01 NEST Primeros Pasos

- Instalar NEST CLI. Ejecutar como administrador

```
npm i -g @nestjs/cli
```

- Crear un proyecto

```
nest new project-name
```

- Para correr la app

```
npm run start
```

- Para correr la app y que escuche los cambios (genera la carpeta dist)

```
npm run start:dev
```

- Por defecto es el puerto 3000

Explicación de src

- Borro todos los archivos, dejo solo el main y el app.module limpio
- Este es el módulo principal. Va a tener referencia a otros módulos, servicios, etc
- app.module.ts

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


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

- Los módulos acoplan y desacoplan un conjunto de funcionalidad específica por dominio
- El **main** tiene una función asíncrona que es *bootstrap* (puedo llamarlo como quiera, main por ejemplo)

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule); // esto crea la app
  await app.listen(3000); //escucha en este puerto
```

```
}
bootstrap();
```

Controladores

- La diferencia entre las clases de los servicios, controladores, son los decoradores
- Controlan rutas. Son los encargados de escuchar la solicitud y emitir una respuesta
- Para generar un controlador

> nest g co path/nombre

- Para mostrar la ayuda *nest --help*
- Creo un módulo llamado cars en */car-dealership*

> nest g mo cars

- Crea la carpeta cars. La clase cars aparece con el decorador **@Module({})**
- Aparece en el array de imports de app.module (el módulo principal)
- Creo el controlador en */car-dealership* con *nest g co car-dealership*
- Crea una clase **CarsController** con el decorador **@Controller('cars')**
- El controlador lo ha añadido en el modulo de cars
    - Si encuentra un módulo con el nombre cars lo coloca ahí, si no lo hará en el módulo más cercano
- Creo un método GET en el controlador
- Le añado el decorador **@GET()**

```
import { Controller, Get } from '@nestjs/common';

@Controller('cars')
export class CarsController {

    @Get()
    getAllCars(){
        return ['Toyota', 'Suzuki', 'Honda']
    }
}
```

- Si apunto a http://localhost:3000/cars con un método GET me retorna el arreglo

Obtener un coche por ID

- Hago un pequeño cambio, guardo el array en una variable privada y uso el this

```
import { Controller, Get } from '@nestjs/common';

@Controller('cars')
```

```
export class CarsController {

    private cars = ['Toyota', 'Suzuki', 'Honda']

    @Get()
    getAllCars(){
        return this.cars
    }
}
```

- Quiero crear el método para buscar por id. Usaré el parámetro de la url para declararlo como posición en el arreglo
- Tengo que decirselo a Nest. Para obtener parámetros / segmentos uso **@Param('id)**
- Para obtener el body de la petición es **@Body()** y **@Query()** para los query. El response es **Res()** (hay que importarlo de Express)
- Debo añadirle el tipo al id

```
    @Get(':id')   //Puedo colocar /:id pero no es necesario el slash
    getCarById( @Param('id') id: string){
        return this.cars[+id]
    }´
```

- Por ahora puedo parsear el id que viene como string (al venir de la url) con +id, pero **Nest tiene los pipes para parsear la data**

Servicios

- Todos los servicios son providers
- No todos los providers son servicios.
- **Los Providers son clases que se pueden inyectar**
- Para generar un Servicio

> nest g s cars --no-spec //El --no-spec es para que no cree el archivo de test

- No es más que una clase llamada **CarsService** con el decorador **@Injectable()**
- El servicio aparece en el array de providers del módulo *CarsModule*
- Voy a mockear la db en el servicio

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

@Injectable()
export class CarsService {

    //al ser private solo se va a poder consumir dentro del servicio
    private cars =[
        {
            id:1,
```

```
            brand: 'Toyota',
            model: 'Corola'
        },
        {
            id:2,
            brand: 'Suzuki',
            model: 'Vitara'
        },
        {
            id:3,
            brand: 'Honda',
            model: 'Civic'
        }
    ]
}
```

- Ahora necesito hacer uso de la **inyección de dependencias** para usar el servicio en el controlador

---

Inyección de dependencias

- Declaro en el constructor el servicio *private* porque no lo voy a usar fuera de este controlador, y *readonly* para que no cambie accidentalmente algo a lo que apunte
- El arreglo de cars no aparece en el autocompletado de *carsService*. porque es **privado**. Debo crear un método para ello

```
import { Controller, Get, Param } from '@nestjs/common';
import { CarsService } from './cars.service';

@Controller('cars')
export class CarsController {

    constructor(private readonly carsService: CarsService){}

    @Get()
    getAllCars(){
        return this.carsService. //no aparece el autocompletado porque no tengo
nada público, necesito crear un método
    }

    @Get(':id')
    getCarById(@Param('id') id: string){

    }
}
```

- Añado el método *findAll()*

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

@Injectable()
export class CarsService {

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

    findAll(){
        return this.cars
    }
}
```

- Ahora si dispongo del arreglo de cars en el servicio con el método *findAll*

```
import { Controller, Get, Param } 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){

    }
}
```

- Creo también el método *findOneById* en el servicio

```
   findOneById(id: number){
        return this.cars[id]  //esta puede ser una manera. Puedo usar el .find
también
    }
```

- Lo uso en el controlador

```
import { Controller, Get, Param } 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)
    }
}
```

Pipes

- Hay que implementar una validación del argumento que le paso como id
- Si pasara algo que no es un numero como 3a me devolvería un *NaN*. Debo manejar este tipo de errores
- Los pipes transforman la data recibida en requests, para asegurar un tipo, valor o instancia de un objeto.
- Pipes integrados por defecto
    - ValidationPipe - más orientado a las validaciones y también hace ciertas transformaciones
    - ParseIntPipe - transforma de string a número
    - ParseBoolPipe - transforma de string a boolean
    - ParseArrayPipe - transforma de string a un arreglo
    - ParseFloatPIpe - transforma de string a un float
    - ParseUUIDPipe - transforma de string a UUID
- Uso ParseIntPipe para la verificación de que sea un entero

```
import { Controller, Get, Param, ParseIntPipe } 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')             //uso el Pipe    ahora si puedo tipar a number el id
    getCarById(@Param('id', ParseIntPipe ) id: number){
            return this.carsService.findOneById(id)
    }
}
```

- Si ahora le paso algo que no sea un número en la url salta el error "*Validation failed (numeric string is expected)*"
- Si yo lanzo un error dentro del controlador con *throw new Error*, el servidor responde con *500 Internal Server Error* y en consola me aparece el error
- La *Exception Zone* incluye cualquier zona menos los middlewares y los *Exception Filters*
- Cualquier error (no controlado) que sea lanzado en la *Exception Zone* será lanzado automáticamente por Nest
- **Falta una cosa**: si yo pongo un id válido (un número) pero que no existe en la DB me manda un *status 200* cómo que todo lo hizo correcto. Esto no debería ser así

Exception Filter

- Maneja los errores de código en mensajes de respuest http. Nest incluye los casos de uso comunes pero se pueden expandir
    - BadRequestException: se estaba esperando un número y recibí un string, por ejemplo
    - NotFoundException: 404, no se encontró lo solicitado
    - UnauthorizedException: no tiene autorización
    - ForbiddenException
    - RequestTimeoutException
    - GoneException
    - PayloadTooLargeException
    - InternalServerErrorException: 500
- Estos solo son los más comunes, hay muchos más (mirar la documentación de Nest)
- Ahora quiero mandar un error si el coche no existe
- cars.service.js

```
findOneById(id: number){
        const car = this.cars.find(car => car.id === id)
        if(!car) throw new NotFoundException() //si el id no existe devuelve Not
Found
        return car
    }
```

- Puedo escribir dentro del paréntesis el mensaje que quiero mostrar

```
    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
    }
```

Post, Patch y Delete

- Creo el método Post en el controlador
- Uso **@Body** para obtener el body de la petición. Lo nombro body y lo tipo *any* (de momento)
- Debo hacer la validación de que me envien un brand y un model en el body y de que sean strings
- Por ahora creo los tres endpoints

```
    @Post()
    createCar(@Body() body: any){
            return body
    }

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

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

- **NOTA:** En el Patch, si no pongo nada en el body me regresa un *status 200* igualmente!
- La implementación de los métodos en la siguiente sección

---

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**

---

03 NEST CLI CRUD

- El comando **resource** me ayuda a crear automáticamente los servicios, controladores, dtos, etc
- En esta sección vamos a aprender la comunicación entre modulos y a crear el módulo de Seed para llenar de coches la DB. Data precargada.
- Los servicios son Singletons. Se crea una instancia y esa instancia es compartida por los demás controladores o servicios mediante inyección de dependencias
- Ahora voy a trabajar con las marcas de los coches, las cuales van a tener su id y el nombre
- Para ello tengo que volver a crear otro modulo, servicio, controlador...pero se hará mediante la linea de comandos
- No lo voy a manejar con una interfaz si no con una entity
- Usaré dtos para la creación y la actualización

Nest CLI Resource - Brands

- En el endpoint /brands voy a tener disponibles todas las marcas con las que voy a trabajar
- Le voy a añadir la fecha de creación y actualización, además del id
- Usaré el CLI para crear el módulo, servicio, controlador, etc (un CRUD completo, se necesita conexión)

> nest g resource brands //g de generar, puedo añadir --no-spec para que no genere los tests

- Selecciono REST API, Generate CRUD entry points? yes
- Automaticamente **añade el módulo** BrandsModule en los imports de *app.module*
- La entity que me ha generado es como una interface. Es una simple clase. Es la representación de una tabla
  - Ahí crearé el nombre, la fecha de creación, etc
- En el package.json puedo ver que instaló **@nestjs/mapped-types**, el resto es igual
- Si tengo problemas de errores puedo deshabilitar "*eslint-config-prettier*" y "*eslint-plugin-prettier*"
- Si analizo el controlador tengo hecha la inyección de dependencias, los dtos en su sitio, etc
- El update-brand.dto hereda de CreateBrandDto con **PartialType**
  - PartialType proviene de *@nestjs/mapped-types*
  - Hace que todas las propiedades sean **opcionales**

CRUD completo de Brands

- Empiezo por la **entity**. Cómo quiero que la información quede grabada en la base de datos
- Las entities no tienen la extension Entity (BrandEntity) porque si no la tabla se llamaría así
- brand.entity.ts

```
export class Brand {
    id: string
    name: string

    createdAt: number
    updatedAt?: number
}
```

- Creo el arreglo de brands (lo que sería la data) de tipo Brand[]
- En *findOne* uso .find con el id para encontrar el coche. Hago la validación y mando la excepción si no lo encuentra. Retorno brand
- En *findAll* solo tengo que devolver el arreglo con this.brands
- En el método create tengo que ver primero cómo quiero que luzca el dto

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

export class CreateBrandDto {
    @IsString()
    @MinLength(1)
    name: string
}
```

- Creo en el método el objeto brand, utilizo el .push y lo retorno
- En *update* tengo que definir el dto
- **NOTA**: cómo solo tengo una propiedad, de momento no voy a usar PartialTypes

```
//import { PartialType } from '@nestjs/mapped-types';
import { CreateBrandDto } from './create-brand.dto';
import { IsString, MinLength } from 'class-validator';

//export class UpdateBrandDto extends PartialType(CreateBrandDto) {}
export class UpdateBrandDto{
    @IsString()
    @MinLength(1)
    name: string
}
```

- La lógica de actualización es la misma que con los coches
- Para el *delete* uso .filter

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { CreateBrandDto } from './dto/create-brand.dto';
import { UpdateBrandDto } from './dto/update-brand.dto';
import { Brand } from './entities/brand.entity';
import { v4 as uuid } from 'uuid';

@Injectable()
export class BrandsService {

  private brands: Brand[] = [
    {
      id: uuid(),
      name: "Volvo",
```

```
      createdAt: new Date().getTime() //debo añadirle getTime para que no choque
con la validación tipo number
    }

  ]

  create(createBrandDto: CreateBrandDto) {
    //lo uso como una interfaz. En la vida real, con una DB, va a ser usado como
una instancia
    const brand: Brand = {
      id: uuid(),
      name: createBrandDto.name.toLowerCase(), //uso toLowerCase porque los quiero
almacenar así
      createdAt: new Date().getTime()
    }

    this.brands.push(brand)

    return brand
  }

  findAll() {
    return this.brands;
  }

  findOne(id: string) {
    const brand = this.brands.find(brand=> brand.id === id)
    if(!brand) throw new NotFoundException(`Brand with id ${id} not found`)
    return brand
  }

  update(id: string, updateBrandDto: UpdateBrandDto) {
    let brandDB = this.findOne(id)
    this.brands = this.brands.map(brand=>{
              if(brand.id === id){
                brandDB={
                  ...brandDB,
                  ...updateBrandDto,
                }
                brandDB.updatedAt = new Date().getTime()
                return brandDB
              }
              return brand
    })
    return brandDB
  }

  remove(id: string) {
    this.brands = this.brands.filter(brand => brand.id !== id)
  }
}
```

- En el controlador me aseguro de que el id sea un string y le paso el pipe de UUID

```
import { Controller, Get, Post, Body, Patch, Param, Delete, ParseUUIDPipe } from
'@nestjs/common';
import { BrandsService } from './brands.service';
import { CreateBrandDto } from './dto/create-brand.dto';
import { UpdateBrandDto } from './dto/update-brand.dto';

@Controller('brands')
export class BrandsController {
  constructor(private readonly brandsService: BrandsService) {}

  @Post()
  create(@Body() createBrandDto: CreateBrandDto) {
    return this.brandsService.create(createBrandDto);
  }

  @Get()
  findAll() {
    return this.brandsService.findAll();
  }

  @Get(':id')
  findOne(@Param('id', ParseUUIDPipe) id: string) {
    return this.brandsService.findOne(id);
  }

  @Patch(':id')
  update(@Param('id', ParseUUIDPipe) id: string, @Body() updateBrandDto:
UpdateBrandDto) {
    return this.brandsService.update(id, updateBrandDto);
  }

  @Delete(':id')
  remove(@Param('id', ParseUUIDPipe) id: string) {
    return this.brandsService.remove(id);
  }
}
```

Crear servicio SEED para cargar datos

- Vamos a generar un **SEED** (semilla)
- Se usa para pre-cargar la data
- Uso el CLI

```
nest g resource seed --no-spec
```

- No voy a usar dtos ni entities. No uso los métodos del CRUD, solo necesito el **GET**, lo llamo *runSeed*
- En el servicio borro todos los métodos, dejo un solo método para el GET, lo llamo *populateDB*
- Cars es una propiedad privada en el servicio de cars. Para cargar la data voy a tener que exponerla con un método

- Lo mismo con las brands
- En seed creo una carpeta llamada data. Podría ser un json pero voy a usar TypeScript porque quiero una estructura específica de mi data
- Creo el archivo cars.seed.ts
- En este caso la interfaz Car no necesito que esté importada en el módulo, pero hay cosas que si necesitan estar importadas
- Si son interfaces o clases que no tienen dependencias o ninguna inyección, se pueden importar directamente
- cars.seed.ts

```
import { Car } from "src/cars/interfaces/car.interface";
import { v4 as uuid } from "uuid";


export const CARS_SEED: Car[] = [
    {
        id: uuid(),
        brand: "Toyota",
        model: "Corolla"

    },
    {
        id: uuid(),
        brand: "Suzuki",
        model: "Vitara"

    },
    {
        id: uuid(),
        brand: "Opel",
        model: "Astra"

    }
]
```

- Contra la DB lo que haría es una función que inserte la data (se hará después)
- Hago brands.seed.ts (la relación con los coches se hará cuando se trabaje con una DB real)

```
import { Brand } from "src/brands/entities/brand.entity";
import { v4 as uuid } from "uuid";


export const BRANDS_SEED: Brand[] = [
    {
        id: uuid(),
        name: "Toyota",
        createdAt: new Date().getTime()
    },
    {
```

```
        id: uuid(),
        name: "Suzuki",
        createdAt: new Date().getTime()

    },
    {
        id: uuid(),
        name: "Opel",
        createdAt: new Date().getTime()

    }
]
```

- El método *populateDB* necesita trabajar mediante inyección de dependencias con los otros servicios
- En este caso como trabajo con arreglos podría retornar los arreglos y ya está, pero no es el objetivo de la lección
- Los servicios, como trabajan a través de inyección de dependencias con los controladores, **si debo declararlos en el módulo**
- Además tengo que **poder acceder a la propiedad privada cars y a la propiedad privada brands para cargar la data**
- Creo el método **fillCarsWithSeed** en el cars.service

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

@Injectable()
export class CarsService {

    private cars: Car[] =[]

    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
    }

    createCar(createCarDto: CreateCarDto ){

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

```
        this.cars.push(car)

        return car
    }

    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
    }

    deleteCar(id: string){
        const car = this.findOneById(id)

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

    //metodo para el seed

    fillCarsWithSeedData(cars: Car[]){
        this.cars = cars
    }
}
```

- Hago exactamente lo mismo para brands.service

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { CreateBrandDto } from './dto/create-brand.dto';
import { UpdateBrandDto } from './dto/update-brand.dto';
import { Brand } from './entities/brand.entity';
import { v4 as uuid } from 'uuid';

@Injectable()
export class BrandsService {
```

```
  private brands: Brand[] = []

  create(createBrandDto: CreateBrandDto) {

    const brand: Brand = {
      id: uuid(),
      name: createBrandDto.name.toLowerCase(),
      createdAt: new Date().getTime()
    }

    this.brands.push(brand)

    return brand
  }

  findAll() {
    return this.brands;
  }

  findOne(id: string) {
    const brand = this.brands.find(brand=> brand.id === id)
    if(!brand) throw new NotFoundException(`Brand with id ${id} not found`)
    return brand
  }

  update(id: string, updateBrandDto: UpdateBrandDto) {
    let brandDB = this.findOne(id)
    this.brands = this.brands.map(brand=>{
              if(brand.id === id){
                brandDB={
                  ...brandDB,
                  ...updateBrandDto,
                }
                brandDB.updatedAt = new Date().getTime()
                return brandDB
              }
              return brand
    })

    return brandDB
  }

  remove(id: string) {
    this.brands = this.brands.filter(brand => brand.id !== id)
  }

  //método SEED
  fillBrandsWithSeedData(brands: Brand[]){
    this.brands = brands
}
}
```

- Ahora tengo que llamar el *cars.service* y el *brands.service* desde mi **seed.service**

Inyectar servicios en otros servicios

- Para resolver la dependencia de **SeedService** *CarsService* y poder inyectarlo en el constructor tiene que ser parte del **SeedModule**
- Debo exportarlo de *CarsService* e importarlo en *SeedService*
- cars.module

```
import { Module } from '@nestjs/common';
import { CarsController } from './cars.controller';
import { CarsService } from './cars.service';

@Module({
  controllers: [CarsController],
  providers: [CarsService],
  exports:[CarsService] //exporto el servicio
})
export class CarsModule {}
```

- En *imports* importo los **módulos**. En este caso importo el *CarsModule*
- seed.module

```
import { Module } from '@nestjs/common';
import { SeedService } from './seed.service';
import { SeedController } from './seed.controller';
import { CarsService } from 'src/cars/cars.service';
import { CarsModule } from 'src/cars/cars.module';

@Module({
  controllers: [SeedController],
  providers: [SeedService],
  imports:[CarsModule]
})
export class SeedModule {}
```

- Ahora puedo inyectar el servicio y llamar al método pasándole *CARS_SEED*
- seed.service

```
import { Injectable } from '@nestjs/common';
import { CarsService } from 'src/cars/cars.service';
import { CARS_SEED } from './data/cars.seed';


@Injectable()
export class SeedService {

  constructor(private readonly carsService: CarsService){}
```

```
  populateDB() {
    this.carsService.fillCarsWithSeedData(CARS_SEED)

  }
}
```

- Hago lo mismo con brands
- Lo correcto sería borrar todo lo que hay en readme (dejo el logo de Nest) y escribo

```
<p align="center">
  <a href="http://nestjs.com/" target="blank"><img
src="https://nestjs.com/img/logo-small.svg" width="200" alt="Nest Logo" /></a>
</p>

# Car Dealship

Populate DB
```
http://localhost:3000/seed
```
```

- Para ver el archivo abrir README.md y *ctrl+shift+P* Markdown: abrir vista previa en el lateral

---

04 NEST MONGODB POKEDEX

- Creo un nuevo proyecto

> nest new pokedex

- En esta sección vamos a trabajar también con **Pipes personalizados** y **Exception Filter**, además de la **conexión con la DB**
- Borro el archivo .spec, el app.service y el app.controller (no los necesito)
- Borro sus referencias en el app.module, dejo el modulo sin dependencias

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


@Module({
  imports: []
})
export class AppModule {}
```

Servir contenido estático

- Normalmente es en la raíz dónde quiero servir un sitio web (una app de React o lo que sea)
- Creo una carpeta en la raíz llamada public
- Dentro creo un index.html y dentro de la carpeta css/styles.css
- Pongo un h1 en el body del html y configuro algunos paddings y font-size en el css
- **Para servir contenido estático** uso **ServeStaticModule**
- Para ello instalo npm i @nestjs/serve-static
- Cuando veas la palabra módulo **siempre va en los imports**
- Las importaciones de Node van al inicio, en este caso uso join del paquete *path*
- app.module.ts

```
import { join } from 'path';
import { Module } from '@nestjs/common';
import {ServeStaticModule} from '@nestjs/serve-static'


@Module({
  imports: [
    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'public')
    })
  ]
})
export class AppModule {}
```

- En la creación de la API REST nos vamos a basar en la API de Pokemon

---

Global Prefix

- Creo la API (res de resource)

```
nest g res pokemon --no-spec
```

- Si voy a app.module veo que en imports tengo **PokemonModule**
- Puedo especificar un segmento como prefijo de la url con **setGlobalPrefix** en el main

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

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

  app.setGlobalPrefix('api/v2')

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

Docker - Docker Compose - MongoDB

- Para el desarrollo es aconsejable usar docker
- Creo el archivo **docker-compose.yaml** en la raíz del proyecto
- Uso la versión de la imagen de mongo (5)
- Puedo obviar el restart:always para que no incie el contenedor con el SO
- Conecto el puerto 270127 de **mi computadora** con el 27017 **del contenedor** (solo ese puerto está expuesto)
- Configuro unas **variables de entorno** cómo dice la documentación
- Para que la data sea persistente pese a que borre el contenedor uso los **volumenes**
  - Viene a ser como el puerto solo que ahora va a ser **una carpeta del file system**
  - Creo la carpeta mongo y la conecto con el comando *./mongo:/data/db,* donde data/db está en la imagen que estoy montando
- De momento no le coloco password (se hará en el despliegue)

```
version: '3'

services:
  db:
    image: mongo:5
    restart: always
    ports:
      - 27017:27017
    environment:
    - MONGODB_DATABASE=nest-pokemon
    volumes:
      - ./mongo:/data/db
```

- En VSCode escribo el comando up para levantarlo, -d para que corra desligada de esta instancia de la terminal

```
docker-compose up -d
```

- Si no existe la imagen la descarga
- Esto me crea también la carpeta mongo en la raíz de mi proyecto
- Si miro en docker tengo la imagen de **mongo 5** y en containers tengo **pokedex**
- Puedo borrar la imagen y volver a usar el comando para levantar la DB
- Creo la conexión con este string usando TablePlus

```
mongodb://localhost:27017/nest-pokemon
```

- **NOTA**: Para evitar **problemas** detengo el container con docker-compose down y paro mi servicio de mongo en windows con

```
net stop MongoDB
```

- Hago un test de conexión con TablePlus con el contenedor de docker UP, todo OK

Creo un README.md

- Uso el README de Nest, borro todo menos el log (porque me gusta)

````
<p align="center">
  <a href="http://nestjs.com/" target="blank"><img
src="https://nestjs.com/img/logo-small.svg" width="200" alt="Nest Logo" /></a>
</p>

# Ejecutar en desarrollo

1. Clonar el repositorio
2. Ejecutar
```
npm i
```

3. Tener el Nest CLI instalado
```
npm i -g @nestjs/cli
```
4. Levantar la base de datos
```
docker-compose up -d
```

## Stack Usado
- MongoDB
- Nest
````

Conectar Nest con Mongo

- Instalo mongoose y los conectores de nest

```
npm i @nestjs/mongoose mongoose
```

- En app.module uso **MongooseModel.forRoot**, tengo que especificarle la url de la DB

```
import { join } from 'path';
import { Module } from '@nestjs/common';
import {ServeStaticModule} from '@nestjs/serve-static'
import { PokemonModule } from './pokemon/pokemon.module';
import { MongooseModule } from '@nestjs/mongoose';


@Module({
  imports: [
    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'public')
```

```
    }),
    MongooseModule.forRoot('mongodb://localhost:27017/nest-pokemon'),
    PokemonModule
  ]
})
export class AppModule {}
```

- Debo tener docker abierto, subir la DB con docker-compose e iniciar nest con nom run start:dev

---

Crear esquemas y modelos

- Se recomienda que **la entidad sea una clase** para poder definir reglas de negocio
- Cada registro en la DB de mi entidad será una nueva instancia de la clase
- Voy a tener **3 identificadores únicos**:
  - El nombre del pokemon
  - El número del pokemon
  - El mongoID (no lo tengo que especificar porque mongo me lo da automaticamente)
- Mongoose se va a encargar de ponerle la "s" a la clase Pokemon(s)
- Hago que la clase **herede de Document** de mongoose
- Necesito especificarle un decorador para decir que es un esquema
- Le añado unas propiedades con el **decorador Props**
- Lo exporto

```
import {Document} from 'mongoose'
import {Schema, SchemaFactory, Prop} from '@nestjs/mongoose'


@Schema()
export class Pokemon extends Document{

    @Prop({
        unique: true,
        index: true
    })
    name: string

    @Prop({
        unique: true,
        index: true
    })
    no: number
}


export const PokemonSchema = SchemaFactory.createForClass( Pokemon )
```

- Hay que conectar esta entidad con la DB
- **Los módulos siempre van en los imports**

- Voy a usar **el módulo mongoose PERO NO VA A SER forRoot**
- Pokemon.name, el name sale **de la herencia del Document**, no es la propiedad
- También debo indicarle el **schema**
- **NOTA**: el forFeature es **un arreglo** de objetos
- pokemon.module

```
import { Module } from '@nestjs/common';
import { PokemonService } from './pokemon.service';
import { PokemonController } from './pokemon.controller';
import { MongooseModule } from '@nestjs/mongoose';
import { Pokemon, PokemonSchema } from './entities/pokemon.entity';


@Module({
  imports:[
    MongooseModule.forFeature([
      {
      name: Pokemon.name,
      schema: PokemonSchema
      }
    ])
  ],
  controllers: [PokemonController],
  providers: [PokemonService]
})
export class PokemonModule {}
```

- Hay veces que aparece la tabla directamente en TablePlus (en este caso pokemons)
- Si no, aparecerá con la primera inserción. Mientras no haya ningún error está bien
- Se puede relacionar este modelo fuera de este módulo, **se verá más adelante**

POST - Recibir y validar data

- Primero voy al createPokemonDto
- Instalo class-validator y class-transformer

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

- el dto

```
import { IsPositive, IsInt, IsString, Min, MinLength } from "@nestjs/class-
validator"

export class CreatePokemonDto {

    @IsInt()
    @IsPositive()
    @Min(1)
```

```
    no: number

    @IsString()
    @MinLength(3)
    name: string

}
```

