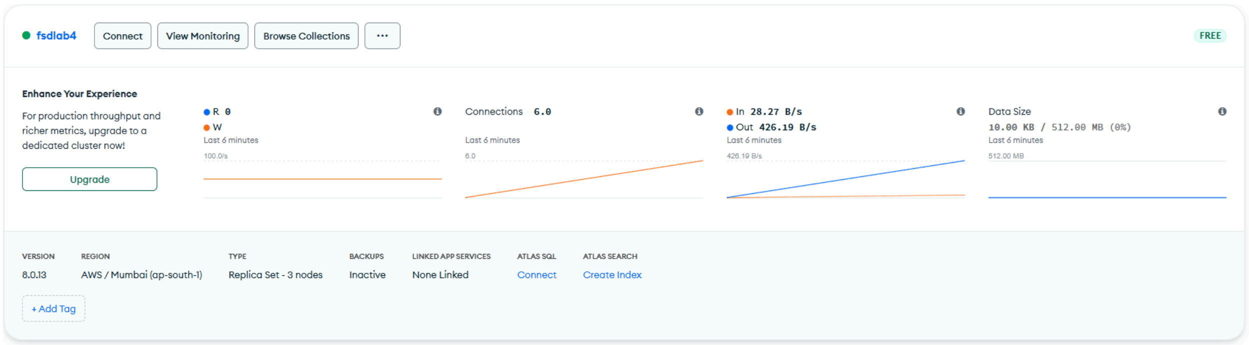
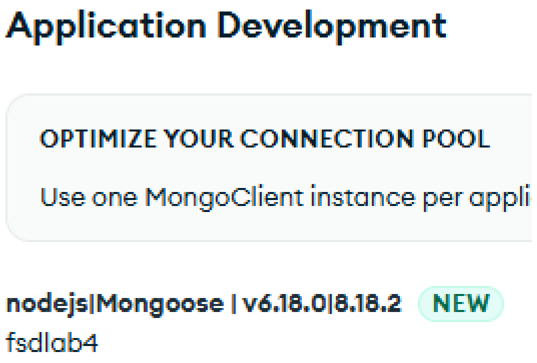


