05NEST MySQL (CONTINUACiÓN API)

Módulo cliente

- Lo suyo es tener la carpeta modules y ahi crear todos los módulos
- Si no crealo directamente en la raíz (src)

> nest g res client

- En el controller le añado "api/v1/client" a la ruta principal

---

Client.dto

- Primero haremos algo básico, luego lo complementaremos

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


export class CreateClientDto {

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

    @IsNotEmpty()
    @IsString()
    name!: string

    @IsNotEmpty()
    @IsEmail()
    email!: string

}
```

- Creo la entidad. Le añado el decorador Entity de typeorm
- Hago el email único

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

@Entity()
export class Client {

    @PrimaryGeneratedColumn()
    id: number
```

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

    @Column({
        type: String,
        nullable: false,
        unique: true,
        length: 30})
    email: string
}
```

- Añado la entity Client en app.module

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

- Ya debería tener la tabla Client en la db

Creando address.dto

- La idea es generar una relación de uno a uno con la entidad address

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

export class AddressDto{

    @IsOptional()
    @IsNumber()
    @IsPositive()
```

```
    id?: number

    @IsNotEmpty()
    @IsString()
    country!: string

    @IsNotEmpty()
    @IsString()
    province!: string

    @IsNotEmpty()
    @IsString()
    city!: string

    @IsNotEmpty()
    @IsString()
    street!: string
}
```

- Creo la entity

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

@Entity()
export class Address{

    @PrimaryGeneratedColumn()
    id: number

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

    @Column({
        type: String,
        nullable: false,
        length: 50
    })
    province: string

    @Column({
        type: String,
        nullable: false,
        length:40
    })
    city: string

    @Column({
        type: String,
```

```
        nullable: false,
        length: 60
    })
    street: string
}
```

- Coloco la entity en app.module

Relación OneToOne con typeORM

- Un usuario tiene una dirección y una dirección tiene un usuario
- Relación OneToOne
- En la entity cliente voy a tener una columna nueva address con una relación one to one
- En este caso la relación no es bidireccional, porque no me interesa un endpoint para buscar a travñes de la dirección
- client.entity

```
@Entity()
export class Client {

    @PrimaryGeneratedColumn()
    id: number

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

    @Column({
        type: String,
        nullable: false,
        unique: true,
        length: 30})
    email: string

    @OneToOne(()=> Address)
    @JoinColumn()
    address: Address
}
```

- Le añado la propiedad al dto

```
import { IsEmail, IsNotEmpty, IsNumber, IsOptional, IsPositive, IsString } from
"class-validator";
import { Address } from "../entities/address.entity";
import { Type } from "class-transformer";
```

```
export class CreateClientDto {

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

    @IsNotEmpty()
    @IsString()
    name!: string

    @IsNotEmpty()
    @IsEmail()
    email!: string

    @Type(()=> Address)
    @IsNotEmpty()
    address: Address
}
```

- Debería aparecer addressId en la tabla Cliente

Crear cliente

- Primero, voy a client.module y en imports uso TypeOrmModule.forFeature e importo las dos entidades
- El forRoot es solo en el app.module

```
import { Module } from '@nestjs/common';
import { ClientService } from './client.service';
import { ClientController } from './client.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Client } from './entities/client.entity';
import { Address } from './entities/address.entity';

@Module({
  imports: [
    TypeOrmModule.forFeature([
      Client,
      Address
    ])
  ],
  controllers: [ClientController],
  providers: [ClientService]
})
export class ClientModule {}
```

- En el controller:

```
@Controller('api/v1/client')
export class ClientController {
  constructor(private readonly clientService: ClientService) {}

  @Post()
  createClient(@Body() createClientDto: CreateClientDto) {
    return this.clientService.createClient(createClientDto);
  }
}
```

- En el service, inyecto el repositorio

```
import { InjectRepository } from '@nestjs/typeorm';
import { Client } from './entities/client.entity';
import { Repository } from 'typeorm';

@Injectable()
export class ClientService {

  constructor(
    @InjectRepository(Client)
    private clientRepository: Repository<Client>
  ){}
}
```

- Voy al método createClient, primero hago la inserción sencilla luego aplico las validaciones

```
createClient(createClientDto: CreateClientDto) {
return this.clientRepository.save(createClientDto)
}
```

- Uso Postman o Thunderclient para crear el usuario con el método POST