- Para que las validaciones sean efectivas debo hacer la validación global en el main (**Recuerda!! Lo del whitelist!!**)

```
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.setGlobalPrefix('api/v2')

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

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

- Con el PartialType del updatePokemonDto tengo configurado el dto del update, ya que hereda del createPokemonDto y hace las propiedades opcionales
- Debo hacer validaciones en el servicio para no hacer duplicados

---

Crear Pokemon en base de datos

- Vohy a insertar este dto en la base de datos
- Paso el nombre a minúsculas
- Voy a necesitar hacer la **inyección de dependencias** en el constructor de mi **entity** pasándoselo al **Model de mongoose como genérico**
- Si quiero inyectarlo debo ponerle el decorador **@InjectModel() de @nestjs/mongoose** y pasarle el nombre (**Pokemon.name**)

```
import { Injectable } from '@nestjs/common';
import {InjectModel} from '@nestjs/mongoose'
import { CreatePokemonDto } from './dto/create-pokemon.dto';
```

```
import { UpdatePokemonDto } from './dto/update-pokemon.dto';
import { Model } from 'mongoose';
import { Pokemon } from './entities/pokemon.entity';

@Injectable()
export class PokemonService {

  constructor(
    @InjectModel(Pokemon.name) //le pongo el decorador y le paso el .name de la
entity
    private readonly pokemonModel: Model<Pokemon>){}//importo Model de mongoose y
le paso como genérico la entity

  create(createPokemonDto: CreatePokemonDto) {

    createPokemonDto.name = createPokemonDto.name.toLowerCase()

    return createPokemonDto;
  }

  findAll() {
    return `This action returns all pokemon`;
  }

  findOne(id: number) {
    return `This action returns a #${id} pokemon`;
  }

  update(id: number, updatePokemonDto: UpdatePokemonDto) {
    return `This action updates a #${id} pokemon`;
  }

  remove(id: number) {
    return `This action removes a #${id} pokemon`;
  }
}
```

- Vamos a hacer una inserción. Dejemos a un lado las validaciones, solo para crear algo facilmente
- Como las inserciones a las DB son asíncronas coloco el async

```
async create(createPokemonDto: CreatePokemonDto) {

    createPokemonDto.name = createPokemonDto.name.toLowerCase()
    const pokemon = await this.pokemonModel.create(createPokemonDto)
    return pokemon;
  }
```

- Hago una petición con ThunderClient a http://localhost:3000/api/v2/pokemon con un numero y un nombre en el body
- Refresco TablePlus con *Ctrl+R*

- Si intento volver a hacer la misma inserción me va a decir "*Internal Server Error*", por un duplicate key
- Esto vendría a suponer que desde el frontend nos digan que es error del backend, siendo **una validación** que nos hizo falta
- Cuantas **menos consultas** se hagan a la **base de datos mucho mejor**
- Vamos a manejarlo de una mejor manera

Responder un error específico

- Meto la inserción en un **try y un catch** y hago la **validación**
- Sé que si me devuelve el error 11000 es **un error de clave duplicada**. Puedo verlo con **un console.log del error**
- Puedo usar eso para **no hacer otra consulta a la base de datos**, si no tendría que usar una consulta para comprobar el número y otra para el nombre
- Si **no es un error 11000** hago un console.log del error y lanzo (ahora si) un error del server
- **NOTA**: Cuando lanzo un error **con throw new Error no hace falta colocar un return después**

```
async create(createPokemonDto: CreatePokemonDto) {

    createPokemonDto.name = createPokemonDto.name.toLowerCase()

    try {
      const pokemon = await this.pokemonModel.create(createPokemonDto)
      return pokemon;

    } catch (error) {
      if(error.code === 11000) throw new BadRequestException(`Pokemon exixts in db ${JSON.stringify(error.keyValue)}`)

      console.log(error)
      throw new InternalServerErrorException("Can't create Pokemon - Check server Logs")
    }
  }
```

- Cuando se trata de un nombre o número duplicado (error 11000), ahora puedo ver en el message que el error es del nombre duplicado o del número
- Para borrar en TablePlus seleccionar la fila, apretar **Supr** y luego **Ctrl+S** para aplicar el commit
- Ctrl+R para recargar
- Si quisiera poner otro código en lugar de un 201, poner un 200, hago uso del decorador **@HttpCode()** en el controlador
- pokemon.controller

```
@Controller('pokemon')
export class PokemonController {
  constructor(private readonly pokemonService: PokemonService) {}

  @Post()
```

```
  @HttpCode(200)
  create(@Body() createPokemonDto: CreatePokemonDto) {
    return this.pokemonService.create(createPokemonDto);
  }
///...rest of code...
```

- También tengo HttpStatus que me ofrecen los códigos de error con autocompletado

```
  @Post()
  @HttpCode(HttpStatus.OK) //HttpStatus.OK === 200
  create(@Body() createPokemonDto: CreatePokemonDto) {
    return this.pokemonService.create(createPokemonDto);
  }
```

---

findOne - Buscar

- Tenemos 3 identificadores: el nombre, el número y el id de mongo
- Si se le pasa un id y no es un mongoId va a dar un error de base de datos (hay que hacer esa validación)
- Si el nombre no existe también devuelve error
- Por id siempre vamos a recibir un string, ya que me puede enviar mongoId, el nombre y si me da un número lo recibo como string
- Le **quito el + del +id** del controlador findOne para no parsear el id a número
- Lo mismo en el servicio,**tipo el id a string**
- En el servicio, declaro la variable pokemon **de tipo Pokemon** (**entity**)
- Uso la negación con NaN para decir : si es un número
- Es un método async porque voy a consultar la DB
- pokemon.service

```
async findOne(id: string) {

  let pokemon:Pokemon

  if(!isNaN(+id)){
    pokemon = await this.pokemonModel.findOne({no:id})
  }

  return pokemon
}
```

- Si le paso un número que no existe me devuelve un status 200 aunque no encontró nada. Obviamente no queremos eso

```
if(!pokemon) throw new NotFoundException("Pokemon not found")
```

- Para el mongoId tengo que balidar que sea un mongoId válido
- Para ello tengo **isValidObjectId de mongoose**
- Agrego la condición para que lo busque si no tengo un pokemon por id
- Si no lo encuentra por id voy a intentar encontrarlo por el nombre. Uso **.trim** para eliminar posibles espacios en blanco
- Y si no lo encuentra lanza el error
- Luego se optimizará este código
- pokemon.service

```
async findOne(id: string) {

    let pokemon:Pokemon

    if(!isNaN(+id)){
      pokemon = await this.pokemonModel.findOne({no:id})
    }

    if(!pokemon && isValidObjectId(id)){
      pokemon = await this.pokemonModel.findById(id)
    }

    if(!pokemon){
      pokemon = await this.pokemonModel.findOne({name: id.toLowerCase().trim()})
    }

    if(!pokemon) throw new NotFoundException("Pokemon not found")

    return pokemon
  }
```

Actualizar Pokemon

- Si miramos en la entity vemos que el nombre está indexado y el número está indexado, por lo que es igaul de rápido que lo busquemos por nombre, por número...
- Hago **los mismos retoques con el id** para tiparlo como string
- El updatePokemonDto está heredando las propiedades (ahora opcionales) de createPokemonDto gracias a **PartialType**
- Uso el método **findOne** creado antes para encontrar el objeto pokemon (de mongo) y verificar el id
- Si viene el nombre lo paso a **lowerCase**
- Hago el update. Este **pokemon es un objeto de mongo**, por lo que tiene todos los métodos y propiedades.
- Le pongo new en true para que me devuelva el objeto actualizado, pero debo guardarlo en una variable para retornarlo
- **Pero aunque lo haga de esta manera no me devuelve el objeto actualizado si no el retorno del update como operación**

```
async update(id: string, updatePokemonDto: UpdatePokemonDto) {
        let pokemon = await this.findOne(id)

        if(updatePokemonDto.name){
          updatePokemonDto.name = updatePokemonDto.name.toLowerCase()
        }

        const updatedPokemon = await pokemon.updateOne(updatePokemonDto, {new:
true}) //aún así no me devuelve el objeto actualizado

        return updatedPokemon
  }
```

- Esparzo todas las propiedades que tiene el pokemon con el **spread.toJSON** y las sobreescribo con el spread de updatedPokemonDto

```
async update(id: string, updatePokemonDto: UpdatePokemonDto) {
      let pokemon = await this.findOne(id)

      if(updatePokemonDto.name){
        updatePokemonDto.name = updatePokemonDto.name.toLowerCase()
      }

      await pokemon.updateOne(updatePokemonDto)

      return {...pokemon.toJSON(), ...updatePokemonDto}
}
```

- Ahora **hay un problema**
- Si intento actualizar el número de un pokemon que ya existe (el 1, por ejemplo, y es bulbasur) con otro nombre me devuelve **error 11000** (de valor duplicado)

Validar valores únicos

- Meto la actualización en un try catch
- En caso de que el error sea 11000 lanzo un *BadRequestException*
- Si no imprimo el error en un console.log para debuggear y lanzo un InternalError

```
async update(id: string, updatePokemonDto: UpdatePokemonDto) {
      let pokemon = await this.findOne(id)

      if(updatePokemonDto.name){
        updatePokemonDto.name = updatePokemonDto.name.toLowerCase()
      }

      try {
```

```
        await pokemon.updateOne(updatePokemonDto)
        return {...pokemon.toJSON(), ...updatePokemonDto}

      } catch (error) {
        if(error.code === 11000){
          throw new BadRequestException(`Pokemon exists in db
${JSON.stringify(error.keyValue)}`)
        }
        console.log(error)
        throw new InternalServerErrorException("Can't update Pokemon. Check server
logs")
      }
}
```

- Voy a crear un método en el servicio para manejar los errores
- Voy a tipar el error como any para dejarlo abierto

```
private handleExceptions(error: any){
  if(error.code === 11000){
    throw new BadRequestException(`Pokemon exists in db
${JSON.stringify(error.keyValue)}`)
  }
  console.log(error)
  throw new InternalServerErrorException("Can't update Pokemon. Check server
logs")
}
```

- Sustituyo el código de error por el método
- Siempre regresa un error (siempre hay que hacer un throw)

```
import { BadRequestException, Injectable, InternalServerErrorException,
NotFoundException } from '@nestjs/common';
import {InjectModel} from '@nestjs/mongoose'
import { CreatePokemonDto } from './dto/create-pokemon.dto';
import { UpdatePokemonDto } from './dto/update-pokemon.dto';
import { Model, isValidObjectId } from 'mongoose';
import { Pokemon } from './entities/pokemon.entity';

@Injectable()
export class PokemonService {

  constructor(
    @InjectModel(Pokemon.name)
    private readonly pokemonModel: Model<Pokemon>){}

  async create(createPokemonDto: CreatePokemonDto) {

    createPokemonDto.name = createPokemonDto.name.toLowerCase()
```

```
    try {
      const pokemon = await this.pokemonModel.create(createPokemonDto)
      return pokemon;

    } catch (error) {
      this.handleExceptions(error)
    }
  }

  async findAll() {
    return this.pokemonModel.find() ;
  }

  async findOne(id: string) {

    let pokemon:Pokemon

    if(!isNaN(+id)){
      pokemon = await this.pokemonModel.findOne({no:id})
    }

    if(!pokemon && isValidObjectId(id)){
      pokemon = await this.pokemonModel.findById(id)
    }

    if(!pokemon){
      pokemon = await this.pokemonModel.findOne({name: id.toLowerCase().trim()})
    }

    if(!pokemon) throw new NotFoundException("Pokemon not found")

    return pokemon
  }

  async update(id: string, updatePokemonDto: UpdatePokemonDto) {
        let pokemon = await this.findOne(id)

        if(updatePokemonDto.name){
          updatePokemonDto.name = updatePokemonDto.name.toLowerCase()
        }

        try {
          await pokemon.updateOne(updatePokemonDto)
          return {...pokemon.toJSON(), ...updatePokemonDto}

        } catch (error) {
          this.handleExceptions(error)
        }
  }

  remove(id: number) {
    return `This action removes a #${id} pokemon`;
  }
```

```
  private handleExceptions(error: any){
    if(error.code === 11000){
      throw new BadRequestException(`Pokemon exists in db
${JSON.stringify(error.keyValue)}`)
    }
    console.log(error)
    throw new InternalServerErrorException("Can't update Pokemon. Check server
logs")
  }
}
```

Eliminar un pokemon

- Hago los retoques del tipado del id como string

```
async remove(id: string) {
  const pokemon = await this.findOne(id)
  await pokemon.deleteOne()
}
```

- Pero yo quiero implementar la lógica que **para borrar el pokemon se tenga que usar el id de mongo**
- Para ello creo un customPipe

CustomPipe - parseMongoIdPipe

- Quiero asegurarme de que el parámetro que le paso a la url sea un mongoId
- Hay una **estructura de módulo recomendada**
  - src
    - common
      - decorators
      - dtos
      - filter
      - guards
      - interceptors
      - middlewares
      - pipes
    - common.controller.ts
    - common.module.ts
    - common.service.ts
- Voy a usar el **CLI** para generar un nuevo módulo llamado common
- Por defecto viene sin servicio ni nada más que el .module

> nest g mo common

- Para crear la carpeta pipes tambien uso el CLI, le digo la carpeta dónde lo quiero (crea la carpeta pipes) y el nombre del pipe

- No le coloco Pipe al final porque lo crea nest directamente

> nest g pi common/pipes/parseMongoId

- Me crea esto.
- Implementa la interfaz de PipeTransform
- Tipo el value a string
- Hago un console.log del value y de la data

```
import { ArgumentMetadata, Injectable, PipeTransform } from '@nestjs/common';

@Injectable()
export class ParseMongoIdPipe implements PipeTransform {
  transform(value: string, metadata: ArgumentMetadata) {

    console.log({value, metadata})
  }
}
```

- Para poder obervar el *console.log* del **ParseMongoIdPipe** uso el Pipe en el controlador del delete (borro el código del servicio y dejo solo un *console.log* del id)
- pokemon.service

```
async remove(id: string) {
  //const pokemon = await this.findOne(id)
  //await pokemon.deleteOne()

  console.log({id})
}
```

- En el controlador

```
@Delete(':id')
remove(@Param('id', ParseMongoIdPipe) id: string) {
  return this.pokemonService.remove(id);
}
```

- Hago una llamada al endpoint desde ThunderClient para observar el *console.log* del value y la metadata
- Como id en la url le paso un 1. Me devuelve esto por consola

```
{
  value: '1',
  metadata: { metatype: [Function: String], type: 'param', data: 'id' }
}
```

- Los Pipes transforman la data
- Si coloco en el *return* value.toUpperCase() me devuelve el id en mayúsculas (al poner un string)
- En consola me devolverá en minúsculas porque en el momento de hacer el *console.log* no le he aplicado el toUpperCase

```
import { ArgumentMetadata, Injectable, PipeTransform } from '@nestjs/common';

@Injectable()
export class ParseMongoIdPipe implements PipeTransform {
  transform(value: string, metadata: ArgumentMetadata) {
    console.log({value, metadata})  //bulbasur    (pasado como id)

    return value.toUpperCase();     //BULBASUR
  }
}
```

- Puedo usar la metadata para hacer validaciones
- Uso el *isValidObjectId* de mongoose para hacer la validación
- Si pasa la validación retorno el value (que es el mongoId que he pasado por parámetro)

```
import { ArgumentMetadata, Injectable, PipeTransform, BadRequestException } from
'@nestjs/common';
import { isValidObjectId } from 'mongoose';

@Injectable()
export class ParseMongoIdPipe implements PipeTransform {
  transform(value: string, metadata: ArgumentMetadata) {

    if(!isValidObjectId(value))
        throw new BadRequestException(`{value} is not a valid MongoId`)

    return value;
  }
}
```

- En el servicio escribo la lógica de negocio

```
async remove(id: string) {

  const pokemon=  await this.pokemonModel.findByIdAndDelete(id)
    return pokemon
  }
```

- De esta manera obtengo el pokemon eliminado. Pero **hay un problema** al hacerlo así
- Si desde el frontend me envían un mongoId válido pero que no existe, el status que me devuelve el remove es un *200*

- Pero la verdad es que no encontró el pokemon
- Quiero **evitar hacer otra consulta a la db**

Validar y eliminar en una sola consulta

```
async remove(id: string) {

   const result = await this.pokemonModel.deleteOne({_id: id})

   return result
  }
```

- Esto me devuelve un valor **"deletedCount"** en 0 si no ha borrado ningún registro, y el acknowledged (boolean) si realizó el procedimiento
- Puedo **desestructurarlo** de la llamada a la db

```
async remove(id: string) {

  const {deletedCount} = await this.pokemonModel.deleteOne({_id: id})

  if(deletedCount === 0){
  throw new BadRequestException(`Pokemon with id ${id} not found`)
  }

  return
}
```

- Podríamos envolver esta llamada de eliminación en un try catch
- Más adelante se creará un *ExceptionFilter* para filtrar todos los endpoints
- Falta el *findAll*, donde haremos paginación y búsqueda mediante expresiones regulares
- También haremos el SEED, sacaremos la data de PokeAPI

> https://pokeapi.co/api/v2/pokemon?limit=500

- Le establezco un límite a la data de 500 pokemon

---

05 NEST SEED PAGINACIÓN

- En esta sección, además de crear el resource del SEED y hacer la paginación, crearemos la documentación

---

Crear módulo de SEED

- El SEED me ayudará a que si viene otro desarrollador pueda hacer las pruebas pertinentes con data en la DB
- También es útil por si empleo algo de código destructivo, poder reestablecer la DB
- El SEED es conveniente en desarrollo

> nest g res seed --no-spec

- Digo de hacer un REST API con los entry points (y)
- Borro dtos ya que no los voy a usar (podría quererlos para que el SEED se cree con ciertos argumentos)
- Borro los dtos del controller, dejo solo el método GET. Lo renombro a executeSEED
- Lo mismo con el service
- Borro la entity

NOTA: Axios está dando problemas con Nest (cannot read properties of undefined). Ser recomienda instalar la versión 0.27.2 hasta que se libere una versión superior sin este problema

> npm i axios@0.27.2

Realizar petición http desde Nest

- Lo que quiero es apuntar a este endpoint de pokeAPI y traerme 500 pokemons e insertarlos en la db

> https://pokeapi.co/api/v2/pokemon?limit=500

- Borro todos los registros de la DB
- En la data me devuelve un nombre, una url (campo que no tengo en la db) y no tengo el número
- Solo dispongo del **fetch en versiones de node 18** o superior
- Quiero tener una forma sencilla de cambiar de fetch a axios u otra librería de peticiones http y no tener que refactorizar de arriba a abajo
- Primero se va a escribir el código tal cómo sale y luego aplicaremos el **patrón adaptador** para esto
- Creo una instancia de axios, para que sea claramente visible que estoy usando axios y no sea una dependencia oculta
- No está inyectada pero es una dependencia de mi servicio
- Puedo desestructurar la data de la respuesta. En results tengo el array de pokemons
- seed.service

```
import { Injectable } from '@nestjs/common';
import axios, { AxiosInstance } from 'axios';


@Injectable()
export class SeedService {

  private readonly axios: AxiosInstance = axios

  async executeSEED() {
    const {data} = await this.axios.get("https://pokeapi.co/api/v2/pokemon?
```

```
limit=500")

    return data;  //en data.results tengo el array de pokemons
  }
}
```

- Apunto al GET de http://localhost:3000/api/v2/seed y obtengo la data
- Pero como decía antes solo tengo el nombre y la url (que no tengo especificada en mi db)
- Quiero tener el tipado de dato de esta respuesta
- Copio la respuesta de data
- Creo la carpeta interfaces en seed/interfaces/poke-response.interface.ts
- Necesito la extension Paste JSON as Code en VSCode
- Ctrl+Shift+P , Paste JSON as code, selecciono Typescript, me pide que nombre la interfaz del nivel superior, la llamo PokeResponse, y con el resultado de la respuesta copiado en el portapapeles (Ctrl+C) le doy a Enter
- Automáticamente me saca las interfaces

```
export interface PokeResponse {
    count:    number;
    next:     string;
    previous: null;
    results:  Result[];
}

export interface Result {
    name: string;
    url:  string;
}
```

- Uso la interfaz y tipo el get cómo un genérico de respuesta tipo PokeResponse

```
import { Injectable } from '@nestjs/common';
import axios, { AxiosInstance } from 'axios';
import { PokeResponse } from './interfaces/poke-response.interface';


@Injectable()
export class SeedService {

  private readonly axios: AxiosInstance = axios

  async executeSEED() {

   const {data}= await this.axios.get<PokeResponse>
("https://pokeapi.co/api/v2/pokemon?limit=500")

    return data;
```

```
  }
}
```

- Ahora si le añado un punto a la data tengo el **autocompletado**.
- Si pongo data.results[0]. tengo el autocompletado de name y url
- Voy a manipular la data para poder insertarla en la db
- El name es fácil, pero el número está dentro de la url de cada pokemon, después de /api/v2/pokemon/número_del_pokemon/lkjj
- Desestructro el name y el url en el forEach
- Uso el **split** para separar por /. Puedo hacer un console.log de la url para ver **en qué posición** queda el número del pokemon
- El número está en el penúltimo lugar, uso **.length** -2 (-1 sería el último)
- Cómo debe ser de tipo número, uso el **+** para parsearlo a número
- seed.service

```
import { Injectable } from '@nestjs/common';
import axios, { AxiosInstance } from 'axios';
import { PokeResponse } from './interfaces/poke-response.interface';


@Injectable()
export class SeedService {

  private readonly axios: AxiosInstance = axios

  async executeSEED() {

   const {data}= await this.axios.get<PokeResponse>
("https://pokeapi.co/api/v2/pokemon?limit=10")

   data.results.forEach(({name, url})=>{
      const segments = url.split('/')
      const no: number = +segments[segments.length -2] //el número está en la
penúltima posición de la url
      console.log({name, no})
   })

    return data;
  }
}
```

- Vamos a crear un **Provider** para poder reemplazar facilmente axios, por request, o cualquier otra librería
- Luego implementaremos este patrón adaptador. Por ahora lo manejo sencillo

---

Insertar Pokemons por lote

- Para insertar el nombre y el número usaré la inyección del modelo de Pokemon y el metodo create
- Inyecto el modelo en el constructor del servicio **SeedService**

```
constructor(
    @InjectModel(Pokemon.name)
    private readonly pokemonModel: Model<Pokemon>){}
```

- PokemonModule debe de estar disponible en SeedModule
- Debo exportarlo de PokemonModule e importarlo en SeedModule
- Con exportar el MongooseModel es suficiente, no necesito el forFeature y el codigo interno porque ya lo exporta como tal
- pokemon.module

```
import { Module } from '@nestjs/common';
import { PokemonService } from './pokemon.service';
import { PokemonController } from './pokemon.controller';
import { MongooseModule } from '@nestjs/mongoose';
import { Pokemon, PokemonSchema } from './entities/pokemon.entity';


@Module({
  imports:[
    MongooseModule.forFeature([
      {
      name: Pokemon.name,
      schema: PokemonSchema
      }
    ])
  ],
  controllers: [PokemonController],
  providers: [PokemonService],
  exports:[MongooseModule] //exporto MongooseModule
})
export class PokemonModule {}
```

- Importo el PokemonModule (que está exportando el MongooseModel) en el seed.module

```
import { Module } from '@nestjs/common';
import { SeedService } from './seed.service';
import { SeedController } from './seed.controller';
import { PokemonModule } from 'src/pokemon/pokemon.module';

@Module({
  imports:[PokemonModule],
  controllers: [SeedController],
  providers: [SeedService]
```

```
})
export class SeedModule {}
```

- Debo usar el async en el foreach para poder usar el await para la inserción

```
import { Injectable } from '@nestjs/common';
import axios, { AxiosInstance } from 'axios';
import { PokeResponse } from './interfaces/poke-response.interface';
import { Model } from 'mongoose';
import { Pokemon } from 'src/pokemon/entities/pokemon.entity';
import { InjectModel } from '@nestjs/mongoose';


@Injectable()
export class SeedService {

  constructor(
    @InjectModel(Pokemon.name)
    private readonly pokemonModel: Model<Pokemon>){}

  private readonly axios: AxiosInstance = axios

  async executeSEED() {

   const {data}= await this.axios.get<PokeResponse>
("https://pokeapi.co/api/v2/pokemon?limit=10")

   data.results.forEach(async ({name, url})=>{
      const segments = url.split('/')
      const no: number = +segments[segments.length -2]

      const pokemon = await this.pokemonModel.create({name, no})
   })

    return 'Seed Executed';
  }
}
```

- Si ahora hago un GET a http://localhost:3000/api/v2/seed tengo mis 10 pokemons en la DB
- El problema es que si ahora tuviera que hacer 1000 inserciones demoraría mucho tiempo y consumiría recursos
- Se pueden hacer todas las inserciones de manera consecutiva de varias maneras más eficientes
- En la próxima lección

Insertar Múltiples registros simultáneamente

- Voy a expresar de dos maneras diferentes la inserción en la tabla
- Para evitar errores primero borro todo lo que haya en la db

- Manera 1

```
async executeSEED() {

    //borro lo que haya en la db
    await this.pokemonModel.deleteMany()

   const {data}= await this.axios.get<PokeResponse>
("https://pokeapi.co/api/v2/pokemon?limit=10")

   const insertPromisesArray = []

   data.results.forEach(async ({name, url})=>{
      const segments = url.split('/')
      const no: number = +segments[segments.length -2]

      insertPromisesArray.push(
        this.pokemonModel.create({name, no})
      )
   })

   //una vez tengo todas las promesas en el array
      await Promise.all( insertPromisesArray)

    return 'Seed Executed';
  }
```

- Manera 2. Con insertMany. Recomendada

```
import { Injectable } from '@nestjs/common';
import axios, { AxiosInstance } from 'axios';
import { PokeResponse } from './interfaces/poke-response.interface';
import { Model } from 'mongoose';
import { Pokemon } from 'src/pokemon/entities/pokemon.entity';
import { InjectModel } from '@nestjs/mongoose';


@Injectable()
export class SeedService {

  constructor(
    @InjectModel(Pokemon.name)
    private readonly pokemonModel: Model<Pokemon>){}

  private readonly axios: AxiosInstance = axios

  async executeSEED() {

    await this.pokemonModel.deleteMany({})
```

```
    const {data}= await this.axios.get<PokeResponse>
("https://pokeapi.co/api/v2/pokemon?limit=500")

    const pokemonToInsert: {name: string, no:number}[] = []

    data.results.forEach(async ({name, url})=>{
       const segments = url.split('/')
       const no: number = +segments[segments.length -2]

     pokemonToInsert.push({name, no})
    })

    await this.pokemonModel.insertMany(pokemonToInsert)

     return 'Seed Executed';
  }
}
```

- Amplio el README con el comando para usar el SEED
- README

```
<p align="center">
  <a href="http://nestjs.com/" target="blank"><img
src="https://nestjs.com/img/logo-small.svg" width="200" alt="Nest Logo" /></a>
</p>

# Ejecutar en desarrollo

1. Clonar el repositorio
2. Ejecutar
```
npm i
```

3. Tener el Nest CLI instalado
```
npm i -g @nestjs/cli
```
4. Levantar la base de datos
```
docker-compose up -d
```
5. Reconstruir la base de datos con la semilla
```
http://localhost:3000/api/v2/seed
```

## Stack Usado

- MongoDB
- Nest
```

Crear un custom Provider (patrón adaptador)

- Queremos que el cambio sea lo más indoloro posible
- Voy a crear un adaptador que va **a envolver axios**, para que en lugar de tener código de terceros incrustado en mi app, tenga el mio
- Quiero sacar esa instancia de axios y crearme **mi propia implementación de una clase**
- Lo creo dentro de *common*
- Va a ser un provider (porque va a poder inyectarse)
- Los **providers** tienen que estar **definidos en el módulo**
- En common creo la carpeta interface y otra llamada adapters
- En interfaces voy a crear http-adapter.interface.ts
- La clase que implemente esta interfaz va a tener el método get que devuelve una promesa de tipo genérico
- El método get está esperando o puede recibir un genérico. **Que la respuesta es de este tipo de dato**

