Experiment No 5

Aim : Create secure, production-ready **RESTful APIs**

Theory :

1. Introduction

This report details the design and implementation of a secure, production-ready RESTful API. Building on a foundational backend architecture of Node.js, Express, MongoDB, and Mongoose, this project incorporates advanced features to handle dynamic data and enhance security. The primary focus of this version is to implement robust file upload functionality, ensuring the API is prepared for real-world use cases beyond basic user management.

2. Core 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. Key Concepts and Middleware

3.1 `multer` for File Uploads (30% Extra)

While `express.json()` and `express.urlencoded()` are sufficient for parsing text-based request bodies, they cannot handle multipart form data, which is necessary for file uploads. **`multer`** is a Node.js middleware specifically designed to process files uploaded via `multipart/form-data`. It adds a `req.file` or `req.files` object to the request, containing information about the uploaded file.

In this project, `multer` is configured with **disk storage**, which saves files to a local directory on the server. The configuration also includes robust **file validation**, ensuring that only allowed file types (e.g., JPEG, PNG) are uploaded and that their size is within a specified limit. This is a critical security measure to prevent a variety of attacks, including those involving malicious scripts or excessive file sizes.

3.2 Static File Serving

After a file is uploaded and saved to the server's file system, it needs to be made accessible to the public. Express provides the `express.static()` middleware for this purpose. By configuring a static route (e.g., `app.use('/uploads', express.static('uploads'))`), the server creates a public endpoint that serves files from a specified directory. The frontend can then request the uploaded image from a URL like `http://localhost:5000/uploads/filename.jpg`, allowing it to display the file without exposing the backend's internal directory structure.

4. Advanced Security Measures

4.1 Cross-Origin Resource Sharing (CORS)

CORS is a browser-based security mechanism that allows a web application on one domain to access resources from another. By default, browsers enforce the Same-Origin Policy, which prevents this. The **`cors`** middleware is used to configure which origins are allowed to make requests to the API, ensuring that only trusted frontend applications can communicate with your backend.

4.2 Rate Limiting

Rate limiting is a crucial defense against brute-force attacks and denial-of-service (DoS) attacks. The **`express-rate-limit`** middleware limits the number of requests an IP address can make to an endpoint within a set time window. This prevents a single client from overwhelming the server with an excessive number of login attempts or requests.

4.3 Protected Routes and Role-Based Access Control (30% Extra)

Protected routes are API endpoints that require a user to be authenticated and authorized. This is achieved using a custom authentication middleware. This middleware checks for a valid **JSON Web Token (JWT)** in the request header. If the token is valid, the middleware allows the request to proceed. This project

takes this a step further by implementing **role-based access control**, as seen in the `adminAuth` middleware. This ensures that certain sensitive endpoints, such as the user management dashboard, are only accessible to users with the 'admin' role, adding a critical layer of security and data privacy.

Code :

routes/uploads.js - upload images using multer (30% extra)

routes > JS uploads.js > storage

```
const express = require('express');
const multer = require('multer');
const path = require('path');
const { auth } = require('../middleware/auth');
const User = require('../models/User');

const router = express.Router();

// Setup Multer for file storage
const storage = multer.diskStorage({
    Windsurf: Refactor | Explain | Generate JSDoc | ×
    destination: (req, file, cb) => {
        cb(null, 'uploads/'); // Store uploaded files in the 'uploads' directory
    },
    Windsurf: Refactor | Explain | Generate JSDoc | ×
    filename: (req, file, cb) => {
        // Create a unique filename with the original extension
        cb(null, `${req.user._id}-${Date.now()}${path.extname(file.originalname)}`);
    }
});

const upload = multer({
    storage: storage,
    limits: { fileSize: 1024 * 1024 * 5 }, // 5MB file size limit
    Windsurf: Refactor | Explain | ×
    fileFilter: (req, file, cb) => {
        const filetypes = /jpeg|jpg|png|gif/;
        const mimetype = filetypes.test(file.mimetype);
        const extname = filetypes.test(path.extname(file.originalname).toLowerCase());

        if (mimetype && extname) {
            return cb(null, true);
        }
        cb(new Error('Only images (JPEG, JPG, PNG, GIF) are allowed.'));
    }
});
```

