# 02 NEST DTOS

- Un **Dto** (*Data Transfer Object*) es una clase que luce de cierta manera.
- Nos va a servir para pasar la data del controlador al servicio o dónde sea
- Es como una interface, pero literalmente es una clase porque nos ayuda a expandirla y añadir funcionalidad, cosa que una interfaz no puede hacer (a una interfaz no se le pueden agregar métodos y lógica). Las interfaces no crean instancias
- Es una clase que tiene ciertas propiedades y yo determino cómo quiero que luzcan estas propiedades
- Con el Dto voy a poder añadir lógica para que la data luzca siempre cómo yo quiera

---

## Interfaces y UUID

- Voy a crear una interfaz para que la data luzca de cierta manera.
- **Va a terminar siendo una clase**, pero por ahora lo hago con una interfaz
- cars/interfaces/car.interface.ts

```
export interface Car{
    id: number
    brand: string
    model: string
}
```

- Implemento la interfaz en cars (en el servicio)

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { Car } from './interfaces/car.interface';

@Injectable()
export class CarsService {

    private cars: Car[] =[
        {
            id:1,
            brand: 'Toyota',
            model: 'Corola'
        },
        {
            id:2,
            brand: 'Suzuki',
            model: 'Vitara'
        },
        {
            id:3,
            brand: 'Honda',
            model: 'Civic'
        }
```

```
    ]

    findAll(){
        return this.cars
    }

    findOneById(id: number){
        const car = this.cars.find(car => car.id === id)
        if(!car) throw new NotFoundException(`Car with id ${id} not found`)
        return car
    }
}
```

- Prefiero trabajar con **UUIDs** para los ids en lugar de usar correlativos (1,2,3...)
- Instalo el paquete y sus tipos

> npm i uuid npm i -D @types/uuid

- *UUID* trabaja con strings, por lo que debo cambiarlo en la interfaz

```
export interface Car{
    id: string
    brand: string
    model: string
}
```

- Uso el paquete en el controlador, **la versión 4**
- Cambio el tipo del id en el método por string

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { Car } from './interfaces/car.interface';
import {v4 as uuid} from 'uuid';

@Injectable()
export class CarsService {

    private cars: Car[] =[
        {
            id: uuid(),
            brand: 'Toyota',
            model: 'Corola'
        },
        {
            id: uuid(),
            brand: 'Suzuki',
            model: 'Vitara'
        },
        {
            id: uuid(),
```

```
            brand: 'Honda',
            model: 'Civic'
        }
    ]

    findAll(){
        return this.cars
    }
            //cambio el tipo a string
    findOneById(id: string){
        const car = this.cars.find(car => car.id === id)
        if(!car) throw new NotFoundException(`Car with id ${id} not found`)
        return car
    }
}
```

- Si hago un *query* a la DB (cuando la DB está grabando el id como un UUID) y no es un *UUID* va a dar un error de DB
- Eso es un *error 500* (**Internal Server Error**)
- Uso el **ParseUUIDPipe** en el controlador. Cambio el tipo a string
- Debo verificar el id antes de hacer la petición (no quiero dejarle ese trabajo a la DB)

```
@Get(':id')
getCarById(@Param('id', ParseUUIDPipe ) id: string){
        return this.carsService.findOneById(id)
}
```

- Puedo crear una nueva instancia de *ParseUUIDPipe* para que trabaje con una versión específica de *UUID*
- Dentro de los pipes también tengo la opción de personalizar mensajes de error

```
@Get(':id')
getCarById(@Param('id', new ParseUUIDPipe({version: '4'}) ) id: string){
        return this.carsService.findOneById(id)
}
```

---

## Dto: Data Transfer Object

- Va a ser una clase que me va a ayudar a decirle a mi controlador que estoy esperando una clase de cierto aspecto, y al pasárselo a mi servicio, mi servicio sabe que esa clase luce de cierta manera
- Se aconseja que los dto sean *readonly*, porque cuando se crea su instancia no cambian las propiedades.
    - Yo no quiero reasignar el valor de un dto porque puede ser un error
- Creo en /cars/dtos/create-car.dto.ts

```
export class CreateCarDto{
   readonly brand: string
   readonly  model: string
}
```

- Entonces, el body que llega del método POST va aser de tipo *CreateCarDto*
- Lo puedo renombrar a *createCarDto*. Todavía no es una instancia, pero es un objeto que espero que luzca como el *CreateCarDto*

```
@Post()
createCar(@Body() createCarDto: CreateCarDto){
        return createCarDto
}
```