```
export interface HttpAdapter{
    get<T>(url: string): Promise<T>
}
```

- En adapters creo axios.adapter.ts
- Esta clase va aenvolver mi código, para que si tengo que cambiar la implementación sólo tenga que cambiar la clase
- Implemento la interfaz. La clase debe tener el método get
- Creo la instancia de axios
- Meto en un try y un catch el get y desestructuro la data. La retorno
- Lanzo un error en el catch
- Debo añadirle el operador **@Injectable** para poder inyectarlo

```
import axios, { AxiosInstance } from "axios";
import { HttpAdapter } from "../interfaces/http-adapter.interface";
import {Injectable} from '@nestjs/common'


@Injectable()
export class AxiosAdapter implements HttpAdapter{

    private axios: AxiosInstance = axios

    async get<T>(url: string): Promise<T> {
        try {
            const {data}=  await this.axios.get<T>(url)
            return data

        } catch (error) {
            throw new Error('This is an error - Check Logs')
```

```
        }
    }
}
```

- Los providers están a nivel de módulo. Para que sea visible por otros módulos **tengo que exportarlo**
- Lo hago dentro del decorador **@Module({})**

```
import { Module } from '@nestjs/common';
import { AxiosAdapter } from './adapters/axios.adapter';

@Module({
    providers:[AxiosAdapter],
    exports:[AxiosAdapter]
})
export class CommonModule {}
```

- Importo el CommonModule en el módulo de seed
- seed.module

```
import { Module } from '@nestjs/common';
import { SeedService } from './seed.service';
import { SeedController } from './seed.controller';
import { PokemonModule } from 'src/pokemon/pokemon.module';
import { CommonModule } from 'src/common/common.module';

@Module({
  imports:[PokemonModule, CommonModule],
  controllers: [SeedController],
  providers: [SeedService]
})
export class SeedModule {}
```

- Ya puedo usar este AxiosAdapter!
- Lo inyecto en el servicio de seed
- No hace falta que desestructure la data porque ya lo he hecho en el adaptador
- Si quiero la respuesta entera de axios lo guardo en una variable en lugar de desestruturarla

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

import { PokeResponse } from './interfaces/poke-response.interface';
import { Model } from 'mongoose';
import { Pokemon } from 'src/pokemon/entities/pokemon.entity';
import { InjectModel } from '@nestjs/mongoose';
import { AxiosAdapter } from 'src/common/adapters/axios.adapter';
```

```
@Injectable()
export class SeedService {

  constructor(
    @InjectModel(Pokemon.name)
    private readonly pokemonModel: Model<Pokemon>,
    //inyecto el adaptador
    private readonly http: AxiosAdapter
    ){}



  async executeSEED() {

   await this.pokemonModel.deleteMany({})

   //guardo la data del método get
   const data = await this.http.get<PokeResponse>
("https://pokeapi.co/api/v2/pokemon?limit=500")

   const pokemonToInsert: {name: string, no:number}[] = []

   data.results.forEach(async ({name, url})=>{
      const segments = url.split('/')
      const no: number = +segments[segments.length -2]

    pokemonToInsert.push({name, no})
   })

   await this.pokemonModel.insertMany(pokemonToInsert)

    return 'Seed Executed';
  }
}
```

- Si ahora quisiera usar otro adaptador, debería crearlo, hacer que implemente la interfaz, hacer el export correspondiente e inyectarlo

---

Paginación Pokemons

- Mediante query parameters puedo indicarle cuántos pokemons quiero por página
- Implementaremos el offset (los siguientes x) y el limit (x pokemons)
- Para que me traiga 2 pokemons de los siguientes 20

> https://pokeapi.co/api/v2/pokemon?offset=20&limit=2

- Si quiero solo 5 pokemons de los siguientes 5 lo haría asi
- pokemon.service

```
async findAll() {
    return await this.pokemonModel.find()
    .limit(5)
    .skip(5) ;
  }
```

- Entonces lo que necesito es extraer los query parameters de la url
- Vamos a necesitar un nuevo dto, para implementar algunas reglas como que tiene que ser un número, tiene que ser positivo...
- Obtengo los query parameters mediante el decorador **@Query()**
- pokemon.controller

```
  @Get()
  findAll(@Query() paginationDto: PaginationDto) {
    return this.pokemonService.findAll(paginationDto);  //le paso el Dto al
servicio para trabajar con la paginación
  }
```

- Creo el dto. Tiene más sentido crearlo en la carpeta **common** ya que es un dto muy genérico y puedo querer usarlo en otros lugares

```
import { IsOptional, IsPositive, Min, IsNumber } from "class-validator"

export class PaginationDto{

    @IsPositive()
    @IsOptional()
    @IsNumber()
    @Min(1)
    limit?: number  //le añado ? para que TypeScript lo considere opcional

    @IsPositive()
    @IsOptional()
    @IsNumber()
    offset?: number
}
```

- Ahora falta parsear el query parameter que me llega cómo un string a número y acabar la paginación

---

Transform Dtos

- Los query parameters, el body... todo va **como string**
- Puedo hacer la transformación **de manera global en el main**
- Dentro del ValidationPipe, le pongo el **transform** en true. También añado **enableImplicitConversion**

```
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.setGlobalPrefix('api/v2')

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

      //añado transform y enableImplicitConversion
      transform: true,
      transformOptions:{
        enableImplicitConversion: true
      }
    })
  )

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

- Desarrollo la lógica en el servicio
- Desestructuro **limit y offset** que vienen en el dto y les asigno **un valor por defecto** por si no vienen
- Los agrego a los métodos

```
async findAll(paginationDto: PaginationDto) {

  const {limit=10, offset=0}= paginationDto

  return await this.pokemonModel.find()
  .limit(limit)
  .skip(offset) ;
}
```

- Para ordenarlos alfabéticamente puedo usar el método **sort**
- Le digo que ordene la columna no de manera ascendente
- Puedo hacer el select de las columnas y restarle el __v para que no lo muestre

```
async findAll(paginationDto: PaginationDto) {

    const {limit=10, offset=0}= paginationDto

    return await this.pokemonModel.find()
```

```
    .limit(limit)
    .skip(offset)
    .sort({
      no:1 //le digo que ordene la columna numero de manera ascendente
    })
    .select('__v') ;
  }
```

NEST VARIABLES DE ENTORNO - DEPLOY

- Levanto la db

```
docker-compose up -d
```

- Levanto el servidor

```
npm run start:dev
```

- Para sembrar la base de datos hago una petición GET al endpoint

```
http://localhost:3000/api/v2/seed
```

- Todo listo!

Configuración de variables de entorno

- El string de conexión con la db y el puerto dónde escucha el servidor deben ser variables de entorno
- Creo el archivo .env en la raíz
- Lo añado al .gitignore para no darle seguimiento

```
PORT=3000
MONGODB=mongodb://localhost:27017/nest-pokemon
```

- Node ya tiene sus variables de entorno
- Puedo colocar un console.log en el constructor de AppModule para visualizarlas

```
import { join } from 'path';
import { Module } from '@nestjs/common';
import {ServeStaticModule} from '@nestjs/serve-static'
import { PokemonModule } from './pokemon/pokemon.module';
import { MongooseModule } from '@nestjs/mongoose';
import { CommonModule } from './common/common.module';
import { SeedModule } from './seed/seed.module';


@Module({
```

```
  imports: [
    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'public')
    }),
    MongooseModule.forRoot('mongodb://localhost:27017/nest-pokemon'),
    PokemonModule,
    CommonModule,
    SeedModule
  ]
})
export class AppModule {
  constructor(){
    console.log(process.env)
  }
}
```

- Hay un monton! Pero **no aparecen las que yo he creado**
- Para decirle a Nest dónde están las variables de interno hago la **instalación**

> npm i @nestjs/config

- Hay que importar en el app.module el **ConfigModule**
- La posición dónde se coloca es importante. Lo coloco al inicio

```
import { join } from 'path';
import { Module } from '@nestjs/common';
import {ServeStaticModule} from '@nestjs/serve-static'
import { PokemonModule } from './pokemon/pokemon.module';
import { MongooseModule } from '@nestjs/mongoose';
import { CommonModule } from './common/common.module';
import { SeedModule } from './seed/seed.module';
import { ConfigModule } from '@nestjs/config';


@Module({
  imports: [
    ConfigModule.forRoot(),
    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'public')
    }),
    MongooseModule.forRoot('mongodb://localhost:27017/nest-pokemon'),
    PokemonModule,
    CommonModule,
    SeedModule
  ]
})
export class AppModule {
  constructor(){
    console.log(process.env)
  }
}
```

- Ahora puedo observar en el console.log que las variables que yo he creado están disponibles
- Para usar las variables PORT y MONGODB solo tengo que escribir **process.env.PORT**, **process.env.MONGODB** donde corresponde
- **OJO! Que PORT aparece como un string en la variable de entorno**. Todas las variables de entorno **son strings**
    - Debo parsearlo a número con **+**

Configuration Loader

- El ConfigModule ofrece un servicio que permite inyectar las variables de entorno
- Para probarlo escribo una nueva variable de entorno que será el limit de la paginación

```
default_limit=5
```

- Si agrego un console.log(process.env.DEFAULT_LIMIT) en el constructor del PokemonService lo imprime en consola

```
@Injectable()
export class PokemonService {

  constructor(
    @InjectModel(Pokemon.name)
    private readonly pokemonModel: Model<Pokemon>){
      console.log(process.env.DEFAULT_LIMIT)
    }
```

- Si agrego directamente la variable de entorno, podría ser que viniera undefined y esto crasheara mi app
- pokemon.service

```
async findAll(paginationDto: PaginationDto) {

const {limit=+process.env.DEFAULT_LIMIT, offset=0}= paginationDto

return await this.pokemonModel.find()
.limit(limit)
.skip(offset)
.sort({
    no:1
}) ;
}
```

- En este caso funciona porque resulta un NaN y JavaScript lo considera un número, lo que no hace el límite y me muestra todos los pokemons

- Pero no es un comportamiento deseable
- Lo que se enseña a continuación vale para la mayoría de casos
- Más adelante se enseñará como ser más estricto y que devuelva un error si no está bien configurado
- Creo en /src la carpeta config con app.config.ts
- Voy a **exportar una función** que va a **mapear mis variables de entorno**
- Regreso **un objeto** entre paréntesis (return implícito)
- En el caso que no esté **NODE_ENV** (que siempre va a estar) lo voy a colocar como 'dev' (desarrollo)
- Coloco el MONGODB como una propiedad de un objeto por lo que lo coloco en lowerCase
- app.config.ts

```
export const EnvConfiguration = ()=>({
    environment: process.env.NODE_ENV || 'dev',
    mongodb: process.env.MONGODB,
    port: process.env.PORT || 3001,
    defaultLimit: process.env.DEFAULT_LIMIT || 5
})
```

- Tengo que decirle a Nest que va a usar este archivo (carga esto!)
- Para ello voy a ConfigModule en app.module

```
import { join } from 'path';
import { Module } from '@nestjs/common';
import {ServeStaticModule} from '@nestjs/serve-static'
import { PokemonModule } from './pokemon/pokemon.module';
import { MongooseModule } from '@nestjs/mongoose';
import { CommonModule } from './common/common.module';
import { SeedModule } from './seed/seed.module';
import { ConfigModule } from '@nestjs/config';
import { EnvConfiguration } from './config/app.config';


@Module({
  imports: [
    ConfigModule.forRoot({
      load: [EnvConfiguration]
    }),
    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'public')
    }),
    MongooseModule.forRoot(process.env.MONGODB),
    PokemonModule,
    CommonModule,
    SeedModule
  ]
})
export class AppModule {
  constructor(){
    console.log(process.env)
```

```
  }
}
```

- No vamos a usar las variables de entorno mediante el process.env
- Las vamos a usar mediante un servicio que proporciona el ConfigModule

---

ConfigurationService

- Inyecto en el constructor del pokemon.service

```
import {ConfigService} from '@nestjs/config'

@Injectable()
export class PokemonService {

  constructor(
    @InjectModel(Pokemon.name)
    private readonly pokemonModel: Model<Pokemon>,
    private readonly configService: ConfigService
    ){}
 }
```

- Esto por si solo da error. Dice que la primera dependencia si puede resolverla, pero no la segunda (en el índice 1)

```
Nest can't resolve dependencies of the PokemonService (PokemonModel, ?). Please
make sure that the argument ConfigService at index [1] is available in the
PokemonModule context.

Potential solutions:
- Is PokemonModule a valid NestJS module?
- If ConfigService is a provider, is it part of the current PokemonModule?
- If ConfigService is exported from a separate @Module, is that module imported
within PokemonModule?
  @Module({
    imports: [ /* the Module containing ConfigService */ ]
  })
```

- El error nos dice que hay que importar el módulo que contenga el ConfigService
- Lo importo de **@nestjs/config**

```
import { Module } from '@nestjs/common';
import { PokemonService } from './pokemon.service';
import { PokemonController } from './pokemon.controller';
import { MongooseModule } from '@nestjs/mongoose';
import { Pokemon, PokemonSchema } from './entities/pokemon.entity';
```

```
import {ConfigModule} from '@nestjs/config'

@Module({
  imports:[
    ConfigModule,
    MongooseModule.forFeature([
      {
      name: Pokemon.name,
      schema: PokemonSchema
      }
    ])
  ],
  controllers: [PokemonController],
  providers: [PokemonService],
  exports:[MongooseModule]
})
export class PokemonModule {}
```

- Para usar las variables de entorno en *PokemonService* uso la inyección de dependencia configService
- Me da dos métodos **get** y **getOrThrow** (si no lo obtiene le decimos que lance un error)
    - Con getOrThrow, si le paso una key que no existe me lanzará un error. Ex: getOrThrow('jwt-seed)

```
async findAll(paginationDto: PaginationDto) {

const {limit=this.configService.get('defaultLimit') , offset=0}= paginationDto

return await this.pokemonModel.find()
.limit(limit)
.skip(offset)
.sort({
    no:1
}) ;
}
```

- Si hago un console.log del this.configService.get('defaultLimit') puedo ver que lo obtengo **como un número**
- configService.get es de tipo genérico, le puedo especificar que es de tipo number y guardarlo en una variable (o directamente en la desestructuración)
    - **OJO que esto no hace ninguna conversión, es solo para decírselo a TypeScript**
- Para que quede más claro, declaro la propiedad como private arriba de todo y la inicializo en el constructor

```
import { BadRequestException, Injectable, InternalServerErrorException,
NotFoundException } from '@nestjs/common';
import {InjectModel} from '@nestjs/mongoose'
import { CreatePokemonDto } from './dto/create-pokemon.dto';
import { UpdatePokemonDto } from './dto/update-pokemon.dto';
import { Model, isValidObjectId } from 'mongoose';
```

```
import { Pokemon } from './entities/pokemon.entity';
import { PaginationDto } from 'src/common/dto/pagination.dto';
import {ConfigService} from '@nestjs/config'

@Injectable()
export class PokemonService {

  private defaultLimit: number     //declaro la propiedad

  constructor(
    @InjectModel(Pokemon.name)
    private readonly pokemonModel: Model<Pokemon>,
    private readonly configService: ConfigService
    ){

      this.defaultLimit= configService.get<number>('defaultLimit') //la inicializo
    }


  async create(createPokemonDto: CreatePokemonDto) {

    createPokemonDto.name = createPokemonDto.name.toLowerCase()

    try {
      const pokemon = await this.pokemonModel.create(createPokemonDto)
      return pokemon;

    } catch (error) {
      this.handleExceptions(error)
    }
  }

  async findAll(paginationDto: PaginationDto) {
    const {limit=this.defaultLimit , offset=0}= paginationDto

    return await this.pokemonModel.find()
    .limit(limit)
    .skip(offset)
    .sort({
      no:1
    }) ;
  }

  async findOne(id: string) {

    let pokemon:Pokemon

    if(!isNaN(+id)){
      pokemon = await this.pokemonModel.findOne({no:id})
    }

    if(!pokemon && isValidObjectId(id)){
      pokemon = await this.pokemonModel.findById(id)
    }
```

```
    if(!pokemon){
      pokemon = await this.pokemonModel.findOne({name: id.toLowerCase().trim()})
    }

    if(!pokemon) throw new NotFoundException("Pokemon not found")

    return pokemon
  }

  async update(id: string, updatePokemonDto: UpdatePokemonDto) {
        let pokemon = await this.findOne(id)

        if(updatePokemonDto.name){
          updatePokemonDto.name = updatePokemonDto.name.toLowerCase()
        }

        try {
          await pokemon.updateOne(updatePokemonDto)
          return {...pokemon.toJSON(), ...updatePokemonDto}

        } catch (error) {
          this.handleExceptions(error)
        }
  }

  async remove(id: string) {

   const {deletedCount} = await this.pokemonModel.deleteOne({_id: id})

   if(deletedCount === 0){
    throw new BadRequestException(`Pokemon with id ${id} not found`)
   }

   return
  }

  private handleExceptions(error: any){
    if(error.code === 11000){
      throw new BadRequestException(`Pokemon exists in db
${JSON.stringify(error.keyValue)}`)
    }
    console.log(error)
    throw new InternalServerErrorException("Can't update Pokemon. Check server
logs")
  }
}
```

- Si la variable de entorno no está definida, **tomará el valor que puse en app.config**
- Si no tengo definida un string de conexión en MONGODB, Nest va a intentar conectarse varias veces sin éxito hasta lanzar un error definitivo

- Deberíamos decirle a la persona que está tratando de levantar la aplicación que tiene que configurar la variable de entorno
- También hay otras formas de establecer valores por defecto
- En el main **no puedo hacer inyección de dependencias** para obtener el PORT, ya que esta fuera del building block
- En este punto puedo usar el **process.env.PORT**
- 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.setGlobalPrefix('api/v2')

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

  await app.listen(process.env.PORT); //uso process.env
}
bootstrap();
```

- Si no tuviera definida la variable PORT daría *undefined*. No sirve la configuración de app.config porque **no está en el building block de Nest**, por lo que no puedo usar el servicio
- En la siguiente lección vamos a establecer unas reglas de validación mediante un Validation Schema, para poder lanzar un error si una de las variables de entorno falla

---

Joi ValidationSchema

- Cuando queremos ser más estrictos y que lance errores en el caso de que un tipo de dato no venga o no sea el esperado podemos usar joi

```
npm i joi
```

- Creo en src/config/joi.validation.ts
- Debo importarlo de esta manera porque solo importando joi no funciona

```
import * as Joi from 'joi'
```

- Básicamente quiero crear un Validation Schema

```
import * as Joi from 'joi'


export const joiValidationSchema = Joi.object({
    MONGODB: Joi.required(),
    PORT: Joi.number().default(3005), //le establezco el puerto 3005 por defecto
    DEFAULT_LIMIT: Joi.number().default(5)
})
```

- **Dónde uso este joiValidationSchema?**
- El ConfigurationLoader puede trabajar de la mano de joi
- En ConfigModule (en app.module)

```
import { join } from 'path';
import { Module } from '@nestjs/common';
import {ServeStaticModule} from '@nestjs/serve-static'
import { PokemonModule } from './pokemon/pokemon.module';
import { MongooseModule } from '@nestjs/mongoose';
import { CommonModule } from './common/common.module';
import { SeedModule } from './seed/seed.module';
import { ConfigModule } from '@nestjs/config';
import { EnvConfiguration } from './config/app.config';
import { joiValidationSchema } from './config/joi.validation';


@Module({
  imports: [
    ConfigModule.forRoot({
      load: [EnvConfiguration],
      validationSchema: joiValidationSchema
    }),
    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'public')
    }),
    MongooseModule.forRoot(process.env.MONGODB),
    PokemonModule,
    CommonModule,
    SeedModule
  ]
})
export class AppModule {}
```

- Pueden trabajar **conjuntamente** el EnvConfiguration con joinValidationSchema
- Prevalece el joinValidationSchema sobre EnvConfiguration. Es decir:

    - Si le doy un valor por default al Schema y pongamos que no viene la variable de entorno, este la setea y para cuando llega a EnvConfiguration, ya esta seteada por el Schema
- Esto hace que trabaje la variable de entorno **como string** (porque está seteada en process.env.VARIABLE) y las variables de entorno **siempre son strings**
- Por ello parseo (por si acaso) la variable

```
export const EnvConfiguration = ()=>({
    environment: process.env.NODE_ENV || 'dev',
    mongodb: process.env.MONGODB,
    port: +process.env.PORT || 3001,
    defaultLimit: +process.env.DEFAULT_LIMIT || 5
})
```

ENV Template README

- Se aconseja que si la app tiene variables de entorno no le dejemos todo el trabajo al nuevo desarrollador o que las adivine
- Debo especificar las variables de entorno necesarias
- Copio el .env a .env.template
- Si hubiera cosas como el SECRET_KEY no lo llenaríamos, solo dejaríamos la definición
- El .env.template si va a estar en el repositorio
- Lo describo en el README. Añado los pasos 5,6,7

````
<p align="center">
  <a href="http://nestjs.com/" target="blank"><img
src="https://nestjs.com/img/logo-small.svg" width="200" alt="Nest Logo" /></a>
</p>

# Ejecutar en desarrollo

1. Clonar el repositorio
2. Ejecutar
```
npm i
```

3. Tener el Nest CLI instalado
```
npm i -g @nestjs/cli
```
4. Levantar la base de datos
```
docker-compose up -d
```
5. Clonar el archivo .env.template y renombrar la copia a .env

6. LLenar las variables de entorno definidas en el .env
````

````
7. Ejecutar la app en dev:
```
npm run start:dev
```

8. Reconstruir la base de datos con la semilla
```
http://localhost:3000/api/v2/seed
```

## Stack Usado

- MongoDB
- Nest
````

- **NOTA**: Para desplegar la aplicación crear la base de datos en MongoDBAtlas. Cambiar la cadena de conexión en .env y en TablePlus. Para desplegar ejecutar el script build. Después la app se ejecuta con start:prod. Así lo ejecuta ya como javascript
- Plataformas como Heroku ejecutan **directamente el comando build y luego el start**
- Por ello cambio los archivos start así

```
"start": "node dist/main",
"start:prod": "nest start"
```

- Debo asegurarme de que el puerto sea process.env.PORT para que le asigne el 3000
- Ingreso en Heroku
- Hago click en New (crear nueva app)
- Le pongo el nombre, el País
- La manera más fácil de subir el código es usar Heroku Git
- Para ello debo usar el Heroku CLI (busco la instalación en la web)
- Para saber la versión:

> heroku -v

- me situo en la carpeta de proyecto en la terminal

> heroku login

- Para subir el código escribo esta linea con el nombre que le puse a la app de heroku
- Debo haber hecho git init, con el último commit y estar en la carpeta de proyecto

> heroku git:remote -a pokedex-migue

> git add . git commit -am "commit a heroku" git push heroku main

- A veces da error el package-lock.json (colócalo en el .gitignore!)
- Para ver los logs (y los posibles errores)

> heroku logs --tail

- En Heroku logeado, en el proyecto, Settings, Reveal Config Varsç
- Aquí puedo setear las variables de entorno
- Ahora puedo hacer una petición al endpoint que me dio heroku, por ejemplo un GET

```
https://pokedex-migue.herokuapp.com/api/v2/pokemon
```

- Debo hacer el seed para llenar la db de MongoDBAtlas de data!!!
- Si hago algún cambio, puedo hacer otra vez el proceso de git add .
- Si quiero hacer un redespliegue sin ninguna modificación

```
git commit --allow-empty -m "Heroku redeploy" git push heroku main
```

- Es main o master, se recomienda cambiar la branch master a main

---

07 NEST BONUS DOCKERIZAR APP

El DockerFile

- Elimino la db pokedex del container de DockerDesktop
- Una vez tengo la imagen solo necesito este comando.
    - Especifico el archivo con -f y le asigno un nuevo .env de producción

```
docker-compose -f docker-compose.prod.yaml --env-file .env.prod up
```

- Creo el Dockerfile en la raíz
- Para construir estas imágenes (la de mongo, por ejemplo) es este procedimiento
- Este Dockerfile es una **versión básica**. En la siguiente lección hay una versión mas **PRO**
- Usualmente vamos a construir imággenes basadas en otras imágenes
    - De aquí el **FROM node:18**, alpine es una imagen super liviana de linux (6MB) con características esenciales
        - Es como tener un linux con node instalado
    - Creo un working directory con **RUN mkdir** (ejecuto el comando mkdir) /var/www/pokedex
    - Le digo que trabaje en el directorio que he creado con **WORKDIR**
- **NOTA**: es cómo tener un SO con el que te comunicas a través de comandos, sin interfaz visual
    - Le digo que copie todo lo que hay con **COPY .** (origen) ./var/www/pokedex (destino)
    - **COPY** los archivos package.json,ts-config, etc. **el último es el path**
        - En teoría no sería necesario pero lo hago por si acaso
    - **RUN** (ejecuta) npm i --prod y npm run build. El build construye la carpeta dist/
        - Debo asegurarme que no tengo la carpeta dist para que no se copie con el comando **COPY .**
            - El dist no lo quiero copiar porque lo voy a ejecutar yo
            - No quiero la data de mongo porque voy a crear la imagen sin el volumen (la data de la db guardada en el directorio)
        - Tampoco quiero los node_modules ya que son especificos del SO
        - Para ignorar estos archivos y carpetas voy a crear el .dockerignore
- .dockerignore
-

```
dist/
node_modules/
.gitignore
.git/
mongo/
```

- **RUN** añadir usuario pokeuser sin password. Conviene crear un nuevo usuario para no usar el root por default
- **RUN** establezco el acceso de pokeuser a solo /var/www/pokedex
    - Si entrara un admin solo podría hacer lo que pokeuser puede hacer, que es en el directorio /var/www/pokedex
- **USER** pokeuser, hago uso del usuario que acabo de crear
- Limpio la caché
- Expongo el puerto 3000
-
- Dockerfile

```
FROM node:18-alpine3.15

# Set working directory
RUN mkdir -p /var/www/pokedex
WORKDIR /var/www/pokedex

# Copiar el directorio y su contenido
COPY . ./var/www/pokedex
COPY package.json package-lock.json tsconfig.json tsconfig.build.json
/var/www/pokedex/
RUN npm install --prod
RUN npm run build


# Dar permiso para ejecutar la applicación
RUN adduser --disabled-password pokeuser
RUN chown -R pokeuser:pokeuser /var/www/pokedex
USER pokeuser

# Limpiar el caché
RUN npm cache clean --force

EXPOSE 3000

CMD [ "node","dist/main" ]
```

