04 NEST Duro de Roer - MySQL

- Creando un proyecto

```
nest new mysql-project
```

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

```
npm i @nestjs/swagger
```

```
npm i @nestjs/typeorm typeorm mysql2
```

- Cojo el boilerplate de swagger de la página de nest que es este
- Lo configuro como yo quiero
- Le añado el uso de globalPipes

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

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

  app.useGlobalPipes(new ValidationPipe({transform: true}))

  const config = new DocumentBuilder()
    .setTitle('MySQL')
    .setDescription('UsandoMySQL')
    .setVersion('1.0')
    .addTag('mysql')
    .build();
  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('api', app, document);

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

- Quito el AppService, el AppController del AppModule (no los quiero para nada)
- Configuro TypeOrm en app.module uso
- El synchronize en true para que esté escuchando y aplique los cambios

NOTA: ormconfig está deprecado. TypeORM ya no lo usa

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

```
@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'shop',
      entities: [],
      synchronize: true,
    }),
  ],
})
export class AppModule {}
```

- Creo la db en MySQL Workbench con el mismo nombre shop, el charset en utf8 con utf8_general_mysql

---

Creando el módulo de Product

- Creo un CRUD completo para Product

> nest g res product

- El CreateProductDto
  - Vamos a hacer un borrado lógico usando deleted con un boolean
  - Lo omito en el dto porque tendrá el valor false por defecto desde la entity
  - Omito también el id, usaré un query string y el body para actualizar

```
import { IsNumber, IsString, IsPositive, IsNotEmpty, IsBoolean, IsOptional } from
"class-validator"

export class CreateProductDto {

    @IsOptional()
    @IsNumber()
    @IsPositive()
    id: number

    @IsString()
    @IsNotEmpty()
    name: string

    @IsNotEmpty()
    @IsNumber()
    @IsPositive()
    stock: number

    @IsNotEmpty()
```

```
    @IsNumber()
    @IsPositive()
    price: number

    @IsOptional()
    @IsBoolean()
    deleted: boolean
}
```

- La entity de Product
- Le tengo que decir que es una entidad con el decorador **@Entity** de typeorm
- PrimaryGeneratedColumn lleva el autoincremental por defecto
- Normalmente el borrado se suele representar con un 0 (false) y un 1 (true) pero vamos a usar un boolean

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

@Entity()
export class Product {

    @PrimaryGeneratedColumn()
    id: number

    @Column({type: String, nullable: false, length: 30})
    name: string

    @Column({type: Number, nullable: false, default: 0 })
    stock: number

    @Column({type: Number, nullable: false})
    price: number

    @Column({type: Boolean, nullable: false, default: false})
    deleted: boolean
}
```

- Tengo que **añadir la entity de Product al array de entities** de TypeOrmModule (en app.module)
- Para inyectar el repo de Producto en el servicio uso **@InjectRepository** de @nestjs/typeorm
    - Será del tipo **Repository** (lo importo de typeorm) de tipo Product

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Product } from './entities/product.entity';
import { Repository } from 'typeorm';

@Injectable()
export class ProductService {
```

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

- Para usar el repo debo **importar en ProductModule dentro de imports el TypeOrmModule, usar el forFeature y pasarle la entity**

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

@Module({
  imports: [TypeOrmModule.forFeature([
    Product
  ])],
  controllers: [ProductController],
  providers: [ProductService]
})
export class ProductModule {}
```

Creando un producto

- Creo el endpoint en el controller

```
@Controller('api/v1/product')
export class ProductController {
  constructor(private readonly productService: ProductService) {}

  @Post()
  create(@Body() createProductDto: CreateProductDto) {
    return this.productService.create(createProductDto);
  }
}
```

- En el servicio, con .save si la entidad no existe la inserta. Si existe, la actualiza
- Uso async await ya que interactúo con la DB

```
@Injectable()
export class ProductService {

  constructor(
    @InjectRepository(Product)
```

```
    private  productRepository :Repository<Product> ){
  }

    async create(createProductDto: CreateProductDto) {
    return await this.productRepository.save(createProductDto)
  }
 }
```

- Le paso en el body de ThunderClient o Postman en producto que encaje con el dto
- El endpoint es de tipo POST http://localhost:3000/api/v1/product

