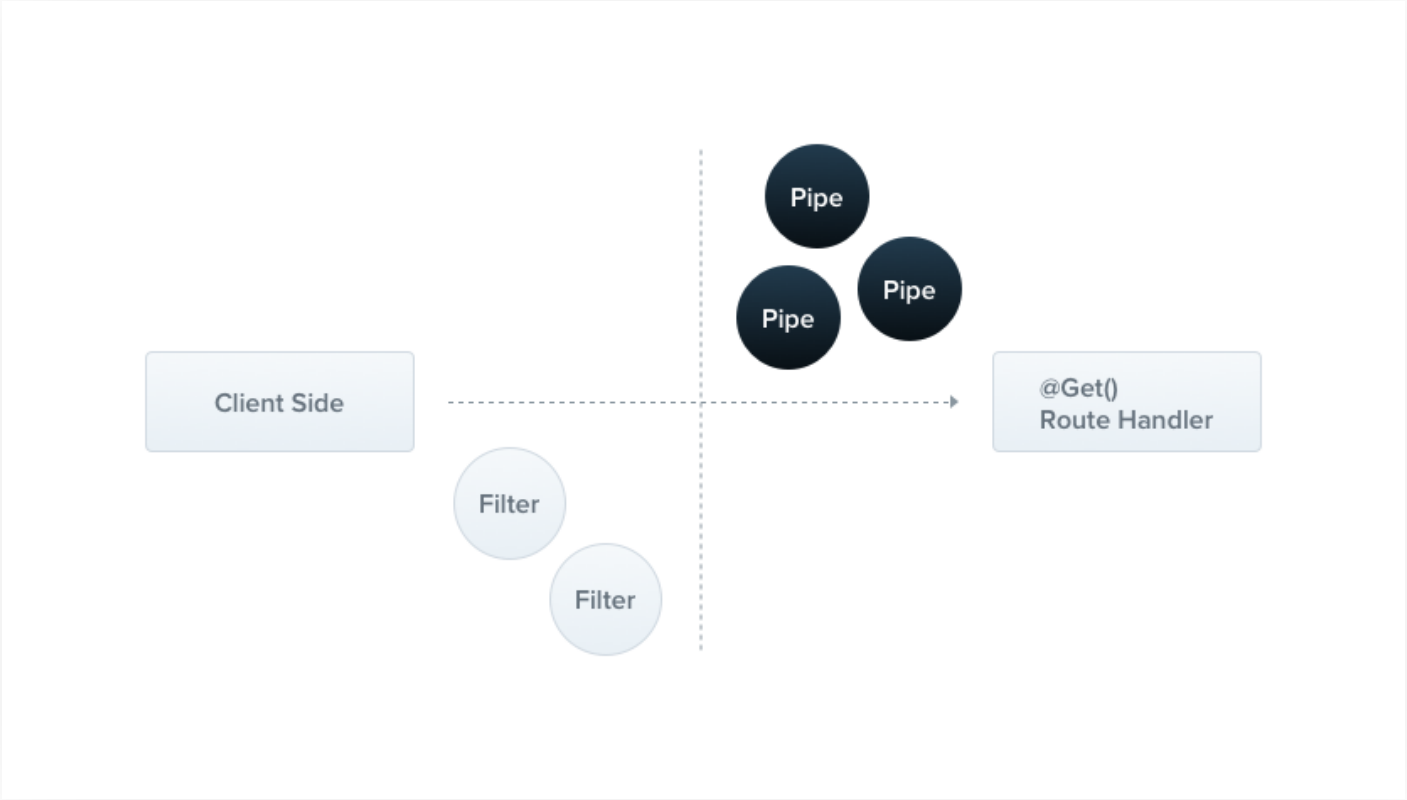
Pipes

A pipe is a class annotated with the `@Injectable()` decorator. Pipes should implement the `PipeTransform` interface.



Pipes have two typical use cases:

- **transformation**: transform input data to the desired output
- **validation**: evaluate input data and if valid, simply pass it through unchanged; otherwise, throw an exception when the data is incorrect

In both cases, pipes operate on the `arguments` being processed by a **controller route handler**. Nest interposes a pipe just before a method is invoked, and the pipe receives the arguments destined for the method. Any transformation or validation operation takes place at that time, after which the route handler is invoked with any (potentially) transformed arguments.

> **HINT**
> Pipes run inside the exceptions zone. This means that when a Pipe throws an exception it is handled by the exceptions layer (global exceptions filter and any **exceptions filters** that are applied to the current context). Given the above, it should be clear that when an exception is thrown in a Pipe, no controller method is subsequently executed.

Built in pipes

Built-in pipes

Nest comes with three pipes available right out-of-the-box: `ValidationPipe`, `ParseIntPipe` and `ParseUUIDPipe`. They're exported from the `@nestjs/common` package. In order to better understand how they work, let's build them from scratch.

