Experiment No 6

Aim : Implement **authentication and user roles** with JWT

Theory:

1. Introduction

This report details the implementation of authentication and user roles using JSON Web Tokens (JWT) in a RESTful API built with Node.js, Express.js, MongoDB, and Mongoose. Building on foundational API design and security features from prior experiments, this lab focuses on enhancing user management by introducing role-based access control (RBAC). RBAC ensures that users can only access resources appropriate to their assigned roles (e.g., 'user', 'admin'), promoting security, scalability, and granular permissions. The objective is to create a robust system where authentication verifies user identity via JWT, while authorization enforces role-specific restrictions on endpoints, preventing unauthorized actions like administrative operations by standard users.

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. Built on the Chrome V8 engine, it excels in event-driven, non-blocking I/O operations, making it suitable for real-time and data-intensive applications. In this lab, Node.js handles the server-side logic for JWT generation, validation, and role enforcement.

2.2 Express.js

Express.js is a minimal, flexible Node.js web framework that simplifies routing, middleware management, and HTTP request/response handling. It enables the definition of protected routes and integration of custom middleware for authentication and role checks. This lab uses Express.js to structure endpoints that differentiate access based on user roles.

2.3 MongoDB & Mongoose

MongoDB is a document-oriented NoSQL database that stores data in flexible, JSON-like BSON documents, offering high scalability for unstructured data. Mongoose, an ODM library, provides schema validation and querying capabilities. Here, the User schema is extended to include a 'role' field (e.g., enum: ['user', 'admin']), allowing role data to be persisted and queried during authorization.

3. Middleware

Middleware functions in Express.js process requests sequentially, accessing req, res, and next(). They are pivotal for authentication and role enforcement.

3.1 Authentication Middleware (auth.js)

This custom middleware verifies JWTs in the Authorization header (Bearer <token>):

Extracts and validates the token using jwt.verify().
If valid, decodes the user ID, fetches the user from MongoDB via Mongoose, and attaches req.user (including role).
Returns 401 Unauthorized if invalid, expired, or missing, halting the request cycle.

3.2 Role-Based Authorization Middleware (adminAuth.js or roleAuth.js)

An extension of authentication middleware for RBAC:

After auth middleware confirms identity, it checks if req.user.role matches the required role (e.g., 'admin').
If mismatched, returns 403 Forbidden.
Supports dynamic roles (e.g., via a function parameter: roleAuth('admin')) for flexible endpoint protection.
Integrates with existing security middleware like Helmet, CORS, and rate-limiting to form a layered defense.

3.3 Body Parsers and Security Middleware

express.json() and express.urlencoded() parse JSON/URL-encoded bodies for registration/login payloads.
Helmet sets secure HTTP headers against XSS and clickjacking.
CORS enables cross-origin requests from trusted frontends.
express-rate-limit prevents brute-force attacks on auth endpoints.

4. User Authentication and Authorization

4.1 JSON Web Tokens (JWT)

JWTs provide stateless authentication as compact, signed tokens divided into Header (algorithm), Payload (claims like user ID and expiration), and Signature (for integrity). Upon login:

Server generates JWT with jwt.sign() using a secret key and user ID.
Client stores and sends it in Authorization headers for protected requests.
Benefits: No server-side session storage, scalable for microservices; drawbacks: Token revocation requires blacklisting or short expiration.

4.2 Password Hashing (bcryptjs)

Passwords are hashed with bcryptjs during registration (bcrypt.hash()). On login, bcrypt.compare() verifies against the stored hash, ensuring one-way security without exposing plaintext.

4.3 Role-Based Access Control (RBAC)

RBAC assigns roles to users in the MongoDB schema (e.g., default 'user', assignable 'admin'). Authorization flow:

Auth middleware validates JWT and attaches user/role to req.
Role middleware enforces permissions (e.g., only 'admin' accesses /api/admin/users). This prevents privilege escalation, enhances data privacy, and simplifies auditing. Roles can be hierarchical (e.g., 'superadmin' inherits 'admin') or granular (e.g., permissions array).

5. Routes and Controllers

5.1 Route Separation

Routes (e.g., routes/auth.js, routes/admin.js) define endpoints, while controllers (e.g., controllers/authController.js) handle logic. Separation promotes maintainability; protected routes chain middleware: app.get('/protected', auth, roleAuth('admin'), controller).

5.2 API Endpoints

POST /api/auth/register: Public; creates user with role 'user' (hashed password).
POST /api/auth/login: Public; returns JWT on success.
GET /api/auth/me: Protected (auth middleware); returns user profile including role.
GET /api/protected: Protected (auth); general access for authenticated users.
GET /api/admin/users: Role-protected (auth + adminAuth); lists all users (admin-only).
PUT /api/admin/users/:id/role: Admin-only; updates user roles.

6. Advanced Concepts and Security (30% Extra)

6.1 JWT Refresh Tokens

For long-lived sessions, issue short-lived access JWTs alongside refresh tokens. On expiration, clients use refresh endpoints to obtain new access tokens without re-login, balancing security and usability. Store refresh tokens securely (e.g., HTTP-only cookies) and revoke on logout.

6.2 Token Blacklisting and Revocation

JWTs are stateless but revocable via a Redis/MongoDB blacklist. On logout or role change, add token ID to blacklist; middleware checks before validation. This mitigates compromised tokens.

6.3 Secure Role Management

Audit logs: Track role changes in a separate MongoDB collection.
Input validation: Sanitize role assignments to prevent injection.
Scalability: Use middleware chaining for multi-role checks (e.g., ['admin', 'moderator']).
6.4 Integration with File Uploads and Static Serving

Building on prior labs, role-protected uploads (e.g., admin-only via multer) ensure sensitive files are role-gated, with express.static serving from /uploads only after authorization.

Code :

Create JWT Token :

controllers > JS authController.js > ...

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

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

Register a user :

```
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
Windsurf: Refactor | Explain | ×
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.'
        });
    }
};
```

Outputs :

```
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 6.1 , 6.2: Creating a new user in mongo db using the rest api and jwt

```
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 6.3 : 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 6.4 : Checking the protected route using JWT Token - 30% Extra

Conclusion :

This lab establishes a secure, role-aware authentication system using JWT, transforming a basic API into an enterprise-ready backend. By leveraging Mongoose for role persistence, custom middleware for enforcement, and Express for routing, the implementation ensures compliance with least-privilege principles. This foundation supports scalable applications with fine-grained access, reducing breach risks while maintaining performance.