Practical Examination 2: Developing a RESTful API with Node.js

Objective:

Create a RESTful task management backend service with a focus on building a web service that includes database integration, authorization, and security considerations. This examination focuses on content from lectures 7 to 12.

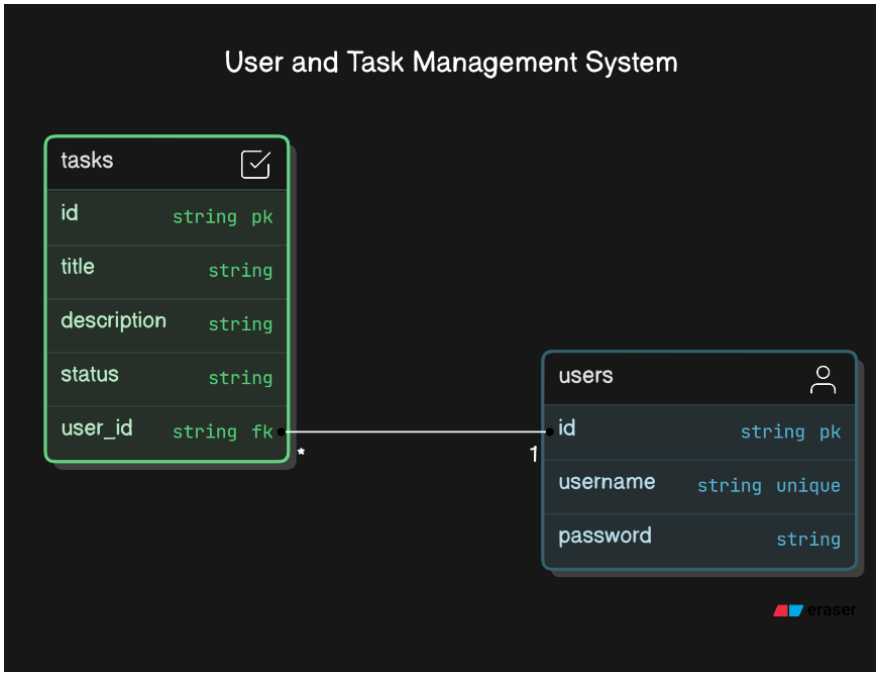
Requirements for Pass:

1. **Backend Application:** Develop a backend application using Node.js to manage user tasks. Users should be able to create, read, update, and delete tasks.

    a. Use Node.js to build a task management backend.
    b. Implement CRUD functionality for user tasks.

2. **Database Coupling:** Connect to a SQL database, develop a simple schema that includes tables for users and tasks. Each task should have fields like title, description, status, and user ID.

    a. Integrate a SQL database (MySQL).
    b. Define a schema with tables for `users` and `tasks`.



(Password is <u>only</u> required for JWT)

3. **RESTful API:** Develop a RESTful API to interact with the task data. Make sure to use appropriate HTTP methods and provide suitable endpoints.

    a. `GET /tasks` – Fetch all tasks
    b. `GET /tasks/:id` – Fetch a specific task
    c. `POST /tasks` – Create a new task
    d. `PUT /tasks/:id` – Update a task
    e. `DELETE /tasks/:id` – Delete a task

4. **Asynchronous Handling:** Use async/await to manage database queries and ensure non-blocking operations.

5. **Testing:** Manually test all endpoints to ensure proper functionality using Postman.

Requirements for Pass with Distinction:

Also see** Testing JWT endpoints **at the end of the document.

1. **User Authentication:**

    a. Implement JWT-based user authentication.
    b. Ensure users authenticate to access or modify their tasks

2. **Advanced Error Handling:** Include detailed error handling for different failure scenarios, such as invalid data, unauthorized access, and missing resources.

    a. Invalid data input
        i. Return a 400 Bad Request response when input data is missing, invalid, or fails validation.
        Include a message indicating the exact problem (e.g., 'Title is required').
    b. Unauthorized access
    c. Missing or unavailable resources
        i. Return a 404 Not Found response when the requested resource does not exist.
        Clearly state in the response that the resource could not be found (e.g., 'Task with ID X not found').

