Certainly! Let's proceed to the next subdirectory in your NestJS application's common/ directory: the guards/ directory. This directory is essential for implementing authorization logic and controlling access to various parts of your application.

**📁 common/guards/ Directory Overview**

In NestJS, **guards** are classes that implement the CanActivate interfaceThey determine whether a given request will be handled by the route handler based on specific conditions, such as user roles, permissions, or authentication statusOrganizing your guards within the common/guards/ directory promotes reusability and maintainability across your application

**🗂️ Suggested Subdirectories within guards/**

To maintain a clean and organized structure, consider categorizing your guards based on their functionalit: 

common/

└── guards/

├── auth/ # Guards related to authentication

│ └── jwt-auth.guard.ts

├── roles/ # Guards handling role-based access control

│ └── roles.guard.ts

└── permissions/ # Guards managing specific permissions

└── permissions.guard.ts



**📄 Detailed Breakdown of Each Subdirectory**

**1. auth/ Subdirectory**

* \**Purpose*: Contains guards that handle authentication logic, such as verifying JWT tokes.
* \**Example*: jwt-auth.guard.s 

import { Injectable } from '@nestjs/common';

import { AuthGuard } from '@nestjs/passport';

@Injectable()

export class JwtAuthGuard extends AuthGuard('jwt') {}

``



\*\*Usage\*\*:

Apply this guard to routes that require authenticated acces.



```typescript

@UseGuards(JwtAuthGuard)

@Get('profile')

getProfile(@Request() req) {

return req.user;

}

``



### 2. \*\*`roles/` Subdirectory\*\*

- \*\*Purpose\*: Houses guards that implement role-based access control (RBA).

- \*\*Example\*: `roles.guard.s`



```typescript

import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';

import { Reflector } from '@nestjs/core';

@Injectable()

export class RolesGuard implements CanActivate {

constructor(private reflector: Reflector) {}

canActivate(context: ExecutionContext): boolean {

const requiredRoles = this.reflector.getAllAndOverride<string[]>('roles', [

context.getHandler(),

context.getClass(),

]);

if (!requiredRoles) {

return true;

}

const { user } = context.switchToHttp().getRequest();

return requiredRoles.some((role) => user.roles?.includes(role));

}

}

``



\*\*Usage\*\*:

Define required roles using a custom decorator and apply the guard to routs.



```typescript

@SetMetadata('roles', ['admin'])

@UseGuards(RolesGuard)

@Get('admin')

getAdminData() {

return 'Admin data';

}

``



### 3. \*\*`permissions/` Subdirectory\*\*

- \*\*Purpose\*: Manages guards that enforce specific permissions beyond general rols.

- \*\*Example\*: `permissions.guard.s`



```typescript

import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';

import { Reflector } from '@nestjs/core';

@Injectable()

export class PermissionsGuard implements CanActivate {

constructor(private reflector: Reflector) {}

canActivate(context: ExecutionContext): boolean {

const requiredPermissions = this.reflector.getAllAndOverride<string[]>('permissions', [

context.getHandler(),

context.getClass(),

]);

if (!requiredPermissions) {

return true;

}

const { user } = context.switchToHttp().getRequest();

return requiredPermissions.every((permission) =>

user.permissions?.includes(permission),

);

}

}

``



\*\*Usage\*\*:

Define required permissions using a custom decorator and apply the guard to routs.



```typescript

@SetMetadata('permissions', ['read:reports'])

@UseGuards(PermissionsGuard)

@Get('reports')

getReports() {

return 'Reports data';

}

``



---

## ✅ Benefits of Organizing Guards in `common/guards/`

- \*\*Centralized Access Control\*: Having all guards in a dedicated directory promotes a single source of truth for authorization logc.

- \*\*Reusability\*: Guards can be easily applied across different parts of the application, reducing code duplicatin.

- \*\*Maintainability\*: A well-structured directory makes it easier to manage and update guards as the application evolvs.

- \*\*Enhanced Readability\*: Categorizing guards by functionality allows developers to quickly locate and understand their purpoe.

--

By structuring your `common/guards/` directory as outlined above, you ensure that your application's access control mechanisms are handled efficiently and consistently, leading to a more secure and maintainable codebae.

If you need further assistance or examples on implementing these components, feel free to ask!