Back End

Carrera Programador full-stack

DTO & VALIDATION PIPE

Data Transfer Object

¿Qué es un D.T.O.?

Un **objeto** que se transfiere por la **red** entre dos sistemas, típicamente usados en aplicaciones **cliente/servidor** y en las aplicaciones web modernas.

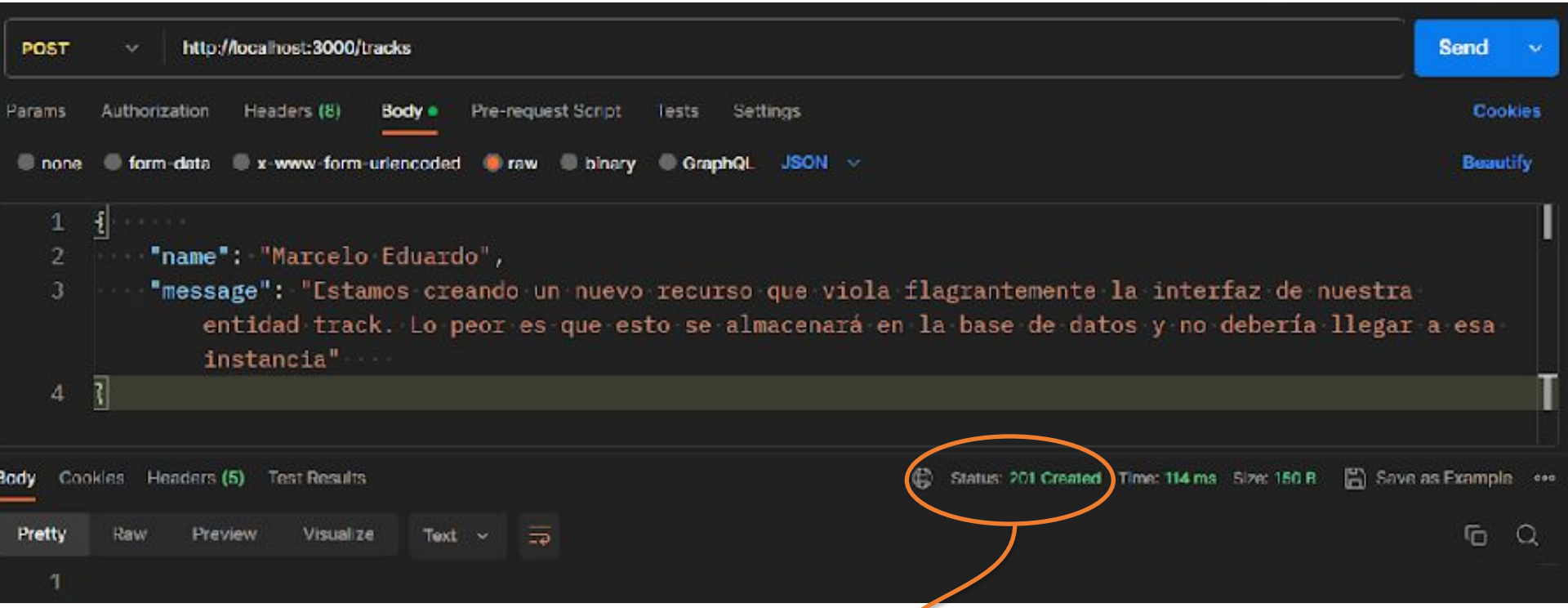
El **DTO** sería el **objeto JSON** que se transfiere desde el cliente al servidor (o viceversa).

¿Por qué usar un D.T.O.?

Para crear aplicaciones robustas es útil tipar los datos que se van a enviar y recibir desde el frontend hacia el backend, especificando qué propiedades tendrán los objetos DTO y de qué tipos.

Hasta aquí estamos usando una interfaz y eso está muy bien en tiempo de desarrollo, pero en tiempo de ejecución esta no evita que nos envíen datos inesperados.

