Back End

Carrera Programador full-stack

MODULES & VALIDATION PIPES

Modules Best Practices

¿Qué es un módulo?

Una pieza fundamental para la organización del código y lograr una arquitectura más avanzada.

Un módulo es una clase de POO que funciona como contenedor de otras clases o artefactos, como son los controladores, servicios y otros componentes desarrollados con Nest.

Los módulos sirven para agrupar elementos, de modo que una aplicación podrá tener varios módulos con clases altamente relacionadas entre sí.

***

Hasta el momento hemos trabajado con un único módulo en la aplicación: el módulo principal instalado en el *boilerplate* inicial. A este módulo también le llamamos **módulo raíz** y siempre existirá en las aplicaciones. De él podemos hacer que dependan todos nuestros artefactos de software (*controllers*, *providers*, etc.).

Sin embargo, es **ideal** que construyamos las **aplicaciones** usando **diversos módulos** que nos permitan **organizar los componentes** de manera limpia y clara.

Módulos: dominio de la aplicación

Los **módulos** que implementamos generalmente surgen del **dominio de la aplicación**.

El modelo del dominio consiste en la definición de todos los elementos con los que una aplicación necesita trabajar y sus relaciones. Por ejemplo, en una aplicación de facturación tendríamos elementos como clientes, facturas, productos, etc. Un cliente puede tener diversas facturas, una factura puede tener diversos productos, y así.

Definir correctamente el modelo de dominio es la primera tarea que normalmente nos planteamos para desarrollar una aplicación.

Nosotros tenemos una carpeta *src/track* que contiene proveedor, controlador e interfaz asociados a esa parte de la lógica del negocio… pero siguen dependiendo del módulo raíz. Vamos resolver eso. Tendremos que quitar las referencias en el módulo raíz y generar un módulo *track.module.ts.*

```
> nest generate module track
                                  O
> nest g mo track
```

El comando anterior crea el módulo *track/track.module.ts*. Está prácticamente vacío. Vamos a completarlo declarando el controlador y el servicio inherentes al módulo.

```
import { Module } from '@nestjs/common';
import { TrackController } from './track.controller';
import { TrackService } from './track.service';
@Module({
  controllers: [TrackController],
  providers: [TrackService],
})
export class TrackModule {}
```

A continuación, podemos eliminar del módulo principal la importación del controlador y el servicio, que ahora ya dependen del módulo apropiado.

```
import { ServeStaticModule } from '@nestjs/serve-static';
import { join } from 'path';
import { Module } from '@nestjs/common';
import { TrackModule } from './track/track.module';
@Module({
  imports: [
    ServeStaticModule.forRoot({ rootPath: join(__dirname, '..', 'client') }),
    TrackModule,
  ],
})
export class AppModule {}
```

DEJEMOS ALGO EN CLARO:

Lo que acabamos de aprender y poner en práctica **no cambia el funcionamiento** de nuestra aplicación pero **sí su arquitectura**.

En bases de código medianas o grandes, esto no solamente es recomendable sino necesario.

Nuestra aplicación es **más fácil de mantener**.

