VALIDACIONES EN NEST.

El tipado en las variables de ambiente nos ayuda a solucionar errores faciles en tiempos de desarrollo.

En el momento de despliegue cuando un servidor externo nos envia variables de ambiente, tenemos que validar los tipos de datos que recibimos en nuestras variables de entorno.

Instalamos npm install joi

app.module.ts importamos joi

import \* as Joi from 'joi';

...

@Module({

imports: [

ConfigModule.forRoot({

envFilePath: environments[process.env.NODE\_ENV] || '.env',

load: [config],

isGlobal: true,

validationSchema: Joi.object({

API\_KEY: Joi.number().required(),

DATABASE\_NAME: Joi.string().required(),

DATABASE\_PORT: Joi.number().required(),

}),

}),

...

})

Corres la aplicacion npm run start:dev

El output de errores en consola daria lo siguiente

Si la variable no se setea:

throw new Error(`Config validation error: ${error.message}`);

^

Error: Config validation error: "DATABASE\_PORT" is required

Si la variable se seteo con otro tipo de dato:

throw new Error(`Config validation error: ${error.message}`);

^

Error: Config validation error: "DATABASE\_PORT" must be a number

Me tocó hacer lo mismo pero con class-validator, dejo mi aporte por si a alguien le sirve:

**.env.validation.ts**

import { IsEnum, IsNumber, IsString, validateSync } from 'class-validator';

import { plainToClass } from 'class-transformer';

import { Environment } from '../src/common/models/environment.model';

class EnvironmentVariables {

// Environment

@IsString()

@IsEnum(Environment)

ENV: Environment;

// Server config

@IsNumber()

PORT: number;

@IsString()

END\_POINT: string;

}

export function validate(config: Record<string, unknown>) {

const validatedConfig = plainToClass(EnvironmentVariables, config, {

enableImplicitConversion: true,

});

const errors = validateSync(validatedConfig, {

skipMissingProperties: false,

});

if (errors.length > 0) {

throw new Error(`ENV validation error` + errors.toString());

}

return validatedConfig;

}

}

**configuration.ts:**

import { registerAs } from '@nestjs/config';

export default registerAs('config', () => {

return {

environment: process.env.ENV,

server: {

port: parseInt(process.env.PORT, 10) || 3000,

endpoint: process.env.END\_POINT,

},

};

});

**app.module.ts**

import { MiddlewareConsumer, Module, NestModule } from '@nestjs/common';

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

import config from '../config/configuration';

import { validate } from 'config/env.validation';

import { LoggerMiddleware } from './common/middleware/appLogger.middleware';

@Module({

imports: [

ConfigModule.forRoot({

load: [config],

validate,

isGlobal: true,

}),

],

})

export class AppModule implements NestModule {

configure(consumer: MiddlewareConsumer): void {

consumer.apply(LoggerMiddleware).forRoutes('\*');

}

}

MODULO DE NEST PERSISTENCIA DE DATOS..

USANDO VARIABLES DE AMBIENTE./..

Agregue las validaciones con joi de la siguiente forma.

Primero arme en un archivo configSchema.ts lo siguiente:

import \* as Joi from 'joi';

const configSchema = Joi.object({

API\_KEY: Joi.number().required(),

DATABASE\_NAME: Joi.string().required(),

DATABASE\_PORT: Joi.number().required(),

POSTGRES\_DB: Joi.string().required(),

POSTGRES\_USER: Joi.string().required(),

POSTGRES\_PASSWORD: Joi.string().required(),

POSTGRES\_PORT: Joi.number().required(),

POSTGRES\_HOST: Joi.string().hostname().required(),

});

export default configSchema;

En el archivo app.module.ts elimine la importacion de joi (ya que lo hago en el configSchema) e importa el configSchema.ts

import configSchema from './configSchema';

@Module({

imports: [

ConfigModule.forRoot({

envFilePath: enviroments[process.env.NODE\_ENV] || '.env',

load: [config],

isGlobal: true,

validationSchema: configSchema,

}),

...

Fernando Cordero

[hace 10 meses](https://platzi.com/comentario/2581439/)

Buen aporte. Por mi parte, yo lo que hice fue crear una nueva carpeta agrupando el config y el configSchema, para ser mas ordenado.

Eduardo Pech

[hace un año](https://platzi.com/comentario/2540790/)

Importante si no les reconoce el port desde variable de entorno porque viene un string.

port: parseInt(process.env.POSTGRES\_PORT, 10),

Fernando Cordero

[hace 10 meses](https://platzi.com/comentario/2636447/)

Chicos hasta los momentos se sigue pasando el puerto del servidor en duro en el main.ts del proyecto.  
Si lo desean arreglar el codigo quedaria asi:

// src/main.ts

async function bootstrap() {

const app = await NestFactory.create(AppModule);

const configService = app.get(ConfigService);

const port = configService.get('API\_PORT');

await app.listen(port);

console.log(`Application is running on: ${await app.getUrl()}`);

}

bootstrap();

TRABAJANDO CON TYPEORM..

* Que hace? Capa extra que le ingresamos a nuestra app
* Se encarga de la conexion y todo en base a modelos y entidades
* En esas entidades definidos propiedades y metodos.
* No vamos a realizar codigo sql para ejecutar las operaciones. Pero el ORM nos permite realizar operaciones y consultas complejas.
* Tambien nos abstrae la conexion, asi que nos podemos conectar a diferentes motores.
* Hay dos famosos ORM que se utilizan y son:
* Sequelize
* TypeORM

**TYPEORM ACTIVE RECORD VS REPOSITORIES**

Por si alguien los necesita para sus notas, los dos trozos de código utilizados para explicar la diferencia entre active records y repositories:

Active records:

const product = new Product();

product.name = "Product 1";

await product.save();

await product.remove();

...

await Product.findOne(1);

Repositories:

const productRepo = connection.getRepository(Product);

const product = new Product();

product.name = "Product 1";

await productRepo.save(product);

await productRepo.remove(product);

await productRepo.findOne(1);

**cuidado con synchronize: true, // 👈 new attr  
en ambientes productivos eso en producion siempre debe estar en false**

**CONFIGURANDO DOS MOTORES DE BASES DE DATOS EN SIMULTANEO**

Les dejo como configure los 2 motores en simultaneo:

Config:

import { registerAs } from '@nestjs/config';

export default registerAs('config', () => {

return {

postgres: {

dbName: process.env.POSTGRES\_DB,

port: parseInt(process.env.POSTGRES\_PORT, 10),

user: process.env.POSTGRES\_USER,

password: process.env.POSTGRES\_PASSWORD,

host: process.env.POSTGRES\_HOST,

},

mysql: {

dbName: process.env.MYSQL\_DB,

user: process.env.MYSQL\_USER,

password: process.env.MYSQL\_PASSWORD,

port: parseInt(process.env.MYSQL\_PORT),

host: process.env.MYSQL\_HOST,

},

apiKey: process.env.API\_KEY,

configuration: process.env.CONFIGURATION,

};

});

Schema:

import \* as Joi from 'joi';

export const pgSchema = Joi.object({

API\_KEY: Joi.number().required(),

POSTGRES\_DB: Joi.string().required(),

POSTGRES\_USER: Joi.string().required(),

POSTGRES\_PASSWORD: Joi.string().required(),

POSTGRES\_HOST: Joi.string().required(),

POSTGRES\_PORT: Joi.number().required(),

PORT: Joi.number().required(),

});

export const mysqlSchema = Joi.object({

MYSQL\_DB: Joi.string().required(),

MYSQL\_USER: Joi.string().required(),

MYSQL\_PASSWORD: Joi.string().required(),

MYSQL\_HOST: Joi.string().required(),

MYSQL\_PORT: Joi.number().required(),

});

Database Module:

imports: [

TypeOrmModule.forRootAsync({

name: 'mysqlDB',

useFactory: (configService: ConfigType<typeof config>) => {

const { dbName, user, password, host, port } = configService.mysql;

return {

type: 'mysql',

host,

port,

username: user,

password,

database: dbName,

synchronize: true,

autoLoadEntities: true,

};

},

inject: [config.KEY],

}),

TypeOrmModule.forRootAsync({

name: 'postgresDB',

useFactory: (configService: ConfigType<typeof config>) => {

const { dbName, user, password, host, port } = configService.postgres;

return {

type: 'postgres',

host,

port,

username: user,

password,

database: dbName,

synchronize: true,

autoLoadEntities: true,

};

},

inject: [config.KEY],

}),

],

Products Module:

@Module({

imports: [TypeOrmModule.forFeature([Product, Category, Brand], 'mysqlDB')],

controllers: [ProductsController, CategoriesController, BrandsController],

providers: [ProductsService, CategoriesService, BrandsService],

exports: [ProductsService],

})

Product Service:

constructor(

@InjectRepository(Product, 'mysqlDB')

private productRepo: Repository<Product>,

) {}

**Formas de controlar los cambios en la base de datos**

**sync mode**

flag synchronize: true esto hace que los cambios que hagamos sobre nuestro modelo de clases impacte directamente en las tablas.

