NEST Duro De Roer - APUNTES INICIALES

- Para generar un nuevo módulo, creo una nueva carpeta llamada modules
- Dentro de la carpeta genero el módulo

> nest g mo names

- Lo incluyo en el imports de app.module
- Para generar un controlador, no hace falta que esté en la misma carpeta names
- Pero si **lo llamo igual que el módulo**

> nest g co names

- En controllers de names.module aparece automáticamente NamesController
- Se acostumbra a poner en el decorador controller *ap/v1* **@Controller('api/v1/names')**
- Se usa **api/v1** porque si la app crece y se añade un nuevo controlador, poder seguir dando soporte al antiguo
- Para crear un servicio, estoy en la carpeta modules

> nest g s names

- Aparece automáticamente en providers de names.module NamesService
- En el servicio aparece el decorador **@Injectable()**
- Significa que puedo inyectar esta clase en el constructor (en este caso del controlador)

```
constructor( private readonly namesService: NamesService)
```

- Para este ejemplo no se usará db, por lo que creo un array de nombres

```
@Injectable()
export class NamesService{

    private _names: string[]

    constructor(){
        this._names = []
    }
}
```

- En el controlador, para obtener la data del body uso @Body
- Aquí usaría un dto en lugar del objeto data, donde tipo el name como un string

```
@Post()
createName(@Body() data:{name: string}){
```

```
    return this.namesService.createname(data.name)
}
```

- En el servicio le paso el name
- Uso el push porque no tenemos uns db y es solo un array
- El return true es solo como mensaje de confirmación de que se ha ejecutado el código

```
createName(name: string){
    this._names.push(name)
    console.log("Names: ", this._names)

    return true
}
```

- Tengo que validar si existe el nombre!
- Uso lowerCase para igualar, trim para eliminar los espacios. Solo elimina espacios delante y detrás, no enmedio
- En caso de que no exista que haga la inserción

```
createName(name: string){

    const nameFound = this._names.find(n => n.toLowerCase().trim() ===
name.toLowerCase().trim())

    if(!nameFound){
       this._names.push(name)
        console.log("Names: ", this._names)
        return true
    }else{
        return false
    }
}
```

- Para el GET es sencillo
- controller

```
@Get()
getNames(){
    return this.namesService.getNames()
}
```

- service

```
getNames(){
    return this._names
}
```

- El decorador @Query me sirve para filtrar elementos a través de la url
- Pongamos que quiero recoger el valor de start

> http://localhost:3000/api/v1/products?start=fer&end=ando

- Cuando sólo va a ser un parámetro puedo especificarlo en @Query()

```
@Get()
getNames(@Query('start') start: string){
    return this.namesService.getNames()
}
```

- Si no le paso ningún query tendrá el valor de undefined
- Cuando van a ser muchos lo puedo dejar **vacío** y tiparlo como query:any
- En el servicio lo marco cómo opcional, porque puede llegar o no
- La lógica es que si viene start lo devuelvo filtrado, si no lo devuelvo tal cual
- filter devuelve un array de algo segun una condición. Uso startsWith para comprobar si empieza por esa letra

```
getNames(start?: string){

    if(!start){
        return this._names
    }else{
        return this._names.filter(n =>
n.toLowerCase().trim().startsWith(start.toLowerCase()))
    }
}
```

- Para el update uso @Put, @Param para capturar el nombre a actualizar y el nuevo nombre
- controller

```
@Put('/:name/:newName')
updateName(@Param('name') name: string, @Param('newName') newName: string){
    return this.namesService.updateName(name, newName)
}
```

- Uso el mismo método find en el servicio para verificar si existe el nombre
- El newName no tiene que existir
- En lugar de find uso findIndex. Si devuelve -1 es que no existe

- service

```
updateName(name: string, newName: string){
    const indexNameFound = this._names.findIndex(n => n.toLowerCase().trim() ===
name.toLowerCase().trim())
    const indexNewNameFound = this._names.findIndex(n => n.toLowerCase().trim()
=== name.toLowerCase().trim())

    if(indexNameFound != -1 && newNameFound == -1){
        this._name[indexNameFound] = newname
        return true
    }else{
        return false
    }
}
```

- Para el delete tengo que extraer del parámetro el nombre y confirmar que el nombre exista
- controller

```
@Delete('/:name')
deleteName(@Param('name') name: string){
    return this.namesService.deleteName(name)
}
```

- Declaro una variable con el número de elementos del array antes de borrar
- Uso el filter para devolver un array con todos menos con el name que he extraido con @Param
- Si deletedBefore y deletedAfter son diferentes es que se ha borrado el elemento
- service

```
deleteName(name: string){
    const deletedBefore = this._names.length

    this._names = this._names.filter(n=> n.toLowerCase().trim() !==
name.toLowerCase().trim())

    const deletedAfter = this._names.length

    return deletedBefore != deletedAfter
}
```

- Limpiando todos los nombres
- Será con otro delete pero le voy a poner un nombre

```
@Delete('clear')
clearNames(){
```

```
    return this.namesService.clearNames()
}
```

- service

```
clearNames(){
    this._names = []
}
```

- El problema aquí es que está detectando el clear como un nombre por el otro delete
- Cuanto más **específico sea, debo colocarlo más arriba en el controlador**
- **Es importante el orden**
- Es decir, si le pongo mancuso, Nest dice: "ok, clear no es" y lo trata como un nombre
- Pero para eso el delete de clear **debe estar antes** del delete por nombre

---

02 NEST Duro de Roer - SWAGGER

- Instalación

> npm i @nestjs/swagger swagger-ui-express

- En el main añado

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

    const config = new DocumentBuilder()
        .setTitle('Names API')
        .setDescription('CRUD names')
        .setVersion('1.0')
        .build()
    const document = SwaggerModule.createDocument(app, config)

    SwaggerModule.setup('documentation', app, document)

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

-*DocumentBuilder* y *SwaggerModule* vienen de @nestjs/swagger

- No hace falta importar este modulo en app.module
- La documentación está en la url documentation (como puse en SwaggerModule.setup)

> http://localhost:3000/documentation

- Aparecen todos los endpoints

ApiTags

- Debajo de **@Controller()** coloco el decorador **@ApiTags('names')** de @nestjs/swagger
- Separaría cada módulo en la documentación con **@ApiTags('otro_modulo')**
- Si quito el .spec, .controller y .service del app (que no me sirven de nada) desaparece el default de la documentación y queda solo names

---

ApiOperation

- En el controller, debajo de **@Post** coloco **@ApiOperation**

```
@Post()
@ApiOperation({
    description: "Crea un nuevo usuario. Retorna true si se inserta correctamente"
})
```

- Hago lo mismo con el resto de endpoints

ApiParam

- Lo usaremos para documentar los parámetros del PUT y el DELETE por nombre

```
@Put()
@ApiParam({
    name: 'name',
    type: 'string',
    description: 'Nombre original'
})
@ApiParam({
    name: 'newName',
    type: 'string',
    description: 'Nombre nuevo'
})
```

ApiQuery

- Lo usaré para el query del getNames que era start
- Debo indicarle que no es obligatorio porque por defecto me marcará que lo es

```
@Get()
@ApiQuery({
    name: 'start',
    type: 'string',
    required: false,
    description: 'Nombres que empiecen por el query'
})
```

ApiBody

- Puedo añadir una descripción al **@Body** de createNames

```
@Post()
@ApiBody({
    description: 'Añadiendo un nombre',
    examples:{
      value:{
        name: 'Migue'
      }
    }
})
@ApiOperation({
    description: "Crea un nuevo usuario. Retorna true si se inserta correctamente"
})
```

- En examples podría poner **ejemplo1**: {values:{name:'Miguel}}

NEST Duro de Roer - Usuarios

- Seguimos la misma estrategia que en la lección anterior
- Creo la carpeta modules, dentro de la carpeta creo el módulo users con el CLI de Nest
- Creo también el servicio y el controlador
- En el controller coloco api/v1

Creando el dto (Data Transfer Object)

- Es una clase común

```
export class UserDto{

    id: number
    name: string
    email:string
```

```
    birthDate: Date
}
```

- Para validar uso class-validator y class-tarnsformer (lo instalo)

> npm i class-validator class-transformer

- Añado los decoradores
- El @Type es de class-transformer. Transforma la data
- Le voy a pasar un string con una data en el body. Quiero que la transforme a tipo Date

```
import {IsNumber, IsNotEmpty, IsString, IsEmail, IsDate} from 'class-validator'
import {Type} from 'class-transformer'

export class UserDto{

    @IsNumber()
    @IsNotEmpty()
    id: number

    @IsString()
    @IsNotEmpty()
    name: string

    @IsEmail()
    @IsNotEmpty()
    email:string

    @IsDate()
    @IsNotEmpty()
    @Type(()=> Date)
    birthDate: Date
}
```

- Para que haga las validaciones uso ValidationPipe en el main antes del app.listen

```
app.useGlobalPipes(
    new ValidationPipe({
        whitelist: true,            //si hay data que no está en el dto la
ignorará
        forbidNonWhitelisted: true, //muestra error si le mando data que no está
en el dto
        transform: true             //permite transformar la data
    })
)
```

Creando Usuarios

- Como no uso una db creo el array en el servicio
- service

```
@Injectable()
export class UsersService{
    private _users: userDto[]

    constructor(){
        this._users = []
    }
}
```

- En el controller está inyectado el servicio en el constructor
- controller

```
@Post()
createUser(@Body() user: UserDto){
    return this.usersService.createUser(user)
}
```

- Creo el servicio createUser
- service

```
createUser(user: UserDto){
    const userFound = this._users.find(u=> u.id === user.id)

    if(!userFound){
        this._user.push(user)
        return true
    }else{
        return false
    }
}
```