Let's start with the `ValidationPipe`. Initially, we'll have it simply take an input value and immediately return the same value, behaving like an identity function.

validation.pipe.ts JS

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

@Injectable()
export class ValidationPipe implements PipeTransform {
  transform(value: any, metadata: ArgumentMetadata) {
    return value;
  }
}
```

> **HINT**
> `PipeTransform<T, R>` is a generic interface in which `T` indicates the type of the input `value`, and `R` indicates the return type of the `transform()` method.

Every pipe has to provide the `transform()` method. This method has two parameters:

- `value`
- `metadata`

The `value` is the currently processed argument (before it is received by the route handling method), while `metadata` is its metadata. The metadata object has these properties:

```
export interface ArgumentMetadata {
  readonly type: 'body' | 'query' | 'param' | 'custom';
  readonly metatype?: Type<any>;
  readonly data?: string;
}
```

These properties describe the currently processed argument.

`type`	Indicates whether the argument is a body `@Body()`, query `@Query()`, param `@Param()`,

	or a custom parameter (read more here).
`metatype`	Provides the metatype of the argument, for example, `String`. Note: the value is `undefined` if you either omit a type declaration in the route handler method signature, or use vanilla JavaScript.
`data`	The string passed to the decorator, for example `@Body('string')`. It's `undefined` if you leave the decorator parenthesis empty.

> **WARNING**
> TypeScript interfaces disappear during transpilation. Thus, if a method parameter's type is declared as an interface instead of a class, the `metatype` value will be `Object`.

Validation use case

Let's take a closer look at the `create()` method of the `CatsController`.

JS

```
@Post()
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

Let's focus in on the `createCatDto` body parameter. Its type is `CreateCatDto`:

create-cat.dto.ts JS

```
export class CreateCatDto {
  readonly name: string;
  readonly age: number;
  readonly breed: string;
}
```

We want to ensure that any incoming request to the create method contains a valid body. So we have to validate the three members of the `createCatDto` object. We could do this inside the route handler method, but we would break the **single responsibility rule** (SRP). Another approach could be to create a **validator class** and delegate the task there, but we would

have to use this validator at the beginning of each method. How about creating a validation middleware? This could be a good idea, but it's not possible to create **generic middleware** which can be used across the whole application (because middleware is unaware of the **execution context**, including the handler that will be called and any of its parameters).

It turns out that this is a case ideally suited for a **Pipe**. So let's go ahead and build one.

Object schema validation

There are several approaches available for object validation. One common approach is to use **schema-based** validation. The **Joi** library allows you to create schemas in a pretty straightforward way, with a readable API. Let's look at a pipe that makes use of Joi-based schemas.

In the code sample below, we create a simple class that takes a schema as a `constructor` argument. We then apply the `Joi.validate()` method, which validates our incoming argument against the provided schema.

As noted earlier, a **validation pipe** either returns the value unchanged, or throws an exception.

In the next section, you'll see how we supply the appropriate schema for a given controller method using the `@UsePipes()` decorator.

JS

```
import * as Joi from 'joi';
import { PipeTransform, Injectable, ArgumentMetadata, BadRequestException } from '@nestjs/common';

@Injectable()
export class JoiValidationPipe implements PipeTransform {
  constructor(private readonly schema: Object) {}

  transform(value: any, metadata: ArgumentMetadata) {
    const { error } = Joi.validate(value, this.schema);
    if (error) {
      throw new BadRequestException('Validation failed');
    }
    return value;
  }
}
```

Binding pipes

Binding pipes (tying them to the appropriate controller or handler) is very straightforward. We use the `@UsePipes()` decorator and create a pipe instance, passing it a Joi validation schema.

JS

```
@Post()
```

```
@UsePipes(new JoiValidationPipe(createCatSchema))
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

Class validator

> **WARNING**
> The techniques in this section require TypeScript, and are not available if your app is written using vanilla JavaScript.

Let's look at an alternate implementation of our validation technique.

Nest works well with the **class-validator** library. This amazing library allows you to use decorator-based validation. Decorator-based validation is extremely powerful, especially when combined with Nest's **Pipe** capabilities since we have access to the `metatype` of the processed property. Before we start, we need to install the required packages:

```
$ npm i --save class-validator class-transformer
```

Once these are installed, we can add a few decorators to the `CreateCatDto` class.

create-cat.dto.ts JS

```
import { IsString, IsInt } from 'class-validator';

export class CreateCatDto {
  @IsString()
  readonly name: string;

  @IsInt()
  readonly age: number;

  @IsString()
  readonly breed: string;
}
```

> **HINT**
> Read more about the class-validator decorators **here**.

Now we can create a `ValidationPipe` class.

validation.pipe.ts JS

