Validation

발표 날짜	@2021년 9월 11일
발표 여부	☑️
발표자	호선우

```
export class CreateMovieDTO {
  readonly title: string;
  readonly year: number;
  readonly genres: string[];
}
```

```
import { Body, Controller, Delete, Get, Param, Patch, Post } from '@nestjs/common';
import { CreateMovieDTO } from './dto/create-movie.dto';
import { Movie } from './entities/movie.entity';
import { MoviesService } from './movies.service';

@Controller('movies')
export class MoviesController {
  constructor(private readonly moviesService: MoviesService) {}

  ...

  @Post()
  create(@Body() movieData: CreateMovieDTO) {
    return this.moviesService.create(movieData);
  }

  @Patch('/:id')
  patch(@Param('id') movieId: string, @Body() updateData: CreateMovieDTO) {
    console.log(updateData);
    console.log(movieId);
    return this.moviesService.update(movieId, updateData);
  }
}
```

🤔언제 사용할까요?

```
if (!email) {
  throw new BadRequestException('이메일 없네요');
}
if (!nickname) {
  throw new BadRequestException('닉네임 없네요');
}
if (!password) {
```

```
  throw new BadRequestException('비밀번호 없네요');
}
```

💡request를 자동으로 검증하기 위해 Nest에서 제공

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

export class JoinRequestDto {
  @IsNotEmpty()
  email: string;

  @IsNotEmpty()
  nickname: string;

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

```
{
  "statusCode": 400,
  "error": "Bad Request",
  "message": ["email should not be empty",
              "nickname should not be empty",
              "password should not be empty"
  ]
}
```

Validation

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

Auto-validation

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

```
// src/users/users.controller.ts
@Post()
  async join(@Body() data: JoinRequestDto) {
    await this.usersService.join(data.email, data.nickname, data.password);
  }
```

```
// src/users/dto/join.request.dto.ts
import { IsNotEmpty } from 'class-validator';

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

  @IsNotEmpty()
  nickname: string;

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

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

▼ Validation decorators

Decorator	Description
Common validation decorators	제목 없음
`@IsDefined(value: any)`	Checks if value is defined (!== undefined, !== null). This is the only decorator that ignores skipMissingProperties option.
`@IsOptional()`	Checks if given value is empty (=== null, === undefined) and if so, ignores all the validators on the property.
`@Equals(comparison: any)`	Checks if value equals ("===") comparison.
`@NotEquals(comparison: any)`	Checks if value not equal ("!==") comparison.
`@IsEmpty()`	Checks if given value is empty (=== '', === null, === undefined).
`@IsNotEmpty()`	Checks if given value is not empty (!== '', !== null, !== undefined).
`@IsIn(values: any[])`	Checks if value is in a array of allowed values.

Decorator	Description
`@IsNotIn(values: any[])`	Checks if value is not in a array of disallowed values.
Type validation decorators	제목 없음
`@IsBoolean()`	Checks if a value is a boolean.
`@IsDate()`	Checks if the value is a date.
`@IsString()`	Checks if the string is a string.
`@IsNumber(options: IsNumberOptions)`	Checks if the value is a number.
`@IsInt()`	Checks if the value is an integer number.
`@IsArray()`	Checks if the value is an array
`@IsEnum(entity: object)`	Checks if the value is an valid enum
Number validation decorators	제목 없음
`@IsDivisibleBy(num: number)`	Checks if the value is a number that's divisible by another.
`@IsPositive()`	Checks if the value is a positive number greater than zero.
`@IsNegative()`	Checks if the value is a negative number smaller than zero.
`@Min(min: number)`	Checks if the given number is greater than or equal to given number.
`@Max(max: number)`	Checks if the given number is less than or equal to given number.
Date validation decorators	제목 없음
`@MinDate(date: Date)`	Checks if the value is a date that's after the specified date.
`@MaxDate(date: Date)`	Checks if the value is a date that's before the specified date.

`@Body` 외에도 다른 요청 객체 속성과 함께 사용

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

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

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

Custom decorator와 함께 사용

```
@Get()
async findOne(
  @User(new ValidationPipe({ validateCustomDecorators: true }))
  user: UserEntity,
) {
  console.log(user);
}
```

`class-validator` 기본 제공 옵션

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

▼ 추가적인 Validation 제공 옵션