routes > JS uploads.js > storage

```
// @desc     Upload a profile picture for the authenticated user
// @access   Private
router.post('/profile-picture', auth, upload.single('profilePicture'), async (req, res) => {
    try {
        if (!req.file) {
            return res.status(400).json({ success: false, message: 'No file uploaded.' });
        }

        const user = await User.findById(req.user._id);
        if (!user) {
            return res.status(404).json({ success: false, message: 'User not found.' });
        }

        // Save the file path to the user document
        user.profilePicture = `/uploads/${req.file.filename}`;
        await user.save();

        res.status(200).json({
            success: true,
            message: 'Profile picture uploaded successfully.',
            filePath: user.profilePicture
        });

    } catch (error) {
        console.error('File upload error:', error);
        res.status(500).json({ success: false, message: 'Server error during file upload.' });
    }
});

module.exports = router;

```

Render all users in frontend for the admins

```
const fetchAndRenderAllUsers = async () => {
    if (!token) {
        renderForm(false);
        return;
    }
    try {
        const response = await fetch(`${API_BASE_URL}/admin/users`, {
            method: 'GET',
            headers: {
                'Authorization': `Bearer ${token}`
            }
        });

        const data = await response.json();

        if (!response.ok) {
            showMessage(data.message || 'Access denied.', false);
            return;
        }

        renderUsersTable(data.data.users);

    } catch (error) {
        console.error(error);
        showMessage('Network error. Could not fetch user list.', false);
    }
};
```

Fetch and Render Dashboard -

```
const fetchAndRenderDashboard = async () => {
        renderForm(false);
        return;
    }
    try {
        const response = await fetch(`${API_BASE_URL}/auth/me`, {
            method: 'GET',
            headers: {
                'Authorization': `Bearer ${token}`
            }
        });

        const data = await response.json();

        if (!response.ok) {
            showMessage(data.message || 'Authentication failed. Please log in again.', false);
            localStorage.removeItem('token');
            token = null;
            currentUser = null;
            renderForm(false);
            return;
        }

        renderDashboard(data.data.user);

    } catch (error) {
        console.error(error);
        showMessage('Network error. Check if the server is running.', false);
        localStorage.removeItem('token');
        token = null;
        currentUser = null;
        renderForm(false);
    }
};
```

Outputs(Frontend Implementation) :