**recomendaciones**

Solo sobre entorno de desarrollo y testing. En produccion es una practica riesgosa, porque cualquier cambio en el codigo, se pueda corromper la base de datos. Para eso se crearon las **migraciones**

**Django**

Dice que las migraciones es la forma que django propaga los CAMBIOS DE LOS O ENTIDADESY modelos a las bases de datos

**Laravel**

son como un control de version de la bd permitiendo que el equipo pueda definir y compartir el esquema de la base de datos.

**TypeORM**

Las migraciones son solo ES UN ARCHIVO un con query tipo sql con el esquema y los cambios aplicados (updates).

**CONFIGURANBDO MIGRACIONES Y NPM SCRIPTS**

Lo que vimos:

No olviden ajustarlo a sus datos  
**.env**

TYPEORM\_CONNECTION = postgres

TYPEORM\_HOST = localhost

TYPEORM\_USERNAME = root

TYPEORM\_PASSWORD = 123456

TYPEORM\_DATABASE = my\_db

TYPEORM\_PORT = 5432

TYPEORM\_SYNCHRONIZE = false

TYPEORM\_LOGGING = true

TYPEORM\_ENTITIES = src/\*\*/\*.entity.ts

TYPEORM\_MIGRATIONS = src/database/migrations/\*.ts

TYPEORM\_MIGRATIONS\_DIR = src/database/migrations

TYPEORM\_MIGRATIONS\_TABLE\_NAME = migrations

**package.json**

"scripts": {

"typeorm": "ts-node -r tsconfig-paths/register ./node\_modules/typeorm/cli.js",

"migrations:generate": "npm run typeorm -- migration:generate -n"

},

Comando para generar la migration

npm run migrations:generate -- init

Si realizaron el test de mysql, no olviden regresarlo a postgres.

**database.module.ts**

imports: [

TypeOrmModule.forRootAsync({

inject: [config.KEY],

useFactory: (configService: ConfigType<typeof config>) => {

const { user, password, host, dbName, port } = configService.postgres;

return {

type: 'postgres',

host,

port,

username: user,

password,

database: dbName,

synchronize: true,

autoLoadEntities: true,

};

},

}),

],

Consulta, por que tendríamos que regresar a postgres? no se puede hacer lo mismo en MySQL

Para evitar futuros dolores de cabeza en sus despliegues, traten de definir siempre sus variables de entorno en archivos .env sin dejar espacios antes/después del signo igual:

# Eviten esto 🚫

TYPEORM\_CONNECTION = postgres

# Así es mejor ✔️

TYPEORM\_CONNECTION=postgres

Me funciono esta configuración en mi package.json:

"typeorm": "node --require ts-node/register ./node\_modules/typeorm/cli.js",

"migration:generate": "npm run typeorm -- migration:generate -n"

Revisando una librería que utilizo de algún tiempo atrás para general las entidades, me he encontrado con esto: [https://github.com/Kononnable/typeorm-model-generator/issues/329](https://github.com/Kononnable/typeorm-model-generator/issues/329" \t "_blank)

Básicamente dice que el futuro de TypeORM es incierto, con errores y un mantenimiento muy pobre, otro artìculo sobre el tema está acá. [https://news.ycombinator.com/item?id=26888369](https://news.ycombinator.com/item?id=26888369" \t "_blank)

Para los que tuvieron problemas en la migración, revisen que tenga node 12 o superior, en mi caso con wsl use [https://github.com/nodesource/distributions/blob/master/README.md](https://github.com/nodesource/distributions/blob/master/README.md" \t "_blank) para instalar la version 17 aunque creo que debí instalar la 15

**Relaciones uno a uno**

Buenas, si no quieren colocar a cada entity los campos createAt y updateAt pueden crear una entidad básica que contenga estos atributos:

import { CreateDateColumn, UpdateDateColumn } from 'typeorm';

export class BasicEntity {

@CreateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

createAt: Date;

@UpdateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

updateAt: Date;

}

Y luego extender cada entidad que necesite estos campos:

import { BasicEntity } from 'src/database/base.entity';

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

@Entity()

export class User extends BasicEntity {

@PrimaryGeneratedColumn()

id: number;

@Column({ type: 'varchar', length: 255 })

email: string;

@Column({ type: 'varchar', length: 255 })

password: string;

@Column({ type: 'varchar', length: 100 })

role: string;

}

De esta forma nos ahorramos un poco de tiempo al crear nuestras entidades

Walter Lensinas

[hace un año](https://platzi.com/comentario/2576425/)

Podria ser una interfaz? porque si es una clase y justo tener que extender de otra clase, no se si typescript tiene herencia multiple.

Guillermo Rodas Gómez

[hace un año](https://platzi.com/comentario/2576425/)

Al ser interfaz, no podrás colocar decoradores a las propiedades de la interfaz, y los atributos createAt y updateAt igualmente tendran que estar en la clase que implementa la interfaz

[4 Respuestas](https://platzi.com/comentario/2576425/)

Ángel David Roque Ayala

[hace un año](https://platzi.com/comentario/2554129/)

## Lo que hicimos

//src/user/entities/customer.entity

import { Column, CreateDateColumn, Entity, OneToOne, PrimaryGeneratedColumn, UpdateDateColumn } from "typeorm";

import { User } from "./user.entity";

@Entity()

export class Customer {

@PrimaryGeneratedColumn()

id: number;

@Column({type: 'varchar', length: 255})

name: string;

@Column({type: 'varchar', length: 255})

lastName: string;

@Column({ type: 'varchar', length: 255 })

phone: string;

@CreateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

createAt: Date;

@UpdateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

updateAt: Date;

@OneToOne(()=>User, (user)=> user.customer)

user:User;

}

//src/user/entities/user.entity

import { Column, CreateDateColumn, Entity, JoinColumn, OneToOne, PrimaryGeneratedColumn, UpdateDateColumn } from "typeorm";

import { Customer } from "./customer.entity";

@Entity()

export class User {

@PrimaryGeneratedColumn()

id: number;

@Column({ type: 'varchar', length: 255})

email: string;

@Column({ type: 'varchar', length: 255})

password: string;

@Column({ type: 'varchar', length: 255})

role: string;

@CreateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP'

})

createAt: Date;

@UpdateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

updateAt: Date;

@OneToOne(()=>Customer, (customer)=>customer.user, {nullable: true})

@JoinColumn()

customer: Customer;

}

//src/user/services/customer.service

import { Injectable, NotFoundException } from '@nestjs/common';

import { Customer } from '../entities/customer.entity';

import { CreateCustomerDto, UpdateCustomerDto } from '../dtos/customer.dto';

import { InjectRepository } from '@nestjs/typeorm';

import { Repository } from 'typeorm';

@Injectable()

export class CustomersService {

constructor(

@InjectRepository(Customer) private customerRepo: Repository<Customer>

){}

async findAll() {

return await this.customerRepo.find()

}

async findOne(id: number) {

const customer = await this.customerRepo.findOne(id);

if (!customer) {

throw new NotFoundException(`Customer #${id} not found`);

}

return customer;

}

create(data: CreateCustomerDto) {

const newCostumer = this.customerRepo.create(data);

return this.customerRepo.save(newCostumer);

}

async update(id: number, changes: UpdateCustomerDto) {

const customer = await this.findOne(id);

this.customerRepo.merge(customer, changes);

return this.customerRepo.save(customer);

}

remove(id: number) {

return this.customerRepo.delete(id);

}

}

//src/user/services/user.service

import { Injectable, NotFoundException, Inject } from '@nestjs/common';

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

import { User } from '../entities/user.entity';

import { Order } from '../entities/order.entity';

import { CreateUserDto, UpdateUserDto } from '../dtos/user.dto';

import { ProductsService } from './../../products/services/products.service';

import { InjectRepository } from '@nestjs/typeorm';

import { Repository } from 'typeorm';

@Injectable()

export class UsersService {

constructor(

@InjectRepository(User) private userRepo: Repository<User>,

private productsService: ProductsService,

private configService: ConfigService,

) {}

async findAll() {

return await this.userRepo.find();

}

async findOne(id: number) {

const user = await this.userRepo.findOne(id);

if (!user) {

throw new NotFoundException(`User #${id} not found`);

}

return user;

}

async create(data: CreateUserDto) {

const newUser = this.userRepo.create(data);

return await this.userRepo.save(newUser);

}

async update(id: number, changes: UpdateUserDto) {

const user = await this.findOne(id);

const updateUser = this.userRepo.merge(user, changes);

return this.userRepo.save(updateUser);

}

remove(id: number) {

return this.userRepo.delete(id);

}

async getOrderByUser(id: number): Promise<Order> {

const user = await this.userRepo.findOne(id);

return {

date: new Date(),

user,

products: await this.productsService.findAll(),

};

}

}

npm run migrations:generate -- create-user-customer

npm run migrations:run

## Apuntes

Para realizar relaciones 1 a 1 debemos hacer lo siguiente

// src/user/entities/user.entity

@OneToOne(()=>Customer, (customer)=>customer.user, {nullable: true})

@JoinColumn()

customer: Customer;

// src/user/entities/customer.entity

@OneToOne(()=>User, (user)=> user.customer)

user:User;

La entidad que tenga el decorador @JoinColumn es el objeto que creara la llave foranea. **Solo una entidad puede tener el decorador @JoinColumn**

TypeORM permite tener una relacion **bidireccional** en las relaciones 1 a 1 sin necesidad de hacer queryes extras.