```
{
  "name": "Pedro",
  "email": "pedro@gmail.com",
  "address":{
    "country": "Spain",
    "province": "Barcelona",
    "city": "Barcelona",
    "street": "Elm street"
  }
}
```

- Esto da error, porque este address no existe. Se tiene que crear para crear un cliente

> ERROR ExceptionsHandler Cannot perform update query because update values are not defined. Call qb.set(...) method to specify updated values.

- Si tuviera este address creado con su id no habría problema
- Para que lo haga automáticamente debo añadir la opción cascade a la client.entity

```
@OneToOne(()=> Address, {cascade: ['insert']})
@JoinColumn()
address: Address
```

- Esto hace que cuando envíe un address lo inserte primero, y luego cree el cliente
- Esto complica a la hora de hacer el borrado (luego se verá)
- Ahora si lo inserta

Validar si existe o no el cliente

- No solo puedo hacerlo con el email, también con el email pues es único
- Creo el método findClient en el servicio. Uso findOne y en la condición coloco un arreglo para que una de las dos se cumpla, o el id o el email
- No uso async await porque lo voy a usar dentro de otros métodos. Si lo usara solo si usaría async await

```
findClient(client: CreateClientDto){
  return this.clientRepository.findOne({
    where:[
      {id: client.id},
      {email: client.email}
    ]
  })
}
```

- En el createClient del servicio:

```
async createClient(cliente: CreateClientDto) {
  const clientExists = await this.findClient(cliente)
  if(clientExists){
    if(cliente.id){
      throw new BadRequestException(`El cliente con id ${cliente.id} ya existe` )
    }else{
      throw new BadRequestException(`El cliente con el email ${cliente.email} ya
existe`)
    }
  }
  return this.clientRepository.save(cliente)
}
```

- La dirección puede llevar id o no. Hago la validación por el id o por el contenido de la address
- Si la address ya existe lanzó un *ConflictException*

```
async createClient(client: CreateClientDto) {
    const clientExists = await this.findClient(client)
    if(clientExists){
      if(client.id){
        throw new BadRequestException(`El cliente con id ${client.id} ya existe` )
      }else{
        throw new BadRequestException(`El cliente con el email ${client.email} ya
existe`)
      }
    }

    let addressExists: Address = null;

    if(client.address.id){
      addressExists = await this.addressRepository.findOne({
        where: {
          id: clientExists.address.id
        }
      })
    }else{
      addressExists = await this.addressRepository.findOne({
        where:{
          country: client.address.country,
          province: client.address.province,
          city: client.address.city,
          street: client.address.street
        }
      })
    }

    if(addressExists){
      throw new ConflictException("Esta dirección ya está registrada")
    }
    return this.clientRepository.save(client)
  }
```

Obtener todos los clientes

- En el controller

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

- En el service

```
async getClients() {
  return await this.clientRepository.find();
}
```

- De esta manera no aparece la address en el resultado
- Para ello tengo que añadir la propiedad eager en la entity
- client.entity.ts

```
@OneToOne(()=> Address, {cascade: ['insert'], eager: true})
@JoinColumn()
address: Address
```

- Si fuera bidireccional el eager posiblemente no sirva

Obtener cliente por id

- En el controller:

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

- En el service debo hacer las validaciones pertinentes

```
async getClient(id: number) {
  const clientExists= await this.clientRepository.findOne({
    where: {id}
  })

  if(!clientExists){
    throw new BadRequestException("El cliente no existe")
  }

  return clientExists;
}
```

Actualizar un cliente

- Al tener un cliente y una dirección va ha haber que hacer algunas validaciones

- Vamos a hacerlo simple
- En el controller

```
@Put()
updateClient(@Body() updateClientDto: UpdateClientDto) {
  return this.clientService.updateClient(updateClientDto);
}
```

- En el service, si no viene el id creo el cliente

```
updateClient(client: UpdateClientDto) {
  if(!client.id){
    return this.createClient(client as CreateClientDto)
  }
  return this.clientRepository.save(client)
}
```

- Tengo que asegurarme de que si cambio de email, ese email no exista ya
- Creo un método específico para buscar por email