Obteniendo usuarios filtrados

- Vamos a filtrar los usuarios por la fecha de nacimiento
- Como puse el transform en true en el ValidationPipe, lo va a transformar a date si es un string con el que puede hacerlo
- Creo el GET en el controlador
- controller

```
@Get()
getUsers(@Query('start') start: Date, @Query('end') end: Date){
    return this.usersService.getUsers(start, end)
}
```

- En el service uso el isNaN para validar si la fecha es válida o no
- Si es una fecha correcta isNaN me va a devolver false
- Puedo negarlo con ! para evaluar lo contrario, es decir **SI SI ES UN NUMERO**
- Tendríamos los casos en el que me pasan las dos fechas, solo una de las dos o ninguna de las dos
- service

```
getUsers(start: Date, end: Date){

    if(!isNaN(start.getTime()) && !isNaN(end.getTime())){
        return this._users.filter(u => u.birthDate.getTime() >= start.getTime() &&
                                       u.birthDate.getTime() <= end.getTime())

    }else if(!isNaN(start.getTime()) && isNaN(end.getTime())){
            return this._users.filter(u => u.birthDate.getTime() >=
start.getTime())

    }else if(isNaN(start.getTime()) && !isNaN(end.getTime())){
            return this._users.filter(u => u.birthDate.getTime() <= end.getTime())
    }else{
        return this._users
    }
}
```

Actualizando Usuarios

- En el caso de que el id no exista no tiene que dar el fallo, tiene que crearse
- En el caso de que exista debe actualizarse
- Se suele hacer así
- Creo el endpoint en el controlador
- controller

```
@Put()
updateUser(@Body() user: UserDto){
    return this.usersService.updateUser(user)
}
```

- En el servicio uso createUser
- Si el usuario existe me va a devolver un falso, con lo que hay que actualizar

- En caso de que exista vamos a buscar el índice en el que se encuentra
- findIndex en lugar de devolver el objeto devuelve el index
- service

```
updateUser(user: userDto){

    const userAdded = this.createUser(user)

    if(!userAdded){
        const index = this._users.findIndex(u => u.id === user.id)

        this._users[index] = user
    }

    return true
}
```

Eliminando usuarios

- controller

```
@Delete('/:id')
deleteUser(@Param('id') id: number){
    return this.usersService.deleteUser(+id)
}
```

- Voy al servicio
- Si index es distinto de -1 significa que existe el elemento. devuelve -1 cuando no lo encuentra
- splice, dado un indice borra n elementos
- service

```
deleteUser(id: number){
    const index = this._users.findIndex(u => u.id === id)

    if(index != -1){
        this._users.splice(index, 1)
        return true
    }
    return false      //si llega aquí es que no existe
}
```

Documentar Dto (ApiProperty)

```
import {IsNumber, IsNotEmpty, IsString, IsEmail, IsDate} from 'class-validator'
import {Type} from 'class-transformer'
import {ApiProperty} from '@nestjs/swagger'

export class UserDto{

    @ApiProperty({
        name: 'id',
        type: 'number',
        description: 'Identificador del usuario'
    })
    @IsNumber()
    @IsNotEmpty()
    id: number

    @ApiProperty({
        name: 'name',
        type: String,
        description: 'Nombre del usuario',
        required: true
    })
    @IsString()
    @IsNotEmpty()
    name: string

     @ApiProperty({
        name: 'email',
        type: String,
        description: 'Email del usuario',
        required: true
    })
    @IsEmail()
    @IsNotEmpty()
    email:string

    @ApiProperty({
        name: 'birthDate',
        type: Date,
        description: 'Fecha de nacimiento del usuario',
        required: true
    })
    @IsDate()
    @IsNotEmpty()
    @Type(()=> Date)
    birthDate: Date
}
```

- Pongo el POST del controlador como ejemplo de documentación

```
@Post()
@ApiOperation({
```

```
    description: 'Crea un usuario'
})
@ApiBody({
    description: 'Crea un usuario mediante un Dto. devuelve true si ha tenido
éxito',
    type: UserDto,
    examples:{
        value:{
            "id": 1,
            "name": "Migue",
            "email": "email@gmail.com",
            "birthDate": "1981-20-01"
        }
    }
})
```

---

04 NEST Duro de Roer - MySQL

- Creando un proyecto

> nest new mysql-project

> npm i class-validator class-transformer

> npm i @nestjs/swagger

> npm i @nestjs/typeorm typeorm mysql2

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

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

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

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

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

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

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

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

NOTA: ormconfig está deprecado. TypeORM ya no lo usa

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

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

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

---

Creando el módulo de Product

- Creo un CRUD completo para Product

> nest g res product

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

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

```
export class CreateProductDto {

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

    @IsString()
    @IsNotEmpty()
    name: string

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

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

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

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

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

@Entity()
export class Product {

    @PrimaryGeneratedColumn()
    id: number

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

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

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

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

```
    deleted: boolean
}
```

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

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

@Injectable()
export class ProductService {

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

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

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

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

Creando un producto

- Creo el endpoint en el controller

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

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

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

```
@Injectable()
export class ProductService {

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

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

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

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

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

Devolviendo un producto por su id

- Para obtener un producto por id en el controller

```
@Get(':id')
findOne(@Param('id') id: string) {
```

```
  return this.productService.findOne(+id);
}
```

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

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

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

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

Validación en createProduct

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

```
  async create(createProductDto: CreateProductDto) {

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

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

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

Devolviendo los productos

- En el controller

```
@Get()
findAll() {
```

```
  return this.productService.findAll();
}
```

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

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

Actualizando Productos

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

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

- En el servicio:

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

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

---

Soft delete

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

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

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

```
  async removeProduct(id: number) {

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

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

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

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

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

  }
```

Recuperando productos borrados

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

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

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

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

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

  return rows.affected == 1
}
```

Creando stock.dto

- stock.dto

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

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

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

Actualizando el stock de un producto

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

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

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

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

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

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

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

```
      {stock: s.stock}
    )

    return rows.affected == 1
  }
```

Incrementando el stock de un producto

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

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

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

```
@Injectable()
export class ProductService {

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

(...)
}
```

- Declro una variable stock = 0

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

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

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

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

```
    }

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

    )

    return rows.affected == 1
  }
```

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

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

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

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

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

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

  )

  return rows.affected == 1
}
```

Documentar los endpoints

- Con Swagger
- Recuerda la configuración del main

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

```
import { AppModule } from './app.module';

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

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

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

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

- Documentando createProduct

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

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

- Puedo usar el decorador @ApiResponse para documentar las respuestas

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

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

- Para documentar DTO

```
export class CreateProductDto {

    @ApiProperty({
        name: "id",
        required: false,
        description: "id del producto",
        type: Number
```

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

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()

}
```

---

01 Autenticación Nest

Creando el proyecto

> nest new authentication

- Intalo el class-validator y el class-transformer
- Tambien voy a necesitar el @nestjs/swagger y swagger-ui-express
- Instalo @nestjs/mongoose mongoose
- Borro app.controller y app.service ya que no me hacen falta. Los quito de app.module

- Configuro en el main el useGlobalPipes para las validaciones y configuro swagger

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe({transform:true}))

  const config = new DocumentBuilder()
  .setTitle('Authentication')
  .setDescription('API AUthenticaction')
  .setVersion('1.0')
  .addTag('auth')
  .build();
const document = SwaggerModule.createDocument(app, config);
SwaggerModule.setup('swagger', app, document);


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

Configuración

> npm i @nestjs/config dotenv

- Importo el ConfigModule en app.module

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


@Module({
  imports: [ConfigModule.forRoot({
    load: [],
    envFilePath: `./env/${process.env.NODE_ENV}.env`,
    isGlobal: true
  })],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

- Creo la carpeta configuration en src con el archivo configuration-mongo.ts (se pueden poner más de uno)

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

export default registerAs('mongo', ()=>({  //creo el objeto mongo

}))
```

- Lo cargo en load como ConfigurationMongo (adopta este nombre automáticamente al ser un objeto de mongo)

```
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import configurationMongo from './configuration/configuration-mongo';
require('dotenv').config();  //configuro dotenv

@Module({
  imports: [ConfigModule.forRoot({
    load: [configurationMongo],
    envFilePath: `./env/${process.env.NODE_ENV}.env`,
    isGlobal: true
  })],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

- Creo el archivo .env en la raiz del proyecto

```
NODE_ENV=development
```

- Creo la carpeta env fuera de src con development.env y production.env

---

Conexión MongoDB

- Creo la carpeta modules en src
- En src/modules creo un modulo y un servicio

> nest mo mongo-connection nest g s mongo-connection

- Exporto el servicio en mongo-connection.module.ts. Los servicios siempre son providers

```
import { Module } from '@nestjs/common';
import { MongoConnectionService } from './mongo-connection.service';
```

```
@Module({
  providers: [MongoConnectionService],
  exports:[
    MongoConnectionService
  ]
})
export class MongoConnectionModule {}
```

- En el servicio creo la propiedad dbConnection
- Hacemos un Singleton, así no lo vuelve a inyectar cuando lo metes en otro lado. Así solo crea una única conexión
- En el constructor llamo al método dbConnection
- En el método dbConnection introduzco todo lo necesario

```
import { Injectable } from '@nestjs/common';
import { Connection, createConnection } from 'mongoose';
import{ ConfigService} from '@nestjs/config'

@Injectable()
export class MongoConnectionService {


    private dbConnection: Connection  //objeto que va a tener siempre la conexión

    constructor( private configService: ConfigService){
        this.createConnectionDB()
    }

   async createConnectionDB(){
            const host = this.configService.get('mongo.host')
            const port = this.configService.get('mongo.port')
            const user = this.configService.get('mongo.user')
            const password = this.configService.get('mongo.password')
            const database = this.configService.get('mongo.database')

        const DB_URI = `mongodb://${user}:${password}@${host}:${port}/${database}?
authSource=admin
        this.dbConnection = await createConnection(DB_URI)`

         this.dbConnection = await createConnection(DB_URI)

        this.dbConnection.once('open', ()=>{
            console.log(`Connected to ${database}!!`)
        })

        this.dbConnection.once('error', ()=>{
            console.log(`Error connecting to ${database}!!`)
        })
```

