Validation

It is best practice to validate the correctness of any data sent into a web application. To automatically validate incoming requests, Nest provides several pipes available right out-of-the-box:

- `ValidationPipe`
- `ParseIntPipe`
- `ParseBoolPipe`
- `ParseArrayPipe`
- `ParseUUIDPipe`

The `ValidationPipe` makes use of the powerful class-validator package and its declarative validation decorators. The `ValidationPipe` provides a convenient approach to enforce validation rules for all incoming client payloads, where the specific rules are declared with simple annotations in local class/DTO declarations in each module.

Overview

In the Pipes chapter, we went through the process of building simple pipes and binding them to controllers, methods or to the global app to demonstrate how the process works. Be sure to review that chapter to best understand the topics of this chapter. Here, we'll focus on various **real world** use cases of the `ValidationPipe`, and show how to use some of its advanced customization features.

Using the built-in ValidationPipe

To begin using it, we first install the required dependency.

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

> info **Hint** The `ValidationPipe` is exported from the `@nestjs/common` package.

Because this pipe uses the `class-validator` and `class-transformer` libraries, there are many options available. 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

`enableDebugMessages`	`boolean`	If set to true, validator will print extra warning messages to the console when something is not right.
`skipUndefinedProperties`	`boolean`	If set to true then validator will skip validation of all properties that are undefined in the validating object.
`skipNullProperties`	`boolean`	If set to true then validator will skip validation of all properties that are null in the validating object.
`skipMissingProperties`	`boolean`	If set to true then validator will skip validation of all properties that are null or undefined 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.
`errorHttpStatusCode`	`number`	This setting allows you to specify which exception type will be used in case of an error. By default it throws `BadRequestException`.
`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.
`always`	`boolean`	Set default for `always` option of decorators. Default can be overridden in decorator options

`strictGroups` `boolean` If `groups` is not given or is empty, ignore decorators with at least one group. `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`. `stopAtFirstError` `boolean` When set to true, validation of the given property will stop after encountering the first error. Defaults to false.

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

Auto-validation

We'll start by binding `ValidationPipe` at the application level, thus ensuring all endpoints are protected from receiving incorrect data.

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

To test our pipe, let's create a basic endpoint.

```
@Post()
create(@Body() createUserDto: CreateUserDto) {
  return 'This action adds a new user';
}
```

> info **Hint** Since TypeScript does not store metadata about **generics or interfaces**, when you use them in your DTOs, `ValidationPipe` may not be able to properly validate incoming data. For this reason, consider using concrete classes in your DTOs.

> info **Hint** When importing your DTOs, you can't use a type-only import as that would be erased at runtime, i.e. remember to `import {{ '{' }} CreateUserDto {{ '}' }}` instead of `import type {{ '{' }} CreateUserDto {{ '}' }}`.

Now we can add a few validation rules in our `CreateUserDto`. We do this using decorators provided by the `class-validator` package, described in detail here. In this fashion, any route that uses the `CreateUserDto` will automatically enforce these validation rules.

```
import { IsEmail, IsNotEmpty } from 'class-validator';

export class CreateUserDto {
  @IsEmail()
  email: string;

  @IsNotEmpty()
  password: string;
}
```

With these rules in place, if a request hits our endpoint with an invalid `email` property in the request body, the application will automatically respond with a `400 Bad Request` code, along with the following response body:

```
{
  "statusCode": 400,
  "error": "Bad Request",
  "message": ["email must be an email"]
}
```

In addition to validating request bodies, the `ValidationPipe` can be used with other request object properties as well. Imagine that we would like to accept `:id` in the endpoint path. To ensure that only numbers are accepted for this request parameter, we can use the following construct:

```
@Get(':id')
findOne(@Param() params: FindOneParams) {
  return 'This action returns a user';
}
```

`FindOneParams`, like a DTO, is simply a class that defines validation rules using `class-validator`. It would look like this:

```
import { IsNumberString } from 'class-validator';

export class FindOneParams {
  @IsNumberString()
  id: number;
}
```

Disable detailed errors

Error messages can be helpful to explain what was incorrect in a request. However, some production environments prefer to disable detailed errors. Do this by passing an options object to the `ValidationPipe`:

```
app.useGlobalPipes(
  new ValidationPipe({
    disableErrorMessages: true,
  }),
);
```

As a result, detailed error messages won't be displayed in the response body.

Stripping properties

Our `ValidationPipe` can also filter out properties that should not be received by the method handler. In this case, we can **whitelist** the acceptable properties, and any property not included in the whitelist is automatically stripped from the resulting object. For example, if our handler expects `email` and `password` properties, but a request also includes an `age` property, this property can be automatically removed from the resulting DTO. To enable such behavior, set `whitelist` to `true`.

```
app.useGlobalPipes(
  new ValidationPipe({
    whitelist: true,
  }),
);
```

When set to true, this will automatically remove non-whitelisted properties (those without any decorator in the validation class).

Alternatively, you can stop the request from processing when non-whitelisted properties are present, and return an error response to the user. To enable this, set the `forbidNonWhitelisted` option property to `true`, in combination with setting `whitelist` to `true`.

Transform payload objects

Payloads coming in over the network are plain JavaScript objects. The `ValidationPipe` can automatically transform payloads to be objects typed according to their DTO classes. To enable auto-transformation, set `transform` to `true`. This can be done at a method level:

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

To enable this behavior globally, set the option on a global pipe:

```
app.useGlobalPipes(
  new ValidationPipe({
    transform: true,
  }),
);
```

With the auto-transformation option enabled, the `ValidationPipe` will also perform conversion of primitive types. In the following example, the `findOne()` method takes one argument which represents an extracted `id` path parameter:

```
@Get(':id')
findOne(@Param('id') id: number) {
  console.log(typeof id === 'number'); // true
  return 'This action returns a user';
}
```

By default, every path parameter and query parameter comes over the network as a `string`. In the above example, we specified the `id` type as a `number` (in the method signature). Therefore, the

`ValidationPipe` will try to automatically convert a string identifier to a number.