```
{
  "id": 1,
  "name": "Producto1",
  "stock":10,
  "price":300
}
```

- Al colocarle el id en el primer producto, en los siguientes ya **lo crea automáticamente de forma autoincremental**
- Si le coloco un id específico, el siguiente producto tendrá el siguiente id correspondiente al anterior
- Por ejemplo: si le coloco el id 6, el siguiente será id 7
- Hay que **validar si el producto existe o no**

Devolviendo un producto por su id

- Para obtener un producto por id en el controller

```
@Get(':id')
findOne(@Param('id') id: string) {
  return this.productService.findOne(+id);
}
```

- En el service, puedo usar el findBy o el findOne expresando una condición con el where

```
async findOne(id: number) {
  return await  this.productRepository.findBy({id});
}

//o también (usaré esta segunda manera para poder hacer la validación)

async findProduct(id: number) {
  return await  this.productRepository.findOne({
    where: {id}
```

```
  });
}
```

Validación en createProduct

- Manejo de la excepción
- Creo una variable productExist. Uso await porque devuelve una promesa

```
  async create(createProductDto: CreateProductDto) {

    const productExists: CreateProductDto= await
this.findProduct(createProductDto.id)

    if(productExists){
      throw new ConflictException(`El producto con el id ${createProductDto.id} ya
existe`)
    }
    return await this.productRepository.save(createProductDto)
  }
```

- Es un conflicto mio. Se usa para este tipo de casos, en el update tiene que existir, etc.
- Para encontrar por id

Devolviendo los productos

- En el controller

```
@Get()
findAll() {
  return this.productService.findAll();
}
```

- En el service, le indico con un where que el deleted debe de ser false

```
findAll() {
    return this.productRepository.find({
      where: {deleted:false}
    });
  }
```

Actualizando Productos

- Con el PUT debo pasarle el id del producto por parámetro.
- Es muy parecido al create
- En el controller

```
@Put(':id')
update(@Param('id') id: string, @Body() updateProductDto: UpdateProductDto) {
  return this.productService.update(+id, updateProductDto);
}
```

- En el servicio:

```
async updateProduct(id: number, updateProductDto: UpdateProductDto) {
  return  await this.productRepository.save(updateProductDto)
}
```

- NOTA: este código plantea los siguientes errores
  - Sin el id por parámetro da error 404 y no crea un nuevo producto
  - Sin el precio en el body salta error interno del server, pues el precio no tiene un valor por defecto

---

Soft delete

- Le paso el id del producto y le pongo el delete a true
- En el controller

```
@Delete(':id')
removeProduct(@Param('id') id: string) {
  return this.productService.removeProduct(+id);
}
```

- En el servicio debo comprobar de que exista, y si ya está borrado
- Con rows obtengo cuantas filas se han actualizado, me devuelve UpdateResult (de TypeORM, válido con mysql)

```
  async removeProduct(id: number) {

    const product: CreateProductDto= await this.findProduct(id)

    if(!product){
      throw new ConflictException(`El producto con el id ${id} no existe`)
    }

    if(product.deleted){
      throw new ConflictException(`El producto con id ${id} ya está borrado`)
    }
```

```
    const rows: UpdateResult= await this.productRepository.update(
      {id}, //los productos que tengan este id ponles eel deleted en true
      {deleted: true}
    )

    return rows.affected == 1 //devolverá un true en caso de borrado exitoso

  }
```

Recuperando productos borrados

- Usaremos PATCH para modificar el valor de deleted
- controller:

```
async restoreProduct(id: number){
  const product: CreateProductDto= await this.findProduct(id)

  if(!product){
    throw new ConflictException(`El producto con el id ${id} no existe`)
  }

  if(!product.deleted){
    throw new ConflictException(`El producto con id ${id} no está borrado`)
  }

  const rows: UpdateResult= await this.productRepository.update(
    {id},
    {deleted: false}
  )

  return rows.affected == 1
}
```

Creando stock.dto

- stock.dto

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

export class StockDto{
    @IsNotEmpty()
    @IsPositive()
    @IsNumber()
    id: number

    @IsNotEmpty()
```

```
    @Min(0)
    @Max(1000)
    @IsNumber()
    stock: number
}
```

Actualizando el stock de un producto

- En el controller, cuando quiero actualizar una propiedad en concreto uso PATCH. Creo una nueva ruta

```
@Patch('/stock')
updateStock(@Body() s: StockDto){
  return this.productService.updateStock(stock)
}
```