```
    }

    //para poder inyectarlo en el modulo que yo quiera
    getConnection(){
        return this.dbConnection
    }


}
```

- En development.env

```
HOST_MONGODB=localhost
PORT_MONGODB=27017
USER_MONGODB=admin
PASSWORD_MONGODB=root
DATABASE_MONGODB=users
```

- En configuration-mongo.ts

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

export default registerAs('mongo', ()=>({
    host: process.env.HOST_MONGODB || 'localhost',
    port: process.env.PORT_MONGODB || 27017,
    user: process.env.USER_MONGODB,
    password: process.env.PASSWORD_MONGODB,
    database: process.env.DATABASE_MONGODB,
}))
```

NOTA: El comando mongo para la terminal esta deprecado. Ir a "Editar Variables de entorno.." /Configuración/Variables de entorno/Path + Editar (añadir la ruta de instalación de mongo (C:\Program Files\MongoDB\Server\6.0\bin)). Debes instalar mongosh de https://www.mongodb.com/try/download/shell el paquete msi

---

- Guia rápida para activar autenticación y usuario en MongoDB

    - Entrar en la terminal con mongosh (si no accediera asegurarse de añadir al Path la variable de entorno de mongosh (la ruta))
    - Escribir:
        - use admin (para entrar en la db admin)
        - escribir lo siguiente

```
db.createUser({ user:"admin", pwd: "123456", roles:
["clusterAdmin","readAnyDatabase","readWriteAnyDatabase",
"userAdminAnyDatabase","dbAdminAnyDatabase"] });
```

- En MongoCompass
- Crear nueva conexión, authentication User/Password
- Colocar el usuario y el password
- El string de conexión tiene que quedar algo así

```
mongodb://admin:123456@localhost:27017/?authSource=admin
```

- Añado el Módulo mongo-connection.module a app.module

```
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import configurationMongo from './configuration/configuration-mongo';
import { MongoConnectionModule } from './modules/mongo-connection/mongo-
connection.module';
require('dotenv').config();

@Module({
  imports: [ConfigModule.forRoot({
    load: [configurationMongo],
    envFilePath: `./env/${process.env.NODE_ENV}.env`,
    isGlobal: true
  }),
  MongoConnectionModule //añado el módulo en imports!!
],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

- Si aparece este error en la terminal al poner en marcha el servidor:

> MongoServerSelectionError: connect ECONNREFUSED ::1:27017

Cambia la palabra localhost por 127.0.0.1 en la variable de entorno HOST_MONGODB en development.env

---

Crear módulo users

- En src/modules creo el modulo de users

> nest g mo users

- De esta manera debería crear también el servicio y el controlador por separado con el CLI de NEST
- Puedo crear el CRUD directamente (con controlador y servicio) con res

> nest g res users

- En el controller añado lo de "/api/v1" en la ruta
- Coloco **@ApiTags('Users')** para documentarlo. recuerda que Swagger debe de estar configurado en el main
- Importo **UsersModule** en el imports de app.module

Creando user-dto

- Creamos el dto
- Uso los decoradores del class-validator (recuerda que para ello debe de estar configurado en el main con **app.useGlobalPipes**)
- Uso **@ApiProperty()** para documentar el dto

```
import { ApiProperty } from "@nestjs/swagger";
import { IsNotEmpty, IsEmail, IsString } from "class-validator";


export class CreateUserDto {

    @ApiProperty({
        name: 'email',
        type: String,
        required: true,
        description: 'Email del usuario'
    })
    @IsNotEmpty()
    @IsEmail()
    email: string;

    @ApiProperty({
        name: 'password',
        type: String,
        required: true,
        description: 'Password del usuario'
    })
    @IsNotEmpty()
```

```
    @IsString()
    password: string;
}
```

Creando interfaz de usuario

- Creo la carpeta interfaces dentro de users y creo user.interface.ts

```
export interface IUser{
    email: string;
    password: string;
}
```

Creando el Schema del usuario

- Al estar trabajando con Mongo, en lugar de trabajar con una clase, trabajaremos con un Schema
- **NOTA** el Schema lo importo de mongoose **NO DE** @nest/mongoose

```
import { Schema } from "mongoose";
import { IUser } from "../interfaces/user.interface";

export const userSchema = new Schema<IUser>({
    email: {type: String, unique: true, trim: true, required: true},
    password: {type: String, required: true}
})
```

Inyectar el modelo en el servicio

- Debo añadir en providers el objeto provide. UseFactory es una función a la que le debo pasar el MongoConnectionModule que tengo een el app.module.
- Por tanto, **SACO el módulo MongoConnectionModule DE app.module** porque lo voy a importar en **users.module**
- En provide le paso un string que usaré posteriormente en el servicio (users.servcie) para inyectar el modelo con el decorador **@Inject**
- Uso la función useFactory y como parámetro le paso el servicio de Mongo, con el método getConnection le paso el modelo con el tipado IUSER, le paso el nombre del modelo (que le pongo ahora), el schema y la colección (la tabla)
- El Servicio **debe de estar exportado en exports de mongo-connection.module.ts**
- **Inject es necesario** para que el useFactory funcione, porque para hacer el .getConnection necesito inyectar el servicio

```

```

```
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { MongoConnectionModule } from '../mongo-connection/mongo-
connection.module';
import { MongoConnectionService } from '../mongo-connection/mongo-
connection.service';
import { IUser } from './interfaces/user.interface';
import { userSchema } from './schema/user-schema';

@Module({
  imports: [MongoConnectionModule],
  controllers: [UsersController],
  providers: [UsersService,
  {
    provide: 'USER_MODEL',
    useFactory: (db: MongoConnectionService) => db.getConnection().model<IUser>
('user', userSchema, 'users'),
    inject: [MongoConnectionService]
  }]
})
export class UsersModule {}
```

- En el users.service uso el decorador Inject y le paso lo que puse en el provide del objeto de configuración anterior
- Inyecto el modelo de tipo IUser

```
import { Inject, Injectable } from '@nestjs/common';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';
import { Model } from 'mongoose';
import { IUser } from './interfaces/user.interface';

@Injectable()
export class UsersService {

  constructor(
    @Inject('USER_MODEL')
    private userModel: Model<IUser>
  ){}
}
```

Creando usuarios

- En el controller me creo el enspoint si no lo he generado con el CLI previamente

```
@Post()
createUser(@Body() createUserDto: CreateUserDto) {
  return this.usersService.createUser(createUserDto);
}
```

- En el servicio, primero debo comprobar si el usuario existe con findOne, donde el email coincida
- Lo paso a minusculas, lo que significa que siempre debe guardarse en minúsculas
- Puedo añadirlo en el schema con lowercase: true

```
export const userSchema = new Schema<IUser>({
    email: {type: String, unique: true, trim: true, required: true, lowercase:
true},
    password: {type: String, required: true}
})
```

- En el servicio hago las validaciones pertinentes si el usuario no existe

```
@Injectable()
export class UsersService {

  constructor(
    @Inject('USER_MODEL')
    private userModel: Model<IUser>
  ){}
  async createUser(createUserDto: CreateUserDto) {
    const userExists = await this.userModel.findOne({email:
createUserDto.email.toLowerCase()})

    if(userExists){
      throw new ConflictException("El usuario ya existe")
    }
    const user = new this.userModel(createUserDto)

    return user.save()
  }
}
```

- Compruebo que funcione con THUNDERCLIENT/POSTMAN
- Creo el objeto en el body

```
{
 "email": "pedro@gmail.com",
  "password": "123456"
}
```

- En el endpoint

> http://localhost:3000/api/v1/users

- Ahora quiero hacer que ne la respuesta no me envie el password de vuelta
- Puedo hacerlo de esta forma

```
async createUser(createUserDto: CreateUserDto) {
    const userExists = await this.userModel.findOne({email:
createUserDto.email.toLowerCase()})

    if(userExists){
      throw new ConflictException("El usuario ya existe")
    }
    const user = new this.userModel(createUserDto)

    await user.save()

    user.password = undefined

    return user
  }
```

- Hay otra manera que explica Herrera bastante más elegante

Encriptando contraseñas

- Con bcrypt

> npm i bcrypt

- Podemos usar algo como beforeSave, aunque bien podríamos hacerlo directamente antes de guardar en el servicio
- Entonces, **no es estrictamente necesario hacerlo asi**
- En el schema, uso .pre y le paso el 'save' y una función. Antes de guardar ejecutará este código
- Le digo que es de tipo IUser
- Uso una función normal (no una de flecha) porque necesito el contexto del this para pasarle el password
- Genero el salt y lo guardo en una constante
- Genero el password con el salt y el this.password
- Entonces le digo que el password es el hash

```
import { Schema } from "mongoose";
import { IUser } from "../interfaces/user.interface";
import * as bcrypt from 'bcrypt'

export const userSchema = new Schema<IUser>({
    email: {type: String, unique: true, trim: true, required: true, lowercase:
```

```
true},
    password: {type: String, required: true}
})

userSchema.pre<IUser>('save', async function(){
    const salt = await bcrypt.genSalt(10)
    const hash = await bcrypt.hash(this.password, salt)
    this.password = hash
})
```

Obteniendo usuarios

- Creo el endpoint en el controller, no vamos a filtrar
- Queremos que no nos muestre el password

```
@Get()
getUsers() {
  return this.usersService.getUsers();
}
```

- En el servicio, uso como segundo parámetro un objeto (antes coloco un objeto vacío) y si le paso password: 0 es como decirle un false
- No lo muestra