Vi un comentario que como haría para evitar estar repitiendo los campos de CreateAt y UpdateAt sin usar un herencia de clase, investigando un poco me tope con [Embedded Entities](https://orkhan.gitbook.io/typeorm/docs/embedded-entities" \t "_blank). Dejo mi ejemplo el cual me funcionó.

* Creé un archivo el cual la puse dentro de database.module: “dateAt.entity.ts”

import { CreateDateColumn, UpdateDateColumn } from 'typeorm';

export class DateAt {

@CreateDateColumn({ type: 'timestamp', default: () => 'CURRENT\_TIMESTAMP' })

createAt: Date;

@UpdateDateColumn({ type: 'timestamp', default: () => 'CURRENT\_TIMESTAMP' })

updateAt: Date;

}

* En las entidades solo importe este archivo y dentro de column realize una arrow function hacia esta nueva clase.

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

import { DateAt } from '../../database/dateAt.entity';

@Entity()

export class Customer {

@PrimaryGeneratedColumn()

id: number;

@Column({ type: 'varchar', length: 100 })

name: string;

@Column({ type: 'varchar', length: 100 })

lastName: string;

@Column({ type: 'varchar', length: 20 })

phone: string;

@Column(() => DateAt)

register: DateAt;

}

Me funcionó, incluso corrí el comando npm run migrations:generate -- dateAt, el cual no me dio problema alguno.

José Mauricio Lemús Rodezno

[hace 2 meses](https://platzi.com/comentario/3409287/)

Podemos configurar ONDELETE y ONUPDATE en nuestras entidades de la siguiente manera:

@OneToOne(() => Customer, (customer) => customer.user, {

nullable: true,

onDelete: 'SET NULL',

onUpdate: 'CASCADE',

})

@JoinColumn()

customer: Customer;

Rodrigo Alejandro Rios

[hace 3 meses](https://platzi.com/comentario/3252600/)

Recomiendo hacer una entidad base con id, updateAd y createAd. Para que luego extender todas las demas entidades de esta base.  
Ej:

import {

CreateDateColumn,

PrimaryGeneratedColumn,

UpdateDateColumn,

} from 'typeorm';

export abstract class BaseEntity {

@PrimaryGeneratedColumn('uuid')

id: string;

@CreateDateColumn({

type: 'timestamp',

name: 'created\_at',

})

createdAt: Date;

@UpdateDateColumn({

type: 'timestamp',

name: 'updated\_at',

})

updatedAt: Date;

}

# **Resolviendo la relación uno a uno en el controlador**

## Cambiando el dto user

// src/users/dtos/user.dto

export class CreateUserDto {

...

// Agregamos atributo

@IsOptional()

@IsNotEmpty()

@ApiProperty()

readonly customerId: number;

}

export class UpdateUserDto extends PartialType(CreateUserDto) {}

## Modificando la funcion crear

Si el objeto contiene un customerId traemos los datos usando el servicio de customer

//src/users/services/user.service

async create(data: CreateUserDto) {

const newUser = this.userRepo.create(data);

if(data.customerId){

const customer = await this.customerService.findOne(data.customerId);

newUser.customer = customer;

}

return await this.userRepo.save(newUser);

}

## Modificando la funcion find

Si queremos que cuando se consulte al usuario tambien se obtengan los datos del customer debemos agregar la siguiente opcion

async findAll() {

return await this.userRepo.find({

relations: ['customer'] // esta

});

}

async findOne(id: number) {

const user = await this.userRepo.findOne(id, {

relations: ['customer'] // esta

});

if (!user) {

throw new NotFoundException(`User #${id} not found`);

}

return user;

}

William Abel Condori Asistiri

[hace 4 meses](https://platzi.com/comentario/3222428/)

#### Para obtener un usuario con su customer, en el método findOne, pueden hacer el siguiente cambio:

async findOne(id: number) {

const user = await this.userRepo.findOne(id, { relations: ['customer'] });

if (!user) {

throw new NotFoundException(`User #${id} not found`);

}

return user;

}

# Resolviendo la relación uno a uno en el controlador

## Cambiando el dto user

// src/users/dtos/user.dto

export class CreateUserDto {

...

// Agregamos atributo

@IsOptional()

@IsNotEmpty()

@ApiProperty()

readonly customerId: number;

}

export class UpdateUserDto extends PartialType(CreateUserDto) {}

## Modificando la funcion crear

Si el objeto contiene un customerId traemos los datos usando el servicio de customer

//src/users/services/user.service

async create(data: CreateUserDto) {

const newUser = this.userRepo.create(data);

if(data.customerId){

const customer = await this.customerService.findOne(data.customerId);

newUser.customer = customer;

}

return await this.userRepo.save(newUser);

}

## Modificando la funcion find

Si queremos que cuando se consulte al usuario tambien se obtengan los datos del customer debemos agregar la siguiente opcion

async findAll() {

return await this.userRepo.find({

relations: ['customer'] // esta

});

}

async findOne(id: number) {

const user = await this.userRepo.findOne(id, {

relations: ['customer'] // esta

});

if (!user) {

throw new NotFoundException(`User #${id} not found`);

}

return user;

}

#### Para obtener un usuario con su customer, en el método findOne, pueden hacer el siguiente cambio:

async findOne(id: number) {

const user = await this.userRepo.findOne(id, { relations: ['customer'] });

if (!user) {

throw new NotFoundException(`User #${id} not found`);

}

return user;

}

# Relaciones uno a muchos

Lo que vimos:

**brand.entity.ts**

import {

PrimaryGeneratedColumn,

Column,

Entity,

CreateDateColumn,

UpdateDateColumn,

OneToMany,

} from 'typeorm';

import { Product } from './product.entity';

@Entity()

export class Brand {

@PrimaryGeneratedColumn()

id: number;

@Column({ type: 'varchar', length: 255, unique: true })

name: string;

@Column({ type: 'varchar', length: 255 })

image: string;

@CreateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

createAt: Date;

@UpdateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

updateAt: Date;

@OneToMany(() => Product, (product) => product.brand)

products: Product;

}

**brands.service.ts**

import { Injectable, NotFoundException } from '@nestjs/common';

import { InjectRepository } from '@nestjs/typeorm';

import { Repository } from 'typeorm';

import { Brand } from '../entities/brand.entity';

import { CreateBrandDto, UpdateBrandDto } from '../dtos/brand.dtos';

@Injectable()

export class BrandsService {

constructor(@InjectRepository(Brand) private brandRepo: Repository<Brand>) {}

findAll() {

return this.brandRepo.find();

}

async findOne(id: number) {

const product = await this.brandRepo.findOne(id);

if (!product) {

throw new NotFoundException(`Brand #${id} not found`);

}

return product;

}

create(data: CreateBrandDto) {

const newBrand = this.brandRepo.create(data);

return this.brandRepo.save(newBrand);

}

async update(id: number, changes: UpdateBrandDto) {

const brand = await this.brandRepo.findOne(id);

this.brandRepo.merge(brand, changes);

return this.brandRepo.save(brand);

}

remove(id: number) {

return this.brandRepo.delete(id);

}

}

**product.entity.ts**

import {

PrimaryGeneratedColumn,

Column,

Entity,

CreateDateColumn,

UpdateDateColumn,

ManyToOne,

} from 'typeorm';

import { Brand } from './brand.entity';

@Entity()

export class Product {

@PrimaryGeneratedColumn()

id: number;

@Column({ type: 'varchar', length: 255, unique: true })

name: string;

@Column({ type: 'text' })

description: string;

@Column({ type: 'int' })

price: number;

@Column({ type: 'int' })

stock: number;

@Column({ type: 'varchar' })

image: string;

@CreateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

createAt: Date;

@UpdateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

updateAt: Date;

@ManyToOne(() => Brand, (brand) => brand.products)

brand: Brand;

}

Ángel David Roque Ayala

[hace un año](https://platzi.com/comentario/2542933/)

Excelente documentación 😄. Apoyo recordando que debemos generar y correr las migraciones

npm run migrations:generate -- create-brands

npm run migrations:run

Jose Daniel Barría Reyes

[hace un año](https://platzi.com/comentario/2542933/)

Te faltaron los corchetes para decir que son muchos productos

@OneToMany(() => Product, (product) => product.brand)

products: Product[];

Muchas gracias por tu aporte!

Carlos Vallejo

[hace 10 meses](https://platzi.com/comentario/2716600/)

He creado dos snippets de VS Code para crear un Servicio y un Controller básico mucho más rápido. (para hacer lo que el profe hace en cámara rápida).

