NEST 11

> nest new TesloShop

- Selecciono npm
- Voy a trabajar a través de TypeORM. Es muy parecido a mongoose, pero aqui voy a tener la forma de mapear las entidades, para poder tener las relaciones entre otras tablas, establecer triggers, procedimientos en cuanto a llaves, etc.
- También tiene decoradores y se integra muy bien con NEST
- Hago limpieza, elimino todo el /src menos el main y el app.module.ts

Docker Instalar y correr Postgres

- Podría instalar postgres en mi máquina, pero conviene usar docker para mantener esa imagen (y si vieniera otro desarrollador todo es más fácil)
- Debe de estar Docker corriendo
- Creo el archivo en la raíz llamado docker-compose.yaml
- Es recomendado trabajar con versiones de DB especificas porque de haber muchos cambios se minimizan errores o la obligación de migrar
- Le indico el puerto de postgres al puerto fisico de mi pc
- Aunque no tengo legible el archivo .env, docker es capaz de leerlo
- Para que la data sea persistente debo guardarla en un volumen, le indico la ruta por defecto

```
version: '3'


services:
  db:
    image: postgres:14.3
    restart: always
    ports:
      - "5432:5432"
    environment:
      POSTGRES_PASSWORD: ${DB_PASSWORD}
      POSTGRESS_DB: ${DB_NAME}
    container_name: teslodb
    volumes:
      - ./postgres:/var/lib/postgresql/data
```

- Levantemos la db

> docker-compose up -d

- -d descarga la imagen si no esta en Docker

- Voy a TablePlus y elijo postgreSQL. Pongo el nombre "Teslo", el puerto "54321", el usuario "postgres" y el password "1234"

Conectar Postgres con NEST

- NEST tiene conectores propios, hay documentación al respecto
- Primero configuraré las variables de entorno
- Instalo el @nestjs/config

```
npm i @nestjs/config
```

- En el app.module, en las importaciones añado el forRoot()

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

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

- Con esto están mis variables de entorno establecidas
- Voy a usar TypeORM. Instalo los decoradores, el paquete en si y el driver de postgres

```
npm i --save @nestjs/typeorm typeorm pg
```

- En el app.module, en imports configuro la conexión
- En host coloco la variable de entorno DB_HOST=localhost, DB_PORT=5432
    - Como el port tiene que ser un numero, fácilmente lo puedo convertir con u +
- El nombre de la base de datos, y el usuario por defecto que es postgres
- El synchronize, si hay algun cambio las entidades automaticamente las sincroniza. No se usa en producción

```
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 {}
```

TypeORM Entity-Product

- El primer endpoint será para manejar los productos
- Productos tendrá descripción, imágenes , tipo, stock, precio, tamaño, tags, slug, titulo, genero
- Creo el resource con el CLI

> nest g res products --no-spec

- Ya tengo la carpeta con los dtos, la entity (que es lo que va a buscar TypeOrm para crearse mi referencia en la DB)
- Comienzo por definir la entity
- Debo usar el decorador

```
import { Entity } from "typeorm";

@Entity()
export class Product {

}
```

- Coloco mi PrimaryGeneratedColumn
- Quiero trabajar con uuid para el id. TypeOrm ofrece diferentes maneras de como manejarlo, ya incorpora uuid
- Como voy a usar otra columna para el titulo del producto uso @Column de tipo 'text'
    - Puedo añadirle una regla, como que tiene que ser único

```
import { ParseUUIDPipe } from "@nestjs/common";
import { Column, Entity, PrimaryGeneratedColumn } from "typeorm";

@Entity()
export class Product {

    @PrimaryGeneratedColumn('uuid')
    id: string
```

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


}
```

- En algún lugar del módulo tiene que estar definida la entity para que el TypeOrm le pueda decir a la base de datos que hay una nueva entidad
- Además definimos el autoLoadEntities en true y el synchronize, con lo que le estoy diciendo que lo sincronice
- Voy a products.module y en el imports defino todas las entidades. No es un forRoot porque solo hay uno, es un forFeature
- Entre corchetes importo la entidad

```
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 {}
```

- Ahora si voy a TablePlus, veo que definió la tabla de productos y una serie de funciones para manejar uuid
- Como estoy en synchronize, si cambio el nombre a titulo2 se refleja en la tabla

Entidad sin relaciones

- Defino otra columna con @Column() para el precio
- No es de tipo number, porque postgres no soporta este tipado. En la ayuda aparecen todos los tipos de mongo, sql,etc
    - En el caso de postgres es numeric. le puedo establecer un valor por defecto
- Muestro dos formas de declarar el tipo de dato

```
import { ParseUUIDPipe } from "@nestjs/common";
import { Column, Entity, PrimaryGeneratedColumn } from "typeorm";
```