```
getUsers() {
  return this.userModel.find({}, {password:0});
}
```

Obteniendo un usario por su email

- No hace falta usar ningún endpoint. Este método se usará para poblar usuarios

```
async findUserByEmail(email: string){
  return await this.userModel.findOne({email: email.toLowerCase()})
}
```

- Lo uso en createUser para comprobar que el usuario existe

```
async createUser(createUserDto: CreateUserDto) {
  const userExists = await this.findUserByEmail(createUserDto.email)

  if(userExists){
```

```
      throw new ConflictException("El usuario ya existe")
    }
    const user = new this.userModel(createUserDto)

    await user.save()

    user.password = undefined

    return user
  }
```

Poblar usuarios

- Creo un método al que llamaré populateUsers con un arreglo
- Lo recorro con un for of (mejor que con un forEach) y hago la inserción con createUser
- Para llamar al método lo hago en el constructor (podría crear un endpoint para llamarlo)

```
import { Model } from 'mongoose';
import { IUser } from './interfaces/user.interface';

@Injectable()
export class UsersService {

  constructor(
    @Inject('USER_MODEL')
    private userModel: Model<IUser>
  ){
    this.populateUsers()
  }

  async populateUsers(){
    const users: CreateUserDto[]= [
      {
        email: "paquita@gmail.com",
        password: "123456"
      },
      {
        email: "jordi@gmail.com",
        password: "123456"
      },
      {
        email: "leonardo@gmail.com",
        password: "123456"
      }
    ]

    for (const user of users){
      const userExists = await this.findUserByEmail(user.email)
      if(!userExists){
        await this.createUser(user)
```

```
      }
    }
  }
```

- Si refresco ya tengo los usuarios en la db

Creando auth module

- Uso el CLI

> nest g res auth

- Importo el AuthModule en app.module dentro de imports
- Creo el endpoint del controller api/v1/auth
- Le añado el @ApiTags('auth') para la documentación con Swagger

```
@Controller('api/v1/auth')
@ApiTags('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}
}
```

Archivo de configuración para auth

- Añado una nueva variable en development.env

```
SECRETKEY_AUTH=secretkey
```

- Creamos otro fichero de configuración para el módulo de auth. Me sirve para cargar variables

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

export default registerAs('auth', ()=>({
    secretkey: process.env.SECRETKEY_AUTH
}))
```

- Debo añadirlo en el app.module, dentro del array de ConfigModule.forRoot, en load

```
@Module({
  imports: [ConfigModule.forRoot({
    load: [configurationMongo, configurationAuth],
    envFilePath: `./env/${process.env.NODE_ENV}.env`,
    isGlobal: true
```

```
    }),
    UsersModule,
    AuthModule
  ],
    controllers: [],
    providers: [],
  })
  export class AppModule {}
```

Creando el dto de auth

- Copio el dto de User, es el mismo

```
import { ApiProperty } from "@nestjs/swagger";
import { IsNotEmpty, IsEmail, IsString } from "class-validator";


export class AuthCredentialsDto {

    @ApiProperty({
        name: 'email',
        type: String,
        required: true,
        description: 'Email del usuario a loguear'
    })
    @IsNotEmpty()
    @IsEmail()
    email: string;

    @ApiProperty({
        name: 'password',
        type: String,
        required: true,
        description: 'Password del usuario a loguear'
    })
    @IsNotEmpty()
    @IsString()
    password: string;
}
```

Passport

- Usaremos un módulo de Nest llamado Passport
- Hay dos formas de autenticación, con Passport local o con **jwt**. Se pueden combinar
- Lo instalo junto a la propia biblioteca con

> npm i @nestjs/passport passport

- Importo PassportModule en auth.module y lo configuro con la estrategia jwt

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { PassportModule } from '@nestjs/passport';

@Module({
  imports:[PassportModule.register({defaultStrategy: "jwt"})],
  controllers: [AuthController],
  providers: [AuthService]
})
export class AuthModule {}
```

Importando JwtModule usadno el secretkey

- Necesitamos colocar la secretkey en el PassportModule
- Instalamos jwt

```
npm i @nestjs/jwt passport-jwt
```

- Con RegisterAsync podemos meter funciones pre, y lo usaremos para obtener la secretkey
- ConfigService es algo global por lo que puedo llamarlo aqui
- LLamo al objeto de configuración donde puse el campo 'auth' y en el callback secretkey, por lo tanto uso auth.secretkey
- En signIn le pongo que expire en 60 segundos por motivos de pruebas. Lo normal es 7 dias o 2 o 3
- Tengo que usar el inject ya que he usado useFactory, si no no funcionará

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { PassportModule } from '@nestjs/passport';
import {JwtModule} from '@nestjs/jwt';
import { ConfigService } from '@nestjs/config';

@Module({
  imports:[PassportModule.register({defaultStrategy: "jwt"}),
  JwtModule.registerAsync({
    useFactory:(configService: ConfigService )=>{
      return {
        secret: configService.get('auth.secretkey'),
        signOptions: {expiresIn: '60s'}
      }
    },
    inject: [ConfigService]
  })
],
  controllers: [AuthController],
```

```
  providers: [AuthService]
})
export class AuthModule {}
```

- El 'auth.secretkey' viene de configuration-auth.ts

```
export default registerAs('auth', ()=>({
    secretkey: process.env.SECRETKEY_AUTH
}))
```

- Nosotros como tal no podemos borrar tokens

Estrategia JWT

- Vamos a implementar la estrategia. Creo la carpeta strategy en /auth donde crearemos un servicio

> cd src/modules/auth/strategy nest g s jwt-strategy

- Tenemos que hacer que la clase herede de PassportStrategy y le paso la Strategy de passport-jwt
- Necesitamos el ConfigService. Lo inyectamos
- Debemos llamar a super poruqe a la clase padre hay que pasarle un objeto con dos parámetros
  - El jwtFromRequest
  - Uso el configService para obtener la secretkey

```
import { Injectable } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { PassportStrategy } from '@nestjs/passport';
import {Strategy, ExtractJwt} from 'passport-jwt'

@Injectable()
export class JwtStrategyService extends PassportStrategy(Strategy) {
    constructor(
        private configService: ConfigService
    ){
        super({
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
            secretOrKey: configService.get('auth.secretkey')
        })
    }
}
```

- Hay que importar en providers del auth.module el servicio JwtStrategyService

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
```

```
import { PassportModule } from '@nestjs/passport';
import {JwtModule} from '@nestjs/jwt';
import { ConfigService } from '@nestjs/config';
import { JwtStrategyService } from './strategy/jwt-strategy/jwt-strategy.service';

@Module({
  imports:[PassportModule.register({defaultStrategy: "jwt"}),
  JwtModule.registerAsync({
    useFactory:(configService: ConfigService )=>{
      return {
        secret: configService.get('auth.secretkey'),
        signOptions: {expiresIn: '60s'}
      }
    },
    inject: [ConfigService]
  })
],
  controllers: [AuthController],
  providers: [AuthService, JwtStrategyService]
})
export class AuthModule {}
```

Validando el usuario

- Hagamos un método para validar el usuario
- NOTA: el JwtStrategyService no lo vamos a llamar nunca, solo va a estar en providers. Al heredar de PassprotStrategy se está usando
- En el método validate le paso un payload, que será un objeto que yo crearé de tipo JwtPayload.
- Lo creo en la carpeta dto/jwt-payload.ts

```
export class JwtPayload{
    email: string
}
```

- Este payload lo crearemos y le pasaremos el email del usuario, pasará por aquí y comprobará que es válido. Es una validación que no se vé
- Para validar necesito buscar al usuario, por lo que necesito el servicio de users
- **Exporto el usersService del usersModule e importo el UsersModulen en auth.module**
- Inyecto el servicio para poder usarlo
- Si el usuario no existe lanzo una excepción
- Le quito el password porque luego vamos a poder ver su información (para no mostrarlo)
- Si no estás autorizado no te va a dejar pasar
- Si le pusiera un return true siempre dejaría

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

```
import { PassportStrategy } from '@nestjs/passport';
import {Strategy, ExtractJwt} from 'passport-jwt'
import { UsersService } from 'src/modules/users/users.service';
import { JwtPayload } from '../../dto/jwt-payload';


@Injectable()
export class JwtStrategyService extends PassportStrategy(Strategy) {
    constructor(
        private configService: ConfigService,
        private usersService: UsersService
    ){
        super({
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
            secretOrKey: configService.get('auth.secretkey')
        })
    }

   async validate(payload: JwtPayload){
        const user = await this.usersService.findUserByEmail(payload.email)
        if(!user){
            throw new NotFoundException("El usuario no existe")
        }
        user.password = undefined

        return user;
    }
}
```

Validar credenciales

- Ahora validaremos las credenciales (no el usuario)
- Creo el método en el AuthService
- Si el usuario está bien compruebo el password (importo bcrypt)
- Devuelve una promesa por lo que uso await
- Si el password está bien retorno el user, si no retorno null

```
import { Injectable } from '@nestjs/common';
import { AuthCredentialsDto } from './dto/create-auth.dto';
import { UsersService } from '../users/users.service';
import * as bcrypt from 'bcrypt'

@Injectable()
export class AuthService {

  constructor(
    private usersService: UsersService
  ){}

  async validateUser(authCredentials: AuthCredentialsDto){
```

```
    const user = await this.usersService.findUserByEmail(authCredentials.email)
    if(user){

      const passwordOk = await bcrypt.compare(authCredentials.password,
user.password)
      if(passwordOk){
          return user
      }
    }
    return null
  }
}
```

Login

- Creo el endpoint en el controller