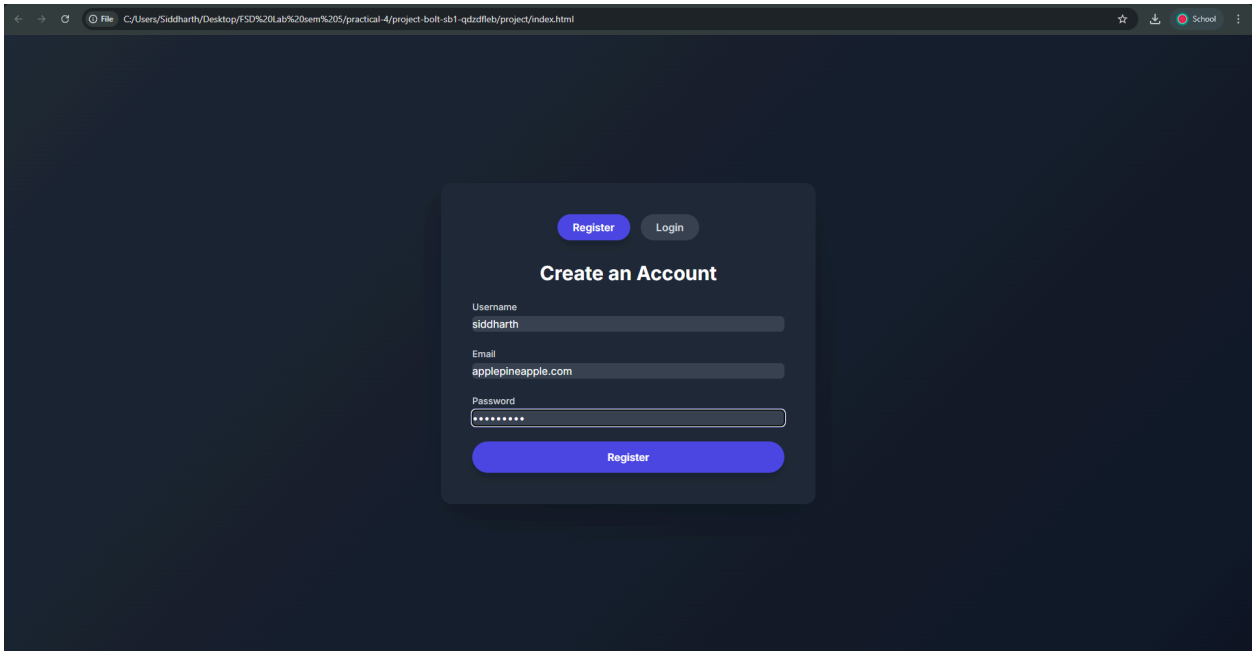

Figure 5.1 : Login and Register Page using the nodejs backend

```
_id: ObjectId('68d2e402f462f00ba1185a08')
username : "siddharth"
email : "applepineapple@gmail.com"
password : "$2a$12$KG/7i9ZFY/PBBguuclyIeeqc2lOeduPsXj0VA/gLFcEmeG8OWEHce"
role : "user"
isActive : true
createdAt : 2025-09-23T18:16:34.971+00:00
updatedAt : 2025-09-23T18:16:34.971+00:00
__v : 0
```


