Formatting nestjs validation errors the right way

June 23, 2023 6 min read

nestjs backend

In this blog post we will learn how to format NestJS validation error messages into key-value pairs for improved usability in client applications.

Introduction

When working with validation in nestjs with `class-validators` and `class-transformers`, the error messages thrown in a single array which isn't much helpful for the client applications. In this blog post, we will look into formatting the error messages from this

```
{
    "statusCode": 400,
    "message": [
        "name should not be empty",
        "name must be a string",
        "email should not be empty",
        "email must be an email",
        "password too weak",
        "password must be longer than or equal to 8 characters",
        "password must be a string"
    ],
    "error": "Bad Request"
}
```

to this

```
 {
  "name": [
    'name should not be empty',
    'name must be a string'
 ],
  "email": ['email should not be empty', 'email must be an email'],
  "password": [
    'password too weak',
    'password must be longer than or equal to 8 characters',
    'password must be a string',
  ],
};
```

The Setup

Let's create a new nestjs project and install necessary dependencies

```
nest new nestjs-validation
cd nestjs-validation
new g resource users

? What transport layer do you use?
❯ REST API
  GraphQL (code first)
  GraphQL (schema first)
  Microservice (non-HTTP)
  WebSockets
? Would you like to generate CRUD entry points? (Y/n) y
```

With these commands, we have created a new nestjs project, and created a users module with dummy CRUD operations. Your `app.modules.ts` should look something like this

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { UsersModule } from './users/users.module';

@Module({
  imports: [UsersModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Let's now install packages to start validating the requests

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

To start using the validation pipe, let's modify `main.ts`

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

Now let's create a `CreateUserDto` that we will use to validate post request to the route `/users`

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

export class CreateUserDto {
  @IsString()
  @IsNotEmpty()
  name: string;

  @IsEmail()
  @IsNotEmpty()
  email: string;

  @IsString()
  @MinLength(8)
  @Matches(/^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[^\da-zA-Z]).{8,}$/, {
    message: 'password too weak',
  })
  password: string;
}
```

We've now added validations for each field. We will now use this dto in the post route

```
  // users.controller.ts
  ...

  @Post()
  create(@Body() createUserDto: CreateUserDto) {
    return this.usersService.create(createUserDto);
  }
 ...
```

At this point if you make a post request to the route `/users` you will get errors in the following format.

```
{
    "statusCode": 400,
    "message": [
        "name should not be empty",
        "name must be a string",
        "email should not be empty",
        "email must be an email",
        "password too weak",
        "password must be longer than or equal to 8 characters",
        "password must be a string"
    ],
    "error": "Bad Request"
}
```

We are validating the request but we can't really associate the error messages to it's respective field. It would be really helpful if we get the error messages with it's respective key.

To achieve that, let's create an exception filter that we can use in our validation pipe. For now to see the structure of the errors we will just log the errors

```
// shared/exceptions/validation.exception.ts
export const validationExceptionFactory = (errors: ValidationError[])
  console.log(errors);
};

//main.ts

app.useGlobalPipes(
  new ValidationPipe({
    exceptionFactory: validationExceptionFactory,
  }),
);
```

If you see the console, you will see the structure of the validation errors.

```
[
  ValidationError {
    target: CreateUserDto {},
    value: undefined,
    property: 'name',
    children: [],
    constraints: {
      isNotEmpty: 'name should not be empty',
      isString: 'name must be a string'
    }
  },
  ...
]
```

Now we know that the `property` will have the key of the field and the error messages will come in the `constraints` so let's use this knowledge and update our exception factory to structure our error messages

```
// shared/exceptions/validation.exception.ts
export const validationExceptionFactory = (errors: ValidationError[])
  const errMsg = {};
  errors.forEach((error: ValidationError) => {
    errMsg[error.property] = [...Object.values(error.constraints)];
  });
  return new ValidationException(errMsg);
};

// shared/exceptions/validation.exception.ts
export class ValidationException extends BadRequestException {
  constructor(public validationErrors: Record<string, unknown>) {
    super(validationErrors);
  }
}
```

Now if you send a post request to the `/users` route, the error messages will be structured in key value pair

```
 {
  "name": [
    'name should not be empty',
    'name must be a string'
 ],
  "email": ['email should not be empty', 'email must be an email'],
  "password": [
    'password too weak',
    'password must be longer than or equal to 8 characters',
    'password must be a string',
  ],
};
```

This is super helpful. Now the client applications can display error messages with their respective fields. But we're not done yet. Our exception factory works for simple fields like but when we pass nested values it will not be able to handle the it. For the sake of the demo, let's say we wan't to create multiple posts along with the user in a single request.

```
import { Type } from 'class-transformer';
import {
  IsEmail,
  IsNotEmpty,
  IsString,
  Matches,
  MinLength,
  ValidateNested,
} from 'class-validator';

export class PostDto {
  @IsString()
  @IsNotEmpty()
  title: string;

  @IsString()
  @IsNotEmpty()
  content: string;
}

export class CreateUserDto {
  ...

  @ValidateNested({ each: true })
  @IsNotEmpty()
  @Type(() => PostDto)
  posts: PostDto[];
}
```

Now try sending a post request to `/users`

```
{
  "posts": [
    {
      "title": '',
    },
  ],
}
```

You will most likely get a 500 error. When you log the validation errors, error with property `posts` will have `children` which will be array of `ValidationError`. Let's re-write our exception factory to handle this.

```
export const validationExceptionFactory = (errors: ValidationError[]) =
  const formatError = (errors: ValidationError[]) => {
    const errMsg = {};
    errors.forEach((error: ValidationError) => {
      errMsg[error.property] = error.children.length
        ? [formatError(error.children)]
        : [...Object.values(error.constraints)];
    });
    return errMsg;
  };
  return new ValidationException(formatError(errors));
};
```

Here we're going through each error and recursively formatting error messages if there's a nested field. Now if you send a post request again, you should see well structured error messages.

```
{
  "name": [
    "name should not be empty",
    "name must be a string"
  ],
  "email": [
    "email should not be empty",
    "email must be an email"
  ],
  "password": [
    "password too weak",
    "password must be longer than or equal to 8 characters",
    "password must be a string"
  ],
  "posts": [
    {
      "0": [
        {
          "title": [
            "title should not be empty"
          ],
          "content": [
            "content should not be empty",
            "content must be a string"
          ]
        }
      ]
    }
  ]
}
```