**📁 common/pipes/ Directory Overview**

In NestJS, **pipes** are classes that implement the PipeTransform interfaceThey operate on the arguments being processed by a controller route handler, allowing you to transform or validate the data before it reaches the route handlerOrganizing your pipes within the common/pipes/ directory promotes reusability and maintainability across your application

**🗂️ Suggested Subdirectories within pipes/**

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

common/

└── pipes/

├── validation/

│ ├── custom-validation.pipe.ts

│ └── another-validation.pipe.ts

├── transformation/

│ ├── trim.pipe.ts

│ └── uppercase.pipe.ts

└── parsing/

├── parse-int.pipe.ts

└── parse-uuid.pipe.ts



**📄 Detailed Breakdown of Each Subdirectory**

**1. validation/ Subdirectory**

* \**Purpose*: Contains pipes that validate incoming data against specific rules or schems.
* \**Example*: custom-validation.pipe.s 

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

@Injectable()

export class CustomValidationPipe implements PipeTransform {

transform(value: any) {

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

throw new BadRequestException('Validation failed');

}

return value;

}

}

``



\*\*Usage\*\*:

Apply this pipe to route parameters or request bodies to enforce custom validation logc.

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

- \*\*Purpose\*: Houses pipes that transform incoming data into the desired formt.

- \*\*Example\*: `trim.pipe.s`



```typescript

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

@Injectable()

export class TrimPipe implements PipeTransform {

transform(value: any) {

return typeof value === 'string' ? value.trim() : value;

}

}

``



\*\*Usage\*\*:

Use this pipe to remove leading and trailing whitespace from string inpus.

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

- \*\*Purpose\*: Manages pipes that parse and convert data types, such as strings to integers or UUIs.

- \*\*Example\*: `parse-int.pipe.s`



```typescript

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

@Injectable()

export class ParseIntPipe implements PipeTransform {

transform(value: any) {

const val = parseInt(value, 10);

if (isNaN(val)) {

throw new BadRequestException('Validation failed');

}

return val;

}

}

``



\*\*Usage\*\*:

Apply this pipe to route parameters that should be integers to ensure proper type conversin.

---

## ✅ Benefits of Organizing Pipes in `common/pipes/`

- \*\*Centralized Data Handling\*: Having all pipes in a dedicated directory promotes a single source of truth for data transformation and validation logc.

- \*\*Reusability\*: Pipes 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 pipes as the application evolvs.

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

--

By structuring your `common/pipes/` directory as outlined above, you ensure that your application's data transformation and validation mechanisms are handled efficiently and consistently, leading to a more robust and maintainable codebae.

**📌 Understanding Pipes in NestJS**

In NestJS, **pipes** are classes that implement the PipeTransform interfaceThey are used for

* **Transformation**:Modifying incoming data to the desired format (e.g., converting a string to an integer)
* **Validation**:Ensuring that incoming data meets specific criteria before it's processed by the route handler Pipes can be applied at different levels: method, controller, or globally across the application citeturn0search0

**🔧 Applying Pipes**

**1. Method-Level Pipes**

Apply a pipe directly to a specific route handler paramete: 

@Get(':id')

getUser(@Param('id', ParseIntPipe) id: number) {

return this.userService.findById(id);

}

``



In this example, `ParseIntPipe` transforms the `id` parameter from a string to an intege.

### 2. \*\*Controller-Level Pipes\*\*

Apply a pipe to all route handlers within a controlle:



```typescript

@UsePipes(ValidationPipe)

@Controller('users')

export class UsersController {

// All methods here will use ValidationPipe

}

``



This ensures that every method within `UsersController` utilizes the `ValidationPipe` for validating incoming request.

### 3. \*\*Global Pipes\*\*

Apply a pipe across the entire applicatio:

- \*\*Without Dependencies\*\* Register in `main.ts:



```typescript

const app = await NestFactory.create(AppModule);

app.useGlobalPipes(new ValidationPipe());

``



- \*\*With Dependencies\*\* Register using the `APP\_PIPE` token in a modul:



```typescript

@Module({

providers: [

{

provide: APP\_PIPE,

useClass: ValidationPipe,

},

],

})

export class AppModule {}

``



This approach allows the pipe to leverage dependency injection, making it suitable for pipes that depend on other service. citeturn0search0

---

## 🛠️ Creating Custom Pipe

To create a custom pipe, implement the `PipeTransform` interfae:



```typescript

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

@Injectable()

export class TrimPipe implements PipeTransform {

transform(value: any) {

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

throw new BadRequestException('Validation failed');

}

return value.trim();

}

}

``



This `TrimPipe` removes leading and trailing whitespace from string inpus.

\*\*Usage\*:



```typescript

@Post()

createUser(@Body('username', TrimPipe) username: string) {

return this.userService.create({ username });

}

``



Here, the `TrimPipe` is applied to the `username` field in the request body, ensuring it's trimmed before processig.

---

## 📁 Organizing Pipes in Your Project Structure

To maintain a clean and scalable codebase, it's advisable to organize pipes within the `common/pipes/` directory, categorizing them based on their functionalty:



```bash

common/

└── pipes/

├── validation/

│ └── custom-validation.pipe.ts

├── transformation/

│ └── trim.pipe.ts

└── parsing/

└── parse-int.pipe.ts

``



This structure promotes reusability and clarity, making it easier to manage and apply pipes across different modules and controllrs.

---

## ✅ Best Practices

- \*\*Scope Appropriately\*: Apply pipes at the most appropriate level (method, controller, or global) based on their intended effct.

- \*\*Leverage Dependency Injection\*: When pipes depend on other services, register them using the `APP\_PIPE` token within a module to enable dependency injecton.

- \*\*Avoid Overuse\*: Be cautious when applying multiple global pipes, as they can introduce complexity and affect performace.

- \*\*Maintain Modularity\*: Keep pipes modular and focused on a single responsibility to enhance maintainabilty.

--