

NEST 04

Interfaces y UUID

- Creo una interface de cars para obligar que mi data luzca de una manera.
- Dentro de la carpeta de cars, nueva carpeta interfaces y dentro de esta car.interface.ts

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

- Ahora puedo obligar que los cars sean de tipo Car, y como es un arreglo lo agrego también

```
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: 'Honda',  
      model: 'Civic'  
    },  
    {  
      id: 3,  
      brand: 'Jeep',  
      model: 'Cherokee'  
    }  
  ]  
  
  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
  }
}

```

- uuid es un paquete de npm para crear ID's únicos a nivel mundial

npm i uuid

- uuid genera strings, por lo que lo cambio en la interface de Car a id: string
- Este paquete de uuid no está escrito en typescript pero se puede tener la definición de los tipos (y tener las ayudas de autocompletado) si instalo @types/uuid

npm i --save-dev @types/uuid

- Usaré la version 4 de uuid, cambio los id de la interface por la función uuid()
- Cambio todos los id: de tipo number por tipo string, ya que uuid usa strings, y quito los ParseIntPipe que convertía los id's a numero

```

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: 'Honda',
      model: 'Civic'
    },
    {
      id: uuid(),
      brand: 'Jeep',
      model: 'Cherokee'
    }
  ]

  findAll(){
    return this.cars
  }

  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
}
}

```

- Debo cambiarlo también en el controlador: quitar el ParseIntPipe y el tipado del id a string

```

import { Body, Controller, Delete, Get, Param, ParseIntPipe, Patch, Post } from
'@nestjs/common';
import { CarsService } from './cars.service';

@Controller('cars')
export class CarsController {

    constructor(
        private readonly carsService: CarsService
    ){}

    @Get()
    getAllCars(){
        return this.carsService.findAll()
    }
    @Get('/:id')
    getCarById( @Param('id') id: string){
        return this.carsService.findOneById( id )
    }
    @Post()
    createCar( @Body() body: any){
        return{
            body
        }
    }
    @Patch('/:id')
    updateCar(
        @Param('id') id: string,
        @Body() body: any){
        return body
    }
    @Delete('/:id')
    deleteCar(@Param('id') id: string){
        return {
            msg: "Delete correcto"
        }
    }
}

```

Pipe ParseUUIDPipe

- Debo asegurar que lo que recibo es un uuid válido (validar los argumentos), porque cuando haga la consulta a la DB, si no es un uuid válido dará un error 500
- Solo tengo que añadir el ParseUUIDPipe a los métodos del controller que usan el uuid
- Puedo usar una versión en concreto de la validación con ParseUUIDPipe creando una nueva instancia

```
import { Body, Controller, Delete, Get, Param, ParseUUIDPipe, Patch, Post } from
 '@nestjs/common';
import { CarsService } from './cars.service';

@Controller('cars')
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() body: any){
    return{
      body
    }
  }
  @Patch('/:id')
  updateCar(
    @Param('id') id: string,
    @Body() body: any){
    return body
  }
  @Delete('/:id')
  deleteCar(@Param('id') id: string){
    return {
      msg: "Delete correcto"
    }
  }
}
```

- También se puede personalizar el mensaje de error, el status, entre otras cosas...
- Sin crear la nueva instancia con new y sin parámetros también sirve

```
@Get('/:id')
getCarById( @Param('id', ParseUUIDPipe) id: string){
```

```

    return this.carsService.findOneById( id )
  }

```

DTO Data Transfer Object

- La data del body que llega en una petición Post debe ser validada
- Puedo crear un customPipe para ello, pero para este caso me voy a servir de un DTO
- **Data Transfer Object** es un objeto que me sirve para transferir la data del body como yo quiero que luzca
- Se recomienda que se haga como una clase
- Si estoy esperando una información con una clase, si le paso esa clase al servicio, este sabe que luce de una cierta manera
- Para eso es el DTO, para asegurarse como es que fluye la info entre diferentes piezas de mi código
- Me va a servir para aplicar muchas reglas de validación automáticas
- En la carpeta de cars, creo la carpeta dto. Cómo se va a encargar del post lo llamo crear-car.dto.ts
- Estoy esperando recibir el brand y el model
- Se aconseja que los dto sean readonly, para que accidentalmente no se reasigne un valor al dto
- Cómo me manda la data el cliente, así será siempre
- Tiene que ser brand y model literal. model no sirve, para que no de error en la DB

```

export class createCarDto {

  readonly brand: string;
  readonly model: string;
}

