OnTrack Documentation

I-ASOS Semestral Assignment

December 14, 2024

Contents

1 Introduction

1.1 Project Overview

OnTrack is a task and note management system designed to help users organize their personal and collaborative workflows. The application supports the following functionalities:

- Creating and managing tasks or notes.
- Assigning tasks and notes to tags for better organization.
- Creating or joining groups using an invite code for collaboration.
- Sharing tasks and notes within groups, visible to all group members.

The system consists of a **frontend** and a **backend**, designed for seamless user experience and scalability. The backend is built with **NestJS**, while the frontend is designed to provide an intuitive and responsive interface.

2 Technologies Used

2.1 Backend

- **NestJS**: A progressive Node.js framework for building efficient and scalable server-side applications.
- **PostgreSQL**: A powerful, open-source object-relational database system.
- **TypeORM**: An Object-Relational Mapper (ORM) for TypeScript and JavaScript.
- **Docker**: Containerization for backend services to ensure portability and consistency.
- **Swagger**: API documentation and testing interface for developers.

2.2 Frontend

- **Vue.js**: A JavaScript framework for building user interfaces and single-page applications.
- **Vite**: A fast development build tool for Vue.js applications.

2.3 Development Tools

- **Prettier**: Code formatting tool to maintain consistent code style.
- **Jest**: A testing framework for writing unit and integration tests.
- **ESLint**: A tool for identifying and fixing linting issues in JavaScript/-TypeScript.

3 Database Design

The database for OnTrack is designed to handle tasks, notes, groups, and user management efficiently. Below is the database diagram:

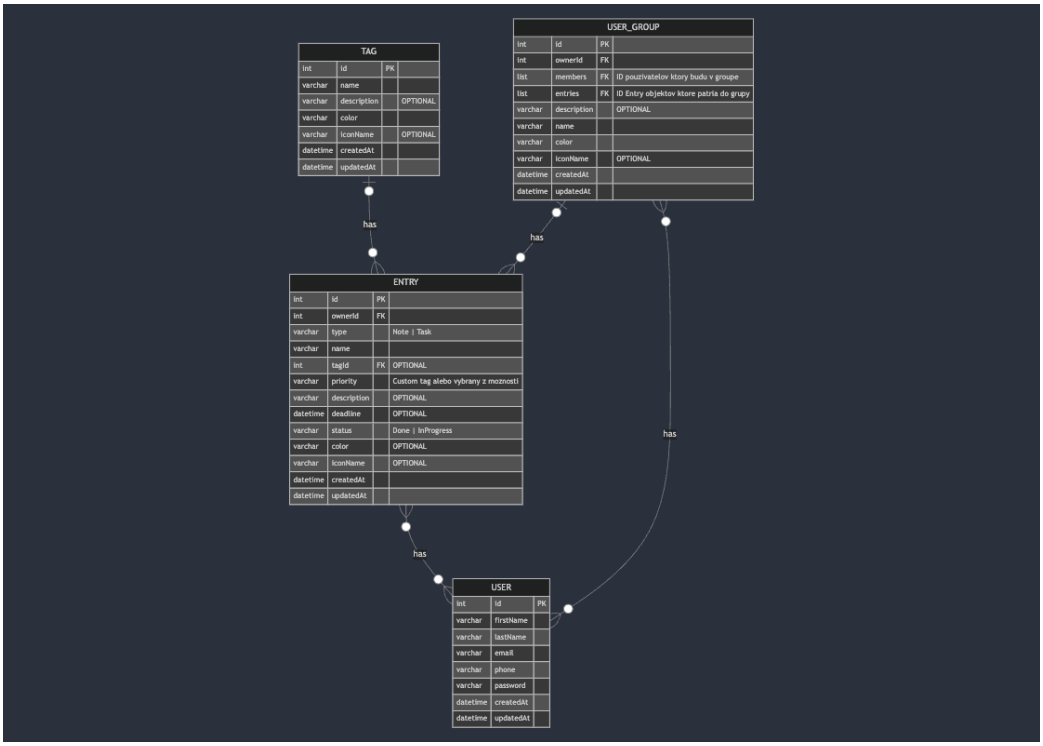

Figure 1: Database Diagram for OnTrack

3.1 Database Tables

- **Users**: Stores information about registered users.
- **Entries**: Represents tasks or notes created by users.
- **Tags**: Used for organizing tasks and notes.
- **User Groups**: Manages collaboration by allowing users to join groups.