```
findClientByEmail(email: string){
  return this.clientRepository.findOne({
    where: {email}
  })
}
```

- Vuelvo al update. Creo con let el *clientExists* porque después voy a buscarlo por el id
- Valido que si hay cambio de email, este email no exista ya

```
  async updateClient(client: UpdateClientDto) {
    if(!client.id){
      return this.createClient(client as CreateClientDto)
    }

    let clientExists = await this.findClientByEmail(client.email)

    if(clientExists && clientExists.id != client.id){
      throw new ConflictException(`El cliente con el email ${client.email} ya
existe`)
    }

    return this.clientRepository.save(client)
  }
```

Validando la dirección al actualizar cuando no tiene id

- Cuando tengamos el id vamos a hacer varias cosas
- Por ahora vamos a copiar la validación de la dirección y la copio en el método de update
- Si actualizo la dirección, la anterior se va a quedar colgada. Para eso creo la variable *deletedAddress*

```
async updateClient(client: UpdateClientDto) {
    if(!client.id){
      return this.createClient(client as CreateClientDto)
    }

    let clientExists = await this.findClientByEmail(client.email)

    if(clientExists && clientExists.id != client.id){
      throw new ConflictException(`El cliente con el email ${client.email} ya
existe`)
    }

    let addressExists: Address = null;

    if(client.address.id){
      addressExists = await this.addressRepository.findOne({
        where: {
          id: clientExists.address.id
        }
      })
    }else{
      addressExists = await this.addressRepository.findOne({
        where:{
          country: client.address.country,
          province: client.address.province,
          city: client.address.city,
          street: client.address.street
        }
      })
    }

    let deletedAddress = false

    if(addressExists){
      throw new ConflictException("La dirección ya existe")
    }else{
      deletedAddress= true
    }

    return this.clientRepository.save(client)
  }
```

- Luego se terminará de jugar con el deletedAddress

Validando la dirección al actualizar cuando tiene el id

- Cuando haya un id debo comprobar si existe. En el caso de que el id coincida con el cliente no debería haber problema

```
 async updateClient(client: UpdateClientDto) {
    if(!client.id){
      return this.createClient(client as CreateClientDto)
    }

    let clientExists = await this.findClientByEmail(client.email)

    if(clientExists && clientExists.id != client.id){
      throw new ConflictException(`El cliente con el email ${client.email} ya
existe`)
    }

    clientExists = await this.getClient(client.id) //busco el cliente por id

    let addressExists: Address = null;
    let deletedAddress = false
    if(client.address.id){
      addressExists = await this.addressRepository.findOne({
        where: {
          id: clientExists.address.id
        }
      })


    }else{
      addressExists = await this.addressRepository.findOne({
        where:{
          country: client.address.country,
          province: client.address.province,
          city: client.address.city,
          street: client.address.street
        }
      })

      if(addressExists){
        throw new ConflictException("La dirección ya existe")
      }else{
        deletedAddress = true
      }
    }



    if(addressExists && addressExists.id != clientExists.address.id){
      throw new ConflictException("La dirección ya existe")
    }else if(JSON.stringify(addressExists) != JSON.stringify(client.address) ){
      //si la dirección que me pasa el cliente (client.address) es distinta a la
```

```
dirección que existe (clientExists)
      addressExists = await this.addressRepository.findOne({
        where:{
          country: client.address.country,
          province: client.address.province,
          city: client.address.city,
          street: client.address.street
        } })

        if(addressExists){
          throw new ConflictException("La dirección ya existe")
        }else{
          deletedAddress = true
        }
    }

    return this.clientRepository.save(client)
  }
```

- Debo decirle en el entity que además del insert, el update también tiene que hacerlo en cascade

```
@OneToOne(()=> Address, {cascade: ['insert', 'update'], eager: true})
  @JoinColumn()
  address: Address
```

Borrando la dirección desreferenciada

- Cuando tenga *deletedAddress* a true vamos a borrar la dirección desreferenciada

```
if(deletedAddress){     //clientExists es el que cogemos de la DB, el que tiene su
id original con la address desreferenciada
      await this.addressRepository.delete({id: clientExists.address.id})
    }
```

- Si hago simplemente esto me dará error porque me dirá que tiene una referencia, etc
- Primero debo actualizar y luego borrarlo, porque de esta manera se queda "sola" (desreferenciada)
- Entonces el clientRepository.save del cliente para el update debo colocarlo antes de este if
- Retorno el cliente actualizado