```

- Entonces el body va a ser de tipo createCarDto, lo importo

```

import { Body, Controller, Delete, Get, Param, ParseUUIDPipe, Patch, Post } from
 '@nestjs/common';
import { CarsService } from './cars.service';
import { createCarDto } from './dto/create-car.dto';

@Controller('cars')
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() body: createCarDto){
        return{
            body
        }
    }
    @Patch('/:id')
    updateCar(
        @Param('id') id: string,
        @Body() body: any){
        return body
    }
    @Delete('/:id')
    deleteCar(@Param('id') id: string){
        return {
            msg: "Delete correcto"
        }
    }
}

```

- Puedo renombrar el body a createCarDto
- Pero si en el body de la petición Post mando otra información adicional, le sigue llegando
- Tengo que decirle a NEST que aplique las validaciones de los dtos
- Por ahora creé una clase, que define cómo espero mover la data dentro de mi aplicación
- En el futuro voy a atacar estos problemas de manera global, para que cuando resuelva uno los resuelva todos

ValidationPipe - Class Validator y Transformer

- El ValidationPipe trabaja de la mano con otras librerías como class-validator y class-transformer. Las instalo

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

- class-validator expone un montón de decoradores:
 - IsOptional
 - IsPositive
 - IsMongoid
 - ...etc
- Hay 4 lugares dónde se pueden aplicar Pipes
 - Por parámetro (como ya he visto)
 - En el controlador
 - A nivel global del controlador
 - A nivel global de aplicación (en el main)

- En el controlador, debajo del @Post() voy a escribir @UsePipes(ValidationPipe)

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

- Todavía falta metadata en el dto.
- Se hace con decoradores
- Yo estoy esperando que el brand y el modelo sean strings
 - Le añado @IsString, lo importo del class-validator

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

export class createCarDto {
    @IsString()
    readonly brand: string;
    @IsString()
    readonly model: string
}
```

- Si ahora en la petición post en lugar de "model": "Volvo" pongo "modelL" con L mayúscula me marca el error "model must be a string"
- Todavía sigue mandando otros campos si los añado al body
- Puedo añadir el mensaje customizado

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

export class createCarDto {
    @IsString({ message: "This brand is not valid"})
    readonly brand: string;

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

- Como voy a tener que usar el ValidationPipe en otros métodos del controlador, lo añado de manera general dentro del controlador
- Lo añado debajo del @Controller, así se haría

```

import { Body, Controller, Delete, Get, Param, ParseUUIDPipe, Patch, Post,
UsePipes, ValidationPipe } from '@nestjs/common';
import { CarsService } from '../cars.service';
import { createCarDto } from '../dto/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') id: string,
    @Body() body: any){
    return body
  }
  @Delete('/:id')
  deleteCar(@Param('id') id: string){
    return {
      msg: "Delete correcto"
    }
  }
}

```

- Pero en realidad, voy a tener que usarlo en otros controladores, así que debería estar en un punto más alto de mi app
- Lo borro y lo coloco en el main (detallado en la próxima lección)

Pipes Globales - A nivel de aplicación

- Voy al main
- Uso el useGlobalPipes porque ValidationPipe es un Pipe


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

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

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

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

- whitelist: true solo deja pasar la data que estoy esperando en el dto
- forbidNonWhitelisted: true regresa el error
- Tengo habilitadas las validaciones a nivel global, en este caso al post de cars
- Puedo añadir más decoradores al dto, como por ejemplo que el modelo tenga mínimo 3 letras

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

export class createCarDto {
  @IsString({ message: "This brand is not valid"})
  readonly brand: string;

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

- Puedo personalizar el mensaje con una coma y abriendo llaves

```
@minLength(3, {"El modelo debe de tener al menos 3 letras"})
```

- Hay muchos decoradores ;)

Crear el nuevo coche

- Creo el método **create** en el cars.service y le paso como argumento createCarDto del tipo crearCarDto

- Recuerda que las propiedades del dto son readonly para que yo no haga createCarDto.brand y pueda cambiarlo
- Los controladores no manejan la lógica. Su único objetivo es escuchar al cliente y regresar una respuesta

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

- Implemento la lógica en el CarsService

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

@Injectable()
export class CarsService {

  private cars: Car[] = [
    {
      id: uuid(),
      brand: 'Toyota',
      model: 'Corola'
    },
    {
      id: uuid(),
      brand: 'Honda',
      model: 'Civic'
    },
    {
      id: uuid(),
      brand: 'Jeep',
      model: 'Cherokee'
    }
  ]

  findAll(){
    return this.cars
  }
  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
  }

  create(createCarDto: createCarDto){
```