4 Project Features

4.1 Core Functionalities

- **Task and Note Management**:
  - Create, update, and delete tasks or notes.
  - Assign tasks and notes to tags for better categorization.
- **Tag Management**:
  - Create, update, and delete tags.
  - Assign tags to tasks and notes for improved organization.
- **Group Management**:
  - Create groups using an invite code.
  - Join groups using an invite code.
  - Share tasks and notes within a group, making them visible to all group members.
- **Profile Management**:
  - View user profile details and statistics.
  - Edit user profile information, such as name and surname.

4.2 Backend API

The backend API is fully documented using Swagger. You can access the Swagger UI at:

- `http://localhost:3000/api`

4.3 Tests

In our backend we have implemented unit tests and end-to-end (E2E) tests, which validate the behavior of individual components or modules in our application.

- **Unit tests:** Validates that the core business logic behaves correctly in isolation.
- **E2E Tests:** Tests the entire application flow, ensuring all parts (controllers, services, middleware, repositories) integrate and work correctly.

4.4 Breakdown of out specific tests

- **users.service.spec.ts** Unit tests for user-related business logic, such as fetching users, validating user credentials, or creating new users.
- **users.controller.spec.ts** Tests whether the UsersController properly interacts with UsersService to handle HTTP requests like GET /users or POST /users.
- **tag.service.spec.ts** Unit tests for tag-related logic, such as managing tags or linking tags to entries.
- **tag.controller.spec.ts** Tests whether the TagController handles requests and interacts with TagService correctly.
- **entry.service.spec.ts** Unit tests for managing entries, such as fetching, creating, or associating entries with users or tags.
- **entry.controller.spec.ts** Tests whether the EntryController handles requests and interacts with EntryService properly.
- **usergroup.service.spec.ts** Unit tests for user group logic, such as creating groups or managing group membership.
- **usergroup.controller.spec.ts** Tests whether the UsergroupController handles requests and interacts with UsergroupService properly.
- **auth.service.spec.ts** Unit tests for authentication, such as validating user credentials, generating JWT tokens, or handling login flows.
- **app.controller.spec.ts** Tests whether the root AppController (if it exists) handles generic requests, such as serving health checks or general application endpoints.

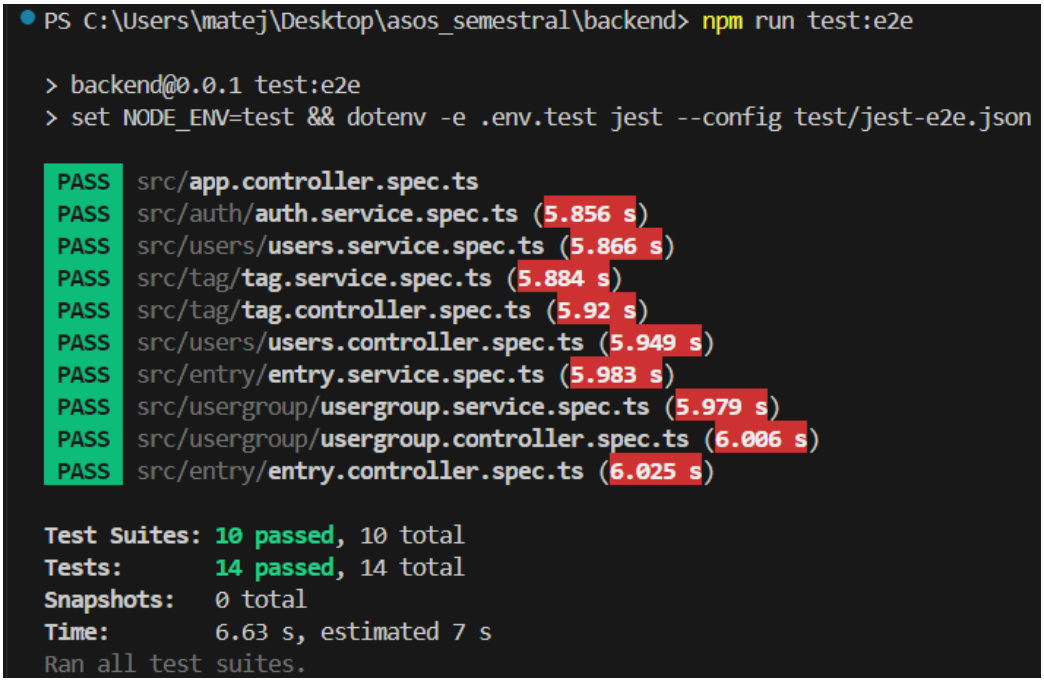

Figure 2: Passed tests

4.5 Load tests

We conducted a load test using Postman to evaluate the performance and stability of our system under concurrent traffic. The test simulated a load of 50 virtual users (VU), each sending parallel HTTP requests to the endpoint over a duration of 5 minutes. This approach allowed us to monitor response times, request throughput, and potential bottlenecks, ensuring the system can handle high traffic efficiently. The results provided insights into scalability and areas for optimization.