```
@Entity()
export class Product {

    @PrimaryGeneratedColumn('uuid')
    id: string

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

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

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

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

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

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

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

- Si voy a TablePlus y recargo (ctrl+R) puedo ver que ya tengo los campos
- En este punto ya está preparado para hacer una inserción

Create Product dto

- Quiero añadir /api a los endpoints, uso en el main el setGlobalPrefix

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

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

  app.setGlobalPrefix('api')

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

- Ahora este endpoint debería devolverme una respuesta en una petición Get

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

- Voy al Dto. recuerda que para usar el class-validator y todo lo demás tengo que instalarlo

> npm i class-validator class-transformer

- Hay que usar el useGlobalPipes en el main para las validaciones

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

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

  app.setGlobalPrefix('api')

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

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

- Voy a definir el Dto, que es como quiero que luzca la data que se va a comunicar con mi controlador
- No espero el id pues ya lo genera la base de datos
- El title, size y gender son obligatorios

```
export class CreateProductDto {
    title: string;
    price?: number;
```

```
    description?: string;
    slug?: string;
    stock?: string;
    sizes: string[];
    gender: string
}
```

- Ahora voy con los decoradores
- each: true para que cada elemento sea una string
- IsIn para que el elemento sea uno de los declarados

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

export class CreateProductDto {

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

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

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

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

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

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

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

Insertar usando TypeORM

- El controlador se queda practicamente igual a como lo configura NEST

- Voy al servicio para trabajar con el create de la petición Post para hacer una inserción
- Yo podría crear una nueva instancia de producto = new Product() para empezar a trabajar con la entidad, pero se aconseja hacerlo mediante el patrón repositorio
- En NEST con el ORM ya lo tengo creado por defecto
- Creo el constructor, le pongo el decorador de inyectable al repositorio, importo Repository de ORM y lo pongo de tipo producto

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

@Injectable()
export class ProductsService {

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

  async create(createProductDto: CreateProductDto) {

  }

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

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

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

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

- Este repositorio no solo me sirve para insertar, también para hacer querybuilders, transacciones, rollbacks y otras cosas
- Voy al create y meto la lógica en un try y un catch por si algo sale mal
- Creo el producto, para guardarlo uso el await

```
  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!')
    }
  }
```

- Le envío una petición Post al enpoint api/products con este body

```
{
  "title": "Migue Shirt",
  "sizes": ["M", "L"],
  "gender": "men"
}
```

- Y me da error:

> null value in column "slug" of relation "product" violates not-null constraint

- Proporciono el slug en el json
- El price aparece como string en la base de datos. El price no es numeric, es un float (corrección en la entity)
- Si lo vuelvo a enviar da error por duplicidad, es un error que voy a tener que manejar, igual que el del slug que si no viene habrá que generarlo a través del título

Manejo de errores

- Hay varias condiciones a evaluar contra la DB: que el titulo sea único, que el slug sea único...muchas evaluaciones
- Para manejar los errores, NEST ofrece los LOGS para ver que es lo que sucedió mal
- Me creo una propiedad de la clase con una instancia del Logger(lo importo)
  - Le añado la clase donde voya usar este Logger
- Estoy creando una propiedad dentro de la clase.
- Uso el this.logger.error en el catch

```
import { Injectable, InternalServerErrorException, Logger } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
```

```
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';
import { Product } from './entities/product.entity';

@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 si vuelvo a enviar el mismo body aparece el error de key duplicated en el LOG, luce de otra manera
- Si hago un console.log del error hay mucha información. Entre otras cosas el code (código) del error, detail con los detalles
- Puedo usar el code para hacer una evaluación

```
@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) {
      if(error.code === "23505")
        throw new BadRequestException(error.detail)

      this.logger.error(error)
      throw new InternalServerErrorException('Unexpected error. Check server
logs')
    }
  }
```

- En lugar de tener el código así puedo crearme un método privado llamado handleDBExceptions

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

@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)
    }
  }

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

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

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

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

  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')
  }
}
```

- De esta manera puedo ir agregando todos los errores en el método y queda más limpio
- Ahora hay que manejar el error del slug que si no lo mando lanza el error

BeforeInsert y BeforeUpdate

- Puedo generar el slug a través del título
- El procedimiento puedo hacerlo e insertarlo antes de guardar en la DB, en la entity
- En la entity añado @BeforeInsert

```
  @BeforeInsert()
  checkSlugInsert(){
      if ( !this.slug){
          this.slug= this.title
      }
      this.slug= this.slug
          .toLowerCase()
          .replace(' ','_')
          .replace("'", '')
  }
```

Get y Delete TypeORM

- Añado la lógica en los métodos del servicio
- En el controlador parseo el UUID del id
- Falta el update(!)