- Todavía tengo que decirle a Nest que aplique las validaciones de los dto

## ValidationPipe - Class validator y Transformer

- Nest proporciona **ValidationPipe** que trabaja con librerías externas como **class-validator** y **class-transformer**
- Algunos decoradores de class-validator:
    - IsOptional, IsPositive, IsMongoId, IsArray,IsString, IsUUID, IsDecimal, IsBoolean, IsEmail, IsDate, IsUrl....
- Podemos aplicar pipes a nivel de parámetro, cómo se ha visto, a nivel de controlador, a nivel global de controlador ( en la clase), o incluso a nivel global de aplicación en el *main.ts*
- Uso **@UsePipes()** con el **ValidationPipe** para validar el dto
- Debo instalar el **class-validator y class-transformer**

```
@Post()
@UsePipes(ValidationPipe)
createCar(@Body() createCarDto: CreateCarDto){
        return createCarDto
}
```

- Todavía no estoy aplicando validaciones porque no las he especificado en el dto
- Voy al dto y uso **decoradores**
- Valido que sean strings
- Debo instalar class-validator y class-transformer

> npm i class-validator class-transformer

```
import { IsString } from "class-validator"

export class CreateCarDto{
```

```
    @IsString()
    readonly brand: string

    @IsString()
    readonly model: string

}
```

- Puedo personalizar el mensaje de error

```
import { IsString } from "class-validator"

export class CreateCarDto{

    @IsString({message: 'I can change the message!'})
    readonly brand: string

    @IsString()
    readonly model: string
}
```

- Voy a tener que hacer esta validación en el PATCH también. Significa que tendría que volver a poner el *@UsePipes*, etc
- Puedo coger el *@UsePipes* y colocarlo a nivel de controlador

```
import { Body, Controller, Get, Param, ParseIntPipe, Patch, Post, Delete,
ParseUUIDPipe, UsePipes, ValidationPipe} from '@nestjs/common';
import { CarsService } from './cars.service';
import { CreateCarDto } from './dtos/create-car.dto';

@Controller('cars')
@UsePipes(ValidationPipe)
export class CarsController {

    constructor(private readonly carsService: CarsService){}

    @Get()
    getAllCars(){
        return this.carsService.findAll()
    }

    @Get(':id')
    getCarById(@Param('id', new ParseUUIDPipe({version: '4'}) ) id: string){
            return this.carsService.findOneById(id)
    }

    @Post()
    createCar(@Body() createCarDto: CreateCarDto){
```

```
            return createCarDto
    }

    @Patch(':id')
    updateCar(
        @Param('id', ParseIntPipe) id: number,
        @Body() body: any){
        return body
    }

    @Delete(':id')
    deleteCar(@Param('id', ParseIntPipe) id: number){
        return id
    }
}
```

- Este pipe se debería aplicar a todos los endpoints que trabajen que reciban dtos, por lo que debería estar a nivel de aplicación

---

## Pipes Globales

- Si escribo app.use en el main puedo ver en el autocompletado varias opciones del use
- Utilizo el **useGlobalPipes**. Puedo separar por comas varios pipes
    - El **whitelist** solo deja la data que estoy esperando. Si hay otros campos en el body los ignorará
    - **forbidNonWhitelisted** en true me muestra el error si le mando data que no corresponde con el dto
- main.ts

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

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

  await app.listen(3000);
}
bootstrap();
```

- Puedo añadir más decoradores al dto
- Por ejemplo, si quiero que como minimo tenga 3 letras puedo usar **@MinLength**

```
import { IsString, MinLength } from "class-validator"

export class CreateCarDto{

    @IsString({message: 'I can change the message!'})
    readonly brand: string

    @IsString()
    @MinLength(3)
    readonly model: string

}
```

---

## Crear nuevo coche

- Creo el método **createCar** en el servicio *CarsService*
- Lo llamo desde el controlador

```
@Post()
createCar(@Body() createCarDto: CreateCarDto){
        return this.carsService.createCar(createCarDto)
}
```

- Tengo que validar que el coche no exista ya en la DB. Más adelante se hará

```
createCar(createCarDto: CreateCarDto ){

    const car: Car = {
        id: uuid(),
        brand: createCarDto.brand,
        model: createCarDto.model
    }

    this.cars.push(car) //evidentemente esto sería una llamada a la DB usando el
await
    return car
}
```

- Puedo usar **desestructuración**