Definir la construcción de la imagen

- Borro lo que hay en el Dockerfile y pego esto

- Nombro la imagen de node como deps
- Instalo la librería libc6 (necesaria)
- Trabajo en el directorio /app
- Copio el package.json
- Uso el npm ci que equivale a yarn install --frozen-lockfile
    - Todo esto crea la imagen únicamente con las dependencias de mi aplicación (el package.json y el package-lock)
    - Muevo las dependencias y hago las instalaciones de los módulos de node ahi
    - De esta manera puedo mantener en caché todas esas dependencias. Y solo si cambian van a instalarse
    - Por lo que hacer diferentes builds, si no han cambiado las dependencias va a ser super rápido porque todo está en caché
    - Si no hiciera este paso, cada vez haría el npm i para instalar todas las dependencias
- Hago el build, es otra imagen de node que llamo builder
    - Trabajo en /app
    - Va a copiar de las dependencias (deps, así llamé la imagen anterior) de /app/node_modules (origen) a node_modules (destino)
    - Si no cambiaron es un paso que está todo en caché e instantáneamente los va a mover
    - Copio todo al WORKINDIRECTORY con **COPY .** a **.** (al WORKINGDIRECTORY)
    - Hago el build de producción
- El siguiente paso es el runner. Es quien va a correr la app
    - El WORKDIR es apuntar a app igual
    - Copio el package.json y el package-lock. El último path es dónde quiero que caigan los archivos (./)
    - Hago la instalación de las dependencias de producción
    - Copio del builder (el paso anterior) la carpeta app/dist a ./dist(destino)
    - Ejecuto node dist/main
- El EXPOSE 3000 no va a hacer falta porque por defecto lo vamos a manejar desde afuera

```
# Install dependencies only when needed
FROM node:18-alpine3.15 AS deps
# Check https://github.com/nodejs/docker-
node/tree/b4117f9333da4138b03a546ec926ef50a31506c3#nodealpine to understand why
libc6-compat might be needed.
RUN apk add --no-cache libc6-compat
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm ci

# Build the app with cache dependencies
FROM node:18-alpine3.15 AS builder
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .
RUN npm run build


# Production image, copy all the files and run next
```

```
FROM node:18-alpine3.15 AS runner

# Set working directory
WORKDIR /usr/src/app

COPY package.json package-lock.json ./

RUN npm install --prod

COPY --from=builder /app/dist ./dist

# # Copiar el directorio y su contenido
# RUN mkdir -p ./pokedex

# COPY --from=builder ./app/dist/ ./app
# COPY ./.env ./app/.env

# # Dar permiso para ejecutar la applicación
# RUN adduser --disabled-password pokeuser
# RUN chown -R pokeuser:pokeuser ./pokedex
# USER pokeuser

# EXPOSE 3000

CMD [ "node","dist/main" ]
```

- Creo el docker-compose.prod.yaml en la raíz de mi proyecto

```
version: '3'  # es un standard

services:
  pokedexapp:     # creo un nuevo servicio
    depends_on:
      - db        # depende de la db. Si la db no se levanta no se levantará
pokedex
    build:        # construcción
      context: .  # que se base en la posición física de mi docker-
compose.prod.yaml
      dockerfile: Dockerfile      # Dockerfile, solo tengo uno
    image: pokedex-docker         # Así se llamará la imagen
    container_name: pokedexapp    # El nombre del container
    restart: always # reiniciar el contenedor si se detiene
    ports:
      - "${PORT}:${PORT}"         # Conecto mi variable de entorno PORT con la del
contenedor
    # working_dir: /var/www/pokedex
    environment:                  # Defino mis variables de entorno. Las uso de mi
.env.prod
      MONGODB: ${MONGODB}
      PORT: ${PORT}
      DEFAULT_LIMIT: ${DEFAULT_LIMIT}
    # volumes:
```

```
    #   - ./:/var/www/pokedex     # Por si quisiera montar con la data existente
en el fs, pero no es lo habitual en un build

  db:
    image: mongo:5              # la imagen que tengo de la db
    container_name: mongo-poke  # el nombre, importante!
    restart: always             # en caso de que se caiga va a volver a qintentar
levantarla
    ports:
      - 27017:27017             # establezco los puertos de comunicación entre la
db y el contenedor
    environment:
      MONGODB_DATABASE: nest-pokemon  # la variable de entorno
    # volumes:
    #   - ./mongo:/data/db  //En lugar de usar un volumen con data vamos a
levantar la imagen de la db desde 0
```

- Antes de ejecutar el comando quiero crear un archivo de variables de entorno de producción

---

Construir la imagen

- Copio el .env en .env.prod
- Con la variable de entorno MONGODB necesito apuntar al container de la db que tengo en el docker-compose.prod.yaml
- Uso el container_name y el puerto que expongo en el docker-compose.prod.yaml para apuntar a él, seguido del nombre de la db
- El container_name podría verse como un DNS de la dirección del container (192.169....)

```
MONGODB=mongodb://mongo-poke:27017/nest-pokemon
```

- No tengo nada corriendo
- Para hacer el build es

```
docker-compose -f docker-compose.prod.yaml --env-file .env.prod up --build
```

- Uso -d para que no ver toda la info de docker en consola
- Cuando quiera ejecutarlo será este comando (sin el --build)

```
docker-compose -f docker-compose.prod.yaml --env-file .env.prod up
```

- docker-compose usa por defecto .env por lo que si no lo especificamos lo tomará por defecto
- Para ejecutar el build debo tener DockerDesktop corriendo y conexión (obvio)
- Ahora puedo usar el endpoint http://localhost:3000/api/v2/seed para ejecutar el seed
- Con ctrl+C cancelo el proceso de docker en la terminal
- Ahora ya no necesito usar el --build, solo docker-compose -f docker-compose.prod.yaml --env-file .env.prod up

---

Conservar la db y analizar la imagen

- Para habilitar la persistencia de la data en la db descomento las dos últimas lineas del docker-compose.prod.yaml

```yaml
version: '3'

services:
  pokedexapp:
    depends_on:
      - db
    build:
      context: .
      dockerfile: Dockerfile
    image: pokedex-docker
    container_name: pokedexapp
    restart: always # reiniciar el contenedor si se detiene
    ports:
      - "${PORT}:${PORT}"
    # working_dir: /var/www/pokedex
    environment:
      MONGODB: ${MONGODB}
      PORT: ${PORT}
      DEFAULT_LIMIT: ${DEFAULT_LIMIT}
    # volumes:
    #   - ./:/var/www/pokedex

  db:
    image: mongo:5
    container_name: mongo-poke
    restart: always
    ports:
      - 27017:27017
    environment:
      MONGODB_DATABASE: nest-pokemon
    #AQUI!!!!
    volumes:
      - ./mongo:/data/db
```

- Estoy apuntando al mismo lugar que docker-compose.yaml
- Quito mongo de .dockerignore
- Si cambiaramos algo de la app habría que hacer de nuevo el --build
- Me puedo conectar directamente a la imagen a través de la terminal
- Puedo abrir el contenedor con la accion (los tres puntitos) de Open in Terminal
- Desde ahi puedo acceder a la carpeta dist, listar con ls, visualizar archivos con cat, podría instalar apt-get e instalar un editor para editar los archivos como nano, etc

---

Actualizar README

- README

````
<p align="center">
  <a href="http://nestjs.com/" target="blank"><img
src="https://nestjs.com/img/logo-small.svg" width="200" alt="Nest Logo" /></a>
</p>

# Ejecutar en desarrollo

1. Clonar el repositorio
2. Ejecutar
```
npm i
```

3. Tener el Nest CLI instalado
```
npm i -g @nestjs/cli
```
4. Levantar la base de datos
```
docker-compose up -d
```
5. Clonar el archivo .env.template y renombrar la copia a .env

6. LLenar las variables de entorno definidas en el .env

7. Ejecutar la app en dev:
```
npm run start:dev
```

8. Reconstruir la base de datos con la semilla
```
http://localhost:3000/api/v2/seed
```
# Build de produccion

1. Crear env.prod
2. Llenar las variables de entorno de producción
3. Crear la imagen con docker-compose
```
docker-compose -f docker-compose.prod.yaml --env-file .env.prod up --build
```

## Stack Usado

- MongoDB
- Nest
````

07 NEST TYPEORM POSTGRES

- Esta será una API de productos
- La subida de imágenes será en la próxima sección
- Las imágenes van a estar relacionadas a la tabla de productos en una tabla aparte
- Manejaremos nuestro porpio uuid correlativo, constraints
- el GetBy lo vamos a manejar por Id, por título y por slot

Inicio de proyecto TesloShop

- Creo el proyecto

```
nest new teslo-shop
```

- Un ORM es muy parecido a lo que ofrece mongoose, solo que aquí voy a poder mapear las entidades para poder tener las relaciones entre otras entidades. Establecer triggers, llaves, etc
- Borro todo lo que hay en /src menos el app.module y el main
- Dejo el app.module limpio

Docker - Instalar y correr Postgres

- Creo el docker-compose.yaml
- Para el password uso una variable de entorno (creo el .env)
- Todavía no he configurado las variables de entorno en Nest, pero el docker-compose por defecto lo puede tomar de .env
- Quiero hacer persistente la data. Creo la carpeta en volumes (si no existe la va a crear)
- Es el lugar por defecto dónde se está grabando en el contenedor

```yaml
version : '3'


services:
  db:
    image: postgres:14.3
    restart: always
    ports:
      - "5432:5432" # el puerto del pc con el del contenedor
    environment:
      POSTGRES_PASSWORD: ${DB_PASSWORD}
      POSTGRES_DB: ${DB_NAME}
    container_name: teslodb
    volumes:
      - ./postgres:/var/lib/postgresql/data
```

- Ahora puedo levantar el contenedor (si no la tengo la imagen la descargará)
- Debo tener Docker Desktop corriendo

> docker-compose up

- No le pongo el -d para observar si hay algún error en consola
- Espero ver: LOG: "database system is ready to accept connections"
- Configuro TablePlus para visualizar la db
    - name: TesloDB
    - host: localhost
    - user: postgres (usuario por defecto)
    - password: lo que haya colcoado en la variable de entorno de password
- Hago el test, todo ok. Save
- Ya tengo la carpeta postgres en mi directorio de trabajo
- La añado a .gitignore

```
postgres/
```

- Escribo en el README los pasos para levantar la db
- README

````
<p align="center">
  <a href="http://nestjs.com/" target="blank"><img
src="https://nestjs.com/img/logo-small.svg" width="200" alt="Nest Logo" /></a>
</p>


# Teslo API

1. Configurar variables de entorno
```
DB_NAME=
DB_PASSWORD=
```


2. Levantar la db
```
docker-compose up -d
```
````

Conectar Postgres con Nest

- Instalar los decoradores y typeorm

> npm i @nestjs/typeorm typeorm

- Configuro las variables de entorno con **ConfigModule.forRoot()** de *@nestjs/config*

> npm i @nestjs/config

- En app.module hago la configuración
- En app.module es dónde uso forRoot. En el resto de módulos usaré forFeature
- El puerto tiene que ser un número. Lo parseo con +
- Después de las variables de entorno coloco dos propiedades
    - autoLoadEntities: true Para que cargue automaticamente las entidades que vaya creando
    - synchronize: true Hace que cuando creo algún cambio en las entidades las sincroniza
- En producción no voy a querer el synchronize en true.

```
import { Module } from '@nestjs/common';
import {ConfigModule} from '@nestjs/config'
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
  imports: [
    ConfigModule.forRoot(),

    TypeOrmModule.forRoot({
      type: 'postgres',
      host: process.env.DB_HOST,
      port: +process.env.DB_PORT,
      database: process.env.DB_NAME,
      username: process.env.DB_USERNAME,
      password: process.env.DB_PASSWORD,
      autoLoadEntities: true,
      synchronize: true
    })
  ],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

- Para hacer la colección necesita un último paquete (el driver)

```
npm i pg
```

- Excluyo el archivo .env añadiéndolo en el .gitignore y copio .env con .env.template

---

TypeOrm Entity Product

- Voy a tener
    - La descripción
    - Imágenes [] Las quiero manejar en filesystem en lugar de urls (archivos jpg)
    - Stock
    - Price
    - Sizes []
    - Slug
    - Type

    - Tags []
    - Title
    - Gender
- Uso el CLI para generar el CRUD de products (--no-spec es para que no me incluya los archivos de test)

> nest g res products --no-spec

- La entity viene a representar una tabla
- Debo decorar la clase como **@Entity()**, decorador de typeorm
- Para el id usaré **@PrimaryGeneratedColumn()**. Ofrece diferentes maneras de cómo manejarlo
    - No usaré uuid
- Defino de qué tipo será la columna, y en un objeto las propiedades
- En el caso de title, no puede haber dos productos con el mismo título
- Le he puesto autoLoadEntities en true, pero todavía no tengo definida la entidad en ningún lugar
- Añado **el módulo** TypeOrmModule (**siempre que es un módulo va en imports**) y esta ves es **forFeature** ya que **forRoot** solo hay uno. En el añado un arreglo donde irán las entidades
- products.module

```
import { Module } from '@nestjs/common';
import { ProductsService } from './products.service';
import { ProductsController } from './products.controller';
import {TypeOrmModule} from '@nestjs/typeorm'
import { Product } from './entities/product.entity';

@Module({
  controllers: [ProductsController],
  providers: [ProductsService],
  imports: [
    TypeOrmModule.forFeature([
      Product
    ])
  ]
})
export class ProductsModule {}
```

- Si levanto docker y el server y voy a TablePlus y me conecto a la DB
- Puedo ver que la tabla Products tiene la columna id y title y tengo una serie de funciones para manejar los uuid

---

Entidad sin relaciones

- Terminemos parcialmente Product. después añadiremos relaciones con otras tablas
- Para añadir el precio *yo podría pensar que la en la Columna es de tipo number* pero no es el tipo que acepta TypeORM
- Para esto habría que mirar la documentación, pero es **float**
- Para la description muestro **otra forma** de definir el tipo usando type

- El slug tiene que ser único, porque me va a servir para identificar un producto, ayuda a tener urls friendly
- Para las sizes, podría pensar en hacer otra tabla. Una manera de saber si hacer otra tabla es pensar si van a haber muchos null, interesa hacer otra tabla para no almacenar null. Pero en este caso todos los productos van a tener un size
    - Le defino **array en true**, es un array de strings

```
import {Entity, PrimaryGeneratedColumn, Column} from 'typeorm'

@Entity()
export class Product {
    @PrimaryGeneratedColumn('uuid')
    id: string

    @Column('text', {
        unique: true
    })
    title: string

    @Column('float',{
        default: 0
    })
    price: number

    @Column({
        type: 'text',
        nullable: true
    })
    description: string

    @Column({
        type: 'text',
        unique: true
    })
    slug: string

    @Column({
        type: 'int',
        default: 0
    })
    stock: number

    @Column({
        type: 'text',
        array: true
    })
    sizes: string[]

    @Column({
        type: 'text',
    })
```

```
    gender: string
}
```

- Todavía faltan campos

Create Product Dto

- Vamos a hacer la configuración de los dtos y también el global prefix para añadir un segmento a la url de la API REST
- En el main, **antes de escuchar el puerto**, añado api a la url

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

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

  app.setGlobalPrefix('api')

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

- Entonces, la url para las peticiones queda asi

> http://localhost:3000/api/products

- Para utilizar el class-validator para los dtos y las validaciones tengo que instalarlo

> npm i class-validator class-transformer

- Hay que usar useGlobalPipes (lo del whitelist) para usar las validaciones
- 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.setGlobalPrefix('api')

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

```
  )

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

- Voy al create-product.dto
- Coloco las propiedades que voy a necesitar o son opcionales en la data de entrada

```
export class CreateProductDto {

    title: string

    price?: number

    description?: string

    slug?: string

    stock?: number

    sizes: string[]

    gender: string
}
```

- Coloco los decoradores
- Uso el each en true para asegurarme que cada valor del array sea un string
- Uso IsIn para establecer que tiene que ser uno de esos valores

```
import { IsString, MinLength, IsNumber, IsOptional, IsInt, IsPositive, IsArray,
IsIn } from "class-validator"

export class CreateProductDto {

    @IsString()
    @MinLength(1)
    title: string

    @IsNumber()
    @IsOptional()
    price?: number

    @IsString()
    @IsOptional()
    description?: string

    @IsString()
    @IsOptional()
```

```
    slug?: string

    @IsInt()
    @IsPositive()
    @IsOptional()
    stock?: number

    @IsString({each: true})
    @IsArray()
    sizes: string[]

    @IsIn(['men', 'women', 'kid','unisex'])
    gender: string
}
```

Insertar usando TypeORM

- El controlador @Post se queda igual
- En el servicio
- Vuelvo el método async ya que consultar una db es una tarea asíncrona
- Para usar la entidad hago uso de la inyección de dependencias en el constructor del servicio
- Hago uso del decorador **@InjectRepository** de typeorm. Le coloco la entidad Product
- En Repository debo colcarle el tipo (que es Product). Repository lo importo de typeorm

```
import { Injectable } from '@nestjs/common';
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { Product } from './entities/product.entity';
import { Repository } from 'typeorm';

@Injectable()
export class ProductsService {

  constructor(
    @InjectRepository(Product)
    private readonly productRepository: Repository<Product> ){}


  async create(createProductDto: CreateProductDto) {


  }
}
```

- Coloco la inserción dentro de un try y un catch porque algo puede salir mal
- Al escribir los paréntesis del método del create() puedo ver que tengo varias opciones
    - Puedo mandar el create vacío create()

    - Puedo mandar el entityLikeArray:DeepPartial< Product >[]
    - Puedo mandar el entityLike:DeepPartial< Product >
- Puedo enviarle el createProductDto ya que es algo que luce como la entidad
- Esto solo crea la instancia del producto con sus propiedades, no lo estoy insertando. Solo **creo el registro**
- **Guardo** con save y le paso el registro (product)

```
  async create(createProductDto: CreateProductDto) {
    try {
      const product = this.productRepository.create(createProductDto)

      await this.productRepository.save(product)

      return product

    } catch (error) {
      console.log(error)
      throw new InternalServerErrorException('Ayuda!')
    }
  }
```

- Creo la peticion POST en ThunderClient al endpoint localhost:3000/api/products
- El description en la entity tiene el nullable en true, con lo que puede no ir
- Pero el slug, por ejemplo, no lo tiene y no tiene ningún valor por defecto, con lo que es obligatorio
- El precio puse en la entidad que tuviera valor 0 por defecto, pero se lo coloco

```
{
  "title": "Migue's trousers",
  "sizes": ["SM", "M", "L"],
  "gender": "men",
  "slug": "migues_trousers",
  "price": 199.99
}
```

- Hay que manejar los errores, por ejemplo el de llave duplicada (que el registro ya exista)
- Vamos a aprender a ejecutar procedimientos antes de la inserción, por ejemplo para evaluar si viene el slug y si no viene generarlo

---

Manejo de errores (LOGGER)

- Hay una serie de condiciones que hay que evaluar. Que el título esté bien, el slug, etc
- Si hiciera la verificación a través de la db para saber si ya hay un título, etc serían muchas consultas a la db
- Para mejorar el console.log del error puedo usar lo que **incorpora Nest**.
- Creo una propiedad privada readonly logger e importo Logger de @nestjs/common
- Cuando abro paréntesis puedo ver las varias opciones que le puedo pasar a la instancia

- Una de ellas es **context:string**. Puedo ponerle **el nombre de la clase** en la que estoy usando este logger
- En lugar del console.log(error) uso **this.logger.error**

```
import { Injectable, InternalServerErrorException, Logger } from '@nestjs/common';
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { Product } from './entities/product.entity';
import { Repository } from 'typeorm';

@Injectable()
export class ProductsService {

  private readonly logger = new Logger('ProductsService')

  constructor(
    @InjectRepository(Product)
    private readonly productRepository: Repository<Product> ){}


  async create(createProductDto: CreateProductDto) {
    try {
      const product = this.productRepository.create(createProductDto)

      await this.productRepository.save(product)

      return product

    } catch (error) {

      this.logger.error(error)
      throw new InternalServerErrorException('Ayuda!')
    }

  }
}
```

- Ahora en la consola tengo un error más específico si intento insertar el mismo título
- Puedo ser más específico. Si hago un console.log(error) obtengo el código del error y los detalles
- Si no es este error concreto puedo mandar el logger para ver que ocurre y lanzar la excepción

```
async create(createProductDto: CreateProductDto) {
  try {
    const product = this.productRepository.create(createProductDto)

    await this.productRepository.save(product)

    return product
```

```
  } catch (error) {

    if(error.code === '23505')
      throw new BadRequestException(error.detail)

    this.logger.error(error)
    throw new InternalServerErrorException('Unexpected error, check Server logs')
  }
```

- Este tipo de error es algo que voy a necesitar en varios lugares.
- Puedo crear un método privado para ello

```
private handleDBExceptions(error: any){
  if(error.code === '23505')
  throw new BadRequestException(error.detail)

  this.logger.error(error)
  throw new InternalServerErrorException('Unexpected error, check Server logs')
}
```

BeforeInsert y BeforeUpdate

- Si no mando el **slug** me da un error de db porque **es requerido**, en la entity no tiene el nullable en true
- Pero **en el dto lo tengo como opcional**
- Yo lo puedo generar basado en el titulo
- Para que no me de error con replaceAll debo cambiar el target a es2021 en el tsconfig
- Reemplazo espacios por guiones bajos y apostrofes por string vacío(no lo voy a colocar)
    - Si viene el slug tengo que quitar esas cosas

```
async create(createProductDto: CreateProductDto) {
    try {

      if(!createProductDto.slug){
        createProductDto.slug = createProductDto.title.toLowerCase().replaceAll('
', '_').replaceAll("'", "")
      }else{
        createProductDto.slug = createProductDto.slug.toLowerCase().replaceAll('
', '_').replaceAll("'", "")
      }

      const product = this.productRepository.create(createProductDto)

      await this.productRepository.save(product)

      return product

    } catch (error) {
```

```
      this.handleDBExceptions(error)
    }

  }
```

- Puedo crear este procedimiento antes de que se inserte en la db
- products.entity

```
import {Entity, PrimaryGeneratedColumn, Column, BeforeInsert} from 'typeorm'

@Entity()
export class Product {
    @PrimaryGeneratedColumn('uuid')
    id: string

    @Column('text', {
        unique: true
    })
    title: string

    @Column('float',{
        default: 0
    })
    price: number

    @Column({
        type: 'text',
        nullable: true
    })
    description: string

    @Column({
        type: 'text',
        unique: true
    })
    slug: string

    @Column({
        type: 'int',
        default: 0
    })
    stock: number

    @Column({
        type: 'text',
        array: true
    })
    sizes: string[]

    @Column({
```

```
        type: 'text',
    })
    gender: string

    @BeforeInsert()
    checkSlugInsert(){
        if(!this.slug){
            this.slug = this.title  //si no viene el slug guardo el titulo en el
slug
        }

        this.slug = this.slug //en este punto ya tengo el slug, lo formateo
        .toLowerCase()
        .replaceAll(' ', '_')
        .replaceAll("'", "")
    }
}
```

Get y Delete TypeORM (CRUD BÁSICO)

- En el controller hago uso del *ParseUUIDPipe* en el findOne y el remove
- Hago uso del repositorio en el servicio
- Extraigo el producto de la db y compruebo de que el producto exista
- En el delete puedo usar el método findOne dónde ya hago la validación

```
import { BadRequestException, Injectable, InternalServerErrorException, Logger,
NotFoundException } from '@nestjs/common';
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { Product } from './entities/product.entity';
import { Repository } from 'typeorm';

@Injectable()
export class ProductsService {

  private readonly logger = new Logger('ProductsService')

  constructor(
    @InjectRepository(Product)
    private readonly productRepository: Repository<Product> ){}


  async create(createProductDto: CreateProductDto) {
    try {
      const product = this.productRepository.create(createProductDto)

      await this.productRepository.save(product)

      return product
```

```
    } catch (error) {

      this.handleDBExceptions(error)
    }
  }

  async findAll() {
    return await this.productRepository.find();
  }

  async findOne(id: string) {
    const product = await this.productRepository.findOneBy({id})
    if(!product) throw new NotFoundException('Product not found')
    return product
  }

  update(id: number, updateProductDto: UpdateProductDto) {
    return `This action updates a #${id} product`;
  }

  async remove(id: string) {

    const product = await this.findOne(id)

    await this.productRepository.delete(id)
  }

  private handleDBExceptions(error: any){
    if(error.code === '23505')
    throw new BadRequestException(error.detail)

    this.logger.error(error)
    throw new InternalServerErrorException('Unexpected error, check Server logs')
  }
}
```

Paginar en TypeORM

- Creo un dto relacionado a la paginación
- No está directamente relacionado a los productos, por lo que lo **creo el módulo common** y dentro creo la carpeta dto

> nest g mo common

- Esto importa directamente en app.module, que es el módulo principal
- Le añado las propiedades a la clase PaginationDto y las decoro
- Para transformar la data, en el proyecto anterior se configuró en el app.useGlobalPipes, dentro del new ValidationPipe
    - transform: true

    - transformOptions: { enableImplicitConversion: true}
- Se puede hacer de esta otra forma, usando **@Type** de *class-transform*
- Cuando pongo **@IsPositive** no es necesario el @IsNumber

```
import { Type } from "class-transformer"
import { IsOptional, IsPositive, Min } from "class-validator"

export class PaginationDto{

    @IsOptional()
    @IsPositive()
    @Type(()=> Number)
    limit?: number

    @IsOptional()
    @Min(0)
    @Type(()=> Number)
    offset?: number
}
```

- En el controller hago uso del dto

```
  @Get()
  findAll(@Query() paginationDto: PaginationDto) {
    return this.productsService.findAll(paginationDto);
  }
```

- En el service hago la paginación
- Extraigo los valores del dto y les doy un valor por defecto

```
 async findAll(paginationDto:PaginationDto) {

    const {limit=10, offset= 0} = paginationDto

    return await this.productRepository.find({
      take: limit,
      skip: offset
      //TODO: relaciones
    })
  }
```

Buscar por slug, título o UUID

- El slug me permite hacer urls friendly
- Yo puedo querer buscar por el slug

- Se podría crear otro endpoint para buscar por el slug pero lo vamos a hacer en el mismo
- En el controller quito el ParseUUIDPipe y cambio id por term, que es más adecuado
- En el service:
  - Necesito instalar uuid para verificar si es un uuid o no. Instalo los tipos también con @types/uuid
  - Importo validate de uuid y lo renombro a isUUID
  - Hago la validación
  - Si solo buscáramos por uuid o slug lo soluciono con un else

```
async findOne(term: string) {

    let product: Product

    if(isUUID(term)){
      product = await this.productRepository.findOneBy({id: term})
    }else{
      product = await this.productRepository.findOneBy({slug: term})
    }


    if(!product) throw new NotFoundException('Product not found')
    return product
  }
```

- Pero quiero también buscar por título. Para eso es el **QueryBuilder**

QueryBuilder

- La consulta se podría hacer con el find y el WHERE, pero aprendamos que es el **queryBuilder**
- TypeORM añade una capa de seguridad que escapa los caracteres especiales para evitar inyección de SQL
- Los : significa que son parámetros, se los paso como segundo argumento
- Solo me interesa uno de los dos(porque podría ser que regrese dos si slug y titulo están ubicados en sitios diferentes), por eso uso getOne()

```
async findOne(term: string) {

  let product: Product

  if(isUUID(term)){
    product = await this.productRepository.findOneBy({id: term})
  }else{
    const queryBuilder = this.productRepository.createQueryBuilder()

    product = await queryBuilder.where(`title = :title or slug = :slug`, {
      title: term,
      slug: term
    }).getOne()
  }
```

```
  if(!product) throw new NotFoundException('Product not found')
  return product
}
```

- El queryBuilder **te permite escribir tus queries** cubriéndote el tema de la seguridad por inyección de SQL
- El queryBuilder es case sensitive. Para evitarlo puedo usar UPPER en el querie y luego pasar el término a mayusculas con toUpperCAse
- El slug lo estoy guardando con toLowerCase con lo que lo uso con el term

```
async findOne(term: string) {

  let product: Product

  if(isUUID(term)){
    product = await this.productRepository.findOneBy({id: term})
  }else{
    const queryBuilder = this.productRepository.createQueryBuilder()

    product = await queryBuilder.where(`UPPER(title) = :title or slug = :slug`, {
      title: term.toUpperCase(),
      slug: term.toLowerCase()
    }).getOne()
  }

  if(!product) throw new NotFoundException('Product not found')
  return product
}
```