```
import { PipeTransform, Injectable, ArgumentMetadata, BadRequestException } from '@nestjs/common';
import { validate } from 'class-validator';
import { plainToClass } from 'class-transformer';

@Injectable()
export class ValidationPipe implements PipeTransform<any> {
  async transform(value: any, { metatype }: ArgumentMetadata) {
    if (!metatype || !this.toValidate(metatype)) {
      return value;
    }
    const object = plainToClass(metatype, value);
    const errors = await validate(object);
    if (errors.length > 0) {
      throw new BadRequestException('Validation failed');
    }
    return value;
  }

  private toValidate(metatype: Function): boolean {
    const types: Function[] = [String, Boolean, Number, Array, Object];
    return !types.includes(metatype);
  }
}
```

> **NOTICE**
> Above, we have used the **class-transformer** library. It's made by the same author as the **class-validator** library, and as a result, they play very well together.

Let's go through this code. First, note that the `transform()` function is `async`. This is possible because Nest supports both synchronous and **asynchronous** pipes. We do this because some of the class-validator validations **can be async** (utilize Promises).

Next note that we are using destructuring to extract the metatype field (extracting just this member from an `ArgumentMetadata`) into our `metatype` parameter. This is just shorthand for getting the full `ArgumentMetadata` and then having an additional statement to assign the metatype variable.

Next, note the helper function `toValidate()`. It's responsible for bypassing the validation step when the current argument being processed is a native JavaScript type (these can't have schemas attached, so there's no reason to run

them through the validation step).

Next, we use the class-transformer function `plainToClass()` to transform our plain JavaScript argument object into a typed object so that we can apply validation. The incoming body, when deserialized from the network request, does not have any type information. Class-validator needs to use the validation decorators we defined for our DTO earlier, so we need to perform this transformation.

Finally, as noted earlier, since this is a **validation pipe** it either returns the value unchanged, or throws an exception.

The last step is to bind the `ValidationPipe`. Pipes, similar to **exception filters**, can be method-scoped, controller-scoped, or global-scoped. Additionally, a pipe can be param-scoped. In the example below, we'll directly tie the pipe instance to the route param `@Body()` decorator.

cats.controller.ts JS

```
@Post()
async create(
  @Body(new ValidationPipe()) createCatDto: CreateCatDto,
) {
  this.catsService.create(createCatDto);
}
```

Param-scoped pipes are useful when the validation logic concerns only one specified parameter.

Alternatively, to set up a pipe at a method level, use the `@UsePipes()` decorator.

cats.controller.ts JS

```
@Post()
@UsePipes(new ValidationPipe())
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

> **HINT**
> The `@UsePipes()` decorator is imported from the `@nestjs/common` package.

In the example above, an instance of `ValidationPipe` has been created immediately in-place. Alternatively, pass the class (not an instance), thus leaving instantiation up to the framework, and enabling **dependency injection**.

cats.controller.ts JS

```
@Post()
@UsePipes(ValidationPipe)
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

Since the `ValidationPipe` was created to be as generic as possible, let's set it up as a **global-scoped** pipe, applied to every route handler across the entire application.

main.ts JS

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(3000);
}
bootstrap();
```

> **NOTICE**
> In the case of **hybrid apps** the `useGlobalPipes()` method doesn't set up pipes for gateways and micro services. For "standard" (non-hybrid) microservice apps, `useGlobalPipes()` does mount pipes globally.

Global pipes are used across the whole application, for every controller and every route handler. In terms of dependency injection, global pipes registered from outside of any module (with `useGlobalPipes()` as in the example above) cannot inject dependencies since this is done outside the context of any module. In order to solve this issue, you can set up a global pipe **directly from any module** using the following construction:

app.module.ts JS

```
import { Module } from '@nestjs/common';
import { APP_PIPE } from '@nestjs/core';

@Module({
  providers: [
    {
      provide: APP_PIPE,
      useClass: ValidationPipe,
    },
  ],
```

```
})
export class AppModule {}
```

> **HINT**
> When using this approach to perform dependency injection for the pipe, note that regardless of the module where this construction is employed, the pipe is, in fact, global. Where should this be done? Choose the module where the pipe (`ValidationPipe` in the example above) is defined. Also, `useClass` is not the only way of dealing with custom provider registration. Learn more **here**.

Transformation use case

Validation isn't the sole use case for **Pipes**. At the beginning of this chapter, we mentioned that a pipe can also **transform** the input data to the desired output. This is possible because the value returned from the `transform` function completely overrides the previous value of the argument. When is this useful? Consider that sometimes the data passed from the client needs to undergo some change - for example converting a string to an integer - before it can be properly handled by the route handler method. Furthermore, some required data fields may be missing, and we would like to apply default values. **Transformer pipes** can perform these functions by interposing a processing function between the client request and the request handler.