```
@Post('/login')
login(@Body() authCredentials: AuthCredentialsDto){
    return this.authService.login(authCredentials)
}
```

- En el auth.service uso el método para comprobar que las credenciales son correctas
- Genero el payload. Necesitaré el servicio de JWTService de @nestjs/jwt
- Retorno un objeto con un accessToken, y con el servicio genero el token

```
import { Injectable, NotFoundException, UnauthorizedException } from
'@nestjs/common';
import { AuthCredentialsDto } from './dto/create-auth.dto';
import { UsersService } from '../users/users.service';
import * as bcrypt from 'bcrypt'
import { JwtPayload } from './dto/jwt-payload';
import { JwtService } from '@nestjs/jwt';

@Injectable()
export class AuthService {

  constructor(
    private usersService: UsersService,
    private jwtService: JwtService
  ){}

  async validateUser(authCredentials: AuthCredentialsDto){
    const user = await this.usersService.findUserByEmail(authCredentials.email)
    if(user){

      const passwordOk = await bcrypt.compare(authCredentials.password,
user.password)
```

```
        if(passwordOk){
            return user
        }
      }
      return null
    }

    async login(authCredentials: AuthCredentialsDto){
      const user = await this.validateUser(authCredentials)

      if(!user){
        throw new UnauthorizedException("Credenciales inválidas")
      }

      const payload: JwtPayload ={
        email: user.email
      }

      return {
        accesToken: this.jwtService.sign(payload)
      }
    }
}
```

- Si voy al endpoint http://localhost:3000/api/v1/auth/login con un usario y contraseña válidos me devuelve esto

```
{
  "accesToken":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlbWFpbCI6InBlZHJvQGdtYWlsLmNvbSIsImlhdCI6
MTcwMjMxODg5MywiZXhwIjoxNzAyMzE4OTIzfQ.56IJzhZSnmhpdtkZ9p7s7e3OT0Xyi7In0LPLL3V75YE
"
}
```

- Lo usaremos para validar que somos nosotros y nos hemos logueado.
- Luego veremos el error de que si no está correcto no me dejaría (cuando el validate(payload))

---

jwt.io

- En esta web puedes introducir el token y comprobar que esté bien lo que ha generado el login

---

Protegiendo los endpoints del método user

- Sólo podremos usar getUsers y createUser si estamos logueados
- En el users.module necesito importar el PassportModule.register. Todo módulo que tenga seguridad lo necesita
- users.module

```
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
import { MongoConnectionModule } from '../mongo-connection/mongo-
connection.module';
import { MongoConnectionService } from '../mongo-connection/mongo-
connection.service';
import { IUser } from './interfaces/user.interface';
import { userSchema } from './schema/user-schema';
import { PassportModule } from '@nestjs/passport';

@Module({
  imports: [PassportModule.register({defaultStrategy: "jwt"}),
  MongoConnectionModule],
  controllers: [UsersController],
  providers: [UsersService,
  {
    provide: 'USER_MODEL',
    useFactory: (db: MongoConnectionService) => db.getConnection().model<IUser>
('user', userSchema, 'users'),
    inject: [MongoConnectionService]
  }],
  exports:[UsersService]
})
export class UsersModule {}
```

- En el users.controller utilizo @UseGuards con AuthGuard y le paso 'jwt'

```
import { Controller, Get, Post, Body, Patch, Param, Delete, UseGuards } from
'@nestjs/common';
import { UsersService } from './users.service';
import { CreateUserDto } from './dto/create-user.dto';
import { UpdateUserDto } from './dto/update-user.dto';
import { ApiTags } from '@nestjs/swagger';
import { AuthGuard } from '@nestjs/passport';

@Controller('api/v1/users')
@ApiTags('Users')
export class UsersController {
  constructor(private readonly usersService: UsersService) {}

  @Post()
  @UseGuards(AuthGuard('jwt'))
  createUser(@Body() createUserDto: CreateUserDto) {
    return this.usersService.createUser(createUserDto);
  }

  @Get()
  @UseGuards(AuthGuard('jwt'))
  getUsers() {
```

```
    return this.usersService.getUsers();
  }
}
```

- Para poder acceder a createUser y getUsers debo añadir el Bearer Token del login en ThunderClient/POSTMAN en Auth/Bearer (sin las comillas!)

Comprobando la validación de la estrategia

- Para qué sirve el validate(payload: JwtPayload)?
- Si le pongo un return false no me va a dejar (pondrá unauthorized) aunque esté bien logueado
- Entonces, este validate es **SUPER IMPORTANTE**. Es quien hace el trabajo por detrás
- Está en el jwt-strategy.service, que solo se incluye en los providers (no se usa como el resto de servicios)

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { PassportStrategy } from '@nestjs/passport';
import {Strategy, ExtractJwt} from 'passport-jwt'
import { UsersService } from 'src/modules/users/users.service';
import { JwtPayload } from '../../dto/jwt-payload';


@Injectable()
export class JwtStrategyService extends PassportStrategy(Strategy) {
    constructor(
        private configService: ConfigService,
        private usersService: UsersService
    ){
        super({
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
            secretOrKey: configService.get('auth.secretkey')
        })
    }

   async  validate(payload: JwtPayload){
        const user = await this.usersService.findUserByEmail(payload.email)
        if(!user){
            throw new NotFoundException("El usuario no existe")
        }
        user.password = undefined

        return user;
    }
}
```

Devolviendo datos del usuario logueado

- Mostraremos los datos del usuario logueado
- Lo haremos con un GET en el auth.controller
- Haremos uso de un nuevo decorador, **@Request**. Lo que metemos de payload se mete en el request

```
  @Get('/data-user')
  dataUser(@Request() req){

    console.log(req)
      return req.user
  }
```

- Le añado el Guard para que solo se pueda hacer cuando estemos logueados. Le añado el Bearer Token a ThunderClient en Auth/Bearer
- Si el user.password no lo paso a undefined en el validate(payload) me mostraría el password también y no interesa

---

Documentar

- En el controller uso @ApiOperation, @ApiBody y @ApiResponse para documentar. Ejemplo del login

```
@Post('/login')
  @ApiOperation({
    description: "Nos loguea en la app",
  })
  @ApiBody({
    description: "Nos logueamos usando las credenciales",
    type: AuthCredentialsDto,
    examples:{
      ejemplo:{
        value:{
          email: "pedro@gmail.com",
          password: "123456"
        }
      }
    }
  })
  @ApiBearerAuth('jwt')
  @ApiResponse({
    status: 401,
    description: "Credenciales inválidas"
  })
    @ApiResponse({
    status: 201,
    description: "Login correcto"
  })
  login(@Body() authCredentials: AuthCredentialsDto){
      return this.authService.login(authCredentials)
  }
```

- Para ver la documentación ir a localhost:3000/swagger

Cron NEST

- Creo el proyecto

```
nest new cron-app
```

- Elimino los archivos que no necesito
- Instalo Swagger y lo configuro
- Creo la carpeta src/modules y genero el módulo de cron

```
nest g res cron
```

- Instalo **@nestjs/schedule**, me va a permitir controlar los crons. Instalo los tipos con **@types/cron**
- En cron.module uso **ScheduleModule.forRoot**

```
import { ScheduleModule } from '@nestjs/schedule';

@Module({
  imports:[
    ScheduleModule.forRoot()  ],
  controllers: [CronController],
  providers: [CronService]
})
export class CronModule {}
```

- Evidentemente, **debo importar CronModule en el imports del app.module**
- En el cron.service cogemos su servicio **ScheduleRegistry** y lo inyectamos

```
import { SchedulerRegistry } from '@nestjs/schedule';

@Injectable()
export class CronService {

  constructor(private readonly scheduleRegistry: SchedulerRegistry){

  }
}
```

Crontab guru

- Creo tres métodos cron en el servicio. Puedo llamarlos cómo quiera
- Uso el decorador **@Cron**. Tenemos el tiempo y los parámetros. Cada asterisco significa cada minuto

- Si quieres añadir los segundos escribe */10
- Aquí podría colocarle la hora de la madrugarda que yo quiera, por ejemplo
- En el objeto le coloco la propiedad name que funciona como id. Tengo también el timezone

NOTA: con Ctrl + i veo las opciones disponibles del objeto (intellisense de Typescript)

```
import { Injectable } from '@nestjs/common';
import { Cron, SchedulerRegistry } from '@nestjs/schedule';

@Injectable()
export class CronService {

  constructor(private readonly scheduleRegistry: SchedulerRegistry){

  }

  @Cron(`*/10 * * * * *`, {
    name: 'cron1'

  })
  cron1(){
    console.log("cron1: Acción cada 10 segundos")
  }

  @Cron(`*/30 * * * * *`, {
    name: 'cron2'

  })
  cron2(){
    console.log("cron1: Acción cada 30 segundos")
  }

  @Cron(`* * * * *`, {
    name: 'cron3'

  })
  cron3(){
    console.log("cron1: Acción cada minuto")
  }
}
```

- No se necesita ejecutar ningún comando específico en la terminal, simplemente arrancar el servidor

Desactivando un cron

- Para desactivar un cron puedo usar PUT o POST
- Elegiré un PUT ya que realmente estoy haciendo algo parecido a modificar su estado

```
@Put('/desactivate/:name')
desactivateCron(@Param('name') name: string ){
return this.cronService.desactivateCron(name)
}
```

- En el service, primero debemos buscar si existe o no ese cron
- Si sitúo el cursor encima de job me dice que es de tipo CronJob. Le pongo tipado

```
desactivateCron(name:string){
    const job: CronJob = this.scheduleRegistry.getCronJob(name)

    if(!job){
      throw new NotFoundException("No se ha encontrado el cron")
    }else{
      job.stop()
      console.log(`El cron con el nombre: ${name} está desactivado`)
      return true
    }
  }
```