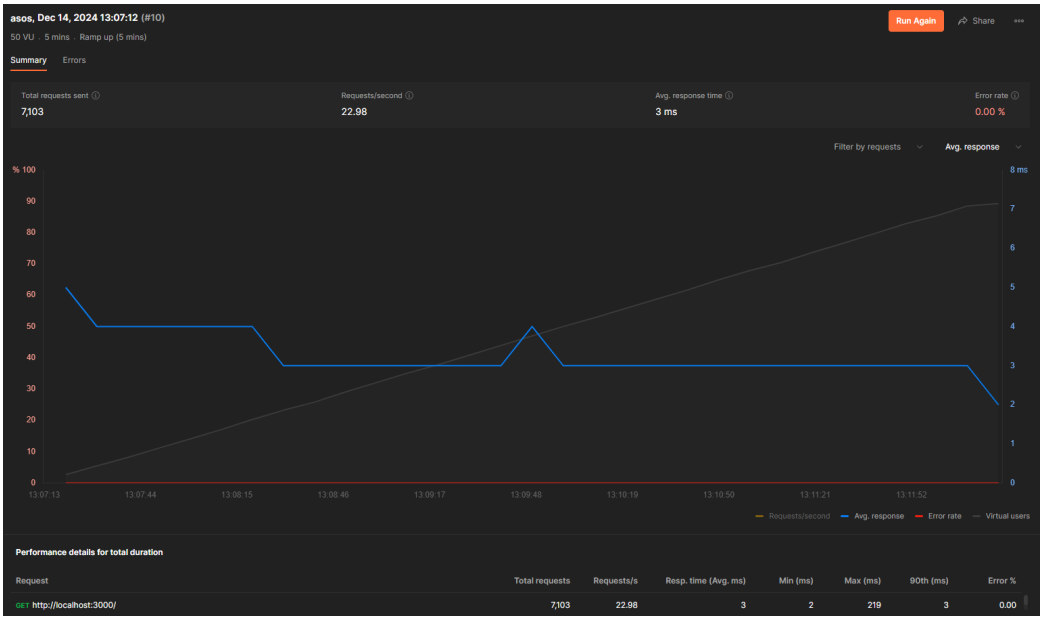

Figure 3: Load test using Postman

5 Installation Instructions

5.1 Run in Docker

Prerequisites:

- Install Docker: https://www.docker.com/

To start the application using Docker, run the following command:

`docker-compose up`

This will spin up the backend, database, and frontend in Docker containers.

5.2 Run Locally

Set Environment Variables: Create a `.env` file in the root directory with the following PostgreSQL database configuration:

```
DB_HOST=localhost
DB_PORT=5432
DB_USER=postgres
DB_PASSWORD=postgres
DB_NAME=postgres
```

Install Dependencies: Run the following commands in the project root directory:

```
npm install
```

Start Backend:

```
cd backend
npm run start:dev
```

Start Frontend:

```
cd frontend
npm run start:frontend
```

6 Usage

1. Open the application in your browser at: `http://localhost:3001`
2. Access the backend Swagger documentation at: `http://localhost:3000/api`

6.1 Functionalities Checklist

The following functionalities have been implemented in the project:

6.1.1 Project Setup and Documentation

- Properly structured project with clear organization and meaningful directory structure.
- Comprehensive README file describing setup instructions, project features, and how to run the application.
- Use of version control with meaningful commit messages (Git).

6.1.2 Backend Implementation

- **API Design**:
  - RESTful principles for data management.
  - Well-structured routes and endpoint documentation via Swagger.
- **Database Integration**:
  - Use of PostgreSQL with TypeORM.
  - Well-defined models with appropriate relationships and indexing.
- **Security**:
  - Implementation of JWT-based authentication.
  - Data validation using class-validator.
  - Protection against common vulnerabilities such as SQL injection.

6.1.3 Frontend Implementation

- **Responsive Design**:
  - Fully responsive application with mobile-first design considerations.
- **User Experience**:
  - Clear and intuitive UI/UX with easy navigation.
- **State Management and Interaction**:
  - Proper state handling with Vue.js.
- **API Integration**:
  - Smooth integration with backend APIs.
  - Proper handling of asynchronous requests.

6.1.4 Containerization

- Use of Docker for consistent development environments.