```
createCar({model, brand}: CreateCarDto ){

    const car: Car = {
        id: uuid(),
        brand,
```

```
        model
    }

    this.cars.push(car)
    return car
}
```

- O usar el operador **spread**

```
createCar(createCarDto: CreateCarDto ){

    const car: Car = {
        id: uuid(),
        ...createCarDto
    }

    this.cars.push(car)
    return car
}
```

- Puedo evaluar si existe la brand y el model y lanzar un **Bad Request** si existe
- Se hará contra la DB más adelante

---

## Actualizar coche

- Creo el método en el service para poder llamarlo desde el controlador
- En lugar de usar el *CreateCarDto* voy a crear otro dto porque puedo querer actualizar solo uno de los valores (brand o model)
- Uso el decorador **@IsOptional()**. les añado ? para que de el lado de Typescript también lo marque como opcional
- Es muy probable que me envien el id en el objeto (en el frontend) para hacer la validación

```
import { IsString, IsOptional, IsUUID} from "class-validator"

export class UpdateCarDto{

    @IsString()
    @IsUUID()
    @IsOptional()
    readonly id?: string

    @IsString()
    @IsOptional()
    readonly brand?: string

    @IsString()
    @IsOptional()
```

```
    readonly model?: string
}
```

- hay algo que se puede hacer para usar las propiedades de *CreateCarDto* y que sean opcionales (**PartialTypes**, se verá más adelante)
- cars.controller.ts

```
@Patch(':id')
updateCar(
    @Param('id', ParseUUIDPipe) id: string,
    @Body() updateCarDto: UpdateCarDto){
    return this.carsService.updateCar(id, updateCarDto)
}
```

- Puedo usar un archivo de barril para las importaciones de los dto
    - Creo un archivo index.ts en /cars/dtos/index.ts
    - Hago los imports de create-car.dto.ts y update-car.dto.ts
    - Cambio la palabra import por export
- Añado la lógica en el servicio. En este caso estamos trabajando con un arreglo pero sería con la DB
- Ya tengo el método *findOneById* (que también maneja la excepción)
- Uso let porque voy a cambiar lo que tengo en car
- Mapeo cars y lo guardo en el propio cars
- Si el id es el mismo uso spread para quedarme con las propiedades existentes, las sobreescribo con el update y me quedo el id existente
- Retorno carDB en el if. Si no es el id simplemente retorno el car
- cars.service.ts

```
    updateCar(id: string, updateCarDto: UpdateCarDto){
      let carDB = this.findOneById(id)

      this.cars = this.cars.map(car=>{
            if(car.id === id){
                carDB={
                    ...carDB, //esto copia las propiedades existentes
                    ...updateCarDto, //esto va a sobreescribir las propiedades
anteriores
                    id //mantengo el id
                }
                return carDB
            }
            return car // si no es el coche del id simplemente regreso el objeto
car
      })

      return carDB // este carDB va a tener la info actualizada
    }
```

- **NOTA:** esto con la DB es mucho más sencillo
- Puedo añadir la validación de que si existe el id en el dto y este es diferente al id que recibo lance un error

```
    updateCar(id: string, updateCarDto: UpdateCarDto){
      let carDB = this.findOneById(id)

      if(updateCarDto.id && updateCarDto !== id){
        throw new BadRequestException('Car id is not valid inside body')
      }

      this.cars = this.cars.map(car=>{
            if(car.id === id){
                carDB={
                    ...carDB,
                    ...updateCarDto,
                    id
                }
                return carDB
            }
            return car
      })

      return carDB
    }
```

---

## Borrar coche

- Creo el método *deleteCar* en el servicio. Lo llamo en el controller
- cars.controller.ts

```
@Delete(':id')
deleteCar(@Param('id', ParseUUIDPipe) id: string){
    return this.carsService.deleteCar(id)
}
```

- Antes de eliminarlo el coche tiene que existir (validación), porque si no le va a dar un falso positivo
- Añado la lógica en el servicio

```
deleteCar(id: string){
    const car = this.findOneById(id) //verificación de que el coche exista

    this.cars = this.cars.filter(car => car.id !== id)
}
```

- **En resumen**: a través de los dtos nos aseguramos de que la data venga como la necesito **usando los decoradores de class-validator**