```
{   {...}
    if(addressExists && addressExists.id != clientExists.address.id){
      throw new ConflictException("La dirección ya existe")
    }else if(JSON.stringify(addressExists) != JSON.stringify(client.address) ){
      //si la dirección que me pasa el cliente es distinta a la dirección que
existe
      addressExists = await this.addressRepository.findOne({
```

```
        where:{
          country: client.address.country,
          province: client.address.province,
          city: client.address.city,
          street: client.address.street
        } })

        if(addressExists){
          throw new ConflictException("La dirección ya existe")
        }else{
          deletedAddress = true
        }
    }

    const updatedClient= await this.clientRepository.save(client)

    if(deletedAddress){     //clientExists es el que cogemos de la DB, el que
tiene su id original con la address desreferenciada
      await this.addressRepository.delete({id: clientExists.address.id})
    }

    return updatedClient;
}
```

- Entonces, lo que sucede es que si yo actualizo una dirección de un cliente, le coloco otro id, la dirección anterior se va a quedar colgada
- De esta manera la borramos

Eliminar un cliente

```
async remove(id: number) {
  const clientExists = await this.getClient(id) //este método creado por mi ya
contempla la excepción

  const rows = await this.clientRepository.delete({id}) //devuelve un DeleteResult

  return rows.affected === 1; // si devuelve true es que se ha hecho bien

}
```

- De esta manera me borra el cliente pero no la dirección asociada a este.
- Añadir 'remove' a la relacion **@OneToOne** para hacerlo en cascada no va a funcionar (es un problema de typeORM)
- Lo haremos manualmente
- Lo que debería pasar es que si borro el cliente con id 1 con la address de id 3 (por ejemplo), ambos se deberían borrar
- Lo haremos a la vieja usanza. Si rows == 1 borramos la address como hicimos con la anterior

```
async remove(id: number) {
    const clientExists = await this.getClient(id) //este método creado por mi ya
contempla la excepción

    const rows = await this.clientRepository.delete({id}) //devuelve un
DeleteResult

    if (rows.affected == 1){
      await this.addressRepository.delete({id: clientExists.address.id})
    }

    return
  }
```

Creando el módulo Order

> nest g res order

- Usaremos un UUID
- createdAt y updatedAt serán un tipo de dato fecha de typeORM que se generan automáticamente
- con **@IsUUID** typeORM genera el id automáticamente
- uso **@Type** para convertirlo a fecha

```
import { Type } from "class-transformer";
import { IsDate, IsOptional, IsUUID } from "class-validator";

export class CreateOrderDto {

    @IsOptional()
    @IsUUID()
    id?: string

    @IsOptional()
    @IsDate()
    @Type(()=> Date)
    createAt?: Date;

    @IsOptional()
    @IsDate()
    @Type(()=> Date)
    updateAt?: Date;

    @IsOptional()
    @IsDate()
    @Type(()=> Date)
    confirmAt?: Date;
}
```

Creando order.entity

- Pongo la id como string con el decorador **@PrimaryGeneratedColumn** con el tipo uuid
- Con **@CreateDateColumn** al hacer el save guardará la fecha automáticamente

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

@Entity()
export class Order {

    @PrimaryGeneratedColumn('uuid')
    id?: string;

    @CreateDateColumn()
    createdAt: Date;

    @UpdateDateColumn()
    updatedAt: Date;

    @Column({type: Date, nullable: true})
    cofirmAt: Date;

}
```

- Las relaciones se establecerán más adelante (ManyToMany, etc)
- Importamos la entity *Order* en el módulo con typeOrmModule.forFeature

```
import { Module } from '@nestjs/common';
import { OrderService } from './order.service';
import { OrderController } from './order.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Order } from './entities/order.entity';

@Module({
  imports:[
    TypeOrmModule.forFeature([
      Order
    ])
  ],
  controllers: [OrderController],
  providers: [OrderService]
})
export class OrderModule {}
```

- Recuerda también añadirla en app.module!

```
@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'root',
      database: 'shop',
      entities: [Product, Client, Address, Order],
      synchronize: true,
    }),
    ProductModule,
    ClientModule,
    OrderModule,
  ],
})
```