Para crearlos van a:

* File
* Preferences
* User Snippets
* Buscan typescript

En el archivo JSON que se les abrió copian este código dentro de las llaves:

* Servicio

"Create NestJS Service": {

"prefix": "nest:s",

"body": [

"import { Injectable, NotFoundException } from '@nestjs/common';",

"import { InjectRepository } from '@nestjs/typeorm';",

"import { Repository } from 'typeorm';",

"",

"import { $2 } from '${1:entity-route}';",

"import { ${4:createDto}, ${5:updateDto} } from '${3:dto-route}';",

"",

"@Injectable()",

"export class ${6:serviceName}Service {",

" constructor(",

" @InjectRepository($2)",

" private ${7:repo}: Repository<$2>,",

" ) {}",

"${0}",

" async findAll() {",

" return await this.$7.find();",

" }",

"",

" async findOne(id: number) {",

" const ${8:object} = await this.$7.findOne(id);",

" if (!$8) throw new NotFoundException(`${9:object} not found.`);",

" return $8;",

" }",

"",

" async create(data: $4) {",

" const ${10:newObject} = this.$7.create(data);",

" return await this.$7.save($10);",

" }",

"",

" async update(id: number, changes: $5) {",

" const $8 = await this.findOne(id);",

" this.$7.merge($8, changes);",

" return await this.$7.save($8);",

" }",

"",

" async remove(id: number) {",

" return await this.$7.delete(id);",

" }",

"}",

],

"description": "This service has a basic CRUD implemented"

},

* Controller

"Create NestJS Controller": {

"prefix": "nest:co",

"body": [

"import {",

" Controller,",

" Get,",

" Post,",

" Put,",

" Body,",

" Param,",

" Delete,",

" ParseIntPipe,",

"} from '@nestjs/common';",

"import { ApiTags } from '@nestjs/swagger';",

"",

"import { ${2:serviceName} } from '${1:service-route}';",

"import { ${4:createDto}, ${5:updateDto} } from '${3:dto-route}';",

"",

"@ApiTags('${6:controllerName}')",

"@Controller('$6')",

"export class ${7:controllerNameInUppercase}Controller {",

" constructor(private ${8:serviceName}: $2) {}",

"",

" @Get()",

" async findAll() {",

" return await this.$8.findAll();",

" }",

"",

" @Get('/:id')",

" async getCategory(@Param('id') id: number) {",

" return await this.$8.findOne(id);",

" }",

"",

" @Post()",

" async create(@Body() payload: $4) {",

" return await this.$8.create(payload);",

" }",

"",

" @Put('/:id')",

" async update(",

" @Param('id', ParseIntPipe) id: number,",

" @Body() payload: $5,",

" ) {",

" return await this.$8.update(id, payload);",

" }",

"",

" @Delete('/:id')",

" async remove(@Param('id', ParseIntPipe) id: number) {",

" return await this.$8.remove(id);",

" }",

"}",

],

"description": "This controller has a basic CRUD implemented"

}

Para usarlos simplemente tienen que crear su servicio/controlador normalmente y borrar lo que hay ahí.  
Luego simplemente ponen nest:s o nest:co y empiezan a llenar todos los datos que se pide.  
Para avanzar solo precionen la tecla tab

Espero les sirva! 😃

# Resolviendo la relación uno a muchos en el controlador

En el servicio para una marca por Id, falta colocar la condición (where) para que busque el Id que enviamos. Quedaría así:

const product = await this.brandsRepo.findOne({

relations: ['products'],

where: {

id,

},

});



Juan Caballero

[hace un año](https://platzi.com/comentario/2557797/)

Pero en la consulta se usa el metodo finOne con el id como parametro

const brand = await this.brandRepo.findOne(id,{

relations: ['products']

});



Walter Lensinas

[hace un año](https://platzi.com/comentario/2557797/)

Gracias Carlos! iba justo a ver como mandarle el id en ese objeto.

Genio!



Lucas Heim

[hace 10 meses](https://platzi.com/comentario/2654369/)

Dejo una pequeña validacion para manejar los errores del create:

async create(payload: CreateProductDto) {

const newProduct = this.productRepo.create(payload);

if (payload.brandId) {

const brand = await this.brandService.findOne(payload.brandId);

newProduct.brand = brand;

}

const product = this.productRepo

.save(newProduct)

.then((res) => {

return res;

})

.catch((err) => {

throw new BadRequestException(`${err.message || 'Unexpected Error'}`);

});

return product;

}



Renato Renzo Luna Herrera

[hace 4 meses](https://platzi.com/comentario/3213416/)

me parece o ese findOne() deberia tener un id en el primer parámetro?, es decir:

const product = await this.brandRepo.findOne(id, {

relations: ['products']

});

# Relaciones muchos a muchos

## Apuntes

#### category.entity

// src\products\entities\category.entity.ts

import { Column, CreateDateColumn, Entity, PrimaryGeneratedColumn, UpdateDateColumn } from "typeorm";

@Entity()

export class Category {

@PrimaryGeneratedColumn()

id: number;

@Column({ type: 'varchar', length: 255, unique: true})

name: string;

@CreateDateColumn({

type: 'timestamptz',

default: ()=> 'CURRENT\_TIMESTAMP'

})

createAt: Date;

@UpdateDateColumn({

type: 'timestamptz',

default: ()=> 'CURRENT\_TIMESTAMP'

})

updateAt: Date;

}

### category.service

// src\products\services\categories.service.ts

import { Injectable, NotFoundException } from '@nestjs/common';

import { Category } from '../entities/category.entity';

import { CreateCategoryDto, UpdateCategoryDto } from '../dtos/category.dtos';

import { InjectRepository } from '@nestjs/typeorm';

import { Repository } from 'typeorm';

import { merge } from 'rxjs';

@Injectable()

export class CategoriesService {

constructor(

@InjectRepository(Category) private categoryRepo: Repository<Category>

){}

findAll() {

return this.categoryRepo.find();

}

async findOne(id: number) {

const category = await this.categoryRepo.findOne(id);

if (!category) {

throw new NotFoundException(`Category #${id} not found`);

}

return category;

}

create(data: CreateCategoryDto) {

const newCategory = this.categoryRepo.create(data);

return this.categoryRepo.save(newCategory);

}

async update(id: number, changes: UpdateCategoryDto) {

const category = await this.findOne(id);

this.categoryRepo.merge(category, changes);

return this.categoryRepo.save(category);

}

remove(id: number) {

return this.categoryRepo.delete(id);

}

}

### products module

// src\products\products.module.ts

@Module({

...

// Agregamos a category

imports:[TypeOrmModule.forFeature([Product, Brand, Category])],

...

})

### Creando la relacion

// src\products\entities\category.entity.ts

@ManyToMany(()=>Product, (product)=>product.categories)

@JoinTable() // Solo debe estar en una entidad

products: Product[]

// src\products\entities\product.entity.ts

@ManyToMany(()=>Category, (category)=>category.products)

categories: Category[];

### Crear y correr la migracion

npm run migrations:generate -- create-categories

npm run migrations:run



ZANONI ALFREDO SALAS TOBON

[hace 5 meses](https://platzi.com/comentario/3121367/)

Tal vez les sirva para nombrar las columnas de las relaciones.

## @JoinColumn

Defines which side of the relation contains the join column with a foreign key and allows you to customize the join column name and referenced column name. Example:

@Entity()  
export class Post {

@ManyToOne(type => Category)

@JoinColumn({

name: "cat\_id",

referencedColumnName: "name"

})

category: Category;

}

## @JoinTable

Used for many-to-many relations and describes join columns of the “junction” table. Junction table is a special, separate table created automatically by TypeORM with columns referenced to the related entities. You can change the name of the generated “junction” table and also the column names inside the junction table and their referenced columns with the joinColumn- and inverseJoinColumn attributes. Example:

@Entity()  
export class Post {

@ManyToMany(type => Category)

@JoinTable({

name: "question\_categories",

joinColumn: {

name: "question",

referencedColumnName: "id"

},

inverseJoinColumn: {

name: "category",

referencedColumnName: "id"

}

})

categories: Category[];

}



ZANONI ALFREDO SALAS TOBON

[hace 5 meses](https://platzi.com/comentario/3121499/)

Para personalizar los nombres de las tablas y llaves o referencias de la tabla

/user.entity.ts

@OneToOne(() => Customer, (customer) => customer.user, { nullable: true })

@JoinColumn({

name: "pk\_customer",

referencedColumnName: "id"

})

customer: Customer

/category.entity.ts

@ManyToMany(() => Product, (products) => products.categories, { nullable: true })

products: Product[];

/product.entity.ts

@ManyToOne(() => Brand, (brand) => brand.products, { nullable: true })

brand: Brand

@ManyToMany(() => Category, (categories) => categories.products, { nullable: true })

@JoinTable({

name: "product\_categories",

joinColumn: {

name: "pk\_product",

referencedColumnName: "id"

},

inverseJoinColumn: {

name: "pk\_category",

referencedColumnName: "id"

}

})

categories: Category[];

# Resolviendo la relación muchos a muchos en el controlador

## Apuntes

**modificamos el dto**

import {

IsString,

IsNumber,

IsUrl,

IsNotEmpty,

IsPositive,

IsArray,

} from 'class-validator';

import { PartialType, ApiProperty } from '@nestjs/swagger';

export class CreateProductDto {

...