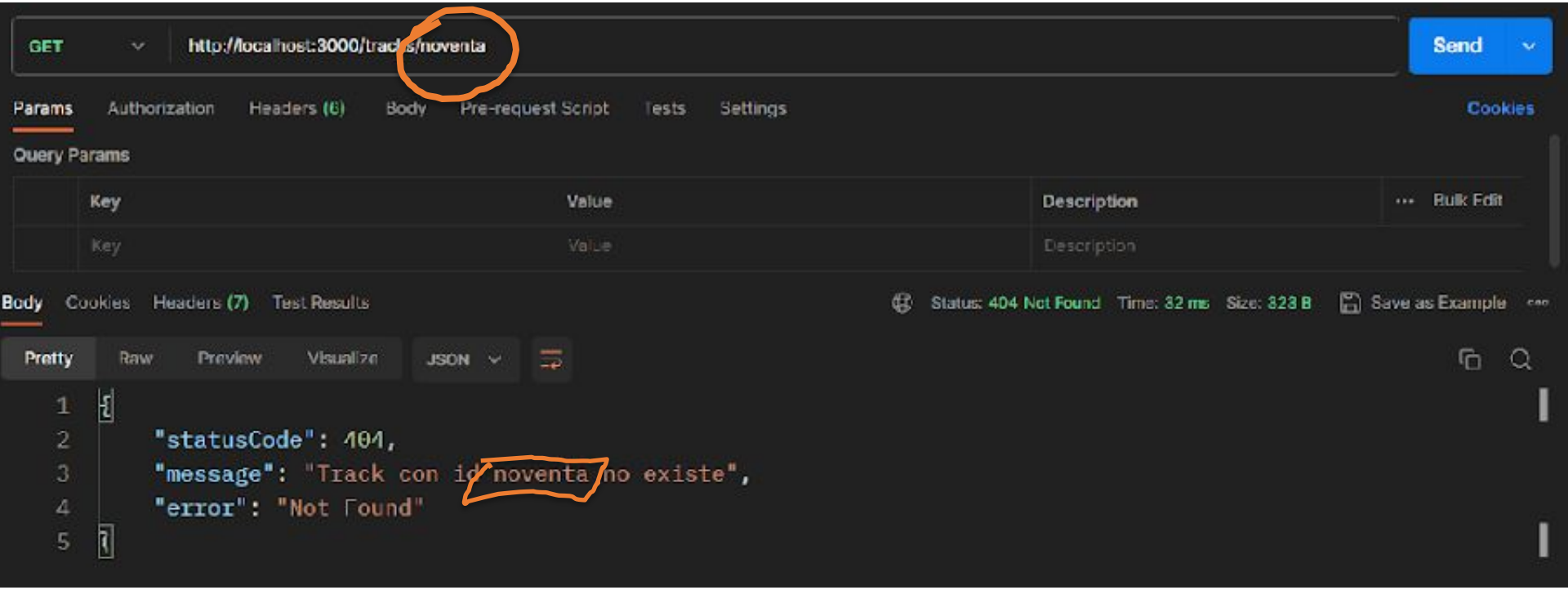
¿Qué ocurre cuando no tenemos la aplicación bien modularizada y surge un bug?

A veces es más difícil rastrear el problema que solucionarlo.

Validaciones en Nest JS

Hasta aquí, si a nuestra API le pasamos en la ruta *tracks* un **id** que no corresponde con un objeto en nuestra base de datos, nos retorna un *status* **404 Not Found**. Eso es correcto. Pero si pasamos un **id** que ni siquiera concuerda con el tipo de dato esperado (un número entero), nos devuelve exactamente lo mismo.

Eso no está mal… pero podría estar mucho mejor.



Validaciones en Nest JS

Para resolver ese problema podríamos validar el **id** que ingresa por **URL Param.** Debe ser un número entero. Eso le agrega lógica extra a nuestro código.

Afortunadamente, Nest nos ofrece una nutrida cantidad de ayudas para que las validaciones y transformaciones de la entrada de datos en los controladores se pueda realizar de manera sencilla, sin prácticamente necesidad de escribir código específico.

Para ello ofrece una serie de **pipes** para validaciones de datos simples y la posibilidad de usar **ValidationPipe** para validar datos mas complejos.

SUS USOS SON BÁSICAMENTE LOS SIGUIENTES:

- **Transformaciones**: para convertir un dato de un tipo en otro, por ejemplo un string en un número entero.
- **Validaciones**: para comprobar si un elemento es lo que se espera, o levantar una excepción en caso contrario.

Validaciones en Nest JS

Para servirnos de un **pipe** tenemos que utilizar decoradores en las firmas de los métodos de controlador que se ejecutan para implementar las rutas de aplicación.

Los **pipes se ejecutan antes del propio método del controlador**, pudiendo trabajar con sus argumentos para validarlos o realizar cualquier transformación. Una vez ejecutado el proceso del pipe pueden pasar dos cosas:

A- Si todo ha ido bien, se ejecutará el propio método del controlador. En caso que el pipe implemente alguna transformación, el dato estará por supuesto transformado.

B- Si el pipe encontró algún problema al realizar la transformación o la validación, entonces levantará una excepción. Como los pipes funcionan en coordinación con el tratamiento de excepciones, el propio framework realizará el envío de los errores a los clientes, sin necesidad de intervenir mediante nuestro propio código.

Validaciones en Nest JS

Existen diversos pipes para realizar operativas habituales en las aplicaciones. Uno de ellos es **ValidationPipe**, que permite realizar todas las validaciones necesarias sobre los datos de entrada de las aplicaciones, lo que resulta muy cómodo para evitar la mayoría del código necesario cuando se desea comprobar que los datos son correctos, antes de operar con ellos.

También hay otros pipes como **ParseIntPipe** o **ParseBoolPipe**, más elementales pero también extremadamente útiles, que realizan las transformaciones adecuadas para proporcionar datos en el tipo que necesitamos dentro de los métodos de los controladores. Si no pueden realizar esas transformaciones, entonces levantarán igualmente excepciones.

