1) Employee Management in a Corporate Environment

REST API | Validation | Authentication | Mongo Db

Imagine you are part of a development team working for a mid-sized company that has recently shifted from a monolithic architecture to a microservices-based approach. Your task is to design and implement a scalable REST API that manages employee data within an organization.

The company currently has 500+ employees, with departments that include HR, Finance, IT, Marketing, and Sales. The existing employee management system is outdated and difficult to maintain. Therefore, you are tasked with creating a modern, scalable API that allows HR to manage employee details, including personal information, department assignments, and roles. This API must also be able to handle large numbers of employee requests efficiently.

Your team needs to ensure that the API meets organizational requirements and scalability expectations, while also addressing security, performance, and maintainability.

You are required to design and implement the following features for the Employee Management REST API:

1. **Create API Endpoints**:
    - A GET endpoint to fetch a list of all employees, supporting pagination.
    - A POST endpoint to add a new employee.
    - A PUT endpoint to update employee details.
    - A DELETE endpoint to remove an employee.
2. **Implement Authentication**:
    - Use JWT tokens for user authentication to secure access to certain routes (e.g., only HR users should be able to add or update employee data).
3. **Database Integration**:
    - Choose either MongoDB or SQL database to store employee details (name, position, department, etc.).
    - Implement necessary database operations for the above CRUD operations.
4. **Handling Scalability**:
    - Implement pagination for the GET endpoint to handle large sets of employee records.
    - Use in-memory caching (using Redis or similar) to reduce the load on the database for frequently accessed data.
5. **Error Handling & Logging**:
    - Implement proper error handling to return meaningful error messages.
    - Integrate logging to keep track of API activity and errors.
6. **Testing**:
    - Write unit tests for each endpoint using a Node.js testing framework (Jest or Mocha).

- Ensure that each route is tested for different scenarios (valid input, invalid input, unauthorized access).

Steps to Solve:

1. **Set Up the Project Environment:**
   - Initialize a new Node.js project using npm init.
   - Install required dependencies: express, mongoose (for MongoDB), jsonwebtoken, bcrypt, cors, morgan, dotenv, redis, and testing libraries (jest or mocha).
2. **Define the Database Schema (MongoDB or SQL):**
   - Create a Mongoose schema or Sequelize model for the employee, which includes fields such as name, role, department, and employeeID.
   - Set up the connection to your database in the app.js file.
3. **Set Up Express Server and Routes:**
   - Create an Express server.
   - Define routes for CRUD operations:
     - GET /employees - to fetch employees with pagination.
     - POST /employees - to add new employees.
     - PUT /employees/:id - to update an employee's details.
     - DELETE /employees/:id - to remove an employee.
4. **Implement Authentication Middleware:**
   - Create middleware to handle JWT authentication for secure routes.
   - Ensure that only authorized users (HR) can create or update employee records.
5. **Implement Error Handling and Logging:**
   - Add custom error handling middleware to return appropriate HTTP status codes and error messages.
   - Set up logging for API requests using morgan to capture request data for debugging and performance monitoring.
6. **Pagination and Caching:**
   - Add query parameters to the GET /employees route to support pagination (e.g., page and limit).
   - Implement caching for frequently requested data using Redis or in-memory caching techniques.
7. **Write Unit Tests:**
   - Write unit tests for each of your endpoints using Jest or Mocha.
   - Ensure tests check for success responses as well as edge cases (e.g., invalid input, unauthenticated requests, etc.).
8. **Run and Test the API:**

- Start the server and test each endpoint using Postman or a similar API testing tool.
- Use unit tests to validate that the application functions as expected.

9. **Push to GitHub:**
   - Commit the code to GitHub, ensuring that your project structure is clean and follows best practices.
   - Ensure that the .env file is included in .gitignore.

Sample Data

```
[
{ "_id": "1", "name": "John Doe", "position": "Software Engineer", "department": "IT", "employeeID": "EMP1001", "email": "johndoe@example.com" },
{ "_id": "2", "name": "Jane Smith", "position": "HR Manager", "department": "HR", "employeeID": "EMP1002", "email": "janesmith@example.com" },
{ "_id": "3", "name": "Peter Brown", "position": "Marketing Specialist", "department": "Marketing", "employeeID": "EMP1003", "email": "peterbrown@example.com" }
]
```

Sample Input & Output for Each API:

1. **GET /employees** – Fetch a list of all employees with pagination.
   Sample Input (Request URL):
   GET /employees?page=1&limit=2
   Sample Output (Response):

```
{
  "page": 1,
  "limit": 2,
  "total": 3,
  "employees": [
    {
      "_id": "1",
      "name": "John Doe",
      "position": "Software Engineer",
      "department": "IT",
      "employeeID": "EMP1001",
      "email": "johndoe@example.com"
    },
    {
      "_id": "2",
      "name": "Jane Smith",
      "position": "HR Manager",
      "department": "HR",
      "employeeID": "EMP1002",
      "email": "janesmith@example.com"
    }
  ]
}
```

2. **POST /employees** – Add a new employee.
   Sample Input (Request Body):

```
{
  "name": "Alice Green",
  "position": "Sales Executive",
  "department": "Sales",
  "employeeID": "EMP1004",
  "email": "alicegreen@example.com"
```

```
}
```

Sample Output (Response):

```
{
  "message": "Employee added successfully",
  "employee": {
    "_id": "4",
    "name": "Alice Green",
    "position": "Sales Executive",
    "department": "Sales",
    "employeeID": "EMP1004",
    "email": "alicegreen@example.com"
  }
}
```

3. **PUT /employees/:id** – Update an employee's details.

Sample Input (Request Body):

```
{
  "name": "John Doe",
  "position": "Senior Software Engineer",
  "department": "IT",
  "email": "john.doe@example.com"
}
```

Sample Output (Response):

```
{
  "message": "Employee details updated successfully",
  "employee": {
    "_id": "1",
    "name": "John Doe",
    "position": "Senior Software Engineer",
    "department": "IT",
    "employeeID": "EMP1001",
    "email": "john.doe@example.com"
  }
}
```

4. **DELETE /employees/:id** – Remove an employee.

Sample Input (Request URL):

```
DELETE /employees/1
```

Sample Output (Response):

```
{
  "message": "Employee deleted successfully"
}
```