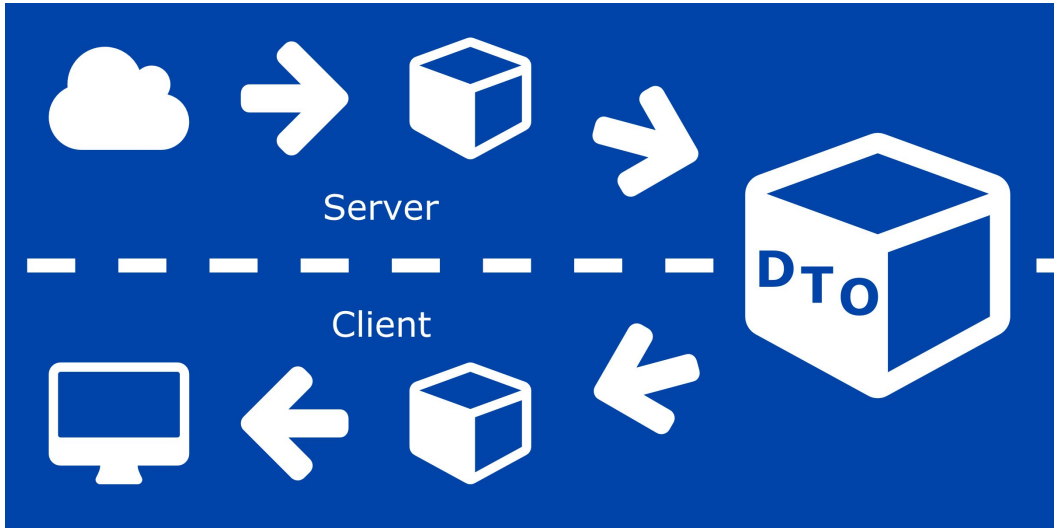
Data Transfer Object



Esto no debería ocurrir.
Vamos a resolverlo con un DTO

PRIMER PASO:
vamos a crear un archivo
src/track/track.dto.ts

```
export class TrackDto {
  title: string;
  duration: number;
  artist: string;
}
```



Como ven, se trata de una clase corriente, en la que solamente estamos especificando propiedades y tipos.

En el **DTO** de creación de un elemento no es necesario especificar el id, puesto que se generará en el momento de crearlo.

Cómo usar el DTO

En el controlador vamos a usar el **DTO** para que, cuando nos piden hacer una inserción, podamos decirle que el dato que se recibirá en el **body** de la **request** será del tipo del **DTO** que acabamos de crear.

Así quedará nuestro nuestro método **@Post**:

```
@Post()
createTrack(@Body() trackDto: TrackDto): Promise<any> {
  return this.trackService.createTrack(trackDto);
}
```

¿Qué pasó aquí?

Con este cambio **no se ha producido ningún comportamiento** real pues aún no estamos haciendo validaciones.

De momento simplemente hemos creado una nueva clase con propiedades y tipos que nos ayudarán a mantener el tipado de los datos con los que vamos a trabajar, por lo que será en el fondo algo parecido a lo que habíamos conseguido por medio de una interfaz.

Lo **importante** es que hemos establecido las **bases** sobre el concepto de **DTO**, que vamos a usar mucho en el futuro.

ValidationPipe

ValidationPipe funciona como otros *pipes* de Nest, estableciendo una serie de comprobaciones que se realizarán antes que se ejecute el cuerpo de un método de un controlador.

El método donde usaremos **ValidationPipe** implementa la ruta de una solicitud de Nest y solamente se ejecutará si todas las validaciones han sido correctas.

Además, Nest se encargará de generar los mensajes de error para los clientes, enviando los correspondientes *status code* en las respuestas y los mensajes de error necesarios.

Todo este funcionamiento ya fue descrito anteriormente, cuando hablamos de los *pipes* en general, así que vamos a concentrarnos en **ValidationPipe**.

Configurar un *pipe* global

Existen diversas maneras de trabajar con **ValidationPipe** y quizás la más práctica es establecer un *pipe* global, de modo que todas las rutas puedan beneficiarse del flujo de validaciones de manera automática.

Para ello, al iniciar la aplicación, en la función **bootstrap()** del archivo **main.ts**, podemos configurar los *pipes* globales con un método del objeto *app* llamado **useGlobalPipes()**, pasándole por parámetro el *pipe* que queremos configurar.

Tenemos que usar un código como este en el **main.ts**:

```
import { ValidationPipe } from '@nestjs/common';

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

Instalar reglas de validación

Las reglas de validación sirven para asegurar que algo sea un entero, una cadena, un booleano o para hacer comprobaciones más complejas.

La clase **ValidationPipe** está incluida en el *framework*, pero las reglas de validación deben ser instaladas:

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

Configurar reglas en el DTO

Ahora, en el archivo del **DTO** que habíamos creado, llamado **track.dto.ts**, vamos a aplicar las reglas de validación para cada campo con anotaciones. Todas las anotaciones vienen en el *package* "**class-validator**", que acabamos de instalar y tiene una lista enorme de decoradores.

El código de nuestra clase **DTO** quedará más o menos así:

```
import { IsInt, IsString } from 'class-validator';
export class TrackDto {
  @IsString()
  title: string;

  @IsInt()
  duration: number;

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

Configurar reglas en el DTO

Ahora el *framework* se encargará de hacer automáticamente las validaciones y enviar correctamente los mensajes de error si las reglas definidas con los decoradores no se han cumplido.

Por supuesto, tenemos que asegurarnos de usar la clase **DTO** en el método donde queramos que las validaciones se ejecuten.

Simplemente con tipar el parámetro con la clase **DTO** se producirán dichas comprobaciones.

```
@Post()
 createTrack(@Body() trackDto: TrackDto): Promise<any> {
   return this.trackService.createTrack(trackDto);
 }
```

Ahora, si intentamos crear un recurso que no cumple con el tipado del DTO…