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[]
```