Update en TypeORM

- Todos los campos son opcionales, pero hay ciertas restricciones. La data tiene que lucir como yo estoy esperando
- Cuando solo hay una tabla involucrada la actualización es sencilla
- En el dto del update uso PartialType que hace las propiedades del dto de create opcionales
- Vamos a hacer que para actualizar siempre vamos a usar un UUID
- Uso el ParseUUIDPipe en el controller, el id es un string

```
@Patch(':id')
update(@Param('id', ParseUUIDPipe) id: string, @Body() updateProductDto:
UpdateProductDto) {
  return this.productsService.update(id, updateProductDto);
}
```

- En el servicio

- Con el preload le digo que busque un producto por el id, y que cargue usando el spread toda la data de las propiedades del dto
- Si no existe el prodcuto lanzo un error
- Guardo el producto actualizado

```
async update(id: string, updateProductDto: UpdateProductDto) {
    const product = await this.productRepository.preload({
      id,
      ...updateProductDto
    })

    if(!product) throw new NotFoundException('Product not found')

    await this.productRepository.save(product)

    return product
  }
```

- Si le paso un título que ya existe me va a devolver un InternalServerError
- Puedo colocar el .save dentro de un try catch para capturar el error y lanzar una excepción

```
async update(id: string, updateProductDto: UpdateProductDto) {
  const product = await this.productRepository.preload({
    id,
    ...updateProductDto
  })

  if(!product) throw new NotFoundException('Product not found')

  try {
    await this.productRepository.save(product)
    return product
  } catch (error) {
    this.handleDBExceptions(error)
  }
}
```

- Los slugs los tengo que validar. Si viene el slug, tiene que cumplir las condiciones que anteriormente establecí
- Para ello usaré **BeforeUpdate**

BeforeUpdate

- Uso **@BeforeUpdate** en la entity

```
import {Entity, PrimaryGeneratedColumn, Column, BeforeInsert, BeforeUpdate} from
'typeorm'

@Entity()
export class Product {
    @PrimaryGeneratedColumn('uuid')
    id: string

    @Column('text', {
        unique: true
    })
    title: string

    @Column('float',{
        default: 0
    })
    price: number

    @Column({
        type: 'text',
        nullable: true
    })
    description: string

    @Column({
        type: 'text',
        unique: true
    })
    slug: string

    @Column({
        type: 'int',
        default: 0
    })
    stock: number

    @Column({
        type: 'text',
        array: true
    })
    sizes: string[]

    @Column({
        type: 'text',
    })
    gender: string

    @BeforeInsert()
    checkSlugInsert(){
        if(!this.slug){
            this.slug = this.title
        }
```

```
        this.slug = this.slug
        .toLowerCase()
        .replaceAll(' ', '_')
        .replaceAll("'", "")
    }

    @BeforeUpdate()
    checkSlugUpdate(){
        this.slug = this.slug
        .toLowerCase()
        .replaceAll(' ', '_')
        .replaceAll("'", "")
    }
}
```

Tags

- Puedo usar tags para mejorar las búsquedas
- Es una nueva columna en mi entity
- Los tags siempre los voy a pedir. Por defecto será un array de strings
- Le añado por defecto un array vacío
- Como tengo el synchronize en true la añade directamente
- entity

```
@Column({
    type: 'text',
    array: true,
    default: []
})
tags: string[]
```

- Hay que enviar los tags en la creación y la actualización como un arreglo
- Para ello actualizo mi Dto
- Como por defecto le mando un arreglo vacío puedo decir que isOptional

```
@IsString({each:true})
@IsArray()
@IsOptional()
tags?: string[]
```

---

08 NEST Relaciones TypeORm

- En esta sección vamos a ver como el cliente a la hora de crear va a ser capaz de mandarme los url de mis imágenes
- Voy a almacenar estas imágenes en una carpeta del filesystme de mis servidor que no va a ser pública
- Apunto a requerir autenticación: no va a ver esta imagen si no está autenticado o si no es el usuario
- Estas imágenes las voy a estar almacenando en una nueva tabla
- Les voy a asignar un id único
- Más adelante se hará un servicio para servir estas imágenes que podamos autenticar y validar
- Cuando envío la actualización, las imágenes que hayan sido guardadas previamente serán eliminadas

---

ProductImage Entity

- No voy a crear un CRUD completo para las imágenes de los productos. No quiero que funcione así
- En src/products/entitys creo product-image.entity
- No digo que la url puede ser null, por lo que tiene que venir

```
import { Entity, PrimaryGeneratedColumn, Column } from "typeorm";

@Entity()
export class ProductImage{

    @PrimaryGeneratedColumn() //va a tener un número autoincremental como id
    id: number

    @Column('text')
    url: string
}
```

- Debo indicar que existe esta entidad, aunque tenga el synchronize en true, por ahora solo he creado un archivo con una clase
- En products.module

```
import { Module } from '@nestjs/common';
import { ProductsService } from './products.service';
import { ProductsController } from './products.controller';
import {TypeOrmModule} from '@nestjs/typeorm'
import { Product } from './entities/product.entity';
import { ProductImage } from './entities/product-image.entity';

@Module({
  controllers: [ProductsController],
  providers: [ProductsService],
  imports: [
    TypeOrmModule.forFeature([
      Product, ProductImage
    ])
  ]
})
export class ProductsModule {}
```

- Ahora debería aparecer en TablePlus
- Como importo las dos entidades desde el mismo lugar me creo un archivo de barril
- La forma fácil es hacer los imports y cambiar la palabra import por export
- index.ts

```
export { ProductImage } from "./product-image.entity";
export { Product } from "./product.entity";
```

- Puedo importarlas así en el módulo

```
import { Product, ProductImage } from './entities';
```

- Cómo hacemos una relación de uno a muchos?
- Necesito una nueva columna en product-image que me diga el id del producto al cual pertenece

---

OneToMany y ManyToOne

- Tengo que decirle a mi entity Product que ahora va a tener las imágenes
- Y a la entity imágenes que va a tener un id producto
- Pongo que las images son opcionales y su relación es OneToMany (lo importo de typeORM) ya que un producto puede tener varias imágenes
    - El callback va a regresar un ProductImage
    - productImage (con minúscula), todavía no aparece .product (pero si .id y .url)
    - Le coloco en las opciones el cascade en true. Esto va ayudar a que si hago una elminación, elimine las imagenes asociadas al producto
    - Usualmente no se borraría, si no que estaría activo inactivo
- product.entity

```
import {Entity, PrimaryGeneratedColumn, Column, BeforeInsert, BeforeUpdate,
OneToMany} from 'typeorm'
import { ProductImage } from './product-image.entity'

@Entity()
export class Product {
    @PrimaryGeneratedColumn('uuid')
    id: string

    @Column('text', {
        unique: true
    })
    title: string

    @Column('float',{
        default: 0
    })
```

```
    price: number

    @Column({
        type: 'text',
        nullable: true
    })
    description: string

    @Column({
        type: 'text',
        unique: true
    })
    slug: string

    @Column({
        type: 'int',
        default: 0
    })
    stock: number

    @Column({
        type: 'text',
        array: true
    })
    sizes: string[]

    @Column({
        type: 'text',
    })
    gender: string

    @Column({
        type: 'text',
        array: true,
        default: []
    })
    tags: string[]

    @OneToMany(
        ()=> ProductImage,
        productImage=> productImage.product,
        {cascade:true}
    )
    images?: ProductImage[]

    @BeforeInsert()
    checkSlugInsert(){
        if(!this.slug){
            this.slug = this.title
        }

        this.slug = this.slug
        .toLowerCase()
        .replaceAll(' ', '_')
```

```
        .replaceAll("'", "")
    }

    @BeforeUpdate()
    checkSlugUpdate(){
        this.slug = this.slug
        .toLowerCase()
        .replaceAll(' ', '_')
        .replaceAll("'", "")
    }

}
```

- En ProductImage voy a tener un product de tipo Product
- Muchas imágenes pueden tener un único producto, por lo que la relación es ManyToOne
- Muchos registros de mi tabla de ProductImage pueden ter un único producto
- El primer callback devuelve algo de TipoProduct
- El segundo callback le paso el product y lo asocio con product.image
- En opciones

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToOne } from "typeorm";
import { Product } from "./product.entity";

@Entity()
export class ProductImage{

    @PrimaryGeneratedColumn() //va a tener un número autoincremental como id
    id: number

    @Column('text')
    url: string

    @ManyToOne(
        ()=> Product,
        product => product.images
    )
    product: Product
}
```

Crear imághenes de producto

- Debo añadir images al create-product.dto para que no me de error

```
import { IsString, MinLength, IsNumber, IsOptional, IsInt, IsPositive, IsArray,
IsIn } from "class-validator"

export class CreateProductDto {
```

```
    @IsString()
    @MinLength(1)
    title: string

    @IsNumber()
    @IsOptional()
    price?: number

    @IsString()
    @IsOptional()
    description?: string

    @IsString()
    @IsOptional()
    slug?: string

    @IsInt()
    @IsPositive()
    @IsOptional()
    stock?: number

    @IsString({each: true})
    @IsArray()
    sizes: string[]

    @IsIn(['men', 'women', 'kid','unisex'])
    gender: string

    @IsString({each:true})
    @IsArray()
    @IsOptional()
    tags?: string[]

    @IsString({each:true})
    @IsArray()
    @IsOptional()
    images?: string[]
}
```

- Salta un error en consola:

```
    Types of property 'images' are incompatible.
      Type 'string[]' is not assignable to type 'DeepPartial<ProductImage>'.

     const product = await this.productRepository.preload({
                                                          ~
       id,
  ~~~~~~~~~
       ...updateProductDto
  ~~~~~~~~~~~~~~~~~~~~~~~~~
     })
```

- Si voy al productService veo que tengo un problema en el create y en el update
- Temporalmente voy a añadir como un arreglo vacío de imágenes en el create y el update.
- Este error es porque debo asegurarme de que las imágenes son una instancia de ProductImage
- Porqué? Porque ahí tienen el productId, el url, el id y no simplemente son strings como le estoy mandando en el body
- Entonces tengo que hacer algún tipo de conversión
- product.service

```
import { BadRequestException, Injectable, InternalServerErrorException, Logger,
NotFoundException } from '@nestjs/common';
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { Product } from './entities/product.entity';
import { Repository } from 'typeorm';
import { PaginationDto } from 'src/common/dto/pagination.dto';
import { validate as isUUID } from 'uuid';


@Injectable()
export class ProductsService {

  private readonly logger = new Logger('ProductsService')

  constructor(
    @InjectRepository(Product)
    private readonly productRepository: Repository<Product> ){}


  async create(createProductDto: CreateProductDto) {
    try {
      const product = this.productRepository.create({...createProductDto, images:
[]}) //añado como arreglo vacío images

      await this.productRepository.save(product)

      return product

    } catch (error) {

      this.handleDBExceptions(error)
    }

  }

  async findAll(paginationDto:PaginationDto) {

    const {limit=10, offset= 0} = paginationDto

    return await this.productRepository.find({
```

```
      take: limit,
      skip: offset
      //TODO: relaciones
    })
  }

  async findOne(term: string) {

    let product: Product

    if(isUUID(term)){
      product = await this.productRepository.findOneBy({id: term})
    }else{
      const queryBuilder = this.productRepository.createQueryBuilder()

      product = await queryBuilder.where(`UPPER(title) = :title or slug = :slug`,
{
        title: term.toUpperCase(),
        slug: term.toLowerCase()
      }).getOne()
    }

    if(!product) throw new NotFoundException('Product not found')
    return product
  }

  async update(id: string, updateProductDto: UpdateProductDto) {
    const product = await this.productRepository.preload({
      id,
      ...updateProductDto,
      images: []      //añado como arreglo vacío images
    })

    if(!product) throw new NotFoundException('Product not found')

    try {
      await this.productRepository.save(product)
      return product
    } catch (error) {
      this.handleDBExceptions(error)
    }
  }

  async remove(id: string) {

    const product = await this.findOne(id)

    await this.productRepository.delete(id)
  }

  private handleDBExceptions(error: any){
    if(error.code === '23505')
    throw new BadRequestException(error.detail)
```

```
    this.logger.error(error)
    throw new InternalServerErrorException('Unexpected error, check Server logs')
  }
}
```

- Extraigo las imágenes con desestructuración del createProductDto, le asigno un arreglo vacío por defecto por si no viene ninguna
- Extraigo el resto de propiedades con el operador **...**
- Para generar las instancias de las imágenes voy a necesitar inyectar el repositorio
- Hago un map, que devuelve un arreglo. Debo indicarle que la url es la imagen
- TypeORM va ainferir por mi, dice:
    - Ah! estás queriendo crear instancias de ProductImage que se encuentran dentro del producto
    - Cuando grabe este nuevo producto, typeORM va a saber que el id que le asigne a este producto va a ser el que le asigne a cada una de las imágenes

```
  async create(createProductDto: CreateProductDto) {
    try {

      const {images = [], ...productDetails} = createProductDto

      const product = this.productRepository.create({
        ...productDetails,
        images: images.map(image=> this.productImageRepository.create({url:
image}))
      })

      await this.productRepository.save(product)

      return product

    } catch (error) {

      this.handleDBExceptions(error)
    }

  }
```

- Ahora apunto al endpoint y en el body le mando las url de las fotos

> POST http://localhost:3000/api/products/

```
{
  "title": "Migue's shoes",
  "sizes": ["SM", "M", "L"],
  "gender": "men",
  "price": 300,
  "tags":["shoes", "Migue"],
```

```
  "images": [
    "http://image1.jpg",
    "http://image2.jpg"
    ]
}
```

- Ahora las imágenes tienen una url y un id
- Me gustaría que estas imágenes que tienen una url y una id me las devuelva solo como un arreglo de url para el frontend
- Lo hago en el return, usando las images que estoy recibiendo para retornarlas tal cual

```
  async create(createProductDto: CreateProductDto) {
    try {

      const {images = [], ...productDetails} = createProductDto

      const product = this.productRepository.create({
        ...productDetails,
        images: images.map(image=> this.productImageRepository.create({url:
image}))
      })

      await this.productRepository.save(product)

      return {...product, images}

    } catch (error) {

      this.handleDBExceptions(error)
    }

  }
```

Aplanar las imágenes

- En product.entity tengo la relación pero **no tengo la columna en la db**
- Con lo que en la tabla de productos no me aparecen las imágenes relacionadas
- En el findAll del servicio establezco la relación con **relations**

```
async findAll(paginationDto:PaginationDto) {

  const {limit=10, offset= 0} = paginationDto

  return await this.productRepository.find({
    take: limit,
    skip: offset,
    relations: {
```

```
      images: true
    }
  })
}
```

- Quiero aplanar las imágenes, que solo me devuelva la url
- El id me puede servir para hacer la eliminación y el update pero de momento lo quiero así
- Yo se que el .find va a regresar un arreglo de Product, lo guardo en una constante

```
async findAll(paginationDto:PaginationDto) {

  const {limit=10, offset= 0} = paginationDto

  const products =  await this.productRepository.find({
    take: limit,
    skip: offset,
    relations: {
      images: true
    }
  })

  return products.map(product =>({
    ...product,
    images: product.images.map(img=> img.url)
  }))
}
```

- Puedo mejorar el código

```
async findAll(paginationDto:PaginationDto) {

  const {limit=10, offset= 0} = paginationDto

  const products =  await this.productRepository.find({
    take: limit,
    skip: offset,
    relations: {
      images: true
    }
  })

  return products.map(({images, ...rest}) =>({
    ...rest,
    images: images.map(img=> img.url)
  }))
}
```

- En el .findOneBy tengo el mismo problema, pero en este no puedo especificar la condición de relations

- En la documentación de typeORM, en Eager relations, veo que solo funcionan con el .find
- Cuando se usa el QueryBuilder están deshabilitadas y hay que usar el leftJoinAndSelect
- En el OneToMany puedo especificar el eager en true
- product.entity

```
@OneToMany(
    ()=> ProductImage,
    productImage=> productImage.product,
    {cascade:true, eager: true}
)
images?: ProductImage[]
```

- Ahora cuando busco por id me aparecen las imágenes
- Pero si busco por el slug no aparecen, porque estoy usando el QueryBuilder
- Uso el leftJoinAndSelect, debo especificar cual es la relación
- Puedo agregarle un alias a la tabla de producto, en este caso prod
- Debo indicarle el punto en el cual quiero hacer el leftJoin, me pide que le ponga un alias en caso de que quiera hacer otros joins, le pongo prodImages

```
async findOne(term: string) {

  let product: Product

  if(isUUID(term)){
    product = await this.productRepository.findOneBy({id: term})
  }else{
    const queryBuilder = this.productRepository.createQueryBuilder('prod')

    product = await queryBuilder.where(`UPPER(title) = :title or slug = :slug`, {
      title: term.toUpperCase(),
      slug: term.toLowerCase()
    })
    .leftJoinAndSelect('prod.images', 'prodImages')
    .getOne()
  }

  if(!product) throw new NotFoundException('Product not found')
  return product
}
```

- Podría usar el mismo método de devolver el ...product, images: images.map, etc
  - Pero no lo voy a manejar así, porque me interesa deviolver una instancia de mi entiad, y no algo que luzca como tal
- Voy a crear un nuevo método que me sirva para aplanarlo

```
async findOnePlane(term: string){
    const {images=[], ...product} = await this.findOne(term)
    return {
      ...product,
      images: images.map(img=>img.url)
    }
  }
```

Query Runner

- Vamos a trabajar la actualización de un producto
- Si actualizo un producto y no le paso las imágenes (y el producto tiene imágenes) aparece el arreglo de imágenes vacío y en la tabla de imágenes he perdido la referencia al producto
- Esto sucede porque tengo el **cascade en true**
- También porque cuando estoy haciendo el **update le estoy diciendo que images es un arreglo vacio**
- **Borro todas las imagenes de la db**
- Quiero que las imágenes que añado en el body sean las nuevas imágenes y borrar las anteriores
- Entonces, son dos cosas que quiero hacer: borrar las anteriores e insertar las nuevas
- Si una de las dos falla quiero revertir el proceso. Para ellos usaré el **Query Runner**
- Desestructuro del dto las imagenes por separado
- Debo verificar si vienen imágenes o no. Si vienen imágenes voy a tener que borrar las anteriores de una manera controlada
- El queryRunner tiene que conocer **la cadena de conexión** que estoy usando
- Para ello usaré la inyección de depndencias con elc **DataSoure** (lo importo de typeORM)
- Uso **createQueryRunner** (el createQueryBuilder serviría si lo creara desde cero sin ninguna entidad)
- Con el queryRunner voy a empezar a **definir una serie de procedimientos**

```
  constructor(
    @InjectRepository(Product)
    private readonly productRepository: Repository<Product>,

    @InjectRepository(ProductImage)
    private readonly productImageRepository: Repository<ProductImage>,

    private readonly dataSource: DataSource

    ){}

async update(id: string, updateProductDto: UpdateProductDto) {

  const {images, ...toUpdate} = updateProductDto


  const product = await this.productRepository.preload({id, ...toUpdate})

  if(!product) throw new NotFoundException(`Product with id : ${id} not found`)
```

```
    const queryRunner = this.dataSource.createQueryRunner()


    try {
      await this.productRepository.save(product)
      return product
    } catch (error) {
      this.handleDBExceptions(error)
    }
  }
```

Transacciones

- Vamos a usar el queryRunner para crear **una transacción**
- Son una serie de querys que pueden impactar la db, hasta que no le haga el commit no lo hará
- Hay que liberar el queryRunner porque si no mantiene esa conexión
- Tengo que evaluar si hay imágenes en el dto para borrar las existentes. Así es como quiero que funcione
- El soft delete mantiene la referencia al objeto, no lo borra. El delete si
- Como primer parámetro recibe la entity, y de segundo los criterios, en este caso en la columna de producto, el id que me están pasando para actualizar
- Entonces, voy a borrar todas las images cuya columna product sea el id
- Coloco product y no productId como aparece en la tabla porque la columna de productId es una relación y me lo permite
- Creo el queryRunner fuera del try para poder usar el rollback en el catch
- Para que me devuelva las imágenes cuando no vienen en el body del update uso findOnePlain
    - Podría hacerlo como la línea de código del else pero tendría que volver a formatear la salida de las url de las imágenes

```
  async update(id: string, updateProductDto: UpdateProductDto) {

    const {images, ...toUpdate} = updateProductDto


    const product = await this.productRepository.preload({id, ...toUpdate})

    if(!product) throw new NotFoundException(`Product with id : ${id} not found`)

    const queryRunner = this.dataSource.createQueryRunner()
    await queryRunner.connect()
    await queryRunner.startTransaction()

    try {

      if(images){
        await queryRunner.manager.delete(ProductImage, {product: {id}}) //con esto
borramos las imágenes anteriores
```

```
        product.images= images.map(image=> this.productImageRepository.create({url: image}))

      }else{
          //product.images = await this.productImageRepository.findBy({product: {id}}) puedo hacerlo así pero usaré findOnePlain
      }

      await queryRunner.manager.save(product)

      await queryRunner.commitTransaction()  //commit
      await queryRunner.release() //desconexión

      return this.findOnePlane( id )

    } catch (error) {
      await queryRunner.rollbackTransaction()
      await queryRunner.release()

      this.handleDBExceptions(error)
    }
  }
```

- Si hago un update sin imágenes respeta las imágenes que había.
- Si añado la misma imágen le asigna un nuevo id

Eliminación en cascada

- Si quiero borrar un producto que tiene una imagen me da error
- Dice que borrar de la tabla producto viola la foreign key de la tabla product_image
- Hay varias formas de resolver este problema
  - Una de ellas es crear una transacción dónde primero borraría las imágenes y luego el producto
- También puedo decirle que al borrar un producto se borren las imágenes relacionadas
- Eliminar en cascada: cuando se afecta una tabla se afectan las demás relacionadas
- En product-image.entity no tengo definido que quiero que suceda en esta tabla si se borra el producto

```
import { Entity, PrimaryGeneratedColumn, Column, ManyToOne } from "typeorm";
import { Product } from "./product.entity";

@Entity()
export class ProductImage{

    @PrimaryGeneratedColumn() //va a tener un número autoincremental como id
    id: number

    @Column('text')
    url: string

    @ManyToOne(
```

```
        ()=> Product,
        product => product.images,
        {onDelete: 'CASCADE'} //le indico que borre en cascada
    )
    product: Product
}
```

- Creo un método en el servicio para borrar todos los productos
- Creo un queryBuilder y le pongo el alias de product
- Borro con query.delete en el where no le pongo nada para que borre todo, y lo ejecuto

```
async deleteAllProducts(){
    const query = this.productRepository.createQueryBuilder('product')

    try {
      return await query
              .delete()
              .where({})
              .execute()

    } catch (error) {
      this.handleDBExceptions(error)
    }
  }
```

Product Seed

- Copio del gist de Herrera la data

> https://gist.github.com/Klerith/1fb1b9f758bb0c5b2253dfc94f09e1b6

- Mi objetivo es apuntar a un endpoint que borre todos los registros anteriores e inserte estos

> nest g res seed --no-spec

- Borro dtos, entitys, todos los endpoints del controlador menos el GET, lo mismo en el servicio
- Nombro el GET como executeSeed (lo mismo en el servicio)
- controller

```
import { Controller,Get} from '@nestjs/common';
import { SeedService } from './seed.service';


@Controller('seed')
export class SeedController {
  constructor(private readonly seedService: SeedService) {}

  @Get()
```

```
  executeSeed() {
    return this.seedService.runSeed();
  }

}
```

- service

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


@Injectable()
export class SeedService {

  async runSeed() {
    return `SEED EXECUTED`;
  }

}
```

- Para borrar los productos necesito acceso al servicio para usar el metodo que creé para borrar todos los productos
- Para ello exporto el servicio en el módulo de Products con exports
- Puedo exportar también el TypeOrmModule si quisiera trabajar con los repositorios

```
import { Module } from '@nestjs/common';
import { ProductsService } from './products.service';
import { ProductsController } from './products.controller';
import {TypeOrmModule} from '@nestjs/typeorm'
import { Product, ProductImage } from './entities';


@Module({
  controllers: [ProductsController],
  providers: [ProductsService],
  imports: [
    TypeOrmModule.forFeature([
      Product, ProductImage
    ])
  ],
  exports: [ProductsService, TypeOrmModule] //exporto el servicio
})
export class ProductsModule {}
```

- E importo el módulo en el módulo de seed con imports

```
import { Module } from '@nestjs/common';
import { SeedService } from './seed.service';
import { SeedController } from './seed.controller';
import { ProductsModule } from 'src/products/products.module';

@Module({
  controllers: [SeedController],
  providers: [SeedService],
  imports:[ProductsModule]
})
export class SeedModule {}
```

- Creo un método privado en el servicio para encapsular la eliminación de los productos y lo ejecuto dentro del servicio

```
import { Injectable } from '@nestjs/common';
import { ProductsService } from 'src/products/products.service';


@Injectable()
export class SeedService {

  constructor(
    private readonly productsService: ProductsService
  ){}

  async runSeed() {

    this.insertNewProducts()

    return `SEED EXECUTED`;
  }

  private async insertNewProducts(){
   await  this.productsService.deleteAllProducts()
  }

}
```

- Esto es lo necesario para la eliminación

Insertar de forma masiva

- En seed creo un directorio llamado data con data-seed.ts con la data a insertar
- data-seed.ts

```
interface SeedProduct {
    description: string;
    images: string[];
    stock: number;
    price: number;
    sizes: ValidSizes[];
    slug: string;
    tags: string[];
    title: string;
    type: ValidTypes;
    gender: 'men'|'women'|'kid'|'unisex'
}

type ValidSizes = 'XS'|'S'|'M'|'L'|'XL'|'XXL'|'XXXL';
type ValidTypes = 'shirts'|'pants'|'hoodies'|'hats';


interface SeedData {
    products: SeedProduct[];
}


export const initialData: SeedData = {
    products: [
        {
            description: "Introducing the Tesla Chill Collection. The Men’s Chill Crew Neck Sweatshirt has a premium, heavyweight exterior and soft fleece interior for comfort in any season. The sweatshirt features a subtle thermoplastic polyurethane T logo on the chest and a Tesla wordmark below the back collar. Made from 60% cotton and 40% recycled polyester.",
            images: [
                '1740176-00-A_0_2000.jpg',
                '1740176-00-A_1.jpg',
            ],
            stock: 7,
            price: 75,
            sizes: ['XS','S','M','L','XL','XXL'],
            slug: "mens_chill_crew_neck_sweatshirt",
            type: 'shirts',
            tags: ['sweatshirt'],
            title: "Men’s Chill Crew Neck Sweatshirt",
            gender: 'men'
        },
        {
            description: "The Men's Quilted Shirt Jacket features a uniquely fit, quilted design for warmth and mobility in cold weather seasons. With an overall street-smart aesthetic, the jacket features subtle silicone injected Tesla logos below the back collar and on the right sleeve, as well as custom matte metal zipper pulls. Made from 87% nylon and 13% polyurethane.",
            images: [
                '1740507-00-A_0_2000.jpg',
                '1740507-00-A_1.jpg',
            ],
```

```
            stock: 5,
            price: 200,
            sizes: ['XS','S','M','XL','XXL'],
            slug: "men_quilted_shirt_jacket",
            type: 'shirts',
            tags: ['jacket'],
            title: "Men's Quilted Shirt Jacket",
            gender: 'men'
        }
     (etc..)
    ]
}
```