Figure 5.2: User Successfully created inside of mongodb

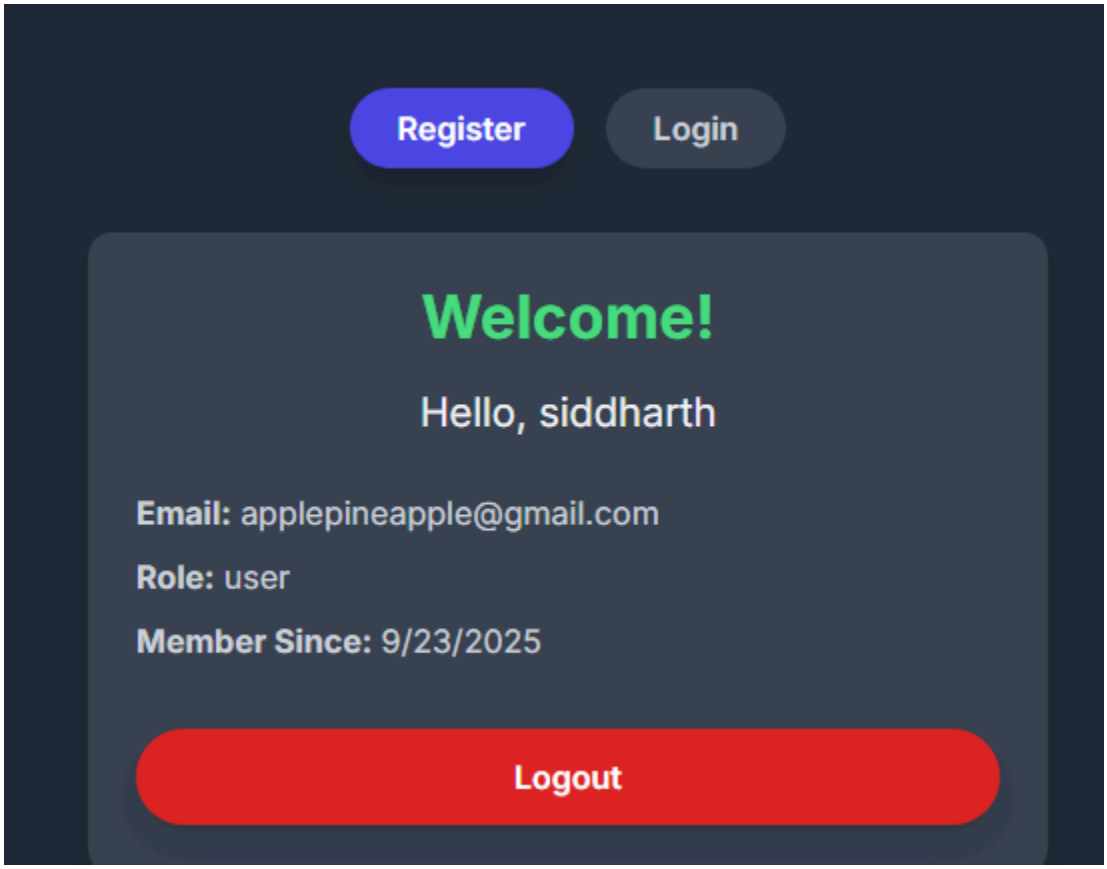

Figure 5.3: Successfully Logged in using JWT key