Activando un cron

- Practicamente hacemos un copy past del controlador anterior y cambiamos desactivate por activate
- controller

```
@Put('/activate/:name')
activateCron(@Param('name') name: string ){
  return this.cronService.activateCron(name)
}
```

- Y con el servicio igual, en lugar de poner .stop ponemos .start

```
activateCron(name:string){
    const job: CronJob = this.scheduleRegistry.getCronJob(name)

    if(!job){
      throw new NotFoundException("No se ha encontrado el cron")
    }else{
      job.start()
      console.log(`El cron con el nombre: ${name} está activado`)
      return true
    }
  }
```

Devolver todos los nombres de los crons

- Creo el endpoint en el controller

```
@Get()
getNameCrons(){
  return this.cronService.getNameCrons()
}
```

- En el servicio, obtengo los crons. .key son los nombres
- Uso un for of para llenar el array de cada uno de ellos

```
getNameCrons(){
    const names = []

    for(const name of this.scheduleRegistry.getCronJobs().keys()){
       names.push(name)
    }
    return names
  }
```

Desactivando todos los crons a la vez

- Usaremos un PUT

```
@Put('desactivate-all')
desactivateAllCrons(){
  return this.cronService.desactivateAllCrons()
}
```

- En el servicio, usaré el método anterior porque necesito los nombres
- Uso un ciclo for para recorrer el array y el metodo de desactivar anterior

```
desactivateAllCrons(){
    const names = this.getNameCrons()

    for(const name of names){
      this.desactivateCron(name)
    }
    return true
  }
```

Activando todos los crons

- Muy similar al anterior, en lugar de desactivar. Uso el método de activar
- En el controller

```
@Put('activate-all')
activateAllCrons(){
  return this.cronService.activateAllCrons()
}
```

- En el servicio

```
activateAllCrons(){
    const names = this.getNameCrons()

    for(const name of names){
      this.activateCron(name)
    }
    return true
  }
```

EMAILS NEST

- Creo la app con nest **new emails-app**
- Borro todos los archivos innecesarios como el app.service, app.controller, los tests .spec
- Instalo swagger con @nestjs/swagger, lo configuro.
- Instalo el class-validator class-transformer
- Lo configuro en el mail con **app.useGlobalPipes**

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

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

  app.useGlobalPipes(new ValidationPipe({
    transform: true
  }))
  const config = new DocumentBuilder()
  .setTitle('Emails example')
  .setDescription('emails app')
  .setVersion('1.0')
  .build();
```

```
const document = SwaggerModule.createDocument(app, config);
SwaggerModule.setup('emails', app, document);
  await app.listen(3000);
}
bootstrap();
```

- Creo el módulo de email con **nest g res emails**
- Creo el dto

```
import { ApiProperty } from "@nestjs/swagger"
import { IsArray, IsNotEmpty, IsString } from "class-validator"

export class EmailDto {

    @ApiProperty({
        name: 'body',
        required: true,
        type: String,
        description: "Cuerpo del mensaje a enviar"
    })
    @IsString()
    @IsNotEmpty()
    body: string

    @ApiProperty({
        name: 'subject',
        required: true,
        type: String,
        description: "Asunto del mensaje a enviar"
    })
    @IsString()
    @IsNotEmpty()
    subject: string

    @ApiProperty({
        name: 'receivers',
        required: true,
        isArray: true,
        type: String,
        description: "Destinatarios del mensaje a enviar"
    })
    @IsArray()
    @IsNotEmpty()
    receivers: string[]
}
```

- Para validar el array de strings voy a crear otro dto que llamaré sender-dto

```
import { ApiProperty } from "@nestjs/swagger";
import { IsEmail, IsNotEmpty} from "class-validator";

export class SenderDto{

    @ApiProperty({
        name: 'email',
        required: true,
        type: String,
        description: "Email del destinatario"
    })
    @IsEmail()
    @IsNotEmpty()
    email: string
}
```

- Uso el decorador **@ValidateNested** en el receivers MessageDto para validar lo que hay dentro del array
- Le digo each: true (dentro de cada elemento que se cumpla lo que te indico)
- Con el class-transformer le indico cin **@Type** que es de tipo SenderDto

```
@ApiProperty({
    name: 'receivers',
    required: true,
    type: String,
    description: "Destinatarios del mensaje a enviar"
})
@IsArray()
@IsNotEmpty()
@ValidateNested({each: true})
@Type(()=> SenderDto[])
receivers: SenderDto[]
```

EmailConfig

- Creo el archivo emai-config.ts dentro de la carpeta emails
- Le coloco el puerto 25 por defecto. No siempre hay que indicarle el puerto, lo pongo como opcional
- Le indico con secure si quiero que sea con seguridad
- Para los servicios creo un enum

```
export class EmailConfig{
    from: string

    password: string

    service: SERVICES
```

```
    port?: number = 25

    secure?: boolean = false
}

export enum SERVICES{
    GMAIL= 'gmail',
    OUTLOOK = 'outlook365'
}
```

Módulo dinámico

- Un módulo dinámico es un módulo al que **le podemos pasar parámetros** y le podemos cambiar los controllers, los providers, etc
- Se usa cuando un módulo necesita comportarse diferente en diferentes casos de uso, a modo de plugin
    - Un buen ejemplo es un módulo de configuración
- Creo un método static llamado register al que le paso unas opciones de tipo **EmailConfig** que devuelve algo de tipo **DynamicModule**
- Para cuando quiera pasar las opciones a mi EmailsService creo un objeto y lo llamo **CONFIG_OPTIONS**

```
import { DynamicModule, Module } from '@nestjs/common';
import { EmailsService } from './emails.service';
import { EmailsController } from './emails.controller';
import { EmailConfig } from './email-config';

@Module({
  controllers: [EmailsController],
  providers: [EmailsService]
})
export class EmailsModule {
  static register(options: EmailConfig): DynamicModule{
    return{
      module: EmailsModule,
      controllers:[EmailsController],
      providers: [
        {
          provide: 'CONFIG_OPTIONS',
          useValue: options
      },
        EmailsService]
    }
  }
}
```

- En el service lo obtengo usando **@Inject**
- Instalo Nodemailer con **npm i nodemailer**

- Lo importo en el servicio

```
import { Inject, Injectable } from '@nestjs/common';
import { UpdateEmailDto } from './dto/update-email.dto';
import { EmailConfig } from './email-config';
import * as nodemailer from 'nodemailer'

@Injectable()
export class EmailsService {

  constructor(
    @Inject('CONFIG_OPTIONS')
    private options: EmailConfig
  ){
    console.log(this.options)
  }
}
```

Creando un endpoint para enviar un email

- Creo un POST en el controller

```
@Post()
sendEmail(@Body() emailDto: EmailDto) {
  return this.emailsService.sendEmail(emailDto);
}
```

Configurando módulo para enviar correos con gmail

- En app.module, con la función register le paso los campos de email-config

```
import { Module } from '@nestjs/common';
import { EmailsModule } from './emails/emails.module';
import { SERVICES } from './emails/email-config';

@Module({
  imports: [EmailsModule.register({
    from: 'miemail@gmail.com',
    password: 'mi-password',
    service: SERVICES.GMAIL
  })],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

- Cuando arrancio el servidor, el console.log(options) en el constructor del servicio me devuelve este objeto

```
{
  from: 'miemail@gmail.com',
  password: 'mi-password',
  service: 'gmail'
}
```

- Con lo que lo está pillando. Quito el console.log
- Hay que **configurar la doble verificación en gmail**
- En Seguridad/Verificación en dos pasos (activarla)
- Hay que sincronizar la cuenta con el teléfono
- Ahora, en Iniciar sesión en Google, aparece **Contraseñas de aplicaciones**
- Selecciona la aplicación o dispositivo: Otro (y le pongo un nombre, por ej: test nest)
- Esto **me va a generar una contraseña**. La usaremos en lugar de mi password

---

Enviar correos con Nodemailer

- Usaremos los "well-known services" que ya tienen su host y puerto pre-definidos, no hace falta indicarlos
- Creo el transporter según la documentación de nodemailer en el método sendEmail
- A la función sendEmail no le puedo pasar el dto tal cual, lo configuro en emailOptions
- Para el to, al ser un array, uso el .map
- Lo coloco todo dentro de un try catch por si falla

```
  sendEmail(message: EmailDto){

    try {
      const transporter = nodemailer.createTransport({
        service: this.options.service,
        auth: {
          user: this.options.from,
          pass: this.options.password
        }
      })


      const to = message.receivers.map(e =>e.email )

      const mailOptions = {
        from: this.options.from,
        to,
        subject: message.subject,
        html: message.body
      }
```

```
      return transporter.sendEmail(mailOptions)
    } catch (error) {
      console.error(error)

      return null
    }
  }
```

- Ahora habría que testearlo con ThunderClient o POSTMAN, indicándole en un objeto el subject, el to, etc...

```
{
  "subject": "Test correo",
  "body": "<h1>Hola mundo! </h1>",
  "receivers": [
   { "email": "perico@gmail.com"}
  ]
}
```

- Para usar un correo de outlook solo tengo que cambiar el SERVICE e indicar mi correo de outlook

---

Documentación endpoint

```
import { Controller, Get, Post, Body } from '@nestjs/common';
import { EmailsService } from './emails.service';
import { EmailDto } from './dto/create-email.dto';
import { ApiBody, ApiOperation, ApiResponse } from '@nestjs/swagger';


@Controller('emails')
export class EmailsController {
  constructor(private readonly emailsService: EmailsService) {}

  @Post()
  @ApiOperation({
    description: 'envía un email',

  })
  @ApiBody({
    description: 'envía un email usando un emailDto',
    type: EmailDto,
    examples:{
      ejemplo1:{
        value: {
          subject: "Test correo",
          body: "<h1>Hola mundo! </h1>",
          receivers: [
           { email: "perico@gmail.com"}
```

```
          ]
        }
      }
    }
  })
  @ApiResponse({
    status: 201,
    description: "Correo enviado correctamente"
  })
  sendEmail(@Body() emailDto: EmailDto) {
    return this.emailsService.sendEmail(emailDto);
  }

}
```