// Agregamos este campo

@IsArray()

@IsNotEmpty()

@ApiProperty()

readonly categoriesIds: number[];

}

export class UpdateProductDto extends PartialType(CreateProductDto) {}

### Create

Modificaremos el metodo para crear un producto

// src\products\services\products.service.ts

async create(data: CreateProductDto) {

const newProduct = this.productRepo.create(data);

if (data.brandId) {

const brand = await this.brandRepo.findOne(data.brandId);

newProduct.brand = brand;

}

// Aqui resolvemos la categoria

if (data.categoriesIds) {

const categories = await this.categoryRepo.findByIds(data.categoriesIds);

newProduct.categories = categories

}

return this.productRepo.save(newProduct);

}

### Find

// src\products\services\products.service.ts

async findOne(id: number) {

const product = await this.productRepo.findOne(id,{

relations: ['brand', 'categories']

});

if (!product) {

throw new NotFoundException(`Product #${id} not found`);

}

return product;

}

### Find of categories

// src\products\services\categories.service.ts

async findOne(id: number) {

const category = await this.categoryRepo.findOne(id,{

relations: ['products']

});

if (!category) {

throw new NotFoundException(`Category #${id} not found`);

}

return category;

}



Jairo Campos Ruiz

[hace 13 días](https://platzi.com/comentario/3556156/)

Buenas la forma correcta de revisar categoriesId en el dto es

@ApiProperty()

@IsArray()

@ArrayMinSize(1)

@IsNumber({}, { each: true })

readonly categoriesId: number[];

De esta manera te revisa que tenga al menos 1 elemento el arrray y que los elementos sean numbers



rdelacruz1

[hace 2 días](https://platzi.com/comentario/3556156/)

También se puede usar el decorador @ArrayNotEmpty() en vez de @ArrayMinSize(1)



rdelacruz1

[hace 2 días](https://platzi.com/comentario/3590437/)

En caso tal estén haciendo cambios y no los vean reflejados al hacer las peticiones, deben hacer build npm run build y luego volver a correr el proyecto. Otra solución es eliminar la carpeta dist y volver a correr el proyecto.

# Manipulación de arreglos en relaciones muchos a muchos

## Apuntes

#### Remover una categoría de un producto

// src\products\services\products.service.ts

asyncremoveCategoryByProduct(productId: number, categoryId: number){

const product = awaitthis.productRepo.findOne(productId, {

relations: ['categories']

});

product.categories = product.categories.filter((item) => {

return item.id !== categoryId

})

}

// src\products\controllers\products.controller.ts

@Delete(':id/category/:categoryId')

deleteCategory(

@Param('id', ParseIntPipe) id: number,

@Param('categoryId', ParseIntPipe) categoryId: number

){

returnthis.productsService.removeCategoryByProduct(id, categoryId);

}

#### Agregar categoría a un producto

// src\products\services\products.service.ts

asyncaddCategoryToProduct(productId: number, categoryId: number){

const product = awaitthis.productRepo.findOne(productId, {

relations: ['categories']

});

const category = awaitthis.categoryRepo.findOne(categoryId);

product.categories.push(category);

returnthis.productRepo.save(product);

}

// src\products\controllers\products.controller.ts

@Put(':id/category/:categoryId')

addCategoryToProduct(

@Param('id', ParseIntPipe) idProduct: number,

@Param('categoryId', ParseIntPipe) idCategory: number,

) {

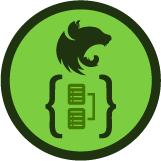
returnthis.productsService.addCategoryToProduct(idProduct, idCategory);

}

[roque-ayala-david-angel](https://platzi.com/p/roque-ayala-david-angel/)

1

hace un año



[Curso de NestJS: Persistencia de Datos con TypeORM](https://platzi.com/cursos/nestjs-typeorm/)•[Manipulación de arreglos en relaciones muchos a muchos](https://platzi.com/clases/2282-nestjs-typeorm/37309-manipulacion-de-arreglos-en-relaciones-muchos-a-mu/)



Escribe tu comentario

[danibarria](https://platzi.com/p/danibarria/)

41426 Puntos

un año

muy buen aporte, por cierto en:

async removeCategoryByProduct(productId: number, categoryId: number)

te falto

returnthis.productRepo.save(product);

Al método de añadir categorías yo le añadí algunas validaciones como comprobar si existe el product o la category y comprobar si ya contiene esta categoría para que no se repita.

asyncaddCategoryToProduct(productId: number, categoryId: number) {

const product = awaitthis.productRepo.findOne(productId, {

relations: ['categories', 'brand'],

});

if (!product) {

thrownew NotFoundException(`Product #${productId} not found`);

}

const category = awaitthis.categoryRepo.findOne(categoryId);

if (!category) {

thrownew NotFoundException(`Category #${categoryId} not found`);

}

if (!product.categories.find((item) => item.id == categoryId)) {

product.categories.push(category);

}

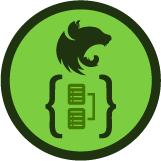
returnthis.productRepo.save(product);

}

[jairocamposruiz](https://platzi.com/p/jairocamposruiz/)

2

hace un año



[Curso de NestJS: Persistencia de Datos con TypeORM](https://platzi.com/cursos/nestjs-typeorm/)•[Manipulación de arreglos en relaciones muchos a muchos](https://platzi.com/clases/2282-nestjs-typeorm/37309-manipulacion-de-arreglos-en-relaciones-muchos-a-mu/)



Escribe tu comentario

[luisrangelc](https://platzi.com/p/luisrangelc/)

34977 Puntos

3 meses

async addCategoryToProduct(productId: number, categoryId: number) {

const product = awaitthis.productRepository.findOne(productId, {

relations: ['categories'],

});

if (!product) {

thrownew NotFoundException(`Product #${productId} not found`);

}

const category = awaitthis.categoriesService.findOne(categoryId);

if (!category) {

thrownew NotFoundException(`Category #${categoryId} not found`);

}

if (!product.categories.find((item) => item.id == categoryId)) {

product.categories.push(category);

} else {

thrownew ConflictException(

`Category #${categoryId} is already present in this product`,

);

}

returnthis.productRepository.save(product);

}

[luissberenguer](https://platzi.com/p/luissberenguer/)

38571 Puntos

6 meses

Perfecto Jairo, en el caso de que una persona quisiera agregar una categoría que ya está presente en el producto yo le agregaría el siguiente código.

A …

if (!product.categories.find((item) => item.id == categoryId)) {

product.categories.push(category);

}

Le añadimos …

else {

thrownew ConflictException(

`Category #${categoryId} is already present in this product`,

);

}

Modifique el código para que no se pueda agregar una categoría existente

async addCategoryByProduct(productId: number, categoryId: number) {

const product = await this.productRepo.findOne(productId, {

relations: ['categories'],

});

const category = await this.categoryRepo.findOne(categoryId);

if (!category) {

throw new NotFoundException(`Category #${categoryId} not found`);

}

const exists = product.categories.some(

(element) => element.id === categoryId,

);

if (!exists) {

product.categories.push(category);

}

return this.productRepo.save(product);

}

# Relaciones muchos a muchos personalizadas

Resumen de la clase

* Modificamos order.entity.

**// src\users\entities\order.entity.ts**

import {

PrimaryGeneratedColumn,

UpdateDateColumn,

CreateDateColumn,

ManyToOne,

Entity,

OneToMany,

} from 'typeorm';

import { Customer } from './customer.entity';

import { OrderItem } from './order-item.entity';

@Entity()

export class Order {

@PrimaryGeneratedColumn()

id: number;

@CreateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

createAt: Date;

@UpdateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

updateAt: Date;

@ManyToOne(() => Customer, (customer) => customer.orders)

customer: Customer;

@OneToMany(() => OrderItem, (item) => item.order)

items: OrderItem[];

}

* Agregamos la relación en customer

**// src\users\entities\customer.entity.ts**

import {

...

OneToMany,

} from 'typeorm';

import { User } from './user.entity';

import { Order } from './order.entity';

@Entity()

export class Customer {

...

@OneToMany(() => Order, (order) => order.customer)

orders: Order[];

}

* Crear tabla ternaria order-item

import {

PrimaryGeneratedColumn,

UpdateDateColumn,

CreateDateColumn,

Entity,

Column,

ManyToOne,

} from 'typeorm';

import { Product } from '../../products/entities/product.entity';

import { Order } from './order.entity';

@Entity()

export class OrderItem {

@PrimaryGeneratedColumn()

id: number;

@CreateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

createAt: Date;

@UpdateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

updateAt: Date;

@Column({ type: 'int' }) // columna que se agrega a la relacion

quantity: number;

@ManyToOne(() => Product) // en este caso no es funcional la relación bidireccional

product: Product;

@ManyToOne(() => Order, (order) => order.items)

order: Order;

}

* Agregar las entidades en el user.module

...

import { Order } from './entities/order.entity';

import { OrderItem } from './entities/order-item.entity';

@Module({

imports: [

...

TypeOrmModule.forFeature([User, Customer, Order, OrderItem]),

],

...