- NOTA IMPORTANTE: debo colocar esta ruta ENCIMA del anterior PATCH con ":id" porque si no hace MATCH primero y da error. Para solucionarlo en lugar de poner directamente ":id" pongo "restore/:id"
- En el service

```
  async updateStock(s: StockDto){
    const product = await this.findProduct(s.id)

    if(!product){
      throw new BadRequestException(`El producto con id ${s.id} no existe`)
    }

    if(product.deleted){
      throw new ConflictException("El producto no está disponible")
    }

    const rows:UpdateResult = await this.productRepository.update(
      {id: s.id},
      {stock: s.stock}
    )

    return rows.affected == 1
  }
```

Incrementando el stock de un producto

- Vamos a sumar un stock al stock actual. En el controller

```
@Patch('/stock-increment')
incrementStock(@Body() s: StockDto){
```

```
    return this.productService.incrementStock(s)
  }
```

- En el servicio reutilizo código del updateStock y calculo la suma del stock con el stock disponible y el stock del dto
- Declaro unas propiedades de minimo stock y máximo stock

```
@Injectable()
export class ProductService {

  private MIN_STOCK: number = 0
  private MAX_STOCK: number = 1000

(...)
}
```

- Declro una variable stock = 0

```
  async incrementStock(s: StockDto){
    const product: UpdateProductDto = await this.findProduct(s.id)

    if(!product){
      throw new BadRequestException(`El producto con id ${s.id} no existe`)
    }

    if(product.deleted){
      throw new ConflictException("El producto no está disponible")
    }

    let stock = 0
    if(s.stock + product.stock > this.MAX_STOCK ){
      throw new ConflictException("El stock excede el máximo permitido")
    }else{
      stock = product.stock + s.stock
    }

    const rows: UpdateResult = await this.productRepository.update(
      {id: s.id},
      {stock}

    )

    return rows.affected == 1
  }
```

- Para decrementar hago lo mismo pero restando en lugar de sumar, mirando que no sea menor que 0

```
async decrementStock(s: StockDto){
  const product: UpdateProductDto = await this.findProduct(s.id)

  if(!product){
    throw new BadRequestException(`El producto con id ${s.id} no existe`)
  }

  if(product.deleted){
    throw new ConflictException("El producto no está disponible")
  }

  let stock = 0
  if(product.stock- s.stock  < this.MIN_STOCK ){
    product.stock = this.MIN_STOCK
  }else{
    stock = product.stock - s.stock
  }

  const rows: UpdateResult = await this.productRepository.update(
    {id: s.id},
    {stock}

  )

  return rows.affected == 1
}
```

Documentar los endpoints

- Con Swagger
- Recuerda la configuración del main

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

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

  app.useGlobalPipes(new ValidationPipe({transform: true}))

  const config = new DocumentBuilder()
    .setTitle('MySQL')
    .setDescription('UsandoMySQL')
    .setVersion('1.0')
    .addTag('mysql')
    .build();
  const document = SwaggerModule.createDocument(app, config);
```

```
  SwaggerModule.setup('api', app, document);

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

- Documentando createProduct

```
  @Post()
  @ApiOperation({
    description: "Crea un producto"

  })
  @ApiBody({
    description: "Crea un producto mediante CreateProductDto",
    type: CreateProductDto,
    examples:{
      ejemplo1: {
        value: {
          "id": 2,
          "name": "Producto2",
          "stock":10,
          "price":300
        }
      },
      ejemplo2:{
        value: {
          "name": "Producto2",
          "stock":10,
          "price":300
        }
      }
    }
  })
  create(@Body() createProductDto: CreateProductDto) {
    return this.productService.create(createProductDto);
  }
```

- Puedo usar el decorador @ApiResponse para documentar las respuestas

```
@Post()
@ApiOperation({
  description: "Crea un producto"

})
@ApiBody({
  description: "Crea un producto mediante CreateProductDto",
  type: CreateProductDto,
  examples:{
```

```
    ejemplo1: {
      value: {
        "id": 2,
        "name": "Producto2",
        "stock":10,
        "price":300
      }
    },
    ejemplo2:{
      value: {
        "name": "Producto2",
        "stock":10,
        "price":300
      }
    }
  }
})
@ApiResponse({
  status: 201,
  description: "Producto creado correctamente"
})
@ApiResponse({
  status:409,
  description: "El producto existe"
})
create(@Body() createProductDto: CreateProductDto) {
  return this.productService.create(createProductDto);
}
```

- Para documentar DTO

```
export class CreateProductDto {

    @ApiProperty({
        name: "id",
        required: false,
        description: "id del producto",
        type: Number
    })
    @IsOptional()
    @IsNumber()
    @IsPositive()
    id?: number
```