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

@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: string, updateProductDto: UpdateProductDto) {
    return `This action updates a #${id} product`;
  }

  async remove(id: string) {
    const product= await this.productRepository.delete(id)
    if(!product) throw new NotFoundException('The product does not exists')
```

```
    return("Product deleted")
  }

  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 nuevo Dto para la paginación. No está relacionado con productos, podría necesitarlo en otro lugar
- Para ello crearé el módulo common

> nest g mo common

- Creo una nueva carpeta en el common llamada Dto

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

export class PaginationDto{

    @IsOptional()
    @IsPositive()
    limit?: number;

    @IsOptional()
    @IsPositive()
    offset?: number

}
```

- En el findAll del controlador, voy a recibir los query parameters mediante el @Query
- Le pongo un console.log al dto

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

- Si coloco un endpoint como este:

> http://localhost:3000/api/products?limit=12

- Me dice que limit must be a positive number. Está llegando como string
- Para hacer la transfromación puedo usar un decorador Type en el dto

```
import { IsNumber, IsOptional, IsPositive } from "class-validator";
import {Type} from 'class-transformer'

export class PaginationDto{

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

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

- Esto es lo mismo que el enableImplicitConversions: true
- Ahora tengo el limit y el offset como numeros
- Añado el paginationDto al findAll (también en el controlador)
- Desestructuro el limit y el offset. Como son opcionales les asigno un valor por defecto
- Aplico el filtro: El take toma la cantidad, y el skip salta

```
  async findAll(paginationDto: PaginationDto) {

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

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

      //TODO: relaciones
    })

  }
```

- En el controller:

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

Buscar por Slug o UUID

- Quito el ParseUUIDPipe del findOne del controller
- Cambio el termino id por term

```
  @Get(':term')
  findOne(@Param('term') term: string) {
    return this.productsService.findOne(term);
  }
```

- En el service también, pero como voy a implementar la búsqueda por titulo y slug también, empiezo con la lógica
- Declaro la variable product con let
- Instalo uuid

> npm i uuid @types/uuid

- Hay una función que es validate de uuid, la renombro a isUUID

```
import { BadRequestException, Injectable, InternalServerErrorException, Logger,
NotFoundException } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { PaginationDto } from 'src/common/dto/pagination.dto';
import { Repository } from 'typeorm';
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';
import { Product } from './entities/product.entity';
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)
      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)){
      await this.productRepository.findOneBy({id: term})

    }else{
      await this.productRepository.findOneBy({slug:term})
    }


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

    return product
  }

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

  async remove(id: string) {
    const product= await this.productRepository.delete(id)
    if(!product) throw new NotFoundException('The product does not exists')

    return("Product deleted")
  }

  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')
  }
}
```

QueryBuilder

```
async findOne(term: string) {

    let product: Product;

    if( isUUID(term)){
      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 problema es que el título no lo estamos parseando a todo minúsculas (o mayúsculas) y eso puede llevar a error de case sensitive
- Paso el título (la definición de la columna) a mayúsculas con UPPER. Esto me escapa los espacios y apóstrofes también
- El slug siempre lo he trabajado en minúsculas

```
  async findOne(term: string) {

    let product: Product;

    if( isUUID(term)){
      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 en la actualización
- Cuando solo hay una tabla involucrada es sencillo actualizar
- update-product.dto

```
import { PartialType } from '@nestjs/mapped-types';
import { CreateProductDto } from './create-product.dto';

export class UpdateProductDto extends PartialType(CreateProductDto) {}
```

- PartialType está expandiendo la config de create-product.dto y las hace opcionales
- Voy a actualizar basado en el id
- Le paso el pipe de UUID
- products.controller:

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

- Con el preload le estoy diciendo: búscate un producto por el id y carga todas las propiedades que estén en el dto
- Esto no lo actualiza, solo lo prepara para la actualización

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

- Tengo que guardarlo

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

```
    if(!product) throw new NotFoundException('This product does not exists')

    await this.productRepository.save(product)

    return product

  }
```

- Si le cambio el titulo y el titulo ya existe voy a tener un error 500
- Puedo manejar el error cómo lo hice anteriormente con el handleDBExceptions
- Meto el save e un try catch
- En el error uso el método que creé

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

    if(!product) throw new NotFoundException('This product does not exists')

    try {
      await this.productRepository.save(product)
      return product

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

    }
  }
```

- Los slugs tengo que validarlos también

BeforeUpdate

- Si recibo el slug debe de estar en minúscula, que tenga el guión bajo en lugar del espacio
- Para ello voy al entity y añado al final

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

    }
```

Nueva columna -Tags

- Añado la columna de tags en la entity.
- Le especifico de tipo array y le mando un arreglo vacío de valor por defecto

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

- Pero hay que configurar en el dto que pueda recibir los tags en el body

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