3. **RESTful Principles:** Ensure the API follows best practices for REST, including proper use of HTTP methods, status codes, and resource structures

    a. Proper HTTP methods (e.g., GET, POST, PUT, DELETE)
    b. Appropriate status codes (e.g., 200, 404, 401)
    c. Resource structures for easy navigation and access

4. **Testing:** Test functionality of JWT secured endpoints with Postman.

Endpoints:

1. Create a New Task

- **Method**: `POST`
- **URL**: `http://localhost:3000/tasks` (replace `localhost:3000` with your server address if different)
- **Headers**:
  - `Content-Type`: `application/json`
- **Body** (JSON):

```
{
  "title": "Sample Task",
  "description": "Description of the task",
  "status": "pending"
}
```

- **Expected Response**:
  - `201 Created` on success, with the created task object.
  - Example:

```
{
  "id": 1,
  "title": "Sample Task",
  "description": "Description of the task",
  "status": "pending",
  "user_id": 1
}
```

---

2. Retrieve All Tasks

- **Method**: `GET`
- **URL**: `http://localhost:3000/tasks`
- **Headers**: None
- **Expected Response**:
  - `200 OK`, with a JSON array of all tasks.
  - Example:

```
[
  {
    "id": 1,
    "title": "Sample Task",
    "description": "Description of the task",
    "status": "pending",
    "user_id": 1
  },
  {
    "id": 2,
    "title": "Another Task",
    "description": "Another description",
    //COMPLETE LIST OF TASKS
```

3. Retrieve a Single Task by ID

- **Method**: `GET`
- **URL**: `http://localhost:3000/tasks/:id` (replace `:id` with the specific task ID)
- **Headers**: None
- **Expected Response**:
  - `200 OK` if task exists, with the task details.
  - `404 Not Found` if no task with the specified ID.
  - Example for a successful response:

```
{
  "id": 1,
  "title": "Sample Task",
  "description": "Description of the task",
  "status": "pending",
  "user_id": 1
}
```

---

4. Update a Task

- **Method**: `PUT`
- **URL**: `http://localhost:3000/tasks/:id` (replace `:id` with the specific task ID)
- **Headers**:
  - `Content-Type`: `application/json`
- **Body** (JSON):

```
{
  "title": "Updated Task Title",
  "description": "Updated description",
  "status": "completed"
}
```

- **Expected Response**:
  - `200 OK` on success, with the updated task object.
  - `404 Not Found` if no task with the specified ID.
  - Example:

```
{
  "id": 1,
  "title": "Updated Task Title",
  "description": "Updated description",
  "status": "completed",
  "user_id": 1
}
```

5. Delete a Task

- **Method**: `DELETE`
- **URL**: `http://localhost:3000/tasks/:id` (replace `:id` with the specific task ID)
- **Headers**: None
- **Expected Response**:
  - `200 OK` with a message confirming deletion.
  - `404 Not Found` if no task with the specified ID.
  - Example response:

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

6. Testing JWT endpoints

- **Step 1: Set Up the Login Endpoint Test**
  - In Postman, set up a `POST` request to your `/login` endpoint (or whichever endpoint issues the JWT).
  - Send user credentials (e.g., username and password) in the request body.
  - Verify that a token is returned in the response if the credentials are correct.
- **Step 2: Testing Authenticated Routes**
  - Copy the JWT from the login response.
  - Set up another request to an authenticated endpoint (e.g., `GET /tasks`).
  - In the request's headers, add `Authorization: Bearer <your_token>` using the token you received.
  - Verify that you receive the expected data if the token is valid, and an authorization error if the token is missing or invalid.
- **Step 3: Test Token Expiration and Revocation**
  - If your JWT has an expiration, wait until it expires and repeat the request. Verify that the API responds with an appropriate error (e.g., 401 Unauthorized).
  - For testing token revocation (if applicable), send a revoked or random invalid token and verify the API rejects it.
    - Return a 401 Unauthorized response when a user tries to access protected resources without a valid JWT or with an expired/revoked token.
      Include a message like *'Authentication token is missing or invalid.'*