Here's a `ParseIntPipe` which is responsible for parsing a string into an integer value.

parse-int.pipe.ts JS

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

@Injectable()
export class ParseIntPipe implements PipeTransform<string, number> {
  transform(value: string, metadata: ArgumentMetadata): number {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      throw new BadRequestException('Validation failed');
    }
    return val;
  }
}
```

We can simply tie this pipe to the selected param as shown below:

JS

```
@Get(':id')
```

```
async findOne(@Param('id', new ParseIntPipe()) id) {
  return await this.catsService.findOne(id);
}
```

If you prefer you can use the `ParseUUIDPipe` which is responsible for parsing a string and validate if is a UUID.

JS

```
@Get(':id')
async findOne(@Param('id', new ParseUUIDPipe()) id) {
  return await this.catsService.findOne(id);
}
```

> **HINT**
> When using `ParseUUIDPipe()` you are parsing UUID in version 3, 4 or 5, if you only requires a specific version of UUID you can pass a version in the pipe options.

With this in place, `ParseIntPipe` or `ParseUUIDPipe` will be executed before the request reaches the corresponding handler, ensuring that it will always receive an integer or uuid (according on the used pipe) for the `id` parameter.

Another useful case would be to select an **existing user** entity from the database by id:

JS

```
@Get(':id')
findOne(@Param('id', UserByIdPipe) userEntity: UserEntity) {
  return userEntity;
}
```

We leave the implementation of this pipe to the reader, but note that like all other transformation pipes, it receives an input value (an `id`) and returns an output value (a `UserEntity` object). This can make your code more declarative and **DRY** by abstracting boilerplate code out of your handler and into a common pipe.

The built-in ValidationPipe

Fortunately, you don't have to build these pipes on your own since the `ValidationPipe` and the `ParseIntPipe` are provided by Nest out-of-the-box. (Keep in mind that `ValidationPipe` requires both `class-validator` and `class-transformer` packages to be installed).

The built in `ValidationPipe` offers more options than in the sample we built in this chapter, which has been kept basic

The built-in `ValidationPipe` offers more options than in the sample we built in this chapter, which has been kept basic for the sake of illustrating the basic mechanics of a pipe. You can find lots of examples here.

One such option is `transform`. Recall the earlier discussion about deserialized body objects being vanilla JavaScript objects (i.e., not having our DTO type). So far, we've used the pipe to validate our payload. You may recall that in the process, we used `class-transform` to temporarily convert our plain object into a typed object so that we could do the validation. The built-in ValidationPipe can also, optionally, return this converted object. We enable this behavior by passing in a configuration object to the pipe. For this option, pass a config object with the field `transform` with a value `true` as shown below:

cats.controller.ts JS

```
@Post()
@UsePipes(new ValidationPipe({ transform: true }))
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

HINT
The `ValidationPipe` is imported from the `@nestjs/common` package.

Because this pipe is based on the `class-validator` and `class-transformer` libraries, there are many additional options available. Like the `transform` option above, you configure these settings via a configuration object passed to the pipe. Following are the built-in options:

```
export interface ValidationPipeOptions extends ValidatorOptions {
  transform?: boolean;
  disableErrorMessages?: boolean;
  exceptionFactory?: (errors: ValidationError[]) => any;
}
```

In addition to these, all `class-validator` options (inherited from the `ValidatorOptions` interface) are available:

Option	Type	Description
`skipMissingProperties`	`boolean`	If set to true, validator will skip validation of all properties that are missing in the validating object.

`whitelist`	`boolean`	If set to true, validator will strip validated (returned) object of any properties that do not use any validation decorators.
`forbidNonWhitelisted`	`boolean`	If set to true, instead of stripping non-whitelisted properties validator will throw an exception.
`forbidUnknownValues`	`boolean`	If set to true, attempts to validate unknown objects fail immediately.
`disableErrorMessages`	`boolean`	If set to true, validation errors will not be returned to the client.
`exceptionFactory`	`Function`	Takes an array of the validation errors and returns an exception object to be thrown.
`groups`	`string[]`	Groups to be used during validation of the object.
`dismissDefaultMessages`	`boolean`	If set to true, the validation will not use default messages. Error message always will be `undefined` if its not explicitly set.
`validationError.target`	`boolean`	Indicates if target should be exposed in `ValidationError`
`validationError.value`	`boolean`	Indicates if validated value should be exposed in `ValidationError`.

> **NOTICE**
> Find more information about the `class-validator` package in its **repository**.

Support us

Nest is an MIT-licensed open source project. It can grow thanks to the support by these awesome people. If you'd like to join them, please read more **here**.

Principal Sponsor

Sponsors / Partners

Become a sponsor