**📁 common/decorators/ Directory Overview**

In NestJS, decorators are functions that can modify the behavior of classes, methods, or properties. They are extensively used for defining routes, injecting dependencies, and more. Custom decorators allow you to encapsulate repetitive logic, making your codebase cleaner and more expressive

**🗂️ Suggested Subdirectories within decorators/**

To maintain an organized structure, consider categorizing your decorators based on their functionalit: 

common/

└── decorators/

├── auth/ # Authentication and authorization related decorators

│ └── roles.decorator.ts

├── request/ # Request-related decorators

│ └── user.decorator.ts

└── validation/ # Validation-related decorators

└── unique-email.decorator.ts

``



---

## 📄 Detailed Breakdown of Each Subdirectory

### 1. \*\*`auth/` Subdirectory\*\*

- \*\*Purpose\*: Contains decorators related to authentication and authorizatin.

- \*\*Example\*: `roles.decorator.s`



```typescript

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

/\*\*

\* Custom decorator to specify required roles for route access.

\* @param roles - Array of roles permitted to access the route.

\*/

export const Roles = (...roles: string[]) => SetMetadata('roles', roles);

``



\*\*Usage\*\*:



```typescript

@Roles('admin', 'user')

@Get('dashboard')

getDashboard() {

// Route accessible by 'admin' and 'user' roles.

}

``



---

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

- \*\*Purpose\*: Houses decorators that extract or manipulate data from the incoming requet.

- \*\*Example\*: `user.decorator.s`



```typescript

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

/\*\*

\* Custom decorator to extract the user object from the request.

\*/

export const User = createParamDecorator(

(data: unknown, ctx: ExecutionContext) => {

const request = ctx.switchToHttp().getRequest();

return request.user;

},

);

``



\*\*Usage\*\*:



```typescript

@Get('profile')

getProfile(@User() user: UserEntity) {

// Access the authenticated user's information.

}

``



---

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

- \*\*Purpose\*: Contains decorators that handle validation logic, often used in conjunction with DTs.

- \*\*Example\*: `unique-email.decorator.s`



```typescript

import { registerDecorator, ValidationOptions, ValidatorConstraint, ValidatorConstraintInterface, ValidationArguments } from 'class-validator';

@ValidatorConstraint({ async: true })

export class IsEmailUniqueConstraint implements ValidatorConstraintInterface {

validate(email: any, args: ValidationArguments) {

// Implement logic to check if the email is unique.

return true; // Return true if unique, false otherwise.

}

}

/\*\*

\* Custom decorator to validate if an email is unique.

\* @param validationOptions - Optional validation options.

\*/

export function IsEmailUnique(validationOptions?: ValidationOptions) {

return function (object: Object, propertyName: string) {

registerDecorator({

target: object.constructor,

propertyName: propertyName,

options: validationOptions,

constraints: [],

validator: IsEmailUniqueConstraint,

});

};

}

``



\*\*Usage\*\*:

```typescript

import { IsEmailUnique } from '../common/decorators/validation/unique-email.decorator';

export class CreateUserDto {

@IsEmailUnique({ message: 'Email already exists' })

email: string;

}

``



---

## ✅ Benefits of Organizing Decorators in `common/decorators/`

- \*\*Reusability\*: Centralizing decorators allows for easy reuse across different moduls.

- \*\*Maintainability\*: Having a dedicated directory makes it straightforward to manage and update decoratos.

- \*\*Clarity\*: Categorizing decorators based on their functionality enhances code readability and organizatin.

--

By structuring your `common/decorators/` directory as outlined above, you promote a clean and maintainable codebase, making it easier for developers to understand and utilize custom decorators effectivey.