Explicit conversion

In the above section, we showed how the `ValidationPipe` can implicitly transform query and path parameters based on the expected type. However, this feature requires having auto-transformation enabled.

Alternatively (with auto-transformation disabled), you can explicitly cast values using the `ParseIntPipe` or `ParseBoolPipe` (note that `ParseStringPipe` is not needed because, as mentioned earlier, every path parameter and query parameter comes over the network as a `string` by default).

```
@Get(':id')
findOne(
  @Param('id', ParseIntPipe) id: number,
  @Query('sort', ParseBoolPipe) sort: boolean,
) {
  console.log(typeof id === 'number'); // true
  console.log(typeof sort === 'boolean'); // true
  return 'This action returns a user';
}
```

> info **Hint** The `ParseIntPipe` and `ParseBoolPipe` are exported from the `@nestjs/common` package.

Mapped types

As you build out features like **CRUD** (Create/Read/Update/Delete) it's often useful to construct variants on a base entity type. Nest provides several utility functions that perform type transformations to make this task more convenient.

> **Warning** If your application uses the `@nestjs/swagger` package, see this chapter for more information about Mapped Types. Likewise, if you use the `@nestjs/graphql` package see this chapter. Both packages heavily rely on types and so they require a different import to be used. Therefore, if you used `@nestjs/mapped-types` (instead of an appropriate one, either `@nestjs/swagger` or `@nestjs/graphql` depending on the type of your app), you may face various, undocumented side-effects.

When building input validation types (also called DTOs), it's often useful to build **create** and **update** variations on the same type. For example, the **create** variant may require all fields, while the **update** variant may make all fields optional.

Nest provides the `PartialType()` utility function to make this task easier and minimize boilerplate.

The `PartialType()` function returns a type (class) with all the properties of the input type set to optional. For example, suppose we have a **create** type as follows:

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

By default, all of these fields are required. To create a type with the same fields, but with each one optional, use `PartialType()` passing the class reference (`CreateCatDto`) as an argument:

```
export class UpdateCatDto extends PartialType(CreateCatDto) {}
```

> info **Hint** The `PartialType()` function is imported from the `@nestjs/mapped-types` package.

The `PickType()` function constructs a new type (class) by picking a set of properties from an input type. For example, suppose we start with a type like:

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

We can pick a set of properties from this class using the `PickType()` utility function:

```
export class UpdateCatAgeDto extends PickType(CreateCatDto, ['age'] as
const) {}
```

> info **Hint** The `PickType()` function is imported from the `@nestjs/mapped-types` package.

The `OmitType()` function constructs a type by picking all properties from an input type and then removing a particular set of keys. For example, suppose we start with a type like:

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

We can generate a derived type that has every property **except** `name` as shown below. In this construct, the second argument to `OmitType` is an array of property names.

```
export class UpdateCatDto extends OmitType(CreateCatDto, ['name'] as
const) {}
```

> info **Hint** The `OmitType()` function is imported from the `@nestjs/mapped-types` package.

The `IntersectionType()` function combines two types into one new type (class). For example, suppose we start with two types like:

```
export class CreateCatDto {
  name: string;
  breed: string;
}

export class AdditionalCatInfo {
  color: string;
}
```

We can generate a new type that combines all properties in both types.

```
export class UpdateCatDto extends IntersectionType(
  CreateCatDto,
  AdditionalCatInfo,
) {}
```

> info **Hint** The `IntersectionType()` function is imported from the `@nestjs/mapped-types` package.

The type mapping utility functions are composable. For example, the following will produce a type (class) that has all of the properties of the `CreateCatDto` type except for `name`, and those properties will be set to optional:

```
export class UpdateCatDto extends PartialType(
  OmitType(CreateCatDto, ['name'] as const),
) {}
```

Parsing and validating arrays

TypeScript does not store metadata about generics or interfaces, so when you use them in your DTOs, `ValidationPipe` may not be able to properly validate incoming data. For instance, in the following code, `createUserDtos` won't be correctly validated:

```
@Post()
createBulk(@Body() createUserDtos: CreateUserDto[]) {
  return 'This action adds new users';
}
```

To validate the array, create a dedicated class which contains a property that wraps the array, or use the `ParseArrayPipe`.

```
@Post()
createBulk(
  @Body(new ParseArrayPipe({ items: CreateUserDto }))
  createUserDtos: CreateUserDto[],
) {
  return 'This action adds new users';
}
```

In addition, the `ParseArrayPipe` may come in handy when parsing query parameters. Let's consider a `findByIds()` method that returns users based on identifiers passed as query parameters.

```
@Get()
findByIds(
  @Query('ids', new ParseArrayPipe({ items: Number, separator: ',' }))
  ids: number[],
) {
  return 'This action returns users by ids';
}
```

This construction validates the incoming query parameters from an HTTP `GET` request like the following:

```
GET /?ids=1,2,3
```

WebSockets and Microservices

While this chapter shows examples using HTTP style applications (e.g., Express or Fastify), the `ValidationPipe` works the same for WebSockets and microservices, regardless of the transport method that is used.

Learn more

Read more about custom validators, error messages, and available decorators as provided by the `class-validator` package here.