Logs NEST

- Creo el proyecto con nest new logs-app
- Borro todos los archivos que no necesito como los test o el app.service y el app.controller (quito las dependencias)
- Instalo swagger y lo configuro
- Vamos a reutilizar el proyecto anterior de Cron
- Borro la carpeta src y la reemplazo por la de cron
- Necesito instalar el @nestjs/schedule y los types de cron
- Instalo Winston

> npm i winston

- Crearemos una instancia de logger por cada tipo de transport que queramos hacer, aunque transports sea un array
- No es como sale en la documentación

Creando nuestro módulo de logger de forma global

- Creo el módulo en src/modules con nest g res logger, aunque no necesito endpoints.
- Le diremos que el módulo es global. Una vez importe este módulo en el app.module no voy a tener que importarlo más
- Tiene lógica, ya que el módulo de logger es necesario en todo el resto de módulos
- Por ejemplo, la configuración de Mongo también puede ser global
- Tengo que exportar el servicio para que esté disponible

```
import { Global, Module } from '@nestjs/common';
import { LoggerService } from './logger.service';
import { LoggerController } from './logger.controller';

@Global()
```

```
@Module({
  controllers: [LoggerController],
  providers: [LoggerService],
  exports:[LoggerService]
})
export class LoggerModule {}
```

- Importo el módulo de Logger en el imports de app.module.
- Se recomienda poner los módulos globales primero en el array de imports

---

Creando el servicio

- Si digo: quiero los loggers de errores, o de info, o de warns, o todos.
- Tengo que crear los campos en el servicio
- En el método createLogger primero formateo el output (para que salga la fecha, la hora...)
- printf nos devuelve un objeto que vamos a llamar log
- Tiene tres propiedades (aunque timestamp no salga con el intellisense)
- Uso un template string para concatenarlas. Uso solo la primera letra en mayúscula del level

```
import { Injectable } from '@nestjs/common';
import { Logger, format } from 'winston';

@Injectable()
export class LoggerService {

    private loggerInfo: Logger
    private loggerError: Logger
    private loggerWarn: Logger
    private loggerAll: Logger


    constructor(){
    }

    createLogger(){
        const textFormat = format.printf((log)=>{
           return `${log.timestamp}- ${log.level.toUpperCase().charAt(0)}-
${log.message}`
        })
    }
}
```

- Formateo la fecha también. Le pongo año, mes, días, horas, minutos y segundos
- Creo el primer logger de info.
- Uso createLogger de Winston
- Uso .combine para combinar los dos formatos
- Transports es para indicar dónde quiero guardarlo

```
    createLogger(){
        const textFormat = format.printf((log)=>{
           return `${log.timestamp}- ${log.level.toUpperCase().charAt(0)}-
${log.message}`
        })

        const dateFormat = format.timestamp({
            format: 'YYYY-MM-DD HH:MM:SS'
        })

        this.loggerInfo = createLogger({
            level: 'info',
            format:format.combine(
                dateFormat,
                textFormat
            ),
            transports:[
                new transports.File({
                    filename: 'log/info/info.log'
                })
            ]

        })
    }
```

- El resto de loggers serían practicamente igual, solo cambio la ruta de ubicación en el transports con el nombre correspondientes
- Al loggerAll no le coloco level porque es en todos y lo guardo en dos ubicaciones
    - Uso un transport para el archivo log y otro transport para la consola

```
this.loggerAll = createLogger({
    format:format.combine(
        dateFormat,
        textFormat
    ),
    transports:[
        new transports.File({
            filename: 'log/all/all.log'
        }),
        new transports.Console()
    ]
})
```

Arrancando nuestro logger

- Hay que crear una instancia del logger en el main
- main.ts

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


async function bootstrap() {
  const app = await NestFactory.create(AppModule, {
    logger: new LoggerService()
  });
  await app.listen(3000);
}
bootstrap();
```

- Esto da error porque el LoggerService debe tener unos métodos concretos. Los creo
- logger.service

```
log(message: string){

}
error(message: string){

}
warn(message: string){

}
debug(message: string){

}
verbose(message: string){

}
```

- Una vez pongo estos métodos en el logger.service desaparece el error
- Uso los loggers que he creado y los coloco dentro de su método correspondiente

```
log(message: string){
    this.loggerInfo.info(message)
    this.loggerAll.info(message)
}
error(message: string){
    this.loggerError.error(message)
    this.loggerAll.error(message)
}
warn(message: string){
    this.loggerWarn.warn(message)
    this.loggerAll.warn(message)
}
```

- Debo llamar al método createLogger del servicio en el constructor

```
import { Injectable } from '@nestjs/common';
import { Logger, createLogger, format, transports } from 'winston';

@Injectable()
export class LoggerService {

    private loggerInfo: Logger
    private loggerError: Logger
    private loggerWarn: Logger
    private loggerAll: Logger


    constructor(){
        this.createLogger()
    }

    createLogger(){
        const textFormat = format.printf((log)=>{
           return `${log.timestamp}- ${log.level.toUpperCase().charAt(0)}-
${log.message}`
        })

        const dateFormat = format.timestamp({
            format: 'YYYY-MM-DD HH:MM:SS'
        })

        this.loggerInfo = createLogger({
            level: 'info',
            format:format.combine(
                dateFormat,
                textFormat
            ),
            transports:[
                new transports.File({
                    filename: 'log/info/info.log'
                })
            ]
        })
        this.loggerError = createLogger({
            level: 'error',
            format:format.combine(
                dateFormat,
                textFormat
            ),
            transports:[
                new transports.File({
                    filename: 'log/error/error.log'
                })
            ]
        })
        this.loggerWarn = createLogger({
```

```
            level: 'warn',
            format:format.combine(
                dateFormat,
                textFormat
            ),
            transports:[
                new transports.File({
                    filename: 'log/warn/warn.log'
                })
            ]
        })
        this.loggerAll = createLogger({
            format:format.combine(
                dateFormat,
                textFormat
            ),
            transports:[
                new transports.File({
                    filename: 'log/all/all.log'
                }),
                new transports.Console()
            ]
        })

    }

    log(message: string){
        this.loggerInfo.info(message)
        this.loggerAll.info(message)
    }
    error(message: string){
        this.loggerError.error(message)
        this.loggerAll.error(message)
    }
    warn(message: string){
        this.loggerWarn.warn(message)
        this.loggerAll.warn(message)
    }
    debug(message: string){

    }
    verbose(message: string){

    }
}
```

Usando LoggerService

- Cómo hicimos global el LoggerModule, si quiero usar el loggerService en el módulo de cron no tengo que importar nada en imports ni exports, solo el servicio como tal en las importaciones como cualquier otro paquete

- Inyecto el servicio
- Sustituimos los console.log de los métodos cron por los del logger

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { Cron, SchedulerRegistry } from '@nestjs/schedule';
import { CronJob } from 'cron';
import { LoggerService } from '../logger/logger.service';

@Injectable()
export class CronService {

  constructor(private readonly scheduleRegistry: SchedulerRegistry,
              private readonly loggerService: LoggerService){

  }

  @Cron(`*/10 * * * * *`, {
    name: 'cron1'

  })
  cron1(){
    this.loggerService.log("cron1: Acción cada 10 segundos")
  }

  @Cron(`*/30 * * * * *`, {
    name: 'cron2'

  })
  cron2(){
    this.loggerService.error("cron1: Acción cada 30 segundos")
  }

  @Cron(`* * * * *`, {
    name: 'cron3'

  })
  cron3(){
    this.loggerService.warn("cron1: Acción cada minuto")
  }

  desactivateCron(name:string){
    const job: CronJob = this.scheduleRegistry.getCronJob(name)

    if(!job){
      throw new NotFoundException("No se ha encontrado el cron")
    }else{
      job.stop()
      console.log(`El cron con el nombre: ${name} está desactivado`)
      return true
    }
  }
  activateCron(name:string){
```

```
    const job: CronJob = this.scheduleRegistry.getCronJob(name)

    if(!job){
      throw new NotFoundException("No se ha encontrado el cron")
    }else{
      job.start()
      console.log(`El cron con el nombre: ${name} está activado`)
      return true
    }
  }

  getNameCrons(){
    const names = []

    for(const name of this.scheduleRegistry.getCronJobs().keys()){
       names.push(name)
    }
    return names
  }

  desactivateAllCrons(){
    const names = this.getNameCrons()

    for(const name of names){
      this.desactivateCron(name)
    }
    return true
  }

  activateAllCrons(){
    const names = this.getNameCrons()

    for(const name of names){
      this.activateCron(name)
    }
    return true
  }
}
```

- Pongo warn y error aleatoriamente solo con fines educativos

Instalar Winston rotate

- Esta dependencia separará los logs por días

```
npm i winston-daily-rotate-file
```

- Para rotar logs por días debo cambiar el transports.File del método createLogger
- Se importa distinto

```
import 'winston-daily-rotate-file';
```

```
this.loggerInfo = createLogger({
    level: 'info',
    format:format.combine(
        dateFormat,
        textFormat
    ),
    transports:[
        new transports.DailyRotateFile({
            filename: 'log/info/info-%DATE%.log', //le añado la fecha en el nombre
del archivo
            datePattern: 'YYY-MM-DD', //formateo la fecha
            maxFiles:'7d', //le digo que conserve los archivos un máximo de 7 días
            zippedArchive: true //que comprima los archivos
        })
    ]
})
```

- Si pongo maxFiles y uso el zippedArchive, los irá manteniendo, porque no borra los .zip
- Hago lo mismo con el resto de transports

Subir Archivos NEST

- Creo el proyecto

```
nest new upload-files
```

- Instalo y configuro swagger

```
npm i @nestjs/swagger swagger-ui-express
```

- En el main añado la configuración

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

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

  const config = new DocumentBuilder()
    .setTitle('Upload files')
    .setDescription('Upload files API')
    .setVersion('1.0')
    .build();
  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('swagger', app, document);
```

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

- Dejo solo el app.module y el main (el resto de archivos los borro y los quito del app.module quedando así)
- app.module

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


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

Creando el módulo

- Creo src/modules y en modules creo el RES con **nest g res upload-file**. Si me sobran endpoints los borro y ya está
- Añado el **UploadFileModule en imports de app.module**
- No habrá conexión con DB

MulterModule

- Necesito instalar los tipos de multer, que es el paquete que viene de forma nativa para subir ficheros

```
npm i -D @types/multer
```

- En el upload-file.module importo el módulo de multer. Lo iremos cumplimentando y mejorando.

