Organizing your NestJS application's common/ directory effectively enhances code reusability, maintainability, and scalability. Here's a comprehensive breakdown of suggested subdirectories and files within the common/ directory, along with detailed explanations and usage examples:

**📁 common/ Directory Structure**

common/

├── decorators/ # Custom decorators to add metadata or modify behavior

│ └── roles.decorator.ts

├── filters/ # Global or route-specific exception filters

│ └── http-exception.filter.ts

├── guards/ # Authorization guards to protect routes

│ └── roles.guard.ts

├── interceptors/ # Intercept and manipulate requests/responses

│ └── logging.interceptor.ts

├── pipes/ # Transform and validate incoming request data

│ └── validation.pipe.ts

├── services/ # Shared services used across the application

│ └── logger.service.ts

└── common.module.ts # Aggregates and exports common components

**📄 Detailed Breakdown**

**1. decorators/**

* **Purpose**: Custom decorators to add metadata or modify behavior.
* **File**: roles.decorator.ts

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

/\*\*

\* Custom decorator to assign roles to route handlers.

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

\*/

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

**Usage**:

@Roles('admin', 'user')

@Get('dashboard')

getDashboard() {

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

}

**2. filters/**

* **Purpose**: Global or route-specific exception filters.
* **File**: http-exception.filter.ts

import {

ExceptionFilter,

Catch,

ArgumentsHost,

HttpException,

HttpStatus,

} from '@nestjs/common';

import { Request, Response } from 'express';

/\*\*

\* Global exception filter to handle all unhandled exceptions.

\*/

@Catch()

export class HttpExceptionFilter implements ExceptionFilter {

catch(exception: unknown, host: ArgumentsHost) {

const ctx = host.switchToHttp();

const response = ctx.getResponse<Response>();

const request = ctx.getRequest<Request>();

const status =

exception instanceof HttpException

? exception.getStatus()

: HttpStatus.INTERNAL\_SERVER\_ERROR;

const message =

exception instanceof HttpException

? exception.getResponse()

: 'Internal server error';

response.status(status).json({

statusCode: status,

timestamp: new Date().toISOString(),

path: request.url,

message,

});

}

}

**Usage**:

Apply globally in main.ts:

import { HttpExceptionFilter } from './common/filters/http-exception.filter';

async function bootstrap() {

const app = await NestFactory.create(AppModule);

app.useGlobalFilters(new HttpExceptionFilter());

await app.listen(3000);

}

bootstrap();

**3. guards/**

* **Purpose**: Authorization guards to protect routes.
* **File**: roles.guard.ts

import {

Injectable,

CanActivate,

ExecutionContext,

ForbiddenException,

} from '@nestjs/common';

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

/\*\*

\* Guard to restrict access based on user roles.

\*/

@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();

if (!user || !requiredRoles.includes(user.role)) {

throw new ForbiddenException('Access denied');

}

return true;

}

}

**Usage**:

Apply to controllers or routes:

import { Roles } from '../common/decorators/roles.decorator';

import { RolesGuard } from '../common/guards/roles.guard';

@UseGuards(RolesGuard)

@Roles('admin')

@Get('admin')

getAdminData() {

// ...

}

**4. interceptors/**

* **Purpose**: Intercept and manipulate requests/responses.
* **File**: logging.interceptor.ts

import {

Injectable,

NestInterceptor,

ExecutionContext,

CallHandler,

Logger,

} from '@nestjs/common';

import { Observable, tap } from 'rxjs';

/\*\*

\* Interceptor to log incoming requests and outgoing responses.

\*/

@Injectable()

export class LoggingInterceptor implements NestInterceptor {

private readonly logger = new Logger(LoggingInterceptor.name);

intercept(context: ExecutionContext, next: CallHandler): Observable<any> {

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

const { method, url } = request;

const now = Date.now();

return next.handle().pipe(

tap(() =>

this.logger.log(`${method} ${url} - ${Date.now() - now}ms`),

),

);

}

}

**Usage**:

Apply globally in main.ts:

import { LoggingInterceptor } from './common/interceptors/logging.interceptor';

async function bootstrap() {

const app = await NestFactory.create(AppModule);

app.useGlobalInterceptors(new LoggingInterceptor());

await app.listen(3000);

}

bootstrap();

Certainly! Let's delve into the pipes/ directory within your NestJS application's common/ folder, providing a comprehensive understanding of its purpose, structure, and usage.