Figure 4.1 : Mongo Cluster

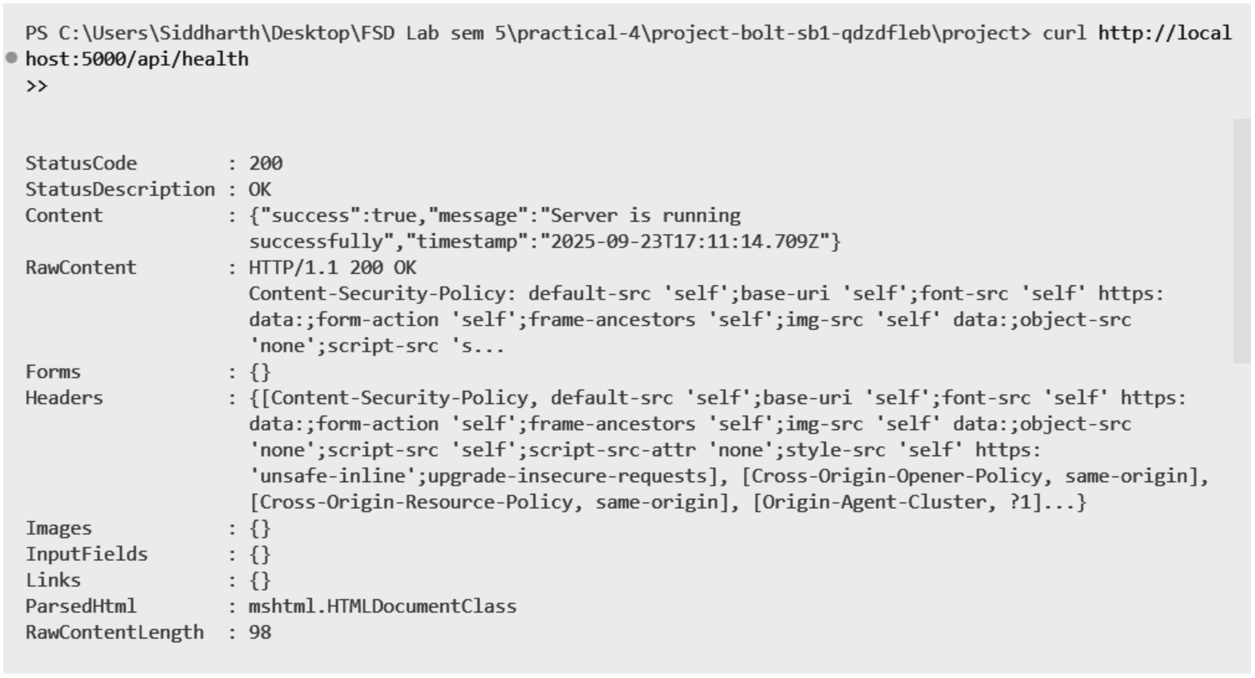

Application Development

OPTIMIZE YOUR CONNECTION POOL

Use one MongoClient instance per appli

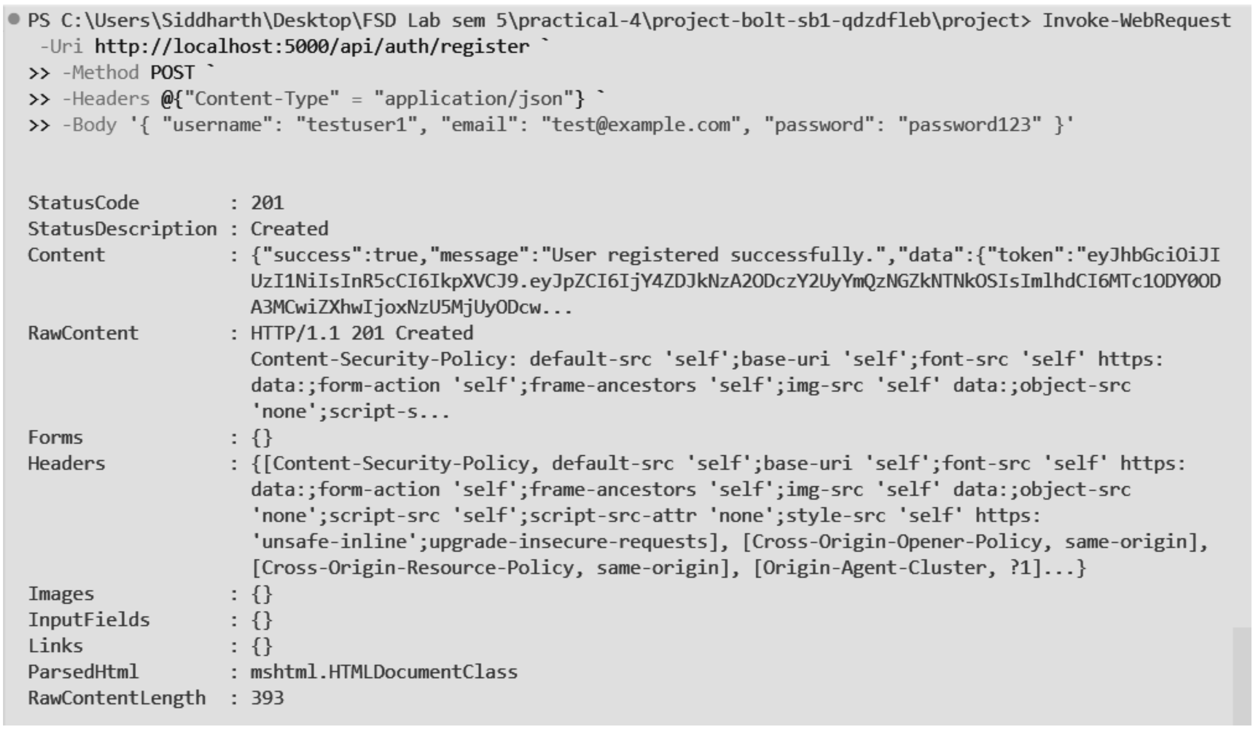
nodejs|Mongoose | v6.18.0|8.18.2 NEW
fsdlab4