})

export class UsersModule {}

* Crear y correr migration

npm run migrations:generate --orders

npm run migrations:run

# Resolviendo la relación muchos a muchos personalizada en el controlador

## Mis apuntes

**CREANDO SERVICIOS Y CONTRALADORES EN NESTJS**

nest g s users/services/orders --flat

nest g co users/controllers/orders --flat

nest g co users/services/order-item --flat

nest g co users/controllers/order-item –flat

**// src\users\dtos\order.dto.ts**

import { ApiProperty, PartialType } from "@nestjs/swagger";

import { IsNotEmpty, IsPositive } from "class-validator";

export class CreateOrderDto {

@IsPositive()

@IsNotEmpty()

@ApiProperty()

readonly customerId: number;

}

export class UpdateOrderDto extends PartialType(CreateOrderDto){}

// src\users\services\orders.service.ts

import { Injectable, NotFoundException } from '@nestjs/common';

import { InjectRepository } from '@nestjs/typeorm';

import { Repository } from 'typeorm';

import { CreateOrderDto, UpdateOrderDto } from '../dtos/order.dto';

import { Customer } from '../entities/customer.entity';

import { Order } from '../entities/order.entity';

@Injectable()

export class OrdersService {

constructor(

@InjectRepository(Order) private orderRepo:Repository<Order>,

@InjectRepository(Customer) private customerRepo:Repository<Customer>

){}

findAll(){

return this.orderRepo.find();

}

async findOne(id: number){

const order = await this.orderRepo.findOne(id);

if(!order) {

throw new NotFoundException(`order ${id} not found`);

}

return order;

}

async create(data: CreateOrderDto){

const order = new Order();

if (data.customerId) {

const customer = await this.customerRepo.findOne(data.customerId);

order.customer = customer;

}

return this.orderRepo.save(order);

}

async update(id:number, changes:UpdateOrderDto){

const order = await this.orderRepo.findOne(id);

if (changes.customerId) {

const customer = await this.customerRepo.findOne(changes.customerId);

order.customer = customer;

}

return this.orderRepo.save(order);

}

delete(id:number){

return this.orderRepo.delete(id);

}

}

// src\users\controllers\orders.controller.ts

import { Body, Controller, Delete, Get, Param, ParseIntPipe, Post, Put } from '@nestjs/common';

import { CreateOrderDto, UpdateOrderDto } from '../dtos/order.dto';

import { OrdersService } from '../services/orders.service';

@Controller('orders')

export class OrdersController {

constructor(private ordersService: OrdersService){}

@Get('')

findAll(){

return this.ordersService.findAll();

}

@Get(':id')

findOne(

@Param('id', ParseIntPipe) id: number

){

return this.ordersService.findOne(id);

}

@Post('')

create(@Body() payload: CreateOrderDto){

return this.ordersService.create(payload);

}

@Put(':id')

update(

@Param('id', ParseIntPipe) id:number,

@Body() payload: UpdateOrderDto

){

return this.ordersService.update(id, payload);

}

@Delete(':id')

delete(@Param('id') id: number){

return this.ordersService.delete(id);

}

}

// src\users\dtos\order-item.dto.ts

import { ApiProperty, PartialType } from "@nestjs/swagger";

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

export class CreateOrderItemDto {

@IsNumber()

@IsPositive()

@IsNotEmpty()

@ApiProperty()

readonly orderId: number;

@IsNumber()

@IsPositive()

@IsNotEmpty()

@ApiProperty()

readonly productId: number;

@IsNumber()

@IsPositive()

@IsNotEmpty()

@ApiProperty()

readonly quantity: number;

}

export class UpdateOrderItemDto extends PartialType(CreateOrderItemDto){}

// src\products\products.module.ts

@Module({

...

exports: [..., TypeOrmModule], // export TypeOrmModule

})

export class ProductsModule {}

// src\users\services\order-item.service.ts

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

import { InjectRepository } from '@nestjs/typeorm';

import { Product } from 'src/products/entities/product.entity';

import { Repository } from 'typeorm';

import { CreateOrderItemDto, UpdateOrderItemDto } from '../dtos/order-item.dto';

import { OrderItem } from '../entities/order-item.entity';

import { Order } from '../entities/order.entity';

@Injectable()

export class OrderItemService {

constructor(

@InjectRepository(Product) private productRepo:Repository<Product>,

@InjectRepository(Order) private orderRepo:Repository<Order>,

@InjectRepository(OrderItem) private orderItemRepo: Repository<OrderItem>

){}

async create(data: CreateOrderItemDto){

const order = await this.orderRepo.findOne(data.orderId);

const product = await this.productRepo.findOne(data.productId);

const newOrderItem = new OrderItem();

newOrderItem.order = order;

newOrderItem.product = product;

newOrderItem.quantity = data.quantity;

return this.orderItemRepo.save(newOrderItem)

}

}

// src\users\controllers\order-item.controller.ts

import { Body, Controller, Post } from '@nestjs/common';

import { CreateOrderItemDto } from '../dtos/order-item.dto';

import { OrderItemService } from '../services/order-item.service';

@Controller('order-item')

export class OrderItemController {

constructor(private orderItemService: OrderItemService){}

@Post('')

create(@Body() payload: CreateOrderItemDto){

return this.orderItemService.create(payload);

}

}

// src\users\services\orders.service.ts

async findOne(id: number){

const order = await this.orderRepo.findOne(id, {

relations: ["items", "items.product"] // Agregamos relacion

});

if(!order) {

throw new NotFoundException(`order ${id} not found`);

}

return order;

}

## **Mi Solucion al Reto**

### findAll, findOne, update, delete

// src\users\services\order-item.service.ts

findAll(){

return this.orderItemRepo.find({

relations: ['order', 'order.customer']

});

}

findOne(id: number){

return this.orderItemRepo.findOne(id,{

relations: ['order', 'product']

});

}

async update(id: number, changes: UpdateOrderItemDto){

const orderItem = await this.orderItemRepo.findOne(id);

if (changes.orderId) {

const order = await this.orderRepo.findOne(changes.orderId);

orderItem.order = order;

}

if (changes.productId) {

const product = await this.productRepo.findOne(changes.productId);

orderItem.product = product;

}

if(changes.quantity){

orderItem.quantity = changes.quantity;

}

return this.orderItemRepo.save(orderItem);

}

delete(id: number){

return this.orderItemRepo.delete(id);

}

// src\users\controllers\order-item.controller.ts

@Get('')

findAll(){

return this.orderItemService.findAll();

}

@Get(':id')

findOne(@Param('id', ParseIntPipe) id: number){

return this.orderItemService.findOne(id);

}

@Put(':id')

update(

@Param('id', ParseIntPipe) id: number,

@Body() payload: UpdateOrderItemDto

){

return this.orderItemService.update(id, payload);

}

@Delete(':id')

delete(@Param('id') id: number){

return this.orderItemService.delete(id);

}

Otro tip, pueden usar nest generate resource <nombre-del-recurso> para autogenerar un CRUD ([link](https://docs.nestjs.com/recipes/crud-generator" \t "_blank)):

$ nest g resource users

> ? What transport layer do you use? GraphQL (code first)

> ? Would you like to generate CRUD entry points? Yes

> CREATE src/users/users.module.ts (224 bytes)

> CREATE src/users/users.resolver.spec.ts (525 bytes)

> CREATE src/users/users.resolver.ts (1109 bytes)

> CREATE src/users/users.service.spec.ts (453 bytes)

> CREATE src/users/users.service.ts (625 bytes)

> CREATE src/users/dto/create-user.input.ts (195 bytes)

> CREATE src/users/dto/update-user.input.ts (281 bytes)

> CREATE src/users/entities/user.entity.ts (187 bytes)

> UPDATE src/app.module.ts (312 bytes)

# **Paginación**

// src\products\dtos\products.dtos.ts

// Creamos un nuevo dto

export class FilterProductDto{

@IsOptional()

@IsPositive()

@ApiProperty()

limit:number;

@IsOptional()

@Min(0)

@ApiProperty()

offset: number;

}

// src\products\controllers\products.controller.ts

@Get()

@ApiOperation({ summary: 'List of products' })

getProducts(

// Cambiamos la forma de recibir el parametro

@Query() params: FilterProductDto

) {

return this.productsService.findAll(FilterProductDto);

}

// src\products\services\products.service.ts

// Modificamos el servicio

async findAll(params?: FilterProductDto) {

if (params) {

const { limit, offset } = params;

return this.productRepo.find({

relations: ['brand'],

take: limit,

skip: offset,

});

}

return this.productRepo.find({

relations: ['brand'],

});

}

// src\main.ts

// Activamos la transformacion de forma implicita

app.useGlobalPipes(

new ValidationPipe({

whitelist: true,

forbidNonWhitelisted: true,

transformOptions: {

enableImplicitConversion: true

}

}),

);



Álvaro Rubio Gómez

[hace un año](https://platzi.com/comentario/2557906/)

Gracias por la info Angel, muy util 😃. Solo decir que tienes un typo , en product.controller.ts sería:

this.productsService.findAll(params);

Un saludo.

# Filtrando precios con operadores

## Lo que hicimios

// src\products\dtos\products.dtos.ts

@IsOptional()

@IsPositive()

minPrice: number;

@ValidateIf((item) => item.minPrice)

@IsPositive()

maxPrice: number;

// src\products\services\products.service.ts

// Cambiamos el servicio

async findAll(params?: FilterProductDto) {

if (params) {

const where: FindConditions<Product> = {};

const { limit, offset } = params;

const { maxPrice, minPrice } = params;

if (minPrice && maxPrice) {

where.price = Between(minPrice, maxPrice);

}

return this.productRepo.find({

relations: ['brand'],

where,

take: limit,

skip: offset,

});

}

return this.productRepo.find({

relations: ['brand'],

});

}



Jordan Salazar

[hace 6 meses](https://platzi.com/comentario/2983046/)

A medida que va avanzando el curso quedo más maravillado con todo lo que puedo hacer con Nest JS ofrece.



