## Task 1: Building a Simple University Authentication System with JWT

**Instructions**:

- Create an Express.js application with routes for user **registration** and **login**.
- For registration, take **`username`**, **`email`**, and **`password`** (hash it before saving), role(e.g. admin, teacher, student).
- Store user data in the MongoDB **`users`** collection.
- For login, authenticate by checking the password hash and generate a JWT upon successful login.
- The JWT should include user ID and username in the payload, and be signed with a secret key.
- Return the JWT to the client upon successful login.

## Testing Steps

Test the authentication endpoints with **Postman** or **vscode plugin "ThunderClient**.

1. **Register a New User**:
   a. **Endpoint**: `POST /api/auth/register`
   b. **Body**: JSON format
   ```
   {
     "username": "john_doe",
     "email": "john@example.com",
     "password": "password123"
   }
   ```
   - **Expected Response**: A confirmation message.
   ```
   {
     "message": "User registered successfully"
   }
   ```
2. **Login User and Get JWT Token**:
   a. **Endpoint**: `POST /api/auth/login`
   b. **Body**: JSON format
   ```
   {
     "email": "john@example.com",
     "password": "password123"
   }
   ```
   - **Expected Response**: A success message with a JWT token.
   ```
   {
     "message": "Login successful",
     "token": "<JWT_TOKEN>"
   }
   ```

**Note**: Copy the `<JWT_TOKEN>`; it will be used in future steps to access protected routes.

## Task 2: Securing Routes with JWT

Implement middleware to protect certain routes.

**Instructions**:

- o Add middleware in the Express.js app to verify JWTs.
- o Implement protected routes (e.g., `/profile`) that can only be accessed if a valid JWT is provided in the request headers.
- o If the JWT is invalid or missing, return a 401 Unauthorized error.
- o On successful verification, retrieve user information from MongoDB and display it.

### Testing Steps

Test the protected route using **Postman** or **VSCode plugin "ThunderClient"**.

1. **Get User Profile (Protected Route)**:
    a. **Endpoint**: `GET /api/users/profile`
    b. **Headers**: Set the `Authorization` header with the JWT token obtained from Task 1:
    `Authorization: Bearer <JWT_TOKEN>`
    - **Expected Response**: If the token is valid, it returns the user profile without the password.
    ```
    {
      "_id": "<USER_ID>",
      "username": "john_doe",
      "email": "john@example.com",
      "role": "user",
      "createdAt": "<DATE>"
    }
    ```
    - **Error Response**: If the token is missing or invalid, it returns an error.
    ```
    {
      "message": "Access denied. No token provided."
    }
    ```

## Task 3: Role-Based Access Control

**Instructions**:

- Use this role field to restrict access to certain routes, such as an admin-only route.
- In the JWT payload, include the user role and implement middleware to check role permissions for each route.

## Testing Steps

Test the authentication endpoints with **Postman** or **vscode plugin “ThunderClient**.

1. **Register a New Admin User**:
   a. **Endpoint**: `POST /api/auth/register`
   b. **Body**: JSON format
   ```
   {
     "username": "admin_user",
     "email": "admin@example.com",
     "password": "password123",
     "role": "admin"  // Specify the role as admin
   }
   ```
   - **Expected Response**: A confirmation message.
   ```
   {
     "message": "User registered successfully"
   }
   ```
2. **Login Admin User and Get JWT Token**:
   a. **Endpoint**: `POST /api/auth/login`
   b. **Body**: JSON format
   ```
   {
     "email": "admin@example.com",
     "password": "password123"
   }
   ```
   - **Expected Response**: A success message with a JWT token.
   ```
   {
     "message": "Login successful",
     "token": "<JWT_TOKEN>"
   }
   ```
3. **Access Admin Dashboard**:
   a. **Endpoint**: `GET /api/users/admin`
   b. **Headers**: Set the `Authorization` header with the JWT token obtained from logging in as the admin:
   ```
   Authorization: Bearer <JWT_TOKEN>
   ```
   - **Expected Response**: If the user is an admin, it returns a welcome message.
   ```
   {
     "message": "Welcome to the admin dashboard!"
   }
   ```