```
import { ValidationPipe } from '@nestjs/common';
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true, // decorator(@)가 없는 속성이 들어오면 해당 속성은 제거하고 받아들입니다.
      forbidNonWhitelisted: true, // DTO에 정의되지 않은 값이 넘어오면 request 자체를 막습니다.
      transform: true, // 클라이언트에서 값을 받자마자 타입을 정의한대로 자동 형변환을 합니다.
      disableErrorMessages: true, // 오류 메시지를 응답에 표시하지 않습니다.
    }),
  );
  await app.listen(5000);
}
bootstrap();
```

Option	Type	Description	Aa 제목

Option	Type	Description	Aa 제목
`skipMissingProperties`	`boolean`	true로 설정하면 유효성 검사기가 객체 유효성 검사에 누락된 모든 속성의 유효성 검사를 건너뜁니다.	제목 없음
`whitelist`	`boolean`	true로 설정되면 유효성 검사기는 유효성 검사 데코레이터를 사용하지 않는 속성의 유효성 검사(반환 된) 객체를 제거합니다.	제목 없음
`forbidNonWhitelisted`	`boolean`	true로 설정하면 화이트리스트에 없는 속성 검사기를 제거하는 대신 예외가 발생합니다.	제목 없음
`forbidUnknownValues`	`boolean`	true로 설정하면 알 수 없는 개체의 유효성을 검사하려는 시도가 즉시 실패합니다.	제목 없음
`disableErrorMessages`	`boolean`	true로 설정하면 유효성 검사 오류가 클라이언트에 반환되지 않습니다.	제목 없음
`errorHttpStatusCode`	`number`	이 설정을 사용하면 오류 발생시 사용할 예외 유형을 지정할 수 있습니다. 기본적으로 `BadRequestException`이 발생합니다.	제목 없음
`exceptionFactory`	`Function`	유효성 검사 오류의 배열을 가져오고 throw할 예외 개체를 반환합니다.	제목 없음
`groups`	`string[]`	개체의 유효성을 검사하는 동안 사용할 그룹입니다.	제목 없음
`dismissDefaultMessages`	`boolean`	true로 설정하면 유효성 검사에서 기본 메시지를 사용하지 않습니다. 오류 메시지는 항상 `undefined`입니다. 명시 적으로 설정되지 않았습니다.	제목 없음

Option	Type	Description	Aa 제목
`validationError.target`	`boolean`	`ValidationError`에서 대상을 노출해야 하는지 여부를 나타냅니다.	제목 없음
`validationError.value`	`boolean`	검증된 값이 `ValidationError`에 노출되어야 하는지 여부를 나타냅니다.	제목 없음

📢 DTO를 가져올 때 런타임시 삭제되므로 타입 전용 가져오기를 사용할 수 없습니다. 즉, import type { CreateUserDto } 대신 import { CreateUserDto }를 기억하세요.

📢 TypeScript는 제네릭 또는 인터페이스에 대한 메타데이터를 저장하지 않으므로 DTO에서 사용할 때 ValidationPipe가 들어오는 데이터의 유효성을 제대로 검사하지 못할 수 있습니다. 이러한 이유로 DTO에서 구체적인 클래스를 사용하는 것이 좋습니다.

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

`ParseArrayPipe`

- 배열의 유효성 검사

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

- 쿼리 매개변수를 구문 분석

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

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

Transform payload objects

암시적 변환

- 메서드 수준에서 수행

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

- 전역적으로 활성화

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

명시적 변환

```
@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';
}
```

Mapped types

- CRUD(만들기/읽기/업데이트/삭제)와 같은 기능을 구축할 때 기본 항목 유형에 대한 변형을 구성하는 것이 종종 유용
- Nest는 이 작업을 보다 편리하게 만들기 위해 유형 변환을 수행하는 여러 유틸리티 함수를 제공

```
export class CreateMovieDTO {
  @IsString()
  readonly title: string;

  @IsNumber()
  readonly year: number;

  @IsString({ each: true })
  readonly genres: string[];
}

// property에 ?만 붙임
export class UpdateMovieDTO {
  @IsString()
  readonly title?: string;

  @IsNumber()
  readonly year?: number;

  @IsString({ each: true })
  readonly genres?: string[];
}
```

`PartialType()`

- 입력 유형의 모든 속성이 선택 사항으로 설정된 유형 (클래스)을 반환

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

  @IsString()
  password: string;

  @IsString()
  nickname: string;

  @IsOnlyDate()
  birthday: string;
}
```

```
export class UpdateUserDto extends PartialType(CreateUserDto) {}
```

PickType()

- 입력 유형에서 속성 집합을 선택하여 새로운 유형(클래스)을 생성

```
@Entity({ schema: 'testDB', name: 'users' })
export class Users{

  @PrimaryGeneratedColumn({ type: 'int', name: 'id' })
  id: number;

  @Column('varchar', { name: 'email', unique: true, length: 30 })
  email: string;

  @IsString()
  @Column('varchar', { name: 'nickname', length: 30 })
  nickname: string;

  @IsString()
  @Column('varchar', { name: 'password', length: 100, select: false })
  password: string;

  @IsOnlyDate()
  @Column('varchar', { name: 'birthday'})
  birthday: string;
}
```

```
export class JoinRequestDto extends PickType(Users, [
  'email',
  'nickname',
  'password',
] as const) {}
```

OmitType()

- 입력 타입에서 모든 속성을 선택한 다음 특정 키 세트를 제거하여 타입을 구성

```
export class JoinRequestDto extends OmitType(Users, ['email'] as const) {}
```

IntersectionType()

- 두 타입을 하나의 새로운 타입 (클래스)으로 결합

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

```
  breed: string;
}

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

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

💡이런 방식으로도 구성할 수 있어요

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

`class-validator` 패키지에 대한 자세한 내용은 여기를 참조하세요

https://github.com/typestack/class-validator