NestJS User Management Project: Detailed Documentation

Project Overview

The **NestJS User Management** project is a simple CRUD application that allows for managing users in a PostgreSQL database. This project uses NestJS, TypeORM, and PostgreSQL. It implements best practices such as server-side validation, logging of user actions, and clean code architecture with modules, services, and controllers.

Core Features:

1. **User CRUD Operations**:
    - **Create a User**
    - **Update a User**
    - **Delete a User**
2. **Action Logging**:
    - All actions (create, update, delete) are logged in a separate `log` table.
3. **Server-side Validation**:
    - Validates user inputs using **pipes**.
4. **Authorization Guard**:
    - Guards ensure the presence of a valid authorization token for secured endpoints.
5. **Best Practices**:
    - Use of **modules**, **services**, **controllers**, and **dependency injection**.
    - Implement **pipes**, **guards**, and **interceptors** for clean code organization.

Database Setup

The project uses **PostgreSQL** as the database, with two main tables: `user` and `log`.

Database Schema:

1. **User Table**:
    - Stores user information such as full name, email, and password.
2. **Log Table**:
    - Logs actions performed on users (create, update, delete), including timestamps.

Entities:

1. **User Entity**: Defined in `user.entity.ts`. It represents the `user` table in the database.

```
ts
Copy code
@Entity()
export class User {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  fullName: string;

  @Column()
  email: string;

  @Column()
  password: string;
}
```

2. **Log Entity**: Defined in `log.entity.ts`. It represents the `log` table in the database.

```
ts
Copy code
@Entity()
export class Log {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  action: string;

  @Column()
  description: string;

  @CreateDateColumn()
  timestamp: Date;
}
```

Modules

The project is split into two main modules:

1. **UserModule**: Manages user-related operations.
2. **LogModule**: Handles logging of actions.

Each module is responsible for organizing the related services, controllers, and entities.

REST API Endpoints

User CRUD Operations

The following endpoints are exposed to manage users:

1. **POST /users**: Create a new user.
    - **Body**:

      ```
      json
      Copy code
      {
        "fullName": "John Doe",
        "email": "john@example.com",
        "password": "password123"
      }
      ```

2. **PUT /users/**

   : Update an existing user.

    - **Body**:

      ```
      json
      Copy code
      {
        "fullName": "Jane Doe"
      }
      ```

3. **DELETE /users/**

   : Delete an existing user.

Logging

Each time a user is created, updated, or deleted, an entry is logged in the `log` table with a description of the action.

Implementation of Pipes, Guards, and Dependency Injection

Pipes: Validation

The project uses **pipes** to validate incoming data. Specifically, `class-validator` and `class-transformer` are used in the DTOs to ensure that the user data (email, fullName, password) is valid before it's processed.

Example: DTO with Validation Rules

CreateUserDto (located in `create-user.dto.ts`) defines validation for creating a new user:

```
ts
Copy code
import { IsEmail, IsNotEmpty } from 'class-validator';

export class CreateUserDto {
```

```
  @IsNotEmpty()
  fullName: string;

  @IsEmail()
  email: string;

  @IsNotEmpty()
  password: string;
}
```

- **IsEmail**: Ensures that the `email` field is a valid email.
- **IsNotEmpty**: Ensures that `fullName` and `password` fields are not empty.

These validation rules are automatically enforced when data is sent to the API, ensuring only valid data is processed.

Guards: Authorization

An **authorization guard** is implemented to protect the routes by checking for a valid authorization token in the request headers.

Example: AuthGuard

AuthGuard (located in `auth.guard.ts`) checks if the incoming request has a valid token in the `Authorization` header.

```
ts
Copy code
import { CanActivate, ExecutionContext, Injectable, UnauthorizedException }
from '@nestjs/common';

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean {
    const request = context.switchToHttp().getRequest();
    const token = request.headers['authorization'];

    if (!token || token !== 'VALID_TOKEN') {
      throw new UnauthorizedException();
    }

    return true;
  }
}
```

- This guard can be applied to any controller method using the `@UseGuards(AuthGuard)` decorator.

Interceptors: Logging Actions

Interceptors can be used to transform data or log responses. Though not explicitly implemented in this project, interceptors could be easily added for tasks such as logging response data.

Dependency Injection

NestJS makes heavy use of **dependency injection** to manage services. In this project:

- The **UserService** is injected into the **UserController** to handle business logic.
- The **LogService** is injected into the **UserService** to log actions.

Example: Dependency Injection in UserService

```
ts
Copy code
@Injectable()
export class UserService {
  constructor(
    @InjectRepository(User) private readonly userRepo: Repository<User>,
// Inject UserRepository
    private readonly logService: LogService,  // Inject LogService
  ) {}

  async createUser(data: CreateUserDto) {
    const user = this.userRepo.create(data);
    await this.userRepo.save(user);
    await this.logService.logAction('Create User', `User ${user.email}
created`);
    return user;
  }
}
```

Logging System

Every action performed on the `user` table (create, update, delete) is logged in the `log` table using the **LogService**. This ensures that you can track all changes made to users over time.

Example of Logging:

```
ts
Copy code
async createUser(data: CreateUserDto) {
  const user = this.userRepo.create(data);
  await this.userRepo.save(user);
  await this.logService.logAction('Create User', `User ${user.email}
created`);
  return user;
}
```

This entry will be saved in the `log` table, with the action and timestamp.

Project Structure

```
bash
Copy code
```

```
/src
  /user
    user.module.ts        # User module
    user.controller.ts    # User controller (API routes)
    user.service.ts       # User service (business logic)
    user.entity.ts        # User entity (database table)
    dto/
      create-user.dto.ts  # Data transfer object for user creation
      update-user.dto.ts  # Data transfer object for user updates
  /log
    log.module.ts         # Log module
    log.service.ts        # Log service (handles logging)
    log.entity.ts         # Log entity (database table)
  /guards
    auth.guard.ts         # Authorization guard
  app.module.ts           # Main application module
  main.ts                 # Application entry point
```

How to Run the Project

1. Install Dependencies

Make sure to install the necessary dependencies by running:

```
bash
Copy code
npm install
```

2. Set Up PostgreSQL Database

1. Ensure PostgreSQL is running.
2. Create a database named `user-management`.
3. Ensure the `TypeOrmModule` is configured with your PostgreSQL credentials in `app.module.ts`.

3. Run the Application

To start the application, run:

```
bash
Copy code
npm run start
```

The application will be available at `http://localhost:3000`.

4. Test the API

You can test the API using Postman or any HTTP client to interact with the following endpoints:

- `POST /users`
- `PUT /users/:id`

- `DELETE /users/:id`

Conclusion

This **NestJS User Management** project implements user CRUD operations with logging and server-side validation, while following the best practices of using pipes, guards, dependency injection, and a modular structure. The project is built to be secure, scalable, and easy to extend.