Jairo Campos Ruiz

[hace 13 días](https://platzi.com/comentario/3559292/)

Aquí dejo un ejemplo con más condiciones, con ordenamiento y alguna que otra cosa más:

Para dividir un poco el Dto me cree una clase de la que heredarán los otros filter con limit y offset ya que estos son muy comunes en cualquier filtro

import { IsOptional, IsPositive, Min } from 'class-validator';

import { ApiProperty } from '@nestjs/swagger';

export class BaseFilterDto {

@ApiProperty()

@IsOptional()

@IsPositive()

readonly limit: number;

@ApiProperty()

@IsOptional()

@Min(0)

readonly offset: number;

}

Al Dto de filtrar productos le añadí varias opciones más como que puedas pasar solo un minimo o maximo precio, ordenar por un parámetro en concreto, orden descendente o ascendente y buscar por id de marca.

export class FilterProductsDto extends BaseFilterDto {

@ApiProperty()

@IsOptional()

@Min(0)

readonly minPrice: number;

@ApiProperty()

@IsOptional()

@IsPositive()

readonly maxPrice: number;

@ApiProperty({ enum: ['price', 'id'] })

@ValidateIf((item) => item.order)

@IsEnum(['price', 'id'])

readonly orderBy: string;

@ApiProperty({ enum: ['ASC', 'DESC'] })

@ValidateIf((item) => item.orderBy)

@IsEnum(['ASC', 'DESC'])

readonly order: string;

@ApiProperty()

@IsOptional()

@IsNumber()

readonly brandId: number;

}

Para terminar el service me queda de la siguiente manera:

async findAll(params?: FilterProductsDto) {

const { limit, offset, maxPrice, minPrice, orderBy, order, brandId } =

params;

const findOptions: FindManyOptions<Product> = {};

const where: FindConditions<Product> = {};

if (minPrice && maxPrice) {

where.price = Between(minPrice, maxPrice);

} else if (minPrice) {

where.price = MoreThanOrEqual(minPrice);

} else if (maxPrice) {

where.price = LessThanOrEqual(maxPrice);

}

if (brandId) where.brand = { id: brandId };

if (orderBy && order) findOptions.order = { [orderBy]: order };

findOptions.where = where;

findOptions.skip = offset;

findOptions.take = limit;

findOptions.relations = ['brand', 'categories'];

return await this.productRepository.find(findOptions);

}



ZANONI ALFREDO SALAS TOBON

[hace 5 meses](https://platzi.com/comentario/3129520/)

Es necesario hacer las validaciones a ambos campos o el otro pasa como vacío

@IsPositive()

@ValidateIf((item) => item.minPrice)

maxPrice: number

@IsPositive()

@ValidateIf((item) => item.maxPrice)

minPrice: number



Ricardo Tribaldos Hernández

[hace 5 meses](https://platzi.com/comentario/3129520/)

Sería mejor ajustar la lógica para que puedas trabajar solo con uno o con el otro que es un caso de uso muy usado, además de validar que minPrice < maxPrice en caso de que te pasen ambos parámetros



ZANONI ALFREDO SALAS TOBON

[hace 5 meses](https://platzi.com/comentario/3129520/)

Si, luego me toco quitarlo, por que generaba un error creo que con la paginación. :C

# Agregando indexadores

## Apuntes

Los indexadores nos sirven para optimizar una búsqueda a través de uno o mas atributos, por defecto el id esta indexado.

**Solo debemos indexar algunos campos, si priorizamos todos los campos al final ningún campo lo será.**

Para ello solo debemos agregar el decorator @Index()

//

...

@Entity()

@Index(['price', 'stock'])

export class Product {

@PrimaryGeneratedColumn()

...

}

No olvides correr las migraciones

npm run migrations:generate -- index-product

npm run migrations:run

# Modificando el naming

Apuntes de la clase

**Naming o Nombramiento de variables en BD**

* El naming es una buena practica.
* Nombre de una variable tiene que ser representativa
* Tiene que ser escrita en camelCase para JavaScript
* En BD una buena práctica es no tener caracteres especiales como la mayúscula.
  + Debido que si se hace consultas directa a la BD habría que escapar caracteres especiales.
* En BD debe ser separadas con \_. (snake\_case)
* Nombre de la tabla debe ser en plural.
* //products/entities/product.entity.ts
* @Entity({ name: 'products'}) // Nombre de la tabla
* @Index(['price', 'stock'])
* export class Product{
* ....
* @CreateDateColumn({
* name: 'created\_at', // nombre de la columna en la BD
* type: 'timestamptz',
* default: () => 'CURRENT\_TIMESTAMP',
* })
* createdAt: Date; // Nombre de la variable en el codigo
* @UpdateDateColumn({
* name: 'update\_at',
* type: 'timestamptz',
* default: () => 'CURRENT\_TIMESTAMP',
* })
* updateAt: Date;
* ...
* }
* En los casos de relaciones OneToOne se hace la modificación en la tabla que tenga el decorator @JoinColumn.
* //users/entities/user.entity
* @OneToOne(() => Customer, (customer) => customer.user, { nullable: true })
* @JoinColumn({ name: 'customer\_id' })
* customer: Customer;
* En una relación ManyToOne se hace la modificación en quien lleve este decorator.
  + Se agrega el decorator @JoinColumn
* //products/entities/product.entity.ts
* @ManyToOne(() => Brand, (brand) => brand.products)
* @JoinColumn({ name: 'brand\_id' })
* brand: Brand;
* En las tablas muchos a muchos que es manejada por TypeORM se modifica en la entidad que tenga el decorator @JoinTable
  + Recuerda que en este caso se crea una tabla ternaria.
* //products/entities/product.entity.ts
* @ManyToMany(() => Category, (category) => category.products)
* @JoinTable({
* name: 'products\_categories', //nombre de la tabla que tambien puede ser products\_has\_categories
* joinColumn: {
* name: 'product\_id', // Relación con la entidad donde estas situado.
* },
* inverseJoinColumn: {
* name: 'category\_id', // Relación con la otra entidad.
* },
* })
* categories: Category[];
* Correr migraciones.
  + Debes ser precavido los nombres deben ser ideado desde el diseño.
  + Cuando se cambia el nombre de una tabla, crea una nueva tabla no migra los datos.
  + Se puede perder datos. Evitar usarlo luego que la BD este en producción.
  + Cuando eliminas toda la BD para volverla crear, sin estar en producción, tienes la opción de eliminar las migraciones anteriores, y crear una nueva como estructura inicial.
  + En caso de vaciar y volver a crear las tablas en la BD ejecuta
* npm run migrations:drop
* npm run migrations:generate -- init
* npm run migrations:run



Nicolas Mendoza

[hace un mes](https://platzi.com/comentario/2574985/)

1 - En cuanto a migracioens enrtonces lo mejor es no usarlas cuando ya tengo un proyecto en produccion?  
2 - Si cambio el nombre e un tabla entonces borra todos los registro de dicha tabla?



Ricardo Tribaldos Hernández

[hace 5 meses](https://platzi.com/comentario/3138795/)

Un tip para no tener que hacer drop: Si ejecutan typeorm migration:create -n <nombre-del-refactor>, TypeORM les generará una plantilla de archivo de migración en blanco donde podrían agregar sus Querys ‘a mano’ como este:

import {MigrationInterface, QueryRunner} from "typeorm";

export class PostRefactoringTIMESTAMP implements MigrationInterface {

async up(queryRunner: QueryRunner): Promise<void> {

}

async down(queryRunner: QueryRunner): Promise<void> {

}

}

Dentro del método up colocan los querys necesarios para hacer los cambios y dentro de down los querys para revertir los cambios en caso de que sea necesario. Por ejemplo:

import {MigrationInterface, QueryRunner} from "typeorm";

export class PostRefactoringTIMESTAMP implements MigrationInterface {

async up(queryRunner: QueryRunner): Promise<void> {

await queryRunner.query(`ALTER TABLE "post" RENAME COLUMN "title" TO "name"`);

}

async down(queryRunner: QueryRunner): Promise<void> {

await queryRunner.query(`ALTER TABLE "post" RENAME COLUMN "name" TO "title"`); // reverts things made in "up" method

}

}



Luis Rangel Castro

[hace 3 meses](https://platzi.com/comentario/3290750/)

creo que el nombre correcto es: created\_at y updated\_at



Carlos Alejandro Hernández Mejía

[hace 3 meses](https://platzi.com/comentario/3368008/)

En TypeORM pueden definir una estrategia para el naming implementando la interfaz **NamingStrategyInterface** o bien, instalando esta librería que ya lo implementa:  
[https://github.com/tonivj5/typeorm-naming-strategies](https://github.com/tonivj5/typeorm-naming-strategies" \t "_blank)



Jordan Salazar

[hace 6 meses](https://platzi.com/comentario/2983161/)

Esta clase es muy importante, siempre tener en cuenta las buenas prácticas y el buen naming de los campos en la DB.