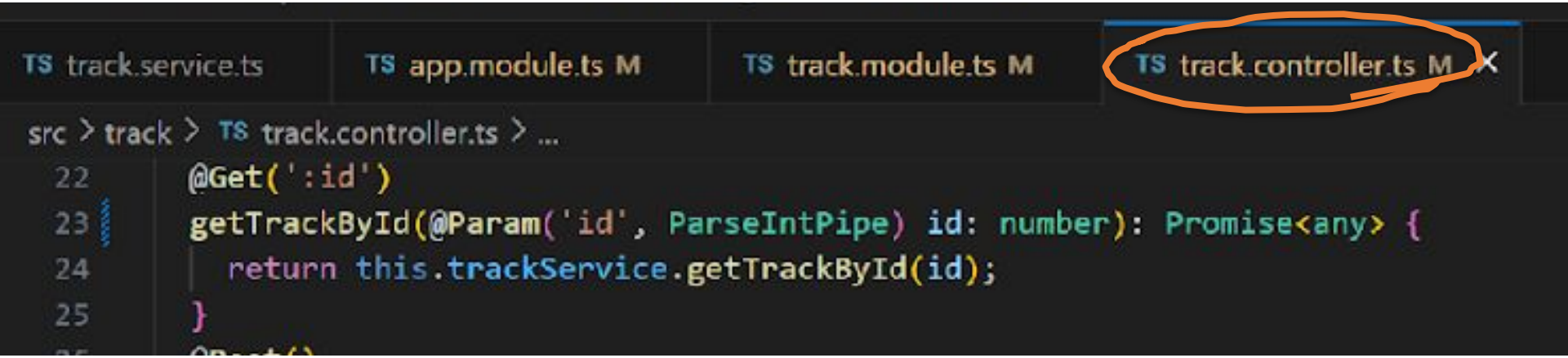
Documentación sobre pipes: https://docs.nestjs.com/pipes

ParseIntPipe

Este pipe sirve para transformar un dato a su valor *integer*. Es por tanto un pipe de **transformación**, sin embargo, en el fondo también realiza una **validación**, puesto que si no es un valor transformable en un entero levantará una excepción y la capa de manejo de excepciones de Nest devolverá el error al cliente.

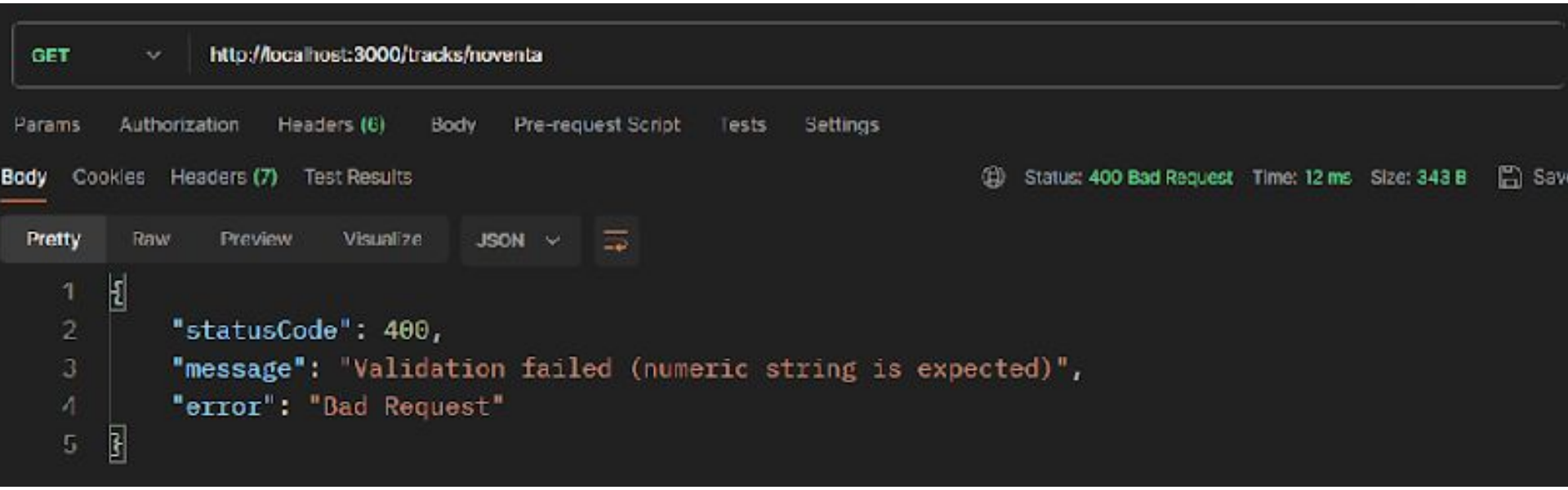
Notará que esto es ideal para nuestra ruta **/tracks/:id** que ejecuta el servicio **getTrackById().**