Relación ManyToOne/OneToMany

- Están relacionadas, obviamente
- A tiene multiples instancias de B pero B solo contiene una instancia
- No puede existir el uno sin el otro
- Order sería **@ManyToOne(()=> Usuario)**
- Usuario sería **@OneToMany(()=> Order)**
- En order.entity

```
import { Client } from "src/client/entities/client.entity";
import { Column, CreateDateColumn, Entity, ManyToOne, PrimaryGeneratedColumn,
UpdateDateColumn } from "typeorm";

@Entity()
export class Order {

    @PrimaryGeneratedColumn('uuid')
    id?: string;

    @CreateDateColumn()
    createdAt: Date;

    @UpdateDateColumn()
    updatedAt: Date;

    @Column({type: Date, nullable: true})
    cofirmAt: Date;

    @ManyToOne(()=> Client, client=> client.orders) //client.orders todavía no
existe
    client!: Client; //marcado con ! como obligatorio
```

```
}
```

- En la entity *Client*

```
import { Column, Entity, JoinColumn, OneToMany, OneToOne, PrimaryGeneratedColumn }
from "typeorm";
import { AddressDto } from "../dto/address.dto";
import { Address } from "./address.entity";
import { Order } from "src/order/entities/order.entity";

@Entity()
export class Client {

    @PrimaryGeneratedColumn()
    id: number

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

    @Column({
        type: String,
        nullable: false,
        unique: true,
        length: 30})
    email: string

    @OneToOne(()=> Address, {cascade: ['insert', 'update'], eager: true})
    @JoinColumn()
    address: Address

    @OneToMany(()=> Order, order=> order.client)
    orders?: Order[];
}
```

Relación ManyToMany

- Relación donde A contiene multiples instancias de A y B contiene múltiples instancias de A
- Una orden puede tener varios productos y un producto puede estar en varias órdenes
- **@JoinTable** es requerido

```
import { Client } from "src/client/entities/client.entity";
import { Product } from "src/product/entities/product.entity";
import { Column, CreateDateColumn, Entity, JoinTable, ManyToMany, ManyToOne,
PrimaryGeneratedColumn, UpdateDateColumn } from "typeorm";
```

```
@Entity()
export class Order {

    @PrimaryGeneratedColumn('uuid')
    id?: string;

    @CreateDateColumn()
    createdAt: Date;

    @UpdateDateColumn()
    updatedAt: Date;

    @Column({type: Date, nullable: true})
    cofirmAt: Date;

    @ManyToOne(()=> Client, client=> client.orders)
    client!: Client;

    @ManyToMany(()=> Product)
    @JoinTable({name: 'order_products'}) //puedo especificarle el nombre del campo
en la tabla
    products: Product[]

}
```

- Que tendrá esta tabla? *orderId* y *productId*

Order.dto

- Hemos añadido el cliente y los productos

```
import { Type } from "class-transformer";
import { ArrayNotEmpty, IsArray, IsDate, IsNotEmpty, IsOptional, IsUUID } from
"class-validator";
import { CreateClientDto } from "src/client/dto/create-client.dto";
import { CreateProductDto } from "src/product/dto/create-product.dto";

export class CreateOrderDto {

    @IsOptional()
    @IsUUID()
    id?: string

    @IsOptional()
    @IsDate()
    @Type(()=> Date)
    createAt?: Date;

    @IsOptional()
```

```
    @IsDate()
    @Type(()=> Date)
    updateAt?: Date;

    @IsOptional()
    @IsDate()
    @Type(()=> Date)
    confirmAt?: Date;

    @IsNotEmpty()
    @Type(()=> CreateClientDto)
    client!: CreateClientDto

    @IsNotEmpty()
    @IsArray()
    @ArrayNotEmpty()
    @Type(()=> CreateProductDto)
    products!: CreateProductDto[]
}
```

Importando el módulo de Cliente y Producto

- Como en order.entity tengo cosas de clientes y de productos (y necesito los servicios) debo exportar e importar los módulos
- En el módulo de order importo los módulos

```
@Module({
  imports:[
    TypeOrmModule.forFeature([
      Order
    ]),
    ClientModule,
    ProductModule
  ],
  controllers: [OrderController],
  providers: [OrderService]
})
export class OrderModule {}
```