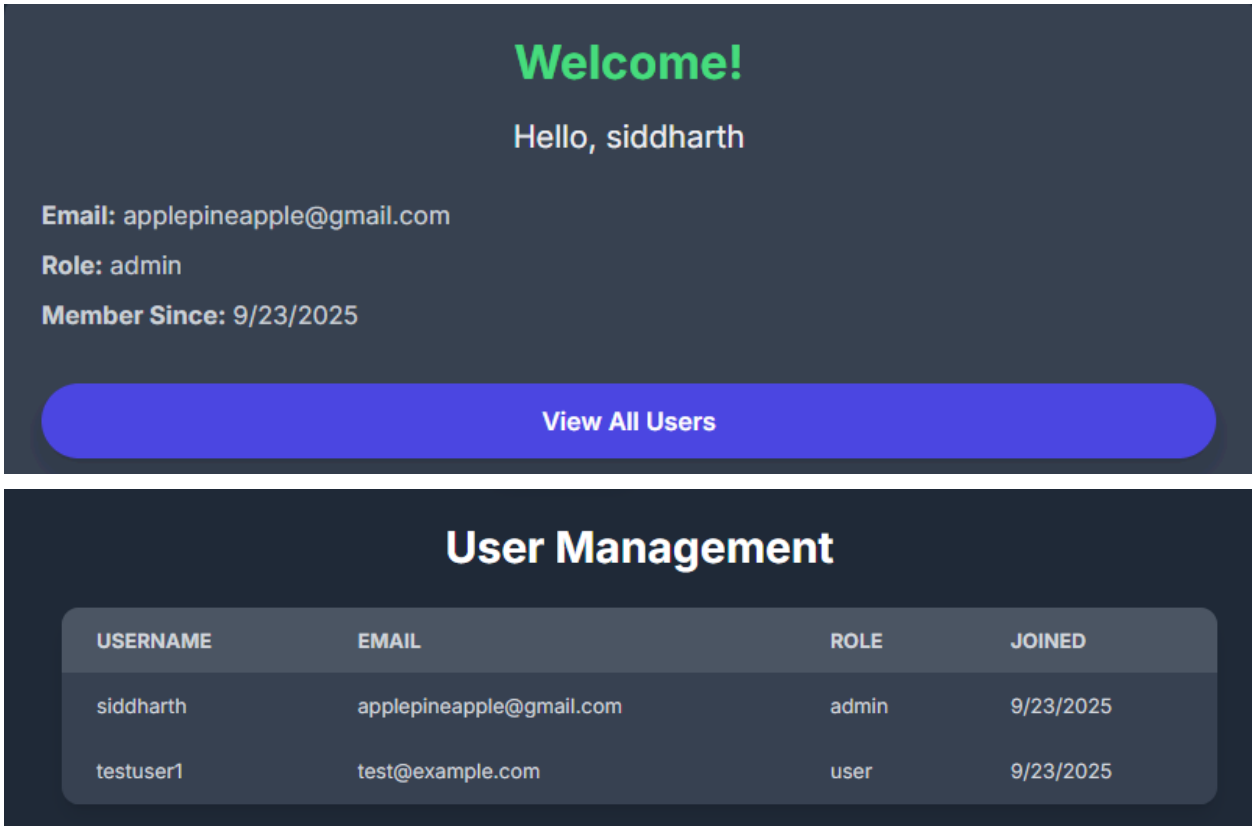

Figure 5.4,5.5: Check all Users for Admins only - Role Based Access **(30% extra)**