- Se podría hacer un insertMany pero tendríamos que insertar el modelo, hacer el repositorio, pero esto es algo que se va a ejecutar una sola vez
- En lugar de eso voy a llamar al método create del productService pasándole algo que luzca como el dto y ejecute el mismo procedimiento
- Guardo los productos del initialData en products con initialData.products
- La interfaz SeedProduct luce muy parecida al dto de create-product.dto

```
import { Injectable } from '@nestjs/common';
import { ProductsService } from 'src/products/products.service';
import { initialData } from './data/seed-data';


@Injectable()
export class SeedService {

  constructor(
    private readonly productsService: ProductsService
  ){}

  async runSeed() {

    this.insertNewProducts()

    const products = initialData.products

    const insertPromises = []

    products.forEach(product=>{
      insertPromises.push(this.productsService.create(product)) //create devuelve
una promesa. Las inserto en el arreglo
    })

    await Promise.all(insertPromises) //Ejecuto todas las promesas

    //Si quisiera el resultado de cada una de esas promesas podría guardar el
Promise.all en una constante results

    //const results = await Promise.all(insertPromises)
```

```
    return `SEED EXECUTED`;
  }

  private async insertNewProducts(){
   await  this.productsService.deleteAllProducts()
  }

}
```

- Añado el seed al README

```
# Teslo API

1. Clonar proyecto
2. ```npm install```
3. Clonar el archivo ```.env.template``` y renombrarlo a ```.env```
4. Cambiar las variables de entorno
5. Levantar la db
```
docker-compose up -d
```
6. Ejecutar SEED
```
localhost:3000/api/seed
```
7. Levantar ```npm run start:dev```
```

Renombrar tablas

- Las tablas, en lugar de llamarse product y product_image deberían llamarse products y product_images
- Voy a las entidades, primero a Product
- Si abro llaves en el decorador Entity puedo ver que hay varias opciones
    - database, engine, name, orderBy, schema, synchronize, etc
    - En este caso me interesa el name

```
import {Entity, PrimaryGeneratedColumn, Column, BeforeInsert, BeforeUpdate,
OneToMany} from 'typeorm'
import { ProductImage } from './product-image.entity'

@Entity({name: 'products'})
export class Product {
    @PrimaryGeneratedColumn('uuid')
    id: string

    @Column('text', {
        unique: true
    })
```

```
    title: string

    @Column('float',{
        default: 0
    })
    price: number
    (etc)
```

- Ahora aparecen las tablas products, product, product_image y product_images
- Las borro todas y hago el seed

---

09 NEST Carga de archivos

- A través de un POST con el nombre de la imagen (UUID) en la url voy a mostrar la fotografía
- Cuando quiero subir una imagen, selecciono el archivo y me devuelve la url que voy a poder utilizar para el frontend

---

Subir un archivo al backend

- Instalo estos tipos

```
npm i -D @types/multer
```

- La carga de archivos es general, por lo que tendrá su propio módulo

```
nest g res file --no-spec
```

- No voy a necesitar ni el dto ni la entity
- Borro todos los endpoints del controller y los métodos del servicio
- La carga de archivos se hará mediante una petición POST
- Le añado el endpoint product
- Recibirá el archivo de tipo **Express** (no hace falta importarlo)

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


@Controller('files')
export class FilesController {
  constructor(private readonly filesService: FilesService) {}

  @Post('product')
  uploadProductFile(file: Express.Multer.File){
    return file
  }
}
```

- Para enviar un archivo desde POSTMAN o ThunderClient, en Body, de tipo form-data, de key le pongo file y al lado puedes elegir el file a subir
- Para poder ver el archivo necesito un decorador, igual que necesito @Body o @Query
- En este caso es **@UploadedFile**
- Necesita saber el nombre de la llave, para esto vamos a usar un **interceptor**
- Los interceptores interceptan las solicitudes y también pueden interceptar y mutar las respuestas
- Dentro de **@UseInterceptors** uso **FileInterceptor** de nest/platform-express
- Debo indicarle el **nombre de la key** que le haya puesto, en este caso file

```
import { Controller, Get, Post, Body, Patch, Param, Delete, UploadedFile,
UseInterceptors } from '@nestjs/common';
import { FilesService } from './files.service';
import { FileInterceptor } from '@nestjs/platform-express';


@Controller('files')
export class FilesController {
  constructor(private readonly filesService: FilesService) {}

  @Post('product')
  @UseInterceptors(FileInterceptor('file'))
  uploadProductFile(@UploadedFile() file: Express.Multer.File){
    return file
  }
}
```

- Por defecto Nest sube el archivo a una carpeta temporal
- No se recomienda guardar el archivo en el filesystem por razones de seguridad
- Usar un servicio de terceros como Cloudinary

Validar archivos

- Cambio el valor del return (devolvía el file), para que me devuelva solo el nombre
- Quiero validar que no me suban un pdf, sólo imágenes
- Cómo esta validación es una tarea común se podría poner en el common
- Lo coloco en /files, ya que es algo que solo voy a usar en este módulo, creo la carpeta helpers
- Dentro creo el fileFilter.helper.ts
- Para poder usarlo en el FileInterceptor debo darle un aspecto característico
- Si coloco unas llaves después de 'file' dentro del FileInterceptor, y escribo fileFilter, la ayuda de Typescript me dice que tiene 3 argumentos
    - La request
    - El file
    - Callback, que tiene como argumentos el error y un boolean llamado acceptFile
- Esta función no regresa nada (void)
- Entonces, debo hacer la función fileFilter con estos 3 argumentos
- No hace falta que importe de express el request y demás porqué ya viene en Nest

```
export const fileFilter=(req: Express.Request, file: Express.Multer.File,
callback:Function )=>{

    console.log({file})

    callback(null, true)

}
```

- Se lo paso al FileInterceptor

```
import { Controller, Get, Post, Body, Patch, Param, Delete, UploadedFile,
UseInterceptors } from '@nestjs/common';
import { FilesService } from './files.service';
import { FileInterceptor } from '@nestjs/platform-express';
import { fileFilter } from './helpers/fileFilter.helper';


@Controller('files')
export class FilesController {
  constructor(private readonly filesService: FilesService) {}

  @Post('product')
  @UseInterceptors(FileInterceptor('file', {
    fileFilter: fileFilter //no lo estoy ejecutando, solo le paso la referencia
  }))
  uploadProductFile(@UploadedFile() file: Express.Multer.File){
    return {
      fileName: file.originalname
    }
  }
}
```

- Si hago el posteo veo en consola la info

```
{
  file: {
    fieldname: 'file',
    originalname: '01NEST_PRIMEROSPASOS.md',
    encoding: '7bit',
    mimetype: 'text/markdown'
  }
}
```

- Si el archivo no existe lanzo un error y le paso un false, que es el boolean diciendo que no aceptó el archivo
- Si el archivo existe, uso el split para dividirlo por / y me quedo con la segunda posición

- Creo un arreglo con las extensiones válidas
- Hago la comparación y devuelvo el calback con un true si el file es una imagen válida, si no con un false

```
export const fileFilter=(req: Express.Request, file: Express.Multer.File,
callback:Function )=>{

    if(!file) return callback(new Error('File is empty'), false)

    const fileExtension = file.mimetype.split('/')[1]
    const validExtensions = ['jpg', 'jpeg', 'png', 'gif']

    if(validExtensions.includes(fileExtension)){
        return callback(null, true)
    }

    callback(null, false)

}
```

- Esto no va alanzar una excepción por parte de Nest, solo me va a validar si el archivo es permitido o no
- Creo un badRequestException si no viene el file

```
import { Controller, Get, Post, Body, Patch, Param, Delete, UploadedFile,
UseInterceptors, BadRequestException } from '@nestjs/common';
import { FilesService } from './files.service';
import { FileInterceptor } from '@nestjs/platform-express';
import { fileFilter } from './helpers/fileFilter.helper';


@Controller('files')
export class FilesController {
  constructor(private readonly filesService: FilesService) {}

  @Post('product')
  @UseInterceptors(FileInterceptor('file', {
    fileFilter: fileFilter
  }))
  uploadProductFile(@UploadedFile() file: Express.Multer.File){

    if(!file) throw new BadRequestException('Make sure that the file is an image')

    return {
      fileName: file.originalname
    }
  }
}
```

- Podría poner la lista de extensiones en variables de entorno para poder expandir rapidamente la funcionalidad

- Ahora vamos a guardar fisicamente la imagen en el filesystem

---

Guardar imagen en filesystem

- Recuerdo que **no es recomendable guardar los archivos en el filesystem** en la vida real
- Lo recomendable es usar un servicio de terceros
- Yo podría crear la carpeta public/products y guardar las imágenes ahí
- Pero el problema es que **cualquier persona autenticada o no va a poder verlo**, porque es público
- En lugar de nombrarla public, la nombro static
- Dentro creo las carpetas uploads y products. Subiré los archivos a products
- Para subir el archivo voy al FileInterceptor
    - Hay muchas cosas que puedo establecer, como limits
    - En storage uso diskStorage de multer
    - Cuando uso ./ me refiero al root del proyecto

```
import { Controller, Get, Post, Body, Patch, Param, Delete, UploadedFile,
UseInterceptors, BadRequestException } from '@nestjs/common';
import { FilesService } from './files.service';
import { FileInterceptor } from '@nestjs/platform-express';
import { fileFilter } from './helpers/fileFilter.helper';
import { diskStorage } from 'multer';


@Controller('files')
export class FilesController {
  constructor(private readonly filesService: FilesService) {}

  @Post('product')
  @UseInterceptors(FileInterceptor('file', {
    fileFilter: fileFilter,
    limits: {fileSize: 10000},
    storage: diskStorage({
      destination: './static/products'
    })
  }))
  uploadProductFile(@UploadedFile() file: Express.Multer.File){

    if(!file) throw new BadRequestException('Make sure that the file is an image')

    return {
      fileName: file.originalname
    }
  }
}
```

- Si voy a la carpeta uploads veo que tengo la imagen
- La guarda con un nombre estraño (único) sin la extensión

- Se suele usat **.gitkeep** para **dar seguimiento a directorios que pueden estar vacíos**, porque git por defecto **no lo hace**
- Vamos a renombrar la imagen que estamos subiendo

Renombrar el archivo subido

- Copio el archivo fileFilter y lo renombro a fileNamer
- En este punto ya debería tener el archivo, pero dejo la validación por si acaso
- Necesito la extensión del archivo, por lo que uso el split

```
export const fileNamer=(req: Express.Request, file: Express.Multer.File,
callback:Function )=>{

    if(!file) return callback(new Error('File is empty'), false)

    const fileExtension = file.mimetype.split('/')[1]

    const fileName = `HolaMundo.${fileExtension}`


    callback(null, fileName)

}
```

- Coloco en la propiedad fileName de diskStorage la función

```
import { Controller, Get, Post, Body, Patch, Param, Delete, UploadedFile,
UseInterceptors, BadRequestException } from '@nestjs/common';
import { FilesService } from './files.service';
import { FileInterceptor } from '@nestjs/platform-express';
import { fileFilter } from './helpers/fileFilter.helper';
import { diskStorage } from 'multer';
import { fileNamer } from './helpers/fileNamer.helper';


@Controller('files')
export class FilesController {
  constructor(private readonly filesService: FilesService) {}

  @Post('product')
  @UseInterceptors(FileInterceptor('file', {
    fileFilter: fileFilter,
    limits: {fileSize: 10000},
    storage: diskStorage({
      destination: './static/uploads',
      filename: fileNamer
    })
  }))
  uploadProductFile(@UploadedFile() file: Express.Multer.File){
```

```
    if(!file) throw new BadRequestException('Make sure that the file is an image')

    return {
      fileName: file.originalname
    }
  }
}
```

- Para colocarle un identificador único como nombre de la imágen (en lugar de usar HolaMUndo y que lo vaya reescribiendo) vamos a usar uuid. Instalo también los tipos

```
npm i uuid @types/uuid
```

```
import {v4 as uuid} from 'uuid'


export const fileNamer=(req: Express.Request, file: Express.Multer.File,
callback:Function )=>{

    if(!file) return callback(new Error('File is empty'), false)

    const fileExtension = file.mimetype.split('/')[1]

    const fileName = `${uuid()}.${fileExtension}`


    callback(null, fileName)
}
```

- Si hago un console.log del file en el controlador veré toda la info
  - fieldname, originalname, encoding, mimetype, destination, filename, path, size
- Nadie desde afuera puede acceder al filesystem, ya que no está en la carpeta pública
- Vamos a ver como devolver la imagen.
- Hay toda una serie de validaciones, de autenticación que hay que hacer que no podría si estuvieran en una carpeta publica

---

Servir archivos de manera controlada

- No puedo usar el filename para servir el archivo porque no lo sé, solo estoy grabando el archivo en el filesystem
- Creo la constante secureURL en el uploadProductImage

```
uploadProductFile(@UploadedFile() file: Express.Multer.File){

  if(!file) throw new BadRequestException('Make sure that the file is an image')
```

```
    const secureURL = `${file.filename}`

    return {
      secureURL
    }
  }
```

- El endpoint sería un GET a api/files/product/:nombre_imagen

> localhost:3000/api/files/product/93479347329847UUID.png

- Creo el endpoint GET en el controller
- Me aseguro de recibir la imagen

```
@Get('product/:imageName')
  findProductImage(@Param('imageName') imageName: string ){
    return imageName
  }
```

- Debo verificar que la imagen exista en /products, voy al servicio para escribir el código
- Para eso debo especificar el path en el que me encuentra, está la función **join del path de node**
- Subo dos escalones en la jerarquía con ../../
- Uso **existSync del fs de node**

```
import { BadRequestException, Injectable } from '@nestjs/common';
import { existsSync } from 'fs';
import { join } from 'path';


@Injectable()
export class FilesService {


    getStaticProductImage(imageName: string){
        const path = join(__dirname, '../../static/products', imageName)

        if(!existsSync) throw new BadRequestException(`No product found with image
${imageName}`)

        return path
    }
}
```

- Podría usar un genérico para saber en qué carpeta buscar
- Uso el servicio en el controller

```
@Get('product/:imageName')
findProductImage(@Param('imageName') imageName: string ){

  const path = this.filesService.getStaticProductImage(imageName)

  return path
}
```

- Obtengo el path de mi computadora donde está el archivo en el controlador
- En lugar de regresa el path yo quiero regresar la imagen. Para eso haré uso de un nuevo decorador **@Res** de nest/common con el **Response de express**. Podría usar **Express.Response para no hacer la importación**
- En el momento que uso **@Res** rompo la funcionalidad de Nest, yo tomo el control de la respuesta manualmente
- Ahora puedo escribir mi respuesta como haría con Express

```
  @Get('product/:imageName')
  findProductImage(
    @Res() res: Response,
    @Param('imageName') imageName: string ){

    const path = this.filesService.getStaticProductImage(imageName)

     res.status(403).json({
      ok: false
     })
  }
```

- Hay que ir **con cuidado con usar Res** porque se salta ciertos interceptores y restricciones que usa Nest
- Uso sendFile para enviar el archivo que esté en el path

```
@Get('product/:imageName')
findProductImage(
  @Res() res: Response,
  @Param('imageName') imageName: string ){

  const path = this.filesService.getStaticProductImage(imageName)

    res.sendFile(path)
}
```

- De esta manera puedo usar la url de cloudinary o AWS, porque estoy ocultando donde está el archivo fisicamente
- Ahora, el secureURL debería ser este path
- Veamos como construir este url para que al subir el archivo quede listo para ser utilizado

- Hay más cosas que puedo hacer, como que cuando se eliminen verificar que las imágenes se han eliminado

---

Retornar el secureURL

- Puede ser que el puerto y la localización sean otros, con lo que es una url volátil
- La idea es que sea una variable de entorno

```
HOST_API=http://localhost:3000/api
PORT=3000
```

- Para usar las variables de entorno inyecto el servicio ConfigService en el controller

```
import { Controller, Get, Post, Param, Delete, UploadedFile, UseInterceptors,
BadRequestException, Res } from '@nestjs/common';
import { FilesService } from './files.service';
import { FileInterceptor } from '@nestjs/platform-express';
import { fileFilter } from './helpers/fileFilter.helper';
import { diskStorage } from 'multer';
import { fileNamer } from './helpers/fileNamer.helper';
import {Response} from 'express'
import { ConfigService } from '@nestjs/config';


@Controller('files')
export class FilesController {
  constructor(private readonly filesService: FilesService,
              private readonly configService: ConfigService
    ) {}

  @Post('product')
  @UseInterceptors(FileInterceptor('file', {
    fileFilter: fileFilter,
    limits: {fileSize: 10000},
    storage: diskStorage({
      destination: './static/products',
      filename: fileNamer
    })
  }))
  uploadProductFile(@UploadedFile() file: Express.Multer.File){

    if(!file) throw new BadRequestException('Make sure that the file is an image')

    const secureURL =
`${this.configService.get('HOST_API')}/files/product/${file.filename}`



    return {
```

```
      secureURL
    }
  }

  @Get('product/:imageName')
  findProductImage(
    @Res() res: Response,
    @Param('imageName') imageName: string ){

    const path = this.filesService.getStaticProductImage(imageName)

     res.sendFile(path)
  }
}
```

- Los módulos están encapsulados. Si quiero usar el servicio debo importar el módulo en files.module
- files.module

```
import { Module } from '@nestjs/common';
import { FilesService } from './files.service';
import { FilesController } from './files.controller';
import { ConfigModule } from '@nestjs/config';

@Module({
  controllers: [FilesController],
  providers: [FilesService],
  imports:[ConfigModule]
})
export class FilesModule {}
```

- Ahora cuando subo el archivo me responde con la secureUrl con el path donde se ubica el archivo

```
{
  "secureURL": "http://localhost:3000/api/files/product/23982de9-89be-4157-b1da-
cd8c629726b7.jpeg"
}
```

- Uso la variable de entorno PORT para el puerto del main
- Uso el logger para imprimir en consola el mensaje

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

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

```
  const logger = new Logger('bootstrap')

  app.setGlobalPrefix('api')

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

  await app.listen(process.env.PORT);
  logger.log(`App running on port ${process.env.PORT}`)
}
bootstrap();
```

- Si hago un GET de todos los productos y miro el arreglo de imágenes de los productos, estas imágenes no existen y no son urls

---

Otras formas de desplegar archivos

- Tengo un paquete comprimido con las imágenes que hacen match con la db
- Si yo sé que estos archivos no van a cambiar y siempre se van a servir de manera estática, accesible para todo el mundo, no hace falta hacer el Restful API
- Creo la carpeta public y copio dentro la carpeta products con todas las imágenes
- products no es el mejor nombre, ya que desde el front esa ruta podría estar tomada, le pongo assets
- Para servir contenido estático debo instalar @nestjs/serve-static y usar ServeStaticModule con el path

> @nestjs/serve-static

- En app.module

```
import { Module } from '@nestjs/common';
import {ConfigModule} from '@nestjs/config'
import { TypeOrmModule } from '@nestjs/typeorm';
import { ProductsModule } from './products/products.module';
import { CommonModule } from './common/common.module';
import { SeedModule } from './seed/seed.module';
import { FilesModule } from './files/files.module';
import { ServeStaticModule } from '@nestjs/serve-static';
import { join } from 'path';

@Module({
  imports: [
    ConfigModule.forRoot(),

    TypeOrmModule.forRoot({
      type: 'postgres',
      host: process.env.DB_HOST,
      port: +process.env.DB_PORT,
```

```
      database: process.env.DB_NAME,
      username: process.env.DB_USERNAME,
      password: process.env.DB_PASSWORD,
      autoLoadEntities: true,
      synchronize: true
    }),

    ProductsModule,

    CommonModule,

    SeedModule,

    FilesModule,

    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'public')
    })
  ],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

- Conviene crear un index.html en la carpeta public, aunque sea de prueba, para que no de error en consola
- En el endpoint localhost:3000/assets/nombre_del_archivo.jpeg en el navegador me muestra la imagen
- De esta manera no puedo controlar quien accede a las imágenes.
- Son recursos públicos, estáticos que no van a cambiar
- Así como lo tengo en la db, no puedo acceder a las imágenes
- Habría que crear un endpoint que deduzca que la imagen de la db es la que tengo guardada y colocar el url completo o actualizar las imágenes
- Una solución viable es actualizar las imágenes de la db, se podría hacer mediante el seed
- Básicamente sería añadirles el **http://localhost:3000/api/assets** y concatenar el campo url

```
update product_images set url = 'http://localhost:3000/api/assets' || url
```

- Pero de esta manera estoy grabando mucha data de manera innecesaria, ya que repite http://localhost:3000/api/assets en cada imagen

---

Colocar imágenes en el directorio estático

- Copio las imágenes y las pego en la carpeta static/products
- De hecho no necesito el directorio public, era solo con fines educativos
- Las imágenes ya tienen una referencia en la db (coincide el nombre)
- Si apunto al endpoint, obtengo la imagen

> http://localhost:3000/api/files/product/1473809-00-A_1_2000.jpg

10 NEST Autenticación

- En esta sección vamos a hacer decoradores personalizados
- Las rutas GET serán públicas, crear, actualizar y borrar si necesitarán autenticación de admin
- Vamos a hacer modificaciones en el SEED para crear usuarios automáticamente en la db y revalidar tokens (en realidad generar uno nuevo basado en el anterior)
- Van a haber varios endpoints nuevos como login, create user, check auth status
- También veremos encriptación de contraseñas
- Hay mucho concepto nuevo en esta sección

Entidad de usuarios

- Voy a proteger rutas. Habrá rutas que solo las podrán ver usuarios con el rol de administrador, por ejemplo
- El objetivo de la entidad es tener una relación entre la db y la aplicación de Nest
- Corresponde a una tabla en la db
- La renombro a user.entity
- Le coloco el decorador **@Entity** de , le paso el nombre 'users'
- No se recomienda usar el mail de id, porque este puede cambiar y dar dolores de cabeza
- Para decirle que es un identificador único uso el decorador **@PrimaryGeneratedColumn**
  - Si no le coloco nada será un numero autoincremental, vamos a manejarlo con uuid
- El isActive servirá para un borrado suave, donde permaneceran los datos pero con el isActive en false
- En el rol le pongo user como valor por defecto
- user.entity

```
import { Column, Entity, PrimaryGeneratedColumn, Unique } from "typeorm";

@Entity('users')
export class User{

    @PrimaryGeneratedColumn('uuid')
    id: string

    @Column('text',{
        unique: true
    })
    email: string


    @Column('text')
    password: string

    @Column('text')
    fullName: string
```

```
    @Column('bool',{
        default: true
    })
    isActive: boolean

    @Column('text',{
        array: true,
        default:['user']
    })
    roles: string[]
}
```

- Para usar la entidad debo especificar en el módulo en imports con **TypeOrmModule** y **forFeature** las entidades que quiero utilizar
- Lo exporto por si lo quiero usar en otro módulo
- En auth.module

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';

@Module({
  controllers: [AuthController],
  providers: [AuthService],
  imports: [
    TypeOrmModule.forFeature([User])
  ],
  exports: [TypeOrmModule]
})
export class AuthModule {}
```

Crear Usuario

- Para crear el usuario voy a usar el endpoint register

> http://localhost:3000/api/auth/register

- Lo añado al controlador
- Borro los dtos y creo CreateUserDto (actualizo también el servicio borrando todo menos el create)

```
import { Controller, Get, Post, Body, Patch, Param, Delete } from
'@nestjs/common';
import { AuthService } from './auth.service';
import { CreateUserDto } from './dto/create-user.dto';
```

```
@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}

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

- En el dto necesito el email, password y fullName
- Usaré una expresión regular para validar el password

```
import { IsEmail, IsString, Matches, MaxLength, MinLength } from "class-validator"
import { Unique } from "typeorm"

export class CreateUserDto{


    @IsEmail()
    email: string

    @IsString()
    @MinLength(1)
    fullName: string

    @IsString()
    @MinLength(6)
    @MaxLength(50)
    @Matches(
    /(?:(?=.*\d)|(?=.*\W+))(?![.\n])(?=.*[A-Z])(?=.*[a-z]).*$/, {
    message: 'The password must have a Uppercase, lowercase letter and a number'
    })
    password: string;
}
```

- Falta implementar la lógica en el servicio
- Siempre en un try catch, async
- El create **no hace la inserción**

```
import { Injectable } from '@nestjs/common';
import { CreateUserDto } from './dto/create-user.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { Repository } from 'typeorm';


@Injectable()
```

```
export class AuthService {

  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>
  ){}

  async create(createUserDto: CreateUserDto) {

    try {
     const user=  this.userRepository.create(createUserDto)

     await this.userRepository.save(user)

     return user

    } catch (error) {
      console.log(error)
    }
  }
}
```

- Evidentemente falta encriptar el password
- Si vuelvo a enviar el mismo usuario salta un error en la terminal, código 23505
- Manejemos la excepción

```
import { BadRequestException, Injectable, InternalServerErrorException } from
'@nestjs/common';
import { CreateUserDto } from './dto/create-user.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { Repository } from 'typeorm';


@Injectable()
export class AuthService {

  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>
  ){}

  async create(createUserDto: CreateUserDto) {

    try {
     const user=  this.userRepository.create(createUserDto)

     await this.userRepository.save(user)

     return user
```

```
    } catch (error) {
      this.handleDBErrors(error)
    }

  }

  private handleDBErrors(error: any):void{    //jamás regresa un valor
    if(error.code === '23505'){
      throw new BadRequestException(error.detail)
    }
    console.log(error)

    throw new InternalServerErrorException("Check logs") //no hace falta poner el
return

  }
}
```

Encriptar contraseña

- No debio regresar la contraseña y por supuesto, debo guardarla encriptada
- Usaremos encriptación de una sola vía con bcrypt. Instalo los tipos

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

- Importo todo como bcrypt (es una manera ligera de hacer el patrón adaptador)
- Uso la desestructuración para extraer el password
- hashSync me pide la data y el número de vueltas de encriptación, se lo paso en un objeto

```
import { BadRequestException, Injectable, InternalServerErrorException } from
'@nestjs/common';
import { CreateUserDto } from './dto/create-user.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { Repository } from 'typeorm';
import * as bcrypt from 'bcrypt'


@Injectable()
export class AuthService {

  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>
  ){}

  async create(createUserDto: CreateUserDto) {

    try {
```

```
    const {password, ...userData} = createUserDto

    const user=  this.userRepository.create({
      ...userData,
      password: bcrypt.hashSync(password, 12)
      })

     await this.userRepository.save(user)

     return user

    } catch (error) {
      this.handleDBErrors(error)
    }
  }

  private handleDBErrors(error: any):void{
    if(error.code === '23505'){
      throw new BadRequestException(error.detail)
    }
    console.log(error)

    throw new InternalServerErrorException("Check logs")
  }
}
```

- No debería regresar la contraseña
- Hay varias tecnicas
- Cuando ya se ha grabado el usuario extraigo el password
- Uso **delete**

```
async create(createUserDto: CreateUserDto) {

try {

const {password, ...userData} = createUserDto

const user=  this.userRepository.create({
    ...userData,
    password: bcrypt.hashSync(password, 12)
    })

    await this.userRepository.save(user)

    delete user.password

    return user
    //TODO: retornar JWT de acceso

} catch (error) {
    this.handleDBErrors(error)
```

```
    }
}
```

- Luego se mejorará este delete!