- Cómo lo que me interesa es el servicio, es lo que exporto en el modulo de cliente y producto

```
@Module({
  imports: [
    TypeOrmModule.forFeature([
      Client,
      Address
    ])
  ],
```

```
  controllers: [ClientController],
  providers: [ClientService],
  exports:[ClientService]
})
export class ClientModule {}
```

- Hago lo mismo en product.module
- En el *OrderService* inyecto los dos servicios
- Inyecto también la entidad para trabajar con ella

```
export class OrderService {

  constructor(

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

    private clientService: ClientService,
    private productService: ProductService,

  ){}
}
```

Creando ordenes

- Vamos a comprobar que un producto no esté borrado y que el cliente exista
- getClient usa el id para encontrar el cliente
- Para el arreglo de Productos vamos a usar un for of

```
  async create(order: CreateOrderDto) {

    const client = await this.clientService.getClient(order.client.id)
    if(!client){
      throw new NotFoundException("El cliente no existe")
    }

    for(let p of order.products){
      const product = await this.productService.findProduct(p.id)
      if(!product){
        throw new NotFoundException("El producto no existe")
      }else if(product.deleted){
        throw new BadRequestException(`No hay existencias del producto con id
${p.id}`)
      }
    }

    return this.orderRepository.save(order)
```

```
  }
```

- Voy a POSTMAN o THUNDERCLIENT a hacer la petición POST. En el body

```
{
  "client": {
    "id": 1
  },
  "products":[
    {
      "id": 1
    },
    {
      "id": 2
    }
    ]
}
```

- Tengo la orden y la orde_products donde aparecen los dos productos con la misma ordenId

---

Obtener orden por id

```
async getOrderById(id: string) {
  const order= await this.orderRepository.findOne({
    where: {id}
  })
  if(!order){
    throw new BadRequestException("La orden no existe")
  }
  return order
}
```

- Hago una petición GET a la url con el id en POSTMAN/THUNDERCLIENT

```
http://localhost:3000/api/v1/order/7b288708-6c78-4df0-a438-cfa5f4e0fc9e
```

- Me devuelve esto:

```
{
  "id": "7b288708-6c78-4df0-a438-cfa5f4e0fc9e",
  "createdAt": "2023-11-27T16:15:16.412Z",
  "updatedAt": "2023-11-27T16:15:16.412Z",
  "cofirmAt": null
}
```

- En el resultado no aparece ni los productos ni el cliente
- Para ello tengo que usar **eager** en la entity
- order.entity

```
@Entity()
export class Order {

    @PrimaryGeneratedColumn('uuid')
    id?: string;

    @CreateDateColumn()
    createdAt: Date;

    @UpdateDateColumn()
    updatedAt: Date;

    @Column({type: Date, nullable: true})
    confirmAt: Date;

    @ManyToOne(()=> Client, client=> client.orders, {eager: true})
    client!: Client;

    @ManyToMany(()=> Product, {eager: true})
    @JoinTable({name: 'order_products'})
    products: Product[]

}
```

- Ahora me devuelve la información completa

```
{
  "id": "7b288708-6c78-4df0-a438-cfa5f4e0fc9e",
  "createdAt": "2023-11-27T16:15:16.412Z",
  "updatedAt": "2023-11-27T16:15:16.412Z",
  "confirmAt": null,
  "client": {
    "id": 3,
    "name": "Pedro",
    "email": "pedro@gmail.com",
    "address": {
      "id": 5,
      "country": "Spain",
      "province": "Barcelona",
      "city": "Barcelona",
      "street": "Elm street"
    }
  },
  "products": [
    {
      "id": 1,
```

```
      "name": "Producto1",
      "stock": 10,
      "price": 300,
      "deleted": false
    },
    {
      "id": 2,
      "name": "Producto2",
      "stock": 330,
      "price": 300,
      "deleted": false
    }
  ]
}
```

Obtener órdenes pendientes

- Creo un endpoint GET getPendingOrders con "/pending". Lo coloco encima del GET con ":id" para que no se confunda y crea que pending es el id. Uso la función isNull (podría usar null, también funcionaría)

```
async getPendingOrders(){
  return await this.orderRepository.find({
    where: {
      confirmAt: IsNull()
    }
  })
}
```