- **Error Response**: If the user does not have the admin role, it returns:
```
{
  "message": "Access forbidden: Admins only"
}
```

**Task 4: Setting Up MongoDB, creating a Simple Collection, and performing MongoDB CRUD operations**

**Instructions**:

- Install MongoDB locally or set up a free MongoDB Atlas cluster.
- Create a new database and a collection named **`Customers`**.
- Insert sample documents with fields like **`username`, `email`, `password`** (hashed), customerType(e.g., regular, VIP, new)., and **`createdAt`**.
- Perform CRUD operations.

## Testing CRUD Operations

Test the authentication endpoints with **Postman** or **vscode plugin "ThunderClient**.

**Create a New customer**:

a. **Endpoint**: `POST /api/customers/create`
b. **Body**: JSON format
```
{
  "username": "john_doe",
  "email": "john@example.com",
  "password": "password123",
  "customerType": "regular"
}
```

- **Expected Response**: A confirmation that the customer record was created.

**Read All Customers**:

- **Endpoint**: GET /api/customers/
- **Expected Response**: A JSON array with all customers.

**Read a Single Customer**:

- **Endpoint**: GET /api/customers/:id
- Replace :id with the actual customer ID from MongoDB.
- **Expected Response**: A JSON object of the specified customer.

**Update a Customer**:

- **Endpoint**: PUT /api/customers/:id
- Replace :id with the actual customer ID from MongoDB.

- **Body**: JSON format (you can update any or all fields)
  ```
  {
    "username": "john_updated",
    "email": "john_updated@example.com",
    "password": "newpassword123",
    "customerType": "regular"
  }
  ```
- **Expected Response**: Confirmation and the updated user data.

5. **Delete a Customer**:
   a. **Endpoint**: `DELETE /api/customers/:id`
   b. Replace `:id` with the actual customer ID from MongoDB.
   c. **Expected Response**: Confirmation that the customer was deleted.

**Task 5: Implement CRUD Operations with MongoDB Using a Relational Schema**

### *Task Description*

1. **Schema Design**: Design a schema where Employees are related to Departments. Each employee document should reference a department, and each department can have multiple employees.
2. **CRUD Operations**: Implement API endpoints to perform CRUD operations on both Employees and Departments.
3. **Testing**: Execute provided test cases to verify the CRUD operations work as expected.

### Relational Schema

**Departments Collection**:

- id: Unique identifier (ObjectId)
- name: Name of the department (string)
- location: Location of the department (string)

**Employees Collection**:

- id: Unique identifier (ObjectId)
- name: Name of the employee (string)
- email: Email of the employee (string)
- position: Position of the employee (string)
- departmentId: Reference to the Department document (ObjectId)

## Testing Steps

1. **Create a Department**:
    a. **POST /api/departments** with a JSON body:
    ```
    {
      "name": "Engineering",
      "location": "Block A"
    }
    ```

2. **Create an Employee Linked to the Department**:
    a. **POST /api/employees** with a JSON body:
    ```
    {
      "name": "Alice Johnson",
      "email": "alice.johnson@example.com",
      "position": "Software Engineer",
      "departmentId": "<DEPARTMENT_ID>"
    }
    ```

3. **Retrieve All Employees with Department Info**:
    **a. GET /api/employees**
4. **Update an Employee's Position**:
    a. **PUT /api/employees/<EMPLOYEE_ID>** with a JSON body:
    ```
    {
      "position": "Senior Software Engineer"
    }
    ```

5. **Delete a Department**:
    **a. DELETE /api/departments/<DEPARTMENT_ID>**

## SUBMISSION GUIDELINES:

Create a report that includes snapshots of all tested endpoints with their results. Then, create a zip file containing both the report and the code folder.