Managers Backend with Nestjs

In this assignment, you're going to implement the backend app used by managers to create and manage their challenges. It's separate from the previous express backend application but it uses the same database. Also, this backend will be integrated with the managers' dashboard in Next js application.

Initial Setup

First, initialize a `nestjs` project and set the serving port to an unutilized port. Next, we are going to use a set of nestjs packages, so make sure they're installed.

- Install `@nestjs/mongoose` and `mongoose` as detailed here.
- Install `class-validator` and `class-transformer` for validation.
- Install `@nestjs/jwt` for jwt-based authentication.

After you install all of these packages, proceed to the tasks.

1. Challenges CRUD Operations

- Create a `challenges` module and add a `controller` and `service` to it. Make sure, that the controller and the service are registered in the module and the module is added to the root module.
- Add routes to `get all the challenges`, `get a challenge by id`, `update a challenge`, and `delete a challenge`.
- Add the `Manager` and `Challenge` mongoose schemas
- The schemas are the same as your previous express backend app.
- Make sure to import these models in the challenges module.
- Make sure to connect the Mongoose module to the root module.
- Create the set of `DTOs` (Data transfer objects) for challenge creation and update. Make sure to add validation decorators and enable the validation pipe for that controller.
- Add the DTOs to their associated routes in the controller and enable validation.

- Implement the different service functions in the challenges service for each one of the previous controller endpoints.

2. Authentication and Authorization

We want to make sure that clients are authenticated and that they're indeed managers so that only managers are able to make requests to this API.

> Note: Managers get their tokens from your express backend via the login endpoints, and they should pass it as an authorization header when they make requests to this application.

- First, create a `Roles` decorator that assigns metadata to the controller. It should accept a list of role strings that signifies the allowed roles to make requests to a specific controller. You can find more about the metadata decorator here.

- Create an authentication guard that checks the existence of the token in the authorization header. If the header is not present, the request is denied.

- If the header is present, make sure to verify it using `JwtService` from `@nestjs/jwt`. If it's not valid, deny the request.

If the request is valid:

- Extract the roles from the payload.
- Extract the list of allowed roles that were declared when decorating the controller.
- Make sure that the request role is included in the declared roles.
- Inject user info in the request object.
- Create a parameter decorator that extracts user info from the request. This decorator will be used in the controller functions like this:

Assume the name of the decorator is `@AuthenticatedUser`. it will be injected in the function as follows:

JavaScript

```
  @Get()

    async findAll(

        @AuthenticatedUser() user: User,

    ) {

        // user contains id and role and it'll be used by
the service

        await this.challengesService.getAll(...);
```