- Hacemos lo mismo con las confirmadas. Creo el endpoint "confirmadas" en el controlador y uso **Not** de typeorm en el servicio

```
async getConfirmedOrders(){
  return await this.orderRepository.find({
    where:{
      confirmAt: Not(IsNull())
    }
  })
}
```

Filtrando órdenes confirmadas

- Filtraremos por la fecha de confirmAt
- En el controlador añadimos los **@Query**

```
@Get("/confirmed")
  getConfirmedOrders(@Query("start") start: Date, @Query("end") end: Date){
    return this.orderService.getConfirmedOrders(start, end)
  }
```

- En el servicio uso getTime que me devuelve la fecha en número
- Uso la negación de isNaN para entrar en el if
- Utilizo OR porque puede ser que uno me devuelva true y otro false
- Uso 'DESC' de descendente para ordenar (de la más nueva a la más antigua)

```
async getConfirmedOrders(start:Date, end: Date){
    if(!isNaN(start.getTime()) || !isNaN(end.getTime())){
      console.log({start,end})

    }else{
      return await this.orderRepository.find({
        where:{
          confirmAt: Not(IsNull())
        },
        order:{
          confirmAt: 'DESC'
        }
      })
    }
```

- Cuando me entreguen unas fechas (y entre en el if) lo filtraré a través de un **queryBuilder**. Sirve para hacer queries
- Para consultas más extensas viene muy bien
- Le coloco de alias "order"
- Pongo order.client porque he llamado al queryBuilder "order". Si le hubiera puesto "o" pondría "o.client"
- Hago los joints correspondientes, ordeno por confirmAt
- Para filtrar por la fecha uso el queryBuilder y añado la condición con andWhere la función MoreThanOrEqual
- Uso LessThanOrEqual para el end
- Ejecuto el query con **getMany()**

```
 async getConfirmedOrders(start:Date, end: Date){
    if(!isNaN(start.getTime()) || !isNaN(end.getTime())){
      const query = this.orderRepository.createQueryBuilder("order")
                    .leftJoinAndSelect("order.client", "client")
                    .leftJoinAndSelect("order.products", "product")
                    .orderBy("order.confirmAt")
      if(!isNaN(start.getTime())){
          query.andWhere({confirmAt: MoreThanOrEqual(start)})
      }
```

```
      if(!isNaN(end.getTime())){
        query.andWhere({confirmAt: LessThanOrEqual(end)})
      }
      return await query.getMany()

    }else{
      return await this.orderRepository.find({
        where:{
          confirmAt: Not(IsNull())
        },
        order:{
          confirmAt: 'DESC'
        }
      })
    }
  }
```

- Puedo formatear la hora para que no de problemas

```
if(!isNaN(end.getTime())){
        end.setHours(24)
        end.setMinutes(59)
        end.setSeconds(59)
        query.andWhere({confirmAt: LessThanOrEqual(end)})
      }
```

Confirmando órdenes

- Usaremos PATCH (también se podría hacer con un PUT)
- En el controller

```
@Patch("/confirm/:id")
confirmOrder(@Param('id') id: string){
  return this.orderService.confirmOrder(id)
}
```

- En el service primero compruebo que la orden exista
- Que la orden no esté confirmada
- El update devuelve una promesa de tipo UpdateResult
- Lo guardo en una variable llamada rows.
- Si rows está afectada(igual a 1) es que ha hecho el update

```
async confirmOrder(id: string){
  const orderExists = await this.getOrderById(id)
```

```
    if(!orderExists){
      throw new NotFoundException("La orden no ha sido encontrada")
    }

    if(orderExists.confirmAt){
      throw new ConflictException("La orden ya está confirmada")
    }
    const rows: UpdateResult = await this.orderRepository.update(
      {id},
      {confirmAt: new Date()}
    )

    return rows.affected ==1
  }
```

Obtener órdenes de un cliente

- Haciendo los joints es como si tuviera las tres tablas, order, client y product
- Le puedo poner un where donde el client.id sea igual al idClient

```
async getOrdersByClient(idClient: number){
    return this.orderRepository.createQueryBuilder("order")
            .leftJoinAndSelect("order.client", "client")
            .leftJoinAndSelect("order.product", "product")
            .where("client.id = :idClient", {idClient})
            .orderBy("order.confirmAt")
            .getMany()

}
```