Renato Renzo Luna Herrera

[hace 4 meses](https://platzi.com/comentario/3225688/)

shait, este tema debimos verlo al inicio del curso, incluso cuando estábamos aprendiendo fundamentos de NestJs. -.-



ZANONI ALFREDO SALAS TOBON

[hace 5 meses](https://platzi.com/comentario/3129547/)

Name of product.entity.ts

@ManyToOne(() => Brand, (brand) => brand.products, { nullable: true })

@JoinColumn({ name: 'brand\_id' })

brand: Brand

@ManyToMany(() => Category, (categories) => categories.products, { nullable: true })

@JoinTable({

name: "product\_categories",

joinColumn: {

name: "pk\_product",

referencedColumnName: "id"

},

inverseJoinColumn: {

name: "pk\_category",

referencedColumnName: "id"

}

})

categories: Category[];

**SERIALIZACIÓN**

## erializar

Esto sirve para transformar la información antes de retórnala

// src/main.ts

// Admitimos serialiacion

app.useGlobalInterceptors(new ClassSerializerInterceptor(app.get(Reflector)));

#### Excluir un campo

Solo debemos agregarle el decorador @Exclude en el campo que queremos excluir

// src\users\entities\order-item.entity.ts

import { Exclude } from "class-transformer";

...

@Exclude()

@CreateDateColumn({

type: 'timestamptz',

default: () => 'CURRENT\_TIMESTAMP',

})

createAt: Date;

#### Transformar la informacion

// src\users\entities\order.entity.ts

@Expose()

get products() {

if (this.items) {

return this.items

.filter((item) => !!item)

.map((item) => ({

...item.product,

quantity: item.quantity,

itemId: item.id,

}));

}

return [];

}

@Expose()

get total() {

if (this.items) {

return this.items

.filter((item) => !!item)

.reduce((total, item) => {

const totalItem = item.product.price \* item.quantity;

return total + totalItem;

}, 0);

}

return 0;

}



johnatan ramos

[hace 6 meses](https://platzi.com/comentario/3066883/)

You are the best !!! 😉



José Mauricio Lemús Rodezno

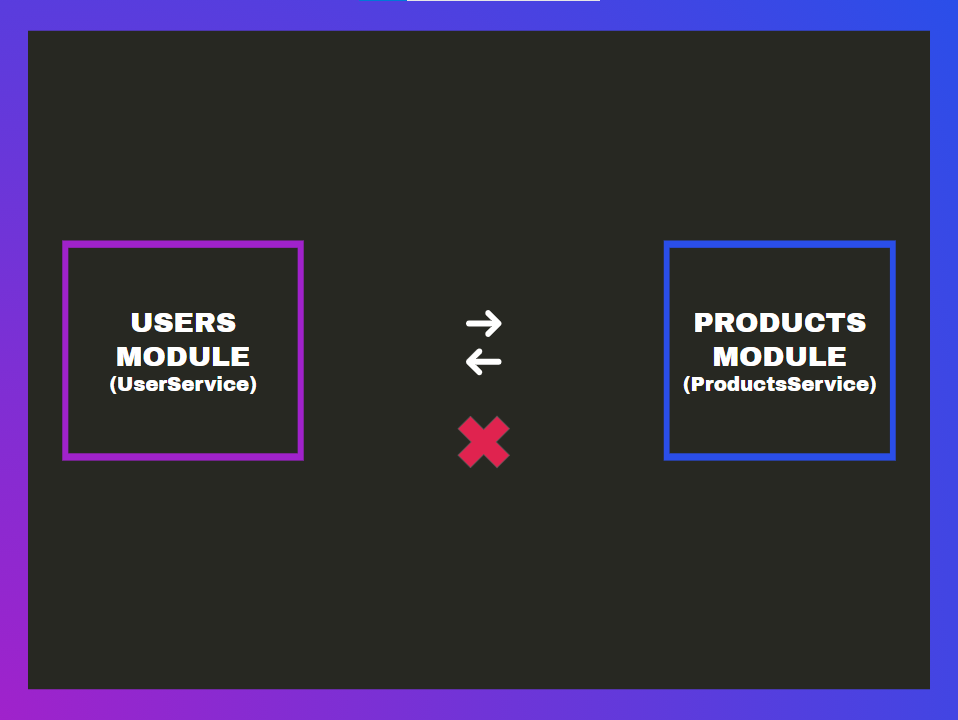
[hace 2 meses](https://platzi.com/comentario/3413997/)

Les dejo el enlace a la documentación sobre este tema: <https://docs.nestjs.com/techniques/serialization>

**Cómo solucionar una referencia circular entre módulos**

NestJS Realmente es un framework potente que nos permite tener una arquitectura sólida y escalable. Pero al conectarnos con bases de datos podemos llegar a enfrentarnos a una **referencia circular**, problema que debemos evitar y/o aprender a solucionar.

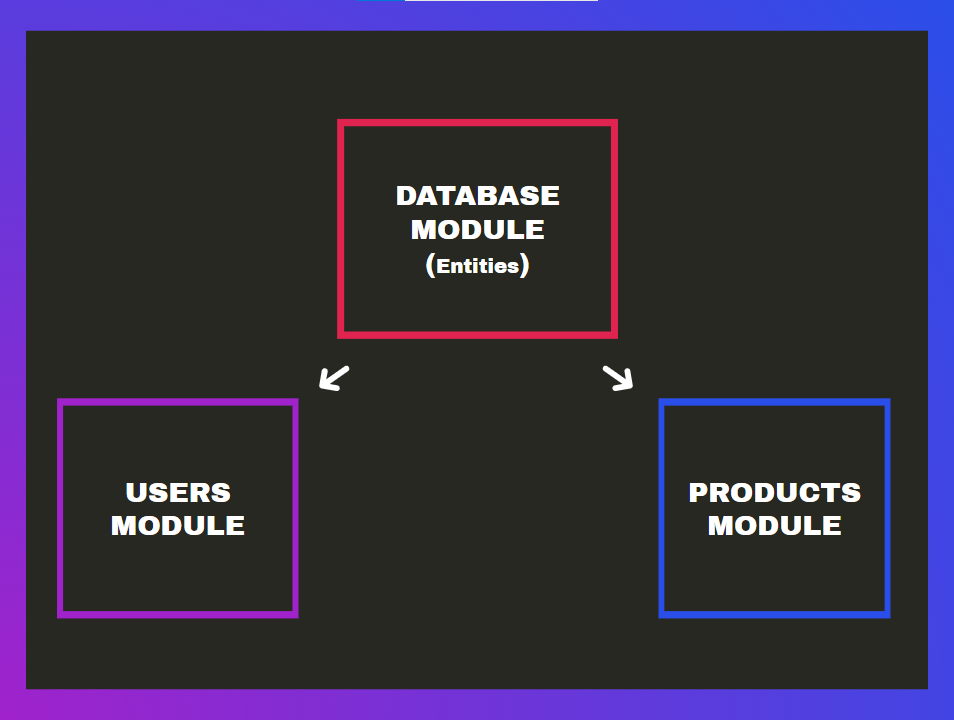
En nuestro proyecto tenemos a *UsersModule* que importa a *ProductsModule* con el fin usar al *ProductsService*. Pero ¿qué pasa si \_ProductsModule\_ necesita hacer una consulta hacia *users*? Recuerda que dentro de *UsersModule* está el *UserService* y la entidad de Users. Si importamos a UserModule dentro de ProductsModule, tendríamos un problema de **referencia circular entre módulos**.



Existen dos maneras de solucionarlo si llegas a tener este caso.

## Solución 1: entidades en un Global Module

Una de las soluciones es poner todas las entidades de tu proyecto en el DatabaseModule de manera global haciendo que cada uno de los demás módulos pueda usar estas entidades sin tener problemas de referencia circular.



De esta manera cada módulo puede usar el Repository Pattern y hacer la consulta que necesite de una entidad, puedes ver un ejemplo de esta solución en la [rama 21](https://github.com/platzi/nestjs-typeorm-datos/tree/21-step) y detalle de [cambios](https://github.com/platzi/nestjs-typeorm-datos/commit/e148130525c0f415389f6089186990524496fdf7) 😎

## Solución 2:  Referencia directa

Una de las formas que tiene NestJS para resolver la referencia circular es tener una referencia directa por ejemplo, si AService y BService dependen el uno del otro, ambos lados de la relación pueden usar @Inject () y la utilidad forwardRef () para resolver la dependencia circular, ejemplo:

@Injectable()

export class AService {

constructor(

@Inject(forwardRef(() => BService ))

private service: BService ,

) {}

}

De la misma manera en el otro servicio.

@Injectable()

export class BService {

constructor(

@Inject(forwardRef(() => AService ))

private service: AService ,

) {}

}

También puedes aplicar lo mismo entre módulos

@Module({

imports: [forwardRef(() => AModule)],

})

export class BModule{}

@Module({

imports: [forwardRef(() => BModule)],

})

export class AModule{}

¿Te has enfrentado a un problema de referencia circular? Compártenos en los comentarios de qué manera lo resolviste.