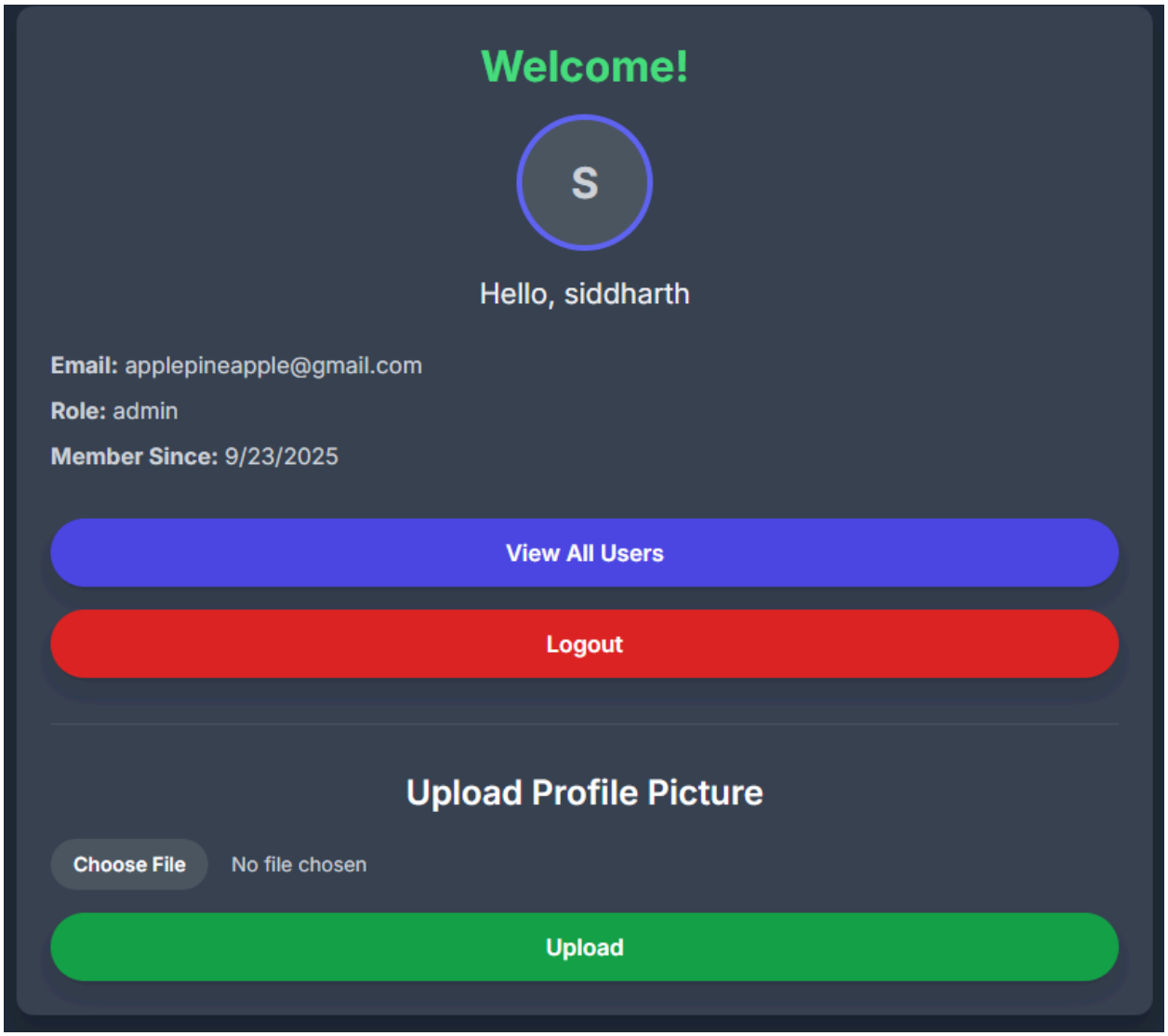

Figure 5.6 : User Dashboard , Upload files functionality **(30% extra)**

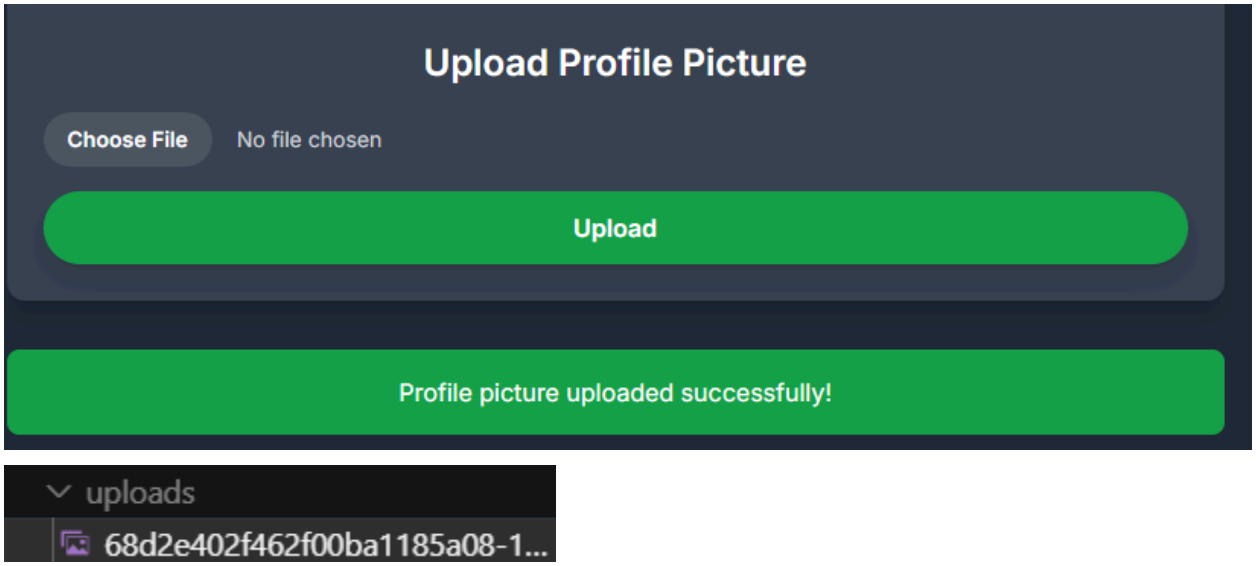

Figure 5.7 : Image Uploaded Successfully using Multer and then stored in the backend

Conclusion :

This project provides a comprehensive and secure foundation for a Node.js REST API. By combining Express.js for routing, MongoDB for data storage, Mongoose for data modeling, and JWT for authentication, the application is ready to be scaled and integrated with a frontend client. The modular design and use of middleware demonstrate key principles of modern web application development, including advanced security measures, dynamic data handling, and file management.