Figure 4.2 : Mongo connected using mongoose

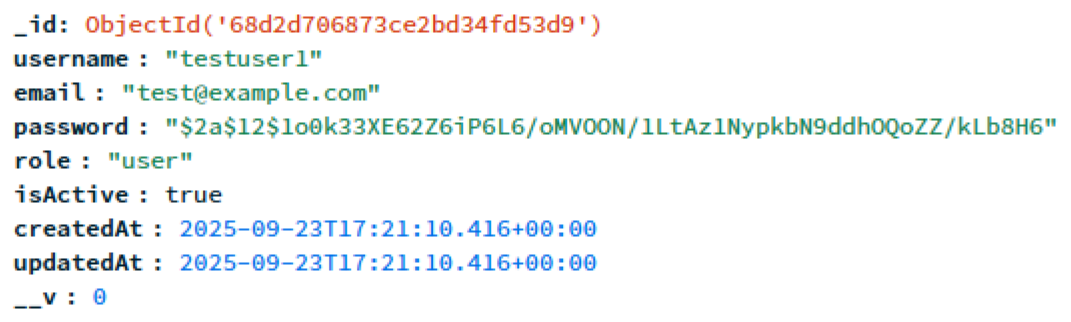


```
PS C:\Users\Siddharth\Desktop\FSD Lab sem 5\practical-4\project-bolt-sb1-qdzdfleb\project> curl http://local
host:5000/api/health
>>


StatusCode        : 200
StatusDescription : OK
Content           : {"success":true,"message":"Server is running
                    successfully","timestamp":"2025-09-23T17:11:14.709Z"}
RawContent        : HTTP/1.1 200 OK
                    Content-Security-Policy: default-src 'self';base-uri 'self';font-src 'self' https:
                    data:;form-action 'self';frame-ancestors 'self';img-src 'self' data:;object-src
                    'none';script-src 's...
Forms             : {}
Headers           : {[Content-Security-Policy, default-src 'self';base-uri 'self';font-src 'self' https:
                    data:;form-action 'self';frame-ancestors 'self';img-src 'self' data:;object-src
                    'none';script-src 'self';script-src-attr 'none';style-src 'self' https:
                    'unsafe-inline';upgrade-insecure-requests], [Cross-Origin-Opener-Policy, same-origin],
                    [Cross-Origin-Resource-Policy, same-origin], [Origin-Agent-Cluster, ?1]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 98
```

Figure 4.3 : Using the FAST API for using the curl commands to check the connection

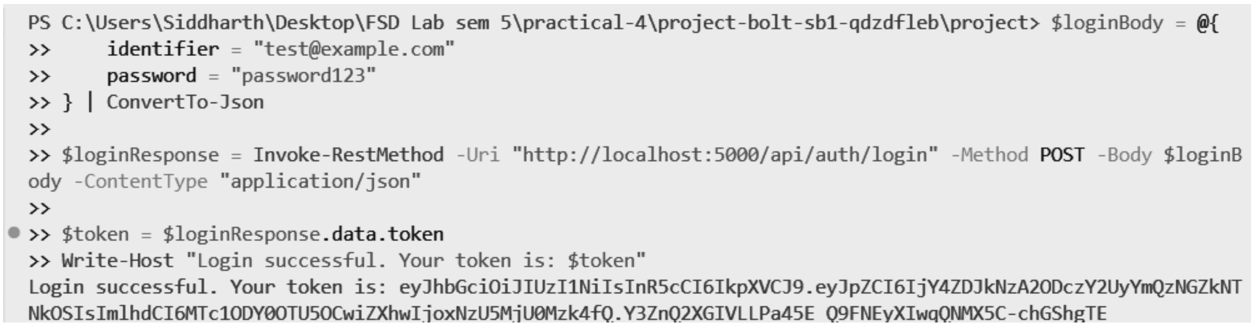


```
PS C:\Users\Siddharth\Desktop\FSD Lab sem 5\practical-4\project-bolt-sb1-qdzdfleb\project> Invoke-WebRequest
 -Uri http://localhost:5000/api/auth/register `
>> -Method POST `
>> -Headers @{"Content-Type" = "application/json"} `
>> -Body '{ "username": "testuser1", "email": "test@example.com", "password": "password123" }'


StatusCode        : 201
StatusDescription : Created
Content           : {"success":true,"message":"User registered successfully.","data":{"token":"eyJhbGciOiJI
                    UzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY4ZDJkNzA2ODczY2UyYmQzNGZkNTNkOSIsImlhdCI6MTc1ODY0OD
                    A3MCwiZXhwIjoxNzU5MjUyODcw...
RawContent        : HTTP/1.1 201 Created
                    Content-Security-Policy: default-src 'self';base-uri 'self';font-src 'self' https:
                    data:;form-action 'self';frame-ancestors 'self';img-src 'self' data:;object-src
                    'none';script-s...
Forms             : {}
Headers           : {[Content-Security-Policy, default-src 'self';base-uri 'self';font-src 'self' https:
                    data:;form-action 'self';frame-ancestors 'self';img-src 'self' data:;object-src
                    'none';script-src 'self';script-src-attr 'none';style-src 'self' https:
                    'unsafe-inline';upgrade-insecure-requests], [Cross-Origin-Opener-Policy, same-origin],
                    [Cross-Origin-Resource-Policy, same-origin], [Origin-Agent-Cluster, ?1]...}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 393
```