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

- Puedo usar la desestructuración para tener un código más limpio

```
create({brand, model}: createCarDto){

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

- O usar el operador spread

```
create(createCarDto: createCarDto){

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

- Finalmente lo agrego al arreglo con un push

```
create(createCarDto: createCarDto){

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

  this.cars.push(car)
```

```
    return car  
  }  
}
```

- Hay validaciones que hacer, por ejemplo para no repetir modelos, pero se hará más adelante contra la base de datos

Actualizar coche

- Creo el archivo update-car.dto.ts
- Debo pensar que voy a recibir un id y que quizá no rellene todos los campos, por eso los pongo con el decorador de es opcional
- Es conveniente colocarles el interrogante para typescript
- Todo esto porque en la actualización puede que venga a cambiar solo un parámetro, no todos

```
import { IsOptional, IsString, IsUUID } from "class-validator";  
  
export class updateCarDto {  
  @IsString()  
  @IsUUID()  
  @IsOptional()  
  readonly id?: string  
  
  @IsString()  
  @IsOptional()  
  readonly brand?: string  
  
  @IsString()  
  @IsOptional()  
  readonly model?: string  
}
```

- Coloco el updateCarDto en el controlador

```
@Patch('/:id')  
updateCar(  
  @Param('id', ParseUUIDPipe) id: string,  
  @Body() updateCarDto: updateCarDto){  
  return updateCarDto  
}
```

- Ahora falta guardar la actualización.
- Creo un nuevo método en CarsService llamado update.
- Aquí si tiene que venir el id cómo parámetro

```
update(id: string, updateCarDto: updateCarDto){ }
```

Actualización de coche

- Coloco el método en el controlador

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

- Volviendo al método de carsService
- Debo verificar que el id exista en el arreglo de cars
- Puedo usar el método creado anteriormente, findOneById
- Si pasa es que el coche existe
- Usaré el punto map para barrerlos y seleccionar con una validación el coche por su id
- Usaré el spread del car, luego el spread del dto que sobrescribirá lo que haya que sobrescribir y le paso también el id
- Si no es el carDB retorno el car tal cual

```
import { Injectable, NotFoundException } from '@nestjs/common';  
import { Car } from '../interfaces/car.interface';  
import { v4 as uuid } from 'uuid';  
import { createCarDto } from '../dto/create-car.dto';  
import { updateCarDto } from '../dto/update-car.dto';  
  
@Injectable()  
export class CarsService {  
  
  private cars: Car[] = [  
    {  
      id: uuid(),  
      brand: 'Toyota',  
      model: 'Corola'  
    },  
    {  
      id: uuid(),  
      brand: 'Honda',  
      model: 'Civic'  
    },  
    {  
      id: uuid(),  
      brand: 'Jeep',  
      model: 'Cherokee'  
    }  
  ]  
}
```

```

    }
  ]

  findAll(){
    return this.cars
  }

  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
  }

  create(createCarDto: createCarDto){

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

    this.cars.push(car)
    return car
  }

  update(id: string, updateCarDto: updateCarDto){
    let carDB= this.findOneById(id)

    this.cars = this.cars.map(car =>{
      if(car.id === id){
        //esta lógica no importa pues se hará
        //contra una DB
        carDB ={
          ...carDB,
          ...updateCarDto,
          id
        }
        return carDB
      }
      return car;
    })

    return carDB //el car actualizado
  }
}

```

- **Con la base de datos es más sencillo**
- Puedo poner la validación de si recibo un id y ese id es diferente del que hay en la DB lance un error

```

update(id: string, updateCarDto: updateCarDto){
  let carDB= this.findOneById(id)

```

```
if(updateCarDto.id && updateCarDto.id !== id){
  throw new BadRequestException("El id no es válido")
}

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

Borrar un coche

- Primero hay que verificar que el coche con ese id existe
- Para ello usaré el método creado anteriormente findOneById. Si pasa es que existe
- Uso el filter para devolver un arreglo sin el del id (coche) correspondiente.

```
delete(id: string){
  let carDB = this.findOneById(id)
  this.cars = this.cars.filter(car=> car.id !== id)
}
```

- En el controlador:

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