**📁 common/pipes/ Directory Overview**

The pipes/ directory is dedicated to housing custom pipes that handle data transformation and validation across your applicationOrganizing pipes in this manner promotes reusability and maintains a clean codebase

**5. 📄 Custom Pipe Example: uppercase.pipe.ts**

Let's create a custom pipe that transforms incoming string data to uppercas. 

// common/pipes/uppercase.pipe.ts

import { PipeTransform, Injectable, ArgumentMetadata, BadRequestException } from '@nestjs/common';

/\*\*

\* Transforms incoming string data to uppercase.

\*/

@Injectable()

export class UppercasePipe implements PipeTransform {

transform(value: any, metadata: ArgumentMetadata): any {

if (typeof value !== 'string') {

throw new BadRequestException('Validation failed: Expected a string');

}

return value.toUpperCase();

}

}

``



---

## 🧪 Usage Exampl

Here's how you can apply the `UppercasePipe` to a route parametr:



```typescript

// users.controller.ts

import { Controller, Get, Param, UsePipes } from '@nestjs/common';

import { UppercasePipe } from '../common/pipes/uppercase.pipe';

@Controller('users')

export class UsersController {

@Get(':username')

@UsePipes(UppercasePipe)

getUserByUsername(@Param('username') username: string) {

// The 'username' parameter is now in uppercase

return `Fetching user with username: ${username}`;

}

}

``



In this example, when a request is made to `/users/johndoe`, the `username` parameter is transformed to `JOHNDOE` before reaching the route handlr.

---

## 🌐 Global Pipe Registratin

To apply a pipe globally across all routes, you can register it in the main application bootstrap fle:



```typescript

// main.ts

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

import { AppModule } from './app.module';

import { UppercasePipe } from './common/pipes/uppercase.pipe';

async function bootstrap() {

const app = await NestFactory.create(AppModule);

app.useGlobalPipes(new UppercasePipe());

await app.listen(3000);

}

bootstrap();

``



With this setup, all incoming string data across your application will be transformed to uppercase by defalt.

---

## 📚 Additional Resources

For more in-depth information on pipes in NestJS, consider exploring the official documentation:

- [NestJS Pipes Documentation](https://docs.nestjs.com/pipes)

If you have further questions or need assistance with other components, feel free to ask!

**6. services/**

* **Purpose**: Houses shared services that provide utility functions or shared logic across the application.
* **File**: logger.service.ts

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

/\*\*

\* A shared logger service that wraps NestJS's Logger.

\* Provides a centralized logging mechanism.

\*/

@Injectable()

export class LoggerService {

private readonly logger = new Logger('AppLogger');

log(message: string) {

this.logger.log(message);

}

error(message: string, trace: string) {

this.logger.error(message, trace);

}

warn(message: string) {

this.logger.warn(message);

}

debug(message: string) {

this.logger.debug(message);

}

verbose(message: string) {

this.logger.verbose(message);

}

}



**Usage**:

import { LoggerService } from '../common/services/logger.service';

@Injectable()

export class SomeService {

constructor(private readonly logger: LoggerService) {}

performAction() {

this.logger.log('Action performed');

}

}



**7. common.module.ts**

* **Purpose**: Aggregates and exports all shared components (decorators, filters, guards, interceptors, pipes, services) for easy import into other modules.

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

import { APP\_FILTER, APP\_GUARD, APP\_INTERCEPTOR, APP\_PIPE } from '@nestjs/core';

import { RolesGuard } from './guards/roles.guard';

import { HttpExceptionFilter } from './filters/http-exception.filter';

import { LoggingInterceptor } from './interceptors/logging.interceptor';

import { ValidationPipe } from './pipes/validation.pipe';

import { LoggerService } from './services/logger.service';

/\*\*

\* CommonModule aggregates and provides shared components across the application.

\*/

@Module({

providers: [

LoggerService,

{

provide: APP\_GUARD,

useClass: RolesGuard,

},

{

provide: APP\_FILTER,

useClass: HttpExceptionFilter,

},

{

provide: APP\_INTERCEPTOR,

useClass: LoggingInterceptor,

},

{

provide: APP\_PIPE,

useClass: ValidationPipe,

},

],

exports: [LoggerService],

})

export class CommonModule {}



**Usage**:

Import CommonModule into other modules to access shared components:

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

import { CommonModule } from '../common/common.module';

import { SomeService } from './some.service';

@Module({

imports: [CommonModule],

providers: [SomeService],

})

export class SomeModule {}



By structuring your common/ directory in this manner, you promote reusability and maintainability across your NestJS application. Each subdirectory serves a specific purpose, and the CommonModule ensures that shared components are easily accessible throughout your project.

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