Modificamos el controlador ***src/track/track.controller.ts,*** *en el que también importaremos* ***ParseIntPipe*** *(sí, siempre importamos las piezas de código que queremos usar ☺)*



ParseIntPipe

Y no necesitamos hacer nada más. La capa de validación y transformación trabajará con la capa de manejo de errores de manera totalmente transparente.



ParseIntPipe

Estamos delegando en Nest la responsabilidad de crear una instancia de **ParseIntPipe** para realizar la transformación.

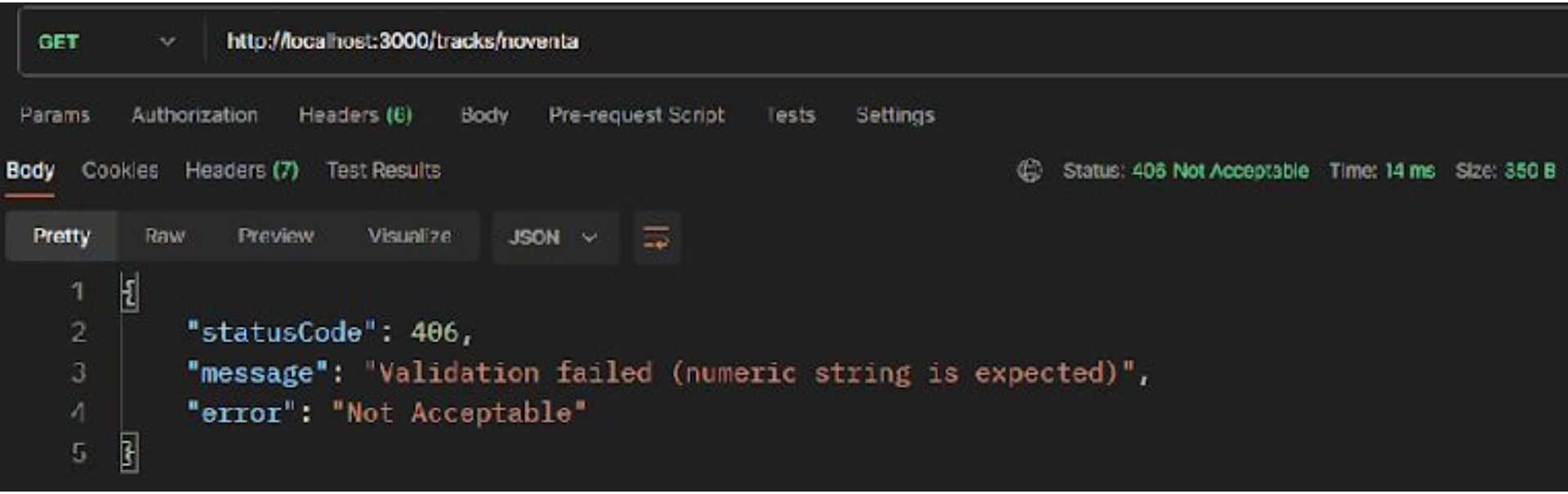
Sin embargo, podemos realizar explícitamente esa instanciación indicando cierta configuración.

Supongamos que queremos levantar una excepción con un **status code** personalizado. Esto es posible enviando el código en un objeto de configuración que recibe el constructor de ParseIntPipe.

En el siguiente código puedes ver cómo enviaríamos un *status code* **406 Not Acceptable**, en lugar del *status code* **400 Bad Request** que tenemos por defecto.

CFS

```
getTrackById(
    @Param(
      'id',
      new ParseIntPipe({ errorHttpStatusCode:
HttpStatus.NOT_ACCEPTABLE }),
    )
    id: number,
  ): Promise<any> {
    return this.trackService.getTrackById(id);
  }
```



NEXT STOP

ValidationPipes + DTO

A continuación veremos un tipo de **pipe** muy útil: el **ValidationPipe**.

Esas validaciones las haremos muy fácilmente gracias a un tipo de clases nuevo: Los **D.T.O.**