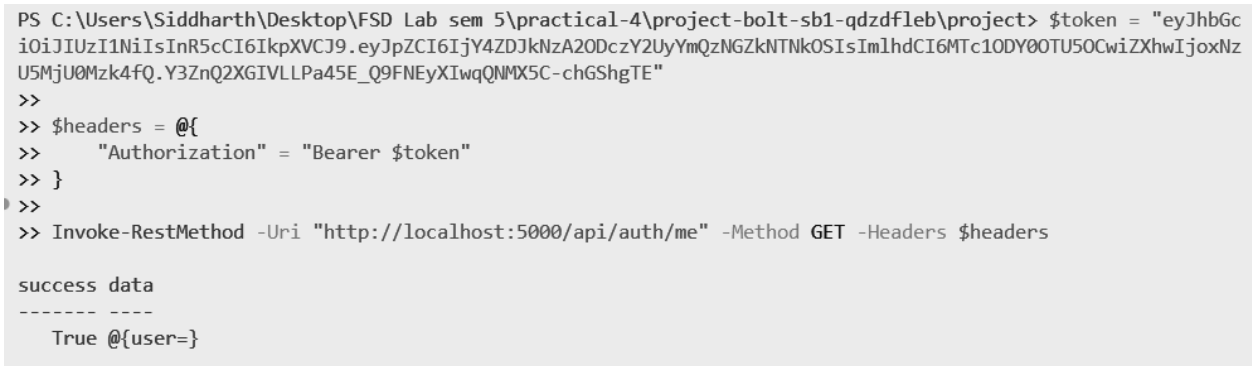
```
_id: ObjectId('68d2d706873ce2bd34fd53d9')
username : "testuser1"
email : "test@example.com"
password : "$2a$12$1o0k33XE62Z6iP6L6/oMVOON/1LtAz1NypkbN9ddhOQoZZ/kLb8H6"
role : "user"
isActive : true
createdAt : 2025-09-23T17:21:10.416+00:00
updatedAt : 2025-09-23T17:21:10.416+00:00
__v : 0
```

Figure 4.4 , 4.5: Creating a new user in mongo db using the rest api



```
PS C:\Users\Siddharth\Desktop\FSD Lab sem 5\practical-4\project-bolt-sb1-qdzdfleb\project> $loginBody = @{
>>     identifier = "test@example.com"
>>     password = "password123"
>> } | ConvertTo-Json
>>
>> $loginResponse = Invoke-RestMethod -Uri "http://localhost:5000/api/auth/login" -Method POST -Body $loginB
ody -ContentType "application/json"
>>
>> $token = $loginResponse.data.token
>> Write-Host "Login successful. Your token is: $token"
Login successful. Your token is: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY4ZDJkNzA2ODczY2UyYmQzNGZkNT
NkOSIsImlhdCI6MTc1ODY0OTU5OCwiZXhwIjoxNzU5MjU0Mzk4fQ.Y3ZnQ2XGIVLLPa45E_Q9FNEyXIwqQNMX5C-chGShgTE
```

Figure 4.6 : Login with the new user using the route and Rest API



```
PS C:\Users\Siddharth\Desktop\FSD Lab sem 5\practical-4\project-bolt-sb1-qdzdfleb\project> $token = "eyJhbGc
iOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY4ZDJkNzA2ODczY2UyYmQzNGZkNTNkOSIsImlhdCI6MTc1ODY0OTU5OCwiZXhwIjoxNz
U5MjU0Mzk4fQ.Y3ZnQ2XGIVLLPa45E_Q9FNEyXIwqQNMX5C-chGShgTE"
>>
>> $headers = @{
>>     "Authorization" = "Bearer $token"
>> }
>>
>> Invoke-RestMethod -Uri "http://localhost:5000/api/auth/me" -Method GET -Headers $headers

success data
------- ----
   True @{user=}
```

Figure 4.7 : Checking the protected route using JWT Token - 30% Extra

## Conclusion :

This report provides a concise summary of the project's architecture. It demonstrates a secure and scalable backend using Node.js, Express, and MongoDB. The modular design, robust authentication, and security middleware make it an ideal foundation for modern web applications.