Login de usuario

- Creo el dto login-user.dto

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


export class LoginUserDto{

    @IsEmail()
    email: string

    @IsString()
    @MinLength(6)
    @MaxLength(50)
    @Matches(
    /(?:(?=.*\d)|(?=.*\W+))(?![.\n])(?=.*[A-Z])(?=.*[a-z]).*$/, {
    message: 'The password must have a Uppercase, lowercase letter and a number'
    })
    password: string
}
```

- En el controlador creo el endpoint 'login'

```
@Post('login')
loginUser(@Body() loginUserDto: LoginUserDto){
  return this.authService.loginUser(loginUserDto)
}
```

- Creo el servicio
- Si uso esto

```
 const user = await this.userRepository.findOneBy({email})
```

- Me devuelve el objeto completo, incluido el password y yo no quiero eso
- El problema es que cuando haga relaciones y mostremos la relación con el usuario también va a venir la contraseña y otras cosas
- Para evitarlo, voy a la entidad y en la propiedad contraseña le coloco select: false
- user.entity

```
@Column('text',{
    select: false
})
password: string
```

- Cuando se haga un find no aparecerá, pero yo ahora necesito el password para validar, por lo que usaré el **where** con **findOne**
- Le paso el mail (solo puede haber 1 y está indexado)
- Le digo que seleccione los campos email y password

```
async login(loginUserDto: LoginUserDto){

    const {email, password} = loginUserDto

    const user = await this.userRepository.findOne({
      where: {email},
      select: {email: true, password: true}
    })

    return user
  }
```

- Hago la validación de si existe usuario y la comparación del password con bcrypt. Si no concuerda devuelvo un error

```
async login(loginUserDto: LoginUserDto){

  const {email, password} = loginUserDto

  const user = await this.userRepository.findOne({
    where: {email},
    select: {email: true, password: true}
  })

  if(!user){
    throw new UnauthorizedException('Credenciales no válidas (email)')
  }

  if(!bcrypt.compareSync(password, user.password)){
    throw new UnauthorizedException('Password incorrect')
  }

  return user
  //TODO: retornar JWT
}
```

Nest Authentication - Passport

- Instalación necesaria

> npm i @nestjs/passport passport @nestjs/jwt passport-jwt npm i -D @types/passport-jwt

- Hay varias estrategias para autenticarse
- En authModule debo definir 2 cosas:
  - **PassportModule**: debo decirle la estrategia que voy a usar. Empleo register (registerAsync es para modulos asíncronos)
    - registerAsync se suele usar para asegurarse que las variables de entorno estan previamente configuradas
    - También si mi configuracion del módulo depende de un servicio externo, un endpoint, etc
  - **JwtModule**: para la palabra secreta usaré una variable de entorno. Expirará en 2 horas

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';

@Module({
  controllers: [AuthController],
  providers: [AuthService],
  imports: [
    TypeOrmModule.forFeature([User]),
    PassportModule.register({defaultStrategy: 'jwt'}),
    JwtModule.register({
      secret: process.env.JWT_SECRET,
      signOptions:{
        expiresIn: '2h'
      }
    })
  ],
  exports: [TypeOrmModule]
})
export class AuthModule {}
```

- Sería mejor usar la manera asíncrona de carga del módulo para asegurarme de que la variable de entorno estará cargada

---

Modulos asíncronos

- En registerAsync tengo opciones como useClass y useExisting muy utiles en la parte del testing
- Voy a usar useFactory, es la función que voy a llamar cuando se intente registrar de manera asincrona el módulo
  - En el return envío el objeto con las opciones del jwt

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';

@Module({
  controllers: [AuthController],
  providers: [AuthService],
  imports: [
    TypeOrmModule.forFeature([User]),
    PassportModule.register({defaultStrategy: 'jwt'}),
    JwtModule.registerAsync({
        imports: [],
        inject: [],
        useFactory: ()=>{

          return {
            secret: process.env.JWT_SECRET,
            signOptions:{
              expiresIn: '2h'
            }
          }
        }
    })
  ],
  exports: [TypeOrmModule]
})
export class AuthModule {}
```

- Puedo inyectar el configService como hice anteriormente para trabajar con las variables de entorno
- Para ello importo el módulo **ConfigModule** e inyecto el servicio en **injects**
- Hago la inyección del servicio igual que lo haría en cualquier clase solo que aqui estoy en una funcion
- ConfigService me da la posibilidad de recibir el dato que yo espero, poder evaluarlo, establecer valores por defecto, etc

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';
import { ConfigModule, ConfigService } from '@nestjs/config';

@Module({
  controllers: [AuthController],
  providers: [AuthService],
```

```
  imports: [
    TypeOrmModule.forFeature([User]),
    PassportModule.register({defaultStrategy: 'jwt'}),

    JwtModule.registerAsync({

        imports: [ConfigModule],
        inject: [ConfigService],
        useFactory: (configService: ConfigService)=>{

          return {
            secret: configService.get('JWT_SECRET'),
            signOptions:{
              expiresIn: '2h'
            }
          }
        }
    })
  ],
  exports: [TypeOrmModule]
})
export class AuthModule {}
```

- Falta saber qué información voy a guardar en el jwt, como validarlo y a qué usuario de la db le corresponde

---

JwtStrategy

- Es recomendable guardar en el jwt algun campo que esté indexado patra que identifique rapidamente al usuario
- Añadir también en qué momento fue creado y la fecha de expiración
- Nunca guardar info sensible: cadenas de conexión, tarjetas de crédito, passwords, etc
- La firma encriptada asegura que el valor no haya sido modificado y que haga match
- Me interesa saber que el usuario esté activo, el rol y el id a través de su correo
- **Solo guardaré el correo en el jwt**
- Vamos a emplear una estrategia personalizada
- En auth creo un nuevo directorio llamado strategies con jwt.strategy.ts
- Esta clase extiende de PassportStrategy (@nestjs/passport) y le paso la estrategia de passport-jwt

```
import {PassportStrategy} from '@nestjs/passport'
import { Strategy } from 'passport-jwt';


export class JwtStrategy extends PassportStrategy(Strategy){


}
```

- Quiero implementar una forma de expandir la validación de jwt
- El passportStrategy va a revisar el jwt basado en la secret_key, tambien si ha expirado o no y la Strategy me va a decir si el token es válido, pero hasta ahí
- Si yo necesito saber si el usuario está activo y todo lo demás, lo haré en base a un método (lo llamaré validate)
- El payload momentaneamente lo pondré de tipo any (lo cambiaré más adelante)
- Devuelve una promesa que va a devolverme una instancia de Usuario de mi db
- Si el jwt es válido y no ha expirado, voy a recibir este payload y puedo validarlo como yo quiera

```
import {PassportStrategy} from '@nestjs/passport'
import { Strategy } from 'passport-jwt';
import { User } from '../entities/user.entity';


export class JwtStrategy extends PassportStrategy(Strategy){

    async validate(payload: any): Promise<User>{

        return
    }
}
```

- Creo el directorio interfaces en /auth para hacer la interfaz del payload
- No voy a incluir la fecha de creación ni expiración

```
export interface JwtPayloadInterface{

    email: string
    //TODO: añadir todo lo que se quiera grabar
}
```

- Se procura que el jwt no lleve mucha info porque viaja de aquí para allá, que sea liviano
- Ahora puedo desestructurar el email del payload

JwtStrategy II

- Añado la lógica para validar el payload
- El método validate **solo se va a llamar si el jwt es válido (la firma hace match) y no ha expirado**
- Necesito ir a la tabla de usuarios y buscar el correo
- Ya tengo importado el módulo user en auth.module por lo que solo debo inyectar el repositorio de usuario
- PassportStrategy me pide invocar el **constructor padre**
- Como tengo que pasarle la secret_key como variable de entorno al constructor padre inyecto el **ConfigService**
- Importo el **ConfigModule en auth.module**

- Le debo indicar también al constructor padre en qué posición voy a esperar que me manden el jwt
    - Lo puedo mandar en los headers, o como un **header de autenticación** de tipo **Bearer Token**

```
import {PassportStrategy} from '@nestjs/passport'
import { ExtractJwt, Strategy } from 'passport-jwt';
import { User } from '../entities/user.entity';
import { JwtPayloadInterface } from '../interfaces/jwt-payload.interface';
import { Repository} from 'typeorm';
import { InjectRepository } from '@nestjs/typeorm';
import { ConfigService } from '@nestjs/config';

export class JwtStrategy extends PassportStrategy(Strategy){


    constructor(

        @InjectRepository(User)
        private readonly userRepository: Repository<User>,

        private readonly configService: ConfigService
    ){
        super({
            secretOrKey: configService.get('JWT_SECRET'),
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken()
        })
    }

    async validate(payload: JwtPayloadInterface): Promise<User>{

        const {email} = payload

         const user = await this.userRepository.findOneBy({email})

        return
    }
}
```

- **Debo importar ConfigModule en imports del auth.module** (no solo en el JwtModule) ya que lo uso en este módulo
- Ahora ya puedo implementar la lógica, validar el usuario, etc
- No tengo el password. Si el token existe significa que el usuario se autenticó en su momento
- Retorno el usuario. Cuando la validación lo que yo retorne se va a añadir en la Request
    - Pasa por interceptores, por los servicios, controladores, **todo lugar donde tenga acceso a la Request**
    - Después se usarán decoradores personalizados para extraer info de la Request y hacer lo que hago en los controladores
- Todavía no he implementado el JwtStrategy, es un archivo flotando en mi app

```
async validate(payload: JwtPayloadInterface): Promise<User>{

      const {email} = payload

        const user = await this.userRepository.findOneBy({email})

        if(!user) throw new UnauthorizedException('Token not valid')

        if(!user.isActive) throw new UnauthorizedException('User is inactive')

      return user
  }
```

- Todas las estrategias son **providers**. Le añado el decorador **@Injectable**
- Como es un provider, debo indicarlo en **el módulo auth.module dónde providers**
- También lo exporto por si quiero usarlo en otro lugar. Exporto los otros módulos
- Después lo vamos a mejorar para que todo sea automático

```
@Module({
  controllers: [AuthController],
  providers: [AuthService, JwtStrategy],
  imports: [
    ConfigModule,
    TypeOrmModule.forFeature([User]),
    PassportModule.register({defaultStrategy: 'jwt'}),

    JwtModule.registerAsync({

        imports: [ConfigModule],
        inject: [ConfigService],
        useFactory: (configService: ConfigService)=>{

          return {
            secret: configService.get('JWT_SECRET'),
            signOptions:{
              expiresIn: '2h'
            }
          }
        }
    })
  ],
  exports: [TypeOrmModule, JwtStrategy, PassportModule, JwtModule]
})
export class AuthModule {}
```

Generar un JWT

- Como voy a crear un jwt en varios lugares voya crear un método en auth.service.ts

- Debo recibir **el payload** con la **info** que quiero en el jwt del tipo **JetPayloadInterface**
- Para generar el token necesito usar el servicio de jwt de nest, hago la inyección de dependencias
- Este servicio lo proporciona el JwtModule
- Uso el servicio con el método sign. Aquí podría pasarle parámetros pero si no queda por defecto como definí en el módulo
- Esparzo con el spread mi user en el return, y añado el token
    - Si coloco directamente donde el payload user.email se me queja porque un string no cumple con el objeto de jwtPayloadInterface, así que lo meto como un objeto *{token: user.email}*
    - Hago lo mismo en el login

```
import { BadRequestException, Injectable, InternalServerErrorException,
NotFoundException, UnauthorizedException } from '@nestjs/common';
import { CreateUserDto } from './dto/create-user.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { Repository } from 'typeorm';
import * as bcrypt from 'bcrypt'
import { LoginUserDto } from './dto/login-user.dto';
import { NotFoundError } from 'rxjs';
import { JwtPayloadInterface } from './interfaces/jwt-payload.interface';
import { JwtService } from '@nestjs/jwt';


@Injectable()
export class AuthService {

  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>,

    private readonly jwtService: JwtService
  ){}

  async create(createUserDto: CreateUserDto) {

    try {

    const {password, ...userData} = createUserDto

    const user=  this.userRepository.create({
      ...userData,
      password: bcrypt.hashSync(password, 12)
      })

     await this.userRepository.save(user)

     delete user.password

       return {
      ...user,
      token: this.getJwt({email: user.email})
```

```
    }

    } catch (error) {
      this.handleDBErrors(error)
    }
  }

  async login(loginUserDto: LoginUserDto){

    const {email, password} = loginUserDto

    const user = await this.userRepository.findOne({
      where: {email},
      select: {email: true, password: true}
    })

    if(!user){
      throw new UnauthorizedException('Credenciales no válidas (email)')
    }

    if(!bcrypt.compareSync(password, user.password)){
      throw new UnauthorizedException('Password incorrect')
    }

       return {
      ...user,
      token: this.getJwt({email: user.email})
    }
  }

  //generar JWT
  private getJwt(payload: JwtPayloadInterface){
      const token = this.jwtService.sign(payload)
      return token
  }


  private handleDBErrors(error: any):void{
    if(error.code === '23505'){
      throw new BadRequestException(error.detail)
    }
    console.log(error)

    throw new InternalServerErrorException("Check logs")
  }
}
```

- Voy al login y coloco usuario y contraseña correctos, en consola me devuelve email, password y el token!
- Quiero guardar todo en minúsculas
- Lo hago en la entidad directamente con **@BeforeInsert**
- Como en el **@BeforeUpdate** es el mismo código llamo al método anterior

```
import { BeforeInsert, BeforeUpdate, Column, Entity, PrimaryGeneratedColumn,
Unique } from "typeorm";

@Entity('users')
export class User{

    @PrimaryGeneratedColumn('uuid')
    id: string

    @Column('text',{
        unique: true
    })
    email: string


    @Column('text',{
        select: false
    })
    password: string

    @Column('text')
    fullName: string

    @Column('bool',{
        default: true
    })
    isActive: boolean

    @Column('text',{
        array: true,
        default:['user']
    })
    roles: string[]

    @BeforeInsert()
    checkFieldsBeforeInsert(){
        this.email = this.email.toLowerCase().trim()
    }

    @BeforeUpdate()
    checkFieldsBeforeUpdate(){
        this.checkFieldsBeforeInsert()
    }
}
```

Priovate Route - General

- Creo mi primera ruta privada que su único objetivo va a asegurar de que hay un jwt, que el usuario esté activo y el token no haya expirado (más adelante se evaluará tambien el rol)
- Voy a usar Get y la llamaré testingPrivateRoute

- auth.controller

```
  @Get('private')
  testingPrivateRoute(){
    return {
      ok: true
    }
  }
```

- Los **Guards** son usados para permitir o prevenir el acceso a una ruta
- **Es dónde se debe de autorizar una solicitud**
- Autenticación y autorización **no son lo mismo**
- Autenticado es cuando el usuario está validado y autorizado es que tiene permiso para acceder
- Para usar el **guard** uso el decorador **@UseGuards** de @nestjs/common (por el momento, se hará un guard personalizado)
- Uso AuthGuard de @nestjs/passport, que usa la estrategia que yo definí por defecto, la configuración que definí, etc
- Para probarlo en Postman/ThunderClient debo añadir el token proporcionado en el login en Auth donde dice Bearer
- Si le cambio el isActive a **FALSE** y le paso el token adecuado, me devuelve un error controlado diciendo que no estoy autorizado porque mi usuario está inactivo
- **Pero de dónde sale eso?**
- Recuerda que en la estrategia, en el validate hago la verificación
- Es la estrategia que está usando por defecto el **Guard**

```
import {PassportStrategy} from '@nestjs/passport'
import { ExtractJwt, Strategy } from 'passport-jwt';
import { User } from '../entities/user.entity';
import { JwtPayloadInterface } from '../interfaces/jwt-payload.interface';
import { Repository} from 'typeorm';
import { InjectRepository } from '@nestjs/typeorm';
import { ConfigService } from '@nestjs/config';
import {UnauthorizedException, Injectable} from '@nestjs/common'


@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy){


    constructor(

        @InjectRepository(User)
        private readonly userRepository: Repository<User>,

        private readonly configService: ConfigService
    ){
        super({
            secretOrKey: configService.get('JWT_SECRET'),
```

```
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken()
        })
    }

    async validate(payload: JwtPayloadInterface): Promise<User>{

        const {email} = payload

         const user = await this.userRepository.findOneBy({email})

         if(!user) throw new UnauthorizedException('Token not valid')

         if(!user.isActive) throw new UnauthorizedException('User is inactive')

        return user
    }
}
```

- Si cambio la secret_key de la variable de entorno, el mismo token va a dar un error de autenticación "Unauthorized"
- Esto esta bien que sea asi

Cambiar el email por el id en el payload

- El email puede cambiar, por lo que conviene usar el uuid
- En el payload del jwt en lugar del email debe ir el uuid. Va a haber que actualizar la estrategia
- Primero, cuando hago el user de retorno en el login debo pedir también el id
- Cambio el email por el id en la generación del token en el return
    - Me marca error porque la interfaz me pide el email. Cambio la interfaz

```
async login(loginUserDto: LoginUserDto){

    const {email, password} = loginUserDto

    const user = await this.userRepository.findOne({
      where: {email},
      select: {email: true, password: true, id: true}
    })

    if(!user){
      throw new UnauthorizedException('Credenciales no válidas (email)')
    }

    if(!bcrypt.compareSync(password, user.password)){
      throw new UnauthorizedException('Password incorrect')
    }

    return {
      ...user,
```

```
      token: this.getJwt({id: user.id})
    }
  }
```

- Hago lo mismo en el método create(en lugar del {mail: user.email},{id: user.id}
- Cambio la interfaz

```
export interface JwtPayloadInterface{

    id: string
}
```

- Falta cambiar la estrategia, ya que desestructuro el email y ya no lo tengo
- Cambio email por id

```
import {PassportStrategy} from '@nestjs/passport'
import { ExtractJwt, Strategy } from 'passport-jwt';
import { User } from '../entities/user.entity';
import { JwtPayloadInterface } from '../interfaces/jwt-payload.interface';
import { Repository} from 'typeorm';
import { InjectRepository } from '@nestjs/typeorm';
import { ConfigService } from '@nestjs/config';
import {UnauthorizedException, Injectable} from '@nestjs/common'


@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy){


    constructor(

        @InjectRepository(User)
        private readonly userRepository: Repository<User>,

        private readonly configService: ConfigService
    ){
        super({
            secretOrKey: configService.get('JWT_SECRET'),
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken()
        })
    }

    async validate(payload: JwtPayloadInterface): Promise<User>{

        const {id} = payload

         const user = await this.userRepository.findOneBy({id})
```

```
            if(!user) throw new UnauthorizedException('Token not valid')

            if(!user.isActive) throw new UnauthorizedException('User is inactive')

          return user
      }
  }
```

- Vuelvo a generar un token, lo pruebo y debería ver la respuesta
- Cuando me autentique en una ruta siempre va a pasar por el JwtStrategy. Ahi ya tengo el usuario, puedo hacer un console.log
- Ahora, si cambia el correo no tengo problema
- Veamos cómo puedo obtener el usuario en los controladores y dónde necesite

---

Custom Property Decorator - GetUser

- Puedo extraer el usuario del **Guard**
- Si se me olvidara que tengo implementado el Guard y quisiera extraer el usuario, debería lanzar un error propio.
    - Es un problema que yo como desarrollador del backend debo resolver
- Hay varias formas. Puedo escribir nest -h para ver la ayuda y usar el CLI

> nest g d nombre_decorador

- Pero este decorador funciona de manera global, por clase y por controlador
- No funciona para propiedad
- **Para extraer el usuario** usaré **@Request** de @nestjs/common
- Si hago un console.log de la request me manda un montón de info en consola

```
  @Get('private')
  @UseGuards( AuthGuard())
  testingPrivateRoute(
    @Request() request: Express.Request
  ){
    return {
      ok: true
    }
  }
```

- **request.user** me devuelve el usuario
- Esto así funcionaría pero no es muy bonito
- Además necesito pasar por el Guard, por lo que habría que hacer un par de validaciones también
- Mejor creemos un **Custom Property Decorator**
- auth/decorators/get-user.decorator.ts
- El createParamDecorator es una función que usa un callback que debe retornar algo

```
import { createParamDecorator } from "@nestjs/common";

export const GetUser = createParamDecorator(
    ()=>{

        return 'Hola mundo'
    }
)
```

- En el controller

```
  @Get('private')
  @UseGuards( AuthGuard())
  testingPrivateRoute(
        @GetUser() user: User
  ){
    console.log({user})  //imprime en consola Hola mundo

    return {
      ok: true
    }
  }
```

- Lo que sea que retorne createParamDecorator es lo que voy a poder extraer
- En el callback de createParamDecorator dispongo de la data y el context (lo importo de @nestjs/common)

```
import { ExecutionContext, createParamDecorator } from "@nestjs/common";

export const GetUser = createParamDecorator(
    (data, ctx: ExecutionContext)=>{
        console.log({data})

    }
)
```

- La consola me devuelve data: undefined.
- Si voy al controlador y escribo 'email' en el decorador **@GetUser('email)** la consola me devuelve data: 'email'
- Puedo pasarle todos los argumentos que quiera en un arreglo

```
  @Get('private')
  @UseGuards( AuthGuard())
  testingPrivateRoute(
        @GetUser(['email', 'role', 'fullName']) user: User
  ){
```

```
    console.log({user})
    return {
      ok: true,
      user
    }
  }
```

- El **ExecutionContext** es el contexto en el que se está ejecutando la función en la app
- Tengo, entre otras cosas, **la Request** (tambien la Response)
- Uso **switchToHttp.getRequest** para extraer la Request. Usaría getResponse para la Response
- Lanzo un error 500 si no está el usuario porque es un error mío ya que debería haber pasado por el Guard

```
import { ExecutionContext, InternalServerErrorException, createParamDecorator }
from "@nestjs/common";

export const GetUser = createParamDecorator(
    (data, ctx: ExecutionContext)=>{

        const req = ctx.switchToHttp().getRequest()

        const user = req.user

        if(!user) throw new InternalServerErrorException('User not found')

        return user
    }
)
```

Tarea Custom Decorators

- Quiero usar el @GetUser dos veces en el mismo endpoint en el controller
- Una sin pasarle ningún argumento que me devuelva el User completo
- Otra pasándole solo el email como parámetro a @GetUser para que me devuelva el email
- Podría usar los Pipes para validar/transformar la data pertfectamente, pero no es el caso

```
  @Get('private')
  @UseGuards( AuthGuard())
  testingPrivateRoute(
        @GetUser() user: User,
        @GetUser('email') email: string
  ){
    console.log({user})
    return {
      ok: true,
      user
```

```
    }
  }
```

- Uso un ternario para devolver si no hay data el user, y si la hay user[propiedad_computada]
- get-user.decorator.ts

```
import { ExecutionContext, InternalServerErrorException, createParamDecorator }
from "@nestjs/common";

export const GetUser = createParamDecorator(
    (data, ctx: ExecutionContext)=>{

        const req = ctx.switchToHttp().getRequest()

        const user = req.user

        if(!user) throw new InternalServerErrorException('User not found')

        return (!data) ? user : user[data]
    }
)
```

- Si hago un console.log de la Request usando el decorador @Request y lo imprimo en consola, puedo crear un decorador que me devuelva lo que yo quiera de ella, por ejemplo los rawHeaders
- Aunque es un decorador que iría más bien en el módulo common, lo pondré junto al otro decorador por tenerlos agrupados
- get-rawheaders.decorator.ts

```
import { ExecutionContext, createParamDecorator } from "@nestjs/common";


export const GetRawHeaders = createParamDecorator(
    (data, ctx: ExecutionContext)=>{

        const req = ctx.switchToHttp().getRequest()


        return  req.rawHeaders
    }
)
```

- auth.controller

```
  @Get('private')
  @UseGuards( AuthGuard())
  testingPrivateRoute(
```

```
        @GetUser() user: User,
        @GetUser('email') email: string,
        @GetRawHeaders() rawHeaders: string[]
  ){
    return {
      ok: true,
      user,
      email,
      rawHeaders
    }
  }
```

- Nest ya tiene su propio decorador **@Headers** para los headers (de @nestjs/common)
- El tipo de headers es IncomingHttpHeaders (importar de http)

Custom Guard y Custom Decorator

- En este momento, si yo quisiera validar el rol podría hacerlo en el controlador con user.roles.includes('admin), por ejemplo
- Pero voy a crear un Guard y un Custom Decorator para esta tarea
- Creo otro Get en el auth.controller

```
@Get('private2')
@UseGuards(AuthGuard())
privateRoute2(
  @GetUser() user: User,
){
  return{
    ok: true,
    user
  }
}
```

- Este Get necesita tener ciertos roles, y quiero crear un decorador que los valide
- Puedo usar **@SetMetaData**

```
@Get('private2')
@UseGuards(AuthGuard())
@SetMetadata('roles', ['admin'])
privateRoute2(
  @GetUser() user: User,
){
  return{
    ok: true,
    user
  }
}
```

- Con esto no es suficiente, debo crear un Guard para que lo evalue
- Puedo hacerlo con el CLI usando gu

> nest g gu auth/guards/userRole --no-spec

- Esto genera por mi

```
import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class UserRoleGuard implements CanActivate {

  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {

    console.log('UserGuard')

    return true;
  }
}
```

- Para que un Guard sea válid tiene que implementar canActivate
- Tiene que retornar un boolean o una Promesa que sea un boolean, si es true lo deja pasar si no no
- También puede devolver un Observable que emita un boolean
- Los Guards por defecto son async
- Coloco el userRoleGuard en el controlador

```
@Get('private2')
@UseGuards(AuthGuard(), UserRoleGuard)
@SetMetadata('roles', ['admin'])
privateRoute2(
  @GetUser() user: User,
){
  return{
    ok: true,
    user
  }
}
```

- Porqué no lleva paréntesis?
- Podría generar una nueva instancia con new
- **AuthGuard ya devuelve la instancia**, por lo que los Guards personalizados no llevan paréntesis, **para usar la misma instancia**
- Se puede hacer usando el new pero eso lo que haría es generar una nueva instancia, y lo que queremos es usar la misma

- Si ejecuto el endpoint private2 con el token en consola imprime el console.log, con lo que ha pasado por el Guard
- Los Guards se encuentran dentro del ciclo de vida de Nest
    - Están dentro de la **Exception Zone**
    - Significa que si devolviera un error en lugar del true va a ser controlado por Nest (BadRequestException o lo que fuera)
- Este Guard se va a encargar de verificar los roles.
- Para ello primero debo extraer la metadata del decorador **@SetMetadata**
- Aquí **no se pone fácil la cosa**. **Tirando de documentación**
- Inyecto Reflector en el constructor
- Lo uso para guardar en la variable roles con el .get('roles') (lo que pone en **@SetMetadata**) y el target es context.getHandler()

```
import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Observable } from 'rxjs';

@Injectable()
export class UserRoleGuard implements CanActivate {

  constructor(
    private readonly reflector: Reflector
  ){}


  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {

    const validRoles: string[] = this.reflector.get('roles', context.getHandler()
)

    console.log({validRoles}) //para testear que los haya extraído bien

    return true;
  }
}
```

- Ahora lo que debo hacer es comparar si existen en el arreglo de roles de mi entidad
- Si no existe niguno voy a devolver un error

Verificar Rol del usuario

- Para obtener el usuario es el mismo código de **ctx.switchToHttp().getrequest()**
- Tipo el usuario con **as User** así obtengo el completado también
- Verifico que venga el usuario para asegurarme de que se usa el Guard de autenticación
- Uso un ciclo for para recorrer el array y verificar el rol
- Si no es un role valido lanzaré un ForbiddenException

```
import { BadRequestException, CanActivate, ExecutionContext, ForbiddenException,
Injectable } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Observable } from 'rxjs';
import { User } from 'src/auth/entities/user.entity';

@Injectable()
export class UserRoleGuard implements CanActivate {

  constructor(
    private readonly reflector: Reflector
  ){}


  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {

    const validRoles: string[] = this.reflector.get('roles', context.getHandler()
)

    const req = context.switchToHttp().getRequest()
    const user = req.user as User

    if(!user) throw new BadRequestException('User not found')

    for(const role of user.roles){
      if(validRoles.includes(role)){
        return true
      }
    }
    throw new ForbiddenException(`User ${user.fullName} needs a valid role`)
  }
}
```

- Para que lo deje pasar añado en TablePlus el role de admin al usuario
- Para usar esta lógica que estoy implementando tengo que memorizar muchas cosas. Establecer el SetMetadata, etc
- Si me olvidara del SetMetadata, al extraer los validRoles mi app reventaría. Debería validarlo
- También es muy volatil el arreglo de roles, me puedo equivocar. El SetMetadata se usa muy poco como decorador directamente
- Mejor crear un **Custom Decorator**