```
import { Module } from '@nestjs/common';
import { UploadFileService } from './upload-file.service';
import { UploadFileController } from './upload-file.controller';
import { MulterModule } from '@nestjs/platform-express';

@Module({
  imports:[
    MulterModule
  ],
  controllers: [UploadFileController],
  providers: [UploadFileService]
})
export class UploadFileModule {}
```

Subir un archivo

- Creo el endpoint POST para subir el archivo
- Uso el decorador **@UseInterceptors**, y con **FileInterceptor** le digo que vendrá el archivo 'file' (es lo que usaré en POSTMAN)
- Para recoger el archivo uso **@UploadedFile** del tipo Express.Multer.File

```
@Post('upload')
@UseInterceptors(FileInterceptor('file'))
uploadFile(@UploadedFile() file: Express.Multer.File) {
return this.uploadFileService.uploadFile(file);
}
```

- En el servicio, lo único que vamos a hacer es gestionar la respuesta. No hay que indicarle la acción de subirlo
- Le indico el nombre original del archivo y el filename que es el que yo le voy a asignar

```
uploadFile(file: Express.Multer.File) {
    if(file){
      const response = {
        originalName: file.originalname,
        filename: file.filename
      }
      return response
    }
    return null
  }
```

Subiendo archivo con POSTMAN/ThunderClient

- En Body/Form, en KEY le pongo el nombre file (como le puse en el FileInterceptor). Le digo que es de tipo File y selecciono el archivo a subir (cualquier .jpg)
- Así, sin configurar el MulterModule, tan solo me devuelve la respuesta, que es el originalName con el nombre original de la foto
- Configuremos el MulterModule
    - Le añado dest, de destino y le especifico una ruta. Si la carpeta no existe la creará
    - Ahora si me devuelve un filename como un id en la respuesta, sin extensión

```
@Module({
  imports:[
    MulterModule.register({
      dest: './upload'
    })
```

```
  ],
  controllers: [UploadFileController],
  providers: [UploadFileService]
})
export class UploadFileModule {}
```

- Para subir varios archivos a la vez creo otro endpoint POST en el controller
- En lugar de usar FileInterceptor usaré **FilesInterceptor** y **UploadedFiles**. Le indico que es de tipo array
- upload-file.controller

```
@Post('upload-files')
@UseInterceptors(FilesInterceptor('files'))
uploadFiles(@UploadedFiles() files: Express.Multer.File[]) {
return this.uploadFileService.uploadFiles(files);
}
```

- En el servicio uso un for of para recorrer el array de files
- Uso el método anterior para gestionar la respuesta
- En caso de que todo esté bien, hago un push al array de respuestas y lo retorno

```
uploadFiles(files: Express.Multer.File[]){
const responses = []

for(const file of files){
    const fileUpload = this.uploadFile(file)
    if(fileUpload){
    responses.push(fileUpload)
    }
}

return responses
}
```

- Ahora en ThunderClient, en Body/Form Files, en Key debo poner files (con s final, como puse en FilesInterceptor) y añado los archivos que quiera subir

---

Limitando el tamaño de los ficheros

- Le añado la propiedad limits y dentro del objeto en fileSize debo expresarlo en bytes
- Si quiero que no sobrepase los 2 Megas, lo multiplico **2 veces por 1024** (de byte pasa a KB, y de KB a MB)
- Si quiero solo subir imágenes uso **fileFilter** que es una función. Esta lleva la request (de Express), el file que subimos y el callback
- Uso una expresión regular. El $ dice que tiene que acabar en uno de estos tipos de archivos

- En caso de que no sea de alguno de estos tipos retorno el callback **con el que puedo lanzar una excepción**
- Si está bien le pongo **null** en el error, **y le digo true** (que está aceptado)

```
@Module({
  imports:[
    MulterModule.register({
      dest: './upload',
      limits:{
        fileSize: 2 * 1024 *1024
      },
      fileFilter: function(req, file, cb){
        if(!file.originalname.match(/\.(jpg|jpeg|png|gif)$/)){
            return cb(new ConflictException("Solo imágenes"), false)
        }
        return cb(null, true)
      }
    })
  ],
  controllers: [UploadFileController],
  providers: [UploadFileService]
})
export class UploadFileModule {}
```

Mejorando dónde almacenar archivos

- Uso la función diskStorage de storage
- Si quisiera filtrar los archivos por extension y guardarlos en diferentes carpetas aquí sería el sitio

```
@Module({
  imports:[
    MulterModule.register({
      limits:{
        fileSize: 2 * 1024 *1024
      },
      fileFilter: function(req, file, cb){
        if(!file.originalname.match(/\.(jpg|jpeg|png|gif)$/)){
            return cb(new ConflictException("Solo imágenes"), false)
        }
        return cb(null, true)
      },
      storage: diskStorage({
        destination: function(req,file,cb){
          cb(null, './upload')
        }
      })
    })
  ],
  controllers: [UploadFileController],
```

```
  providers: [UploadFileService]
})
export class UploadFileModule {}
```

Cambiar el nombre de los ficheros al subirlos

- Quiero añadir al archivo la fecha en la que se ha añadido
- Dentro del diskStorage uso otra función en filename
- Uso el split para dividir el archivo por el punto
- Con el slice le digo y la posición 0 al length - 1 le digo que me de todo desde el principio menos la útima parte después del punto
- Vuelvo a unir las partes (menos la extensión que ya no la tengo)
- Si tiene un mimetype hago el split separando por / y me quedo con lo que hay después del / (la extensión)
- Llamo al callback, le paso null en el error y le concateno el Date.now al nombre del archivo
- En caso de que no haya mimetype devuelveme lo mismo (con el Date.now) pero sin la extensión

```
import { ConflictException, Module } from '@nestjs/common';
import { UploadFileService } from './upload-file.service';
import { UploadFileController } from './upload-file.controller';
import { MulterModule } from '@nestjs/platform-express';
import { diskStorage } from 'multer';

@Module({
  imports:[
    MulterModule.register({
      limits:{
        fileSize: 2 * 1024 *1024
      },
      fileFilter: function(req, file, cb){
        if(!file.originalname.match(/\.(jpg|jpeg|png|gif)$/)){
            return cb(new ConflictException("Solo imágenes"), false)
        }
        return cb(null, true)
      },
      storage: diskStorage({
        destination: function(req,file,cb){
          cb(null, './upload')
        },
        filename: function(req,file,cb){
          let filenameParts = file.originalname.split('.')
          filenameParts = filenameParts.slice(0, filenameParts.length -1)
          const filename = filenameParts.join('.')

          if(file.mimetype){
            let ext = file.mimetype.split('/')[1]
            cb(null, filename + '-'+ Date.now()+ '.'+ ext)
          }else{
            cb(null, filename + '-'+ Date.now())
          }
```

```
        }
      })
    })
  ],
  controllers: [UploadFileController],
  providers: [UploadFileService]
})
export class UploadFileModule {}
```

- El mimetype se ve tipo **image/jpeg**. Puedes hacer un console.log para verlo

---

Descargar un archivo

- Para descargar creamos un GET en el controller
- Usare el response con **@Response** y el **@Body**

```
@Get('download')
download(@Response() res, @Body() body:any ) {
  console.log(res)
  console.log(body)
  return this.uploadFileService.download(res, body.filename);
}
```

- En el service coloco un return true para hacer las pruebas, me interesan los console.logs
- upload-file.service

```
download(res, filename: string){
  return true
}
```

- en el thunderClient en Body/x-www-form-urlencoded Files en la KEY añado filename y en el value el nombre del archivo a descargar
- res tiene una función llamada download donde le puedo indicar la url del fichero que quiero descargar

```
download(res, filename: string){

  return res.download('./upload/' + filename)

}
```

- Nos devuelve la imágen (si estuviera en un navegador lo abriria o lo guardaría)
- Para comprobar si existe o no podemos usar existsSync

```
download(res, filename: string){

    if(existsSync('./upload/'+ filename)){

      return res.download('./upload/' + filename)
    }

    return new NotFoundException("El fichero no existe")

  }
```

Documentando endpoints

- controller

```
import { Controller, Get, Post, Body, Patch, Param, Delete, UseInterceptors,
UploadedFile, UploadedFiles, Response } from '@nestjs/common';
import { UploadFileService } from './upload-file.service';
import { CreateUploadFileDto } from './dto/create-upload-file.dto';
import { UpdateUploadFileDto } from './dto/update-upload-file.dto';
import { FileInterceptor, FilesInterceptor } from '@nestjs/platform-express';
import { ApiOperation, ApiResponse } from '@nestjs/swagger';

@Controller('api/v1/upload-file')
export class UploadFileController {
  constructor(private readonly uploadFileService: UploadFileService) {}

  @Post('upload')
  @ApiOperation({
    description: "Sube un fichero"
  })
  @ApiResponse({