Custom Decorator RoleProtected

- Si no son decoradores de propiedades, perfectamente puedo usar el CLI
- ¿El decorador que voy a crear esta fuertemente ligado al módulo auth o es algo general que podría ir en common?
    - Me va aservir para establecer los roles que el usuario ha de tener para poder ingresar a la ruta

- Por lo que SI está amarrado al módulo de auth

> nest g d auth/decorators/roleProtected --no-spec

- Esto me genera este código

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

export const RoleProtected = (...args: string[]) => SetMetadata('role-protected',
args);
```

- Cambio 'role-protected' en el SetMetadata por 'roles'
- Defino el string con una variable para tenerla en un solo lugar, por si hubiera cambios
- Importo META_ROLES en el UserRoleGuard para añadirlo en el this.reflector.get

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

export const META_ROLES= 'roles'

export const RoleProtected = (...args: string[]) =>{

    SetMetadata(META_ROLES, args);
}
```

- Creo una enum en la carpeta de interfaces para especificar los roles que voy a permitir
- Tienen que ser strings, Typescript les asigna un número 0,1,2

```
export enum ValidRoles{

    admin= 'admin',
    superUser= 'super-user',
    user= 'user'
}
```

- Le paso el enum como tipo como parámetro del decoradorrole-protected

```
import { SetMetadata } from '@nestjs/common';
import { ValidRoles } from '../interfaces/valid-roles';

export const META_ROLES= 'roles'

export const RoleProtected = (...args: ValidRoles[]) =>{

   return  SetMetadata(META_ROLES, args);
}
```

- Uso el **@RoleProtected** en el controller
- Si lo pusiera sin parámetros, cualquier usuario tendría acceso a la ruta
- Uso el **enum**

```
@Get('private2')
@UseGuards(AuthGuard(), UserRoleGuard)
@RoleProtected(ValidRoles.admin)
privateRoute2(
  @GetUser() user: User,
){
  return{
    ok: true,
    user
  }
}
```

- Puedo pasarle varios valores separados por comas, **@RoleProtected(ValidRoles.admin, ValidRoles.user)**
- Es fácil que me olvide de implementar el AuthGuard (autenticación), o el RoleProtected(autorización)
- Podemos crear un único decorador que lo haga todo

Composición de decoradores

- Con **applyDecorators de @nestjs/common** podemos hacer composición de decoradores
- Muy útil para agrupar varios decoradores en uno
- Creo un tercer endpoint privateRoute3
    - Va a funcionar igual solo que en lugar de tener tantos decoradores tendrñe uno que haga todo el trabajo
- controller

```
@Get('private3')
@UseGuards(AuthGuard(), UserRoleGuard)
@RoleProtected(ValidRoles.admin)
privateRoute3(
  @GetUser() user: User,
){
  return{
    ok: true,
    user
  }
}
```

- Creo el auth.decorator.ts en /auth/decorators/
- En lugar de usar el SetMetadata puedo usar el RoleProtected
- AuthGuard de @nestjs/passport hay que ejecutarlo porque así funciona
- Le paso jwt

```
import { UseGuards, applyDecorators } from "@nestjs/common";
import { META_ROLES, RoleProtected } from "./role-protected.decorator";
import { AuthGuard } from "@nestjs/passport";
import { ValidRoles } from "../interfaces/valid-roles";
import { UserRoleGuard } from "../guards/user-role/user-role.guard";



export function Auth(...roles: ValidRoles[]){

    return applyDecorators(
        RoleProtected(...roles),
        UseGuards(AuthGuard('jwt'), UserRoleGuard)
    )
}
```

- Lo uso en el controller
- Si lo envio sin nada entre paréntesis debe querer decir que no necesita ningún role especial y pasar

```
  @Get('private3')
  @Auth()
  privateRoute3(
    @GetUser() user: User,
  ){
    return{
      ok: true,
      user
    }
  }
```

- NOTA: EN USERROLEGUARD FALTABAN DOS LINEAS DE CÓDIGO PARA QUE PUEDA PASAR SIN ROLES
- UserRoleGuard

```
import { BadRequestException, CanActivate, ExecutionContext, ForbiddenException,
Injectable } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Observable } from 'rxjs';
import { META_ROLES } from 'src/auth/decorators/role-protected.decorator';
import { User } from 'src/auth/entities/user.entity';

@Injectable()
export class UserRoleGuard implements CanActivate {

  constructor(
    private readonly reflector: Reflector
  ){}
```

```
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {

    const validRoles: string[] = this.reflector.get(META_ROLES,
context.getHandler() )

    //faltaba este código!!!
    if(!validRoles) return true  //<----------------
    if (validRoles.length === 0) return true //<-----------
    //

    const req = context.switchToHttp().getRequest()
    const user = req.user as User

    if(!user) throw new BadRequestException('User not found')

    for(const role of user.roles){
      if(validRoles.includes(role)){
        return true
      }
    }
    throw new ForbiddenException(`User ${user.fullName} needs a valid role`)
  }
}
```

- Debo estar autenticado con el token
- Si quiero que deba tener algún role en particular uso el ValidRoles

```
@Get('private3')
@Auth(ValidRoles.admin)
privateRoute3(
  @GetUser() user: User,
){
  return{
    ok: true,
    user
  }
}
```

- Si no pusiera el @Auth tendría un error porque necesitamos el usuario en la Request
- Para usarlo en otros endpoints de otros módulos, como el SEED por ejemplo, que solo debería hacerlo el admin, debo **importar el PassportModule** en el módulo dónde quiera utilizar el **@Auth**

Auth en otros módulos

- Quiero usar en mi **SeedController** el decorador **@Auth**
  - **@Auth** está usando **@AuthGuard** que está asociado a **Passport**, y Passport es un **módulo**

    - En el error en consola al intentar usar **@Auth** fuera del módulo lo que está pidiendo el el **"defaultStrategy"**
- En el modulo de **Auth** tengo exportado el **JwtStrategy** y el **PassportModule**

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';
import { ConfigModule, ConfigService } from '@nestjs/config';
import { JwtStrategy } from './strategies/jwt.strategy';

@Module({
  controllers: [AuthController],
  providers: [AuthService, JwtStrategy],
  imports: [
    ConfigModule,
    TypeOrmModule.forFeature([User]),
    PassportModule.register({defaultStrategy: 'jwt'}),

    JwtModule.registerAsync({

        imports: [ConfigModule],
        inject: [ConfigService],
        useFactory: (configService: ConfigService)=>{

          return {
            secret: configService.get('JWT_SECRET'),
            signOptions:{
              expiresIn: '2h'
            }
          }
        }
    })
  ],
  exports: [TypeOrmModule, JwtStrategy, PassportModule, JwtModule]
})
export class AuthModule {}
```

- Es lo que necesito para exponer todo lo que está relacionado a Passport fuera de este módulo
- Importo AuthModule en el módulo de SEED. Es todo

```
import { Module } from '@nestjs/common';
import { SeedService } from './seed.service';
import { SeedController } from './seed.controller';
import { ProductsModule } from 'src/products/products.module';
import { AuthModule } from 'src/auth/auth.module';
```

```
@Module({
  controllers: [SeedController],
  providers: [SeedService],
  imports:[ProductsModule, AuthModule]
})
export class SeedModule {}
```

- Ahora puedo usar el decorador **@Auth** en el SEED controller

```
import { Controller,Get} from '@nestjs/common';
import { SeedService } from './seed.service';
import { Auth } from 'src/auth/decorators/auth.decorators';
import { ValidRoles } from 'src/auth/interfaces/valid-roles';


@Controller('seed')
export class SeedController {
  constructor(private readonly seedService: SeedService) {}

  @Get()
  @Auth(ValidRoles.admin)
  executeSeed() {
    return this.seedService.runSeed();
  }
}
```

- Si quiero proteger las rutas de productos solo tengo que repetir el procedimiento
- Si lo que quiero es que para cualquiera de las rutas el usuario **deba estar autenticado** coloco @Auth en el Controlador padre (sin ningún rol como parámetro)
- El usuario deberá tener el token de autorización (independientemente del rol)
- Falta crear en el Seed una forma de crear usuarios admin

Usuario que creó el producto

- Sería útil saber qué usuario creó el producto. Tenemos una autenticación en marcha que me lo puede decir
- Cómo se relaciona un usuario con un producto. Un usuario puede crear muchos productos
- Es una relación de uno a muchos **OneToMany**
- En productos, muchos productos pueden ser de un usuario, por lo que es una relación de muchos a uno **ManyToOne**
- En user.entity (en el módulo auth)
- El OneToMany no va a hacer que cree ningún valor nuevo en la columna, pero en Product si. Importo Product para tipar el valor
- Lo primero que debo añadir es **la relación con la otra entidad**
- Luego, como la entidad se relaciona con esta tabla, seria **product.user**, pero este user no existe todavía

```
@OneToMany(
    ()=>Product,
    (product)=> product.user //<--------este .user no existe todavía
)
product: Product
```

- En product.entity

```
@ManyToOne(
    ()=>User,
    (user)=>user.product
)
user: User
```

- Ahora en los productos, en TablePlus, hay una nueva columna que es userId. TypeOrm lo hizo por nosotros
- Lo normal es que cuando haga una consulta sobre el producto vaya a querer también el usuario que creó el producto
- Para que lo muestre en la consulta debo añadir eñl eager en true, para que cargue automáticamente esta relación

```
@ManyToOne(
    ()=>User,
    (user)=>user.product,
    {eager: true}
)
user: User
```

- Por ahora la columna de usuarios en producto solo tiene valores NULL porque en el SEED no había usuarios asignados a productos
- Esto es un error que debemos resolver.
- No debería permitir la creación de productos con el campo de usuario en NULL
- Borro toda la tabla de productos en TablePlus
- Ahora falta que al crear un producto, especifique que usuario lo creó a través de la autenticación

---

Insertar userId en los productos

- En el módulo de products **debo importar el AuthModule** para usar la autenticación con **@Auth** en el controller
- Solo los admin van a poder crear productos
- Uso el decorador **@GetUser** para extraer el usuario
- Se lo paso al servicio
- products.controller

```
@Post()
@Auth(ValidRoles.admin)
create(
  @Body() createProductDto: CreateProductDto,
  @GetUser() user: User
  ) {
        return this.productsService.create(createProductDto, user);
  }

  @Patch(':id')
  update(
    @Param('id', ParseUUIDPipe) id: string,
    @Body() updateProductDto: UpdateProductDto,
    @GetUser()user: User) {
    return this.productsService.update(id, updateProductDto, user);
  }
```

- Voy al servicio
- Actualizo create y update. Le paso el user a product, y antes de salvar en el update guardo el user en product.user

```
 async create(createProductDto: CreateProductDto, user:User) {
    try {

      const {images = [], ...productDetails} = createProductDto

      const product = this.productRepository.create({
        ...productDetails,
        images: images.map(image=> this.productImageRepository.create({url:
image})),
        user
      })

      await this.productRepository.save(product)

      return {...product, images}

    } catch (error) {

      this.handleDBExceptions(error)
    }

  }

   async update(id: string, updateProductDto: UpdateProductDto, user: User) {

    const {images, ...toUpdate} = updateProductDto


    const product = await this.productRepository.preload({id, ...toUpdate})
```

```
    if(!product) throw new NotFoundException(`Product with id : ${id} not found`)

    const queryRunner = this.dataSource.createQueryRunner()
    await queryRunner.connect()
    await queryRunner.startTransaction()

    try {

      if(images){
        await queryRunner.manager.delete(ProductImage, {product: {id}}) //con esto
borramos las imágenes anteriores

       product.images= images.map(image=> this.productImageRepository.create({url:
image}))

      }else{
          //product.images = await this.productImageRepository.findBy({product:
{id}}) puedo hacerlo así pero usaré findOnePlain
      }

      product.user= user
      await queryRunner.manager.save(product)

      await queryRunner.commitTransaction()  //commit
      await queryRunner.release() //desconexión

      return this.findOnePlane( id )

    } catch (error) {
      await queryRunner.rollbackTransaction()
      await queryRunner.release()

      this.handleDBExceptions(error)
    }
  }
```

- Tengo un error en el SEED porque llamo al productService.create y no le estoy pasando el user
- Muteo la linea de código donde está el error para poder compilar y crear un producto

```
//insertPromises.push(this.productsService.create(product))
```

NOTA: Si al crear el producto aparece un error que dice Cannot read properties of undefined(reading 'challenge') es porque en la composición del decorador @Auth, en su decorador @AuthGuard le falta pasarle 'jwt'

- Como tengo eager en true en la respuesta me carga directamente el usuario. Si no tendría que hacerlo manualmente
- Falta hacer funcional el SEED ya que le falta el usuario

SEED de usuarios, productos e imágenes

- Voy a crear un método para purgar las tablas de manera manual en el orden respectivo
    - Si intento borrar primero los usuarios, estos están siendo utilizados por los productos, la integridad referencial me va a molestar
- Borro todos los productos con el servicio de productos
- Para los usuarios debo inyectar el repositorio de usuarios
    - Uso el decorador **@InjectRepository**
    - Importo User y Repository
- Estoy exportando TypeORM y en TypeORM ya venía el usuario, por eso no da error
- Con el queryBuilder hago el delete, al no poner el nada en el where es todo, lo ejecuto
    - Recuerda que al tener el cascade en true va a borrar las imágenes también
- LLamo el método que he creado deleteTables en el runSEED
- Antes de insertar productos debo insertar usuarios
- Creo la interfaz en seed-data.ts
- Añado users al SeedData
- Añado los users a initialData

```
export interface SeedUser{
    email: string
    fullName: string
    password: string
    roles: string[]
}

export interface SeedData {
    users: SeedUser[]
    products: SeedProduct[]
}


export const initialData: SeedData = {

    users:[
        {
            email: 'test1@google.com',
            fullName: 'Test One',
            password: 'Abc123',
            roles: ['admin']
        },
        {
            email: 'test2@google.com',
            fullName: 'Test Two',
            password: 'Abc123',
            roles: ['user', 'super']
        }

    ],

    products: [ (etc...etc)
```

- Ahora puedo usar los usuarios para insertarlos masivamente desde el servicio
- Este firstUser que me retorna se lo paso a insertNewProducts (se lo paso al método para que no de error) y se lo paso al forEach, para que lo inserte en cada producto, por eso necesitaba retornar el user[0]
- seed.service

```
import { Injectable } from '@nestjs/common';
import { ProductsService } from 'src/products/products.service';
import { initialData } from './data/seed-data';
import { InjectRepository } from '@nestjs/typeorm';
import { User } from 'src/auth/entities/user.entity';
import { Repository } from 'typeorm';


@Injectable()
export class SeedService {

  constructor(
    private readonly productsService: ProductsService,

    @InjectRepository(User)
    private readonly userRepository: Repository<User>
  ){}

  private async insertUsers(){

    const seedUsers= initialData.users
    const users: User[] = []

    seedUsers.forEach(user=>{
      users.push(this.userRepository.create(user)) //esto no salva el usuario en
la db
    })

    const dbUsers = await this.userRepository.save(seedUsers)

    return dbUsers[0] //retorno el primer usuario para que le pueda mandar
insertUsers a insertProducts

  }

  async runSeed() {

    await this.deleteTables()
    const firstUser = await this.insertUsers()

    this.insertNewProducts(firstUser)

    const products = initialData.products


  }
```

```
  private async deleteTables(){

    await this.productsService.deleteAllProducts()

    const queryBuilder = this.userRepository.createQueryBuilder()

    await queryBuilder
    .delete()
    .where({})
    .execute()



  }

  private async insertNewProducts(user: User){
   await  this.productsService.deleteAllProducts()

   const products = initialData.products
   const insertPromises = []


   products.forEach(product=>{
     insertPromises.push(this.productsService.create(product, user))
   })

   await Promise.all(insertPromises)

   return `SEED EXECUTED`;
  }
}
```

- Creo los usuarios, regreso un usuario, ese usuario es el que utilizo para insertar los productos
- El password no está encriptado, por lo que necesita encriptación si quiero que haga match!!
- Para ello no hay más que usar bcrypt en la data en seed-data.ts

```
import * as bcrypt from 'bcrypt'

export const initialData: SeedData = {

    users:[
        {
            email: 'test1@google.com',
            fullName: 'Test One',
            password: bcrypt.hashSync('Abc123',10),
            roles: ['admin']
        },
        {
            email: 'test2@google.com',
```

```
            fullName: 'Test Two',
            password: bcrypt.hashSync('Abc123',10) ,
            roles: ['user', 'super']
        }

    ],

    (etc...etc)
```

Check AuthStatus

- Falta poder revalidar el token. No es revalidar exactamente
- Es usar el token suministrado y generar un nuevo token basado en el anterior
- Si no hago esto, si el usuario refresca el navegador no va a estar autenticado
- Creo un nuevo endpoint en auth.controller (con su respectivo servicio)
- Un Get que llamaré checkAuthStatus

```
@Get('check-auth')
@Auth()
checkAuthStatus(
  @GetUser() user: User,
){
  return this.authService.checkAuthStatus(user)
}
```

- Esparzo el user, genero un nuevo JWT con el id que es el user.id
- auth.service

```
async checkAuthStatus(user: User){
  return {
    ...user,
    token: this.getJwt({id: user.id})
  }
}
```

- En la respuesta regreso un nuevo JWT y la info de name, fullName, email, etc por si le sirve al frontend
- El usuario tiene que estar activo

NEST DOCUMENTACION SWAGGER

> npm i --save @nestjs/swagger

- En el main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { Logger, ValidationPipe } from '@nestjs/common';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';

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

  const logger = new Logger('bootstrap')

  app.setGlobalPrefix('api')

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

  //SWAGGER-------------------------------------------------
  const config = new DocumentBuilder()
    .setTitle('Teslo RESTFul API')
    .setDescription('Teslo Shop')
    .setVersion('1.0')
    .build()

  const document= SwaggerModule.createDocument(app, config)
  SwaggerModule.setup('api', app, document) //en el endpoint api va a crear la
documentación
 //SWAGGER-------------------------------------------------------

  await app.listen(process.env.PORT);
  logger.log(`App running on port ${process.env.PORT}`)
}
bootstrap();
```

> localhost:3000/api

- Evidentemente faltan configuraciones, no hay data de ejemplo, ni referencias...

Tags, ApiProperty, ApiResponse

- Quiero agrupar los endpoints de productos, auth, seed, etc
- Importo **@ApiTags** a nivel de controlador
- En products.controller

```
import { ApiTags } from '@nestjs/swagger';

@ApiTags('Products')
```

```
@Controller('products')
export class ProductsController
```

- Hago lo mismo en el resto de controladores (seed, auth, files)
- En el POST me gustaría saber **qué tipo de data** está esperando, **que es obligatorio** o no y **qué tipo de respuestas** hay
- Para ello uso **@ApiResponse** de swagger
- products.controller

```
@Post()
  @Auth(ValidRoles.admin)
  @ApiResponse({status: 201, description: 'Product was created'})
  @ApiResponse({status: 400, description: 'Bad request'})
  @ApiResponse({status: 403, description: 'Forbidden. Token related'})
  create(
    @Body() createProductDto: CreateProductDto,
    @GetUser() user: User,
    ) {
          return this.productsService.create(createProductDto, user);
    }
```

- Pongamos que quiero saber cómo va a lucir la respuesta
- Para ello puedo usar el type
- Si todo sale bien estaría regresando un producto. Lo coloco en type

```
@Post()
@Auth(ValidRoles.admin)
@ApiResponse({status: 201, description: 'Product was created', type: Product})
@ApiResponse({status: 400, description: 'Bad request'})
@ApiResponse({status: 403, description: 'Forbidden. Token related'})
create(
@Body() createProductDto: CreateProductDto,
@GetUser() user: User,
) {
        return this.productsService.create(createProductDto, user);
}
```

- Debo ir a la entity para añadir **@ApiProperty** a cada propiedad
- El usuario no lo decoro porque daría error porque no hay establecida la relación directamente

```
import {Entity, PrimaryGeneratedColumn, Column, BeforeInsert, BeforeUpdate,
OneToMany, ManyToOne} from 'typeorm'
import { ProductImage } from './product-image.entity'
import { User } from 'src/auth/entities/user.entity'
import { ApiProperty } from '@nestjs/swagger'
```

```
@Entity({name: 'products'})
export class Product {

    @ApiProperty()
    @PrimaryGeneratedColumn('uuid')
    id: string

    @ApiProperty()
    @Column('text', {
        unique: true
    })
    title: string

    @ApiProperty()
    @Column('float',{
        default: 0
    })
    price: number

    @ApiProperty()
    @Column({
        type: 'text',
        nullable: true
    })
    description: string

    @ApiProperty()
    @Column({
        type: 'text',
        unique: true
    })
    slug: string

    @ApiProperty()
    @Column({
        type: 'int',
        default: 0
    })
    stock: number

    @ApiProperty()
    @Column({
        type: 'text',
        array: true
    })
    sizes: string[]

    @ApiProperty()
    @Column({
        type: 'text',
    })
    gender: string

    @ApiProperty()
```

```
    @Column({
        type: 'text',
        array: true,
        default: []
    })
    tags: string[]


    @ApiProperty()
    @OneToMany(
        ()=> ProductImage,
        productImage=> productImage.product,
        {cascade:true, eager: true}
    )
    images?: ProductImage[]

    @ManyToOne(
        ()=>User,
        (user)=>user.product,
        {eager: true}
    )
    user: User

    @BeforeInsert()
    checkSlugInsert(){
        if(!this.slug){
            this.slug = this.title
        }

        this.slug = this.slug
        .toLowerCase()
        .replaceAll(' ', '_')
        .replaceAll("'", "")
    }

    @BeforeUpdate()
    checkSlugUpdate(){
        this.slug = this.slug
        .toLowerCase()
        .replaceAll(' ', '_')
        .replaceAll("'", "")
    }

}
```

- Me gustaría proporcionar más info, por ejemplo que el id no solo es un string, es un UUID, y es único
- Se hace en un objeto dentro de **@ApiProperty**

Expandir el ApiProperty

- En el id puedo añadir un id de ejemplo, una descripción y marcarlo como unique

```
@ApiProperty({
    example: '8da88a62-cd23-4662-a6ab-5a6c85e97bf6',
    description: 'Product ID',
    uniqueItems: true
})
@PrimaryGeneratedColumn('uuid')
id: string
```

- Si voy al Schema, en la documentación, ahora voy a tener más info
- Puedo hacer algo parecido con el título

```
@ApiProperty({
    example: "T-Shirt Teslo",
    description: "Product Title",
    uniqueItems: true
})
@Column('text', {
    unique: true
})
title: string
```

- Y así con el resto de propiedades...

```
import {Entity, PrimaryGeneratedColumn, Column, BeforeInsert, BeforeUpdate,
OneToMany, ManyToOne} from 'typeorm'
import { ProductImage } from './product-image.entity'
import { User } from 'src/auth/entities/user.entity'
import { ApiProperty } from '@nestjs/swagger'

@Entity({name: 'products'})
export class Product {

    @ApiProperty({
        example: '8da88a62-cd23-4662-a6ab-5a6c85e97bf6',
        description: 'Product ID',
        uniqueItems: true
    })
    @PrimaryGeneratedColumn('uuid')
    id: string

    @ApiProperty({
        example: "T-Shirt Teslo",
        description: "Product Title",
        uniqueItems: true
    })
    @Column('text', {
        unique: true
    })
```

```
    title: string

    @ApiProperty({
        example: 0,
        description: 'Product Price',
    })
    @Column('float',{
        default: 0
    })
    price: number

    @ApiProperty({
        example: "This is a very weird t-shirt with weird colors",
        description: 'Product description',
        default: null
    })
    @Column({
        type: 'text',
        nullable: true
    })
    description: string

    @ApiProperty({
        example: 't_shirt_teslo',
        description: 'slug for SEO',
        uniqueItems: true
    })
    @Column({
        type: 'text',
        unique: true
    })
    slug: string

    @ApiProperty({
        example: '10',
        description:'Product Stock',
        default: 0
    })
    @Column({
        type: 'int',
        default: 0
    })
    stock: number

    @ApiProperty({
        example: ['M', 'S', 'L', 'XL'],
        description: 'Product Size',
    })
    @Column({
        type: 'text',
        array: true
    })
    sizes: string[]
```

```
    @ApiProperty({
        example: 'women',
        description: 'Product gender'
    })
    @Column({
        type: 'text',
    })
    gender: string

    @ApiProperty()
    @Column({
        type: 'text',
        array: true,
        default: []
    })
    tags: string[]


    @ApiProperty()
    @OneToMany(
        ()=> ProductImage,
        productImage=> productImage.product,
        {cascade:true, eager: true}
    )
    images?: ProductImage[]

    @ManyToOne(
        ()=>User,
        (user)=>user.product,
        {eager: true}
    )
    user: User

    @BeforeInsert()
    checkSlugInsert(){
        if(!this.slug){
            this.slug = this.title
        }

        this.slug = this.slug
        .toLowerCase()
        .replaceAll(' ', '_')
        .replaceAll("'", "")
    }

    @BeforeUpdate()
    checkSlugUpdate(){
        this.slug = this.slug
        .toLowerCase()
        .replaceAll(' ', '_')
        .replaceAll("'", "")
    }

}
```

- Me falta documentar los DTOS. Y cómo documento el update-product.dto si es una expansión de otro DTO?

Documentar DTOS

- Documentar dtos es fundamental, ya que si no el endpoint responderá un error
- Es sencillo. Si los dtos no fueran clases perderiamos la oportunidad de decorar las propiedades
- pagination.dto

```
import { ApiProperty } from "@nestjs/swagger"
import { Type } from "class-transformer"
import { IsOptional, IsPositive, Min } from "class-validator"

export class PaginationDto{

    @ApiProperty({
        default: 10,
        description: 'How many rows do you need?'
    })
    @IsOptional()
    @IsPositive()
    @Type(()=> Number)
    limit?: number

    @ApiProperty({
        default: 0,
        description:'How many rows do you want to skip?'
    })
    @IsOptional()
    @Min(0)
    @Type(()=> Number)
    offset?: number
}
```

- Ahora en la documentación puedo ver en el GET de /api/products el limit y el offset
- En el create-product.dto

```
import { ApiProperty } from "@nestjs/swagger"
import { IsString, MinLength, IsNumber, IsOptional, IsInt, IsPositive, IsArray,
IsIn } from "class-validator"

export class CreateProductDto {

    @ApiProperty({
        example: 'Blue Trousers',
        description: 'Product Title',
        nullable: false,
```

```
        minLength: 1
    })
    @IsString()
    @MinLength(1)
    title: string

    @ApiProperty()
    @IsNumber()
    @IsOptional()
    price?: number

    @ApiProperty()
    @IsString()
    @IsOptional()
    description?: string

    @ApiProperty()
    @IsString()
    @IsOptional()
    slug?: string

    @ApiProperty()
    @IsInt()
    @IsPositive()
    @IsOptional()
    stock?: number

    @ApiProperty()
    @IsString({each: true})
    @IsArray()
    sizes: string[]

    @ApiProperty()
    @IsIn(['men', 'women', 'kid','unisex'])
    gender: string

    @ApiProperty()
    @IsString({each:true})
    @IsArray()
    @IsOptional()
    tags?: string[]

    @ApiProperty()
    @IsString({each:true})
    @IsArray()
    @IsOptional()
    images?: string[]
}
```

- Pero no tengo el update-product.dto
- En lugar de importar PartialType de @nestjs/mapped-types lo importo de @nestjs/swagger

```ts
//import { PartialType } from '@nestjs/mapped-types';
import { PartialType } from '@nestjs/swagger';
import { CreateProductDto } from './create-product.dto';

export class UpdateProductDto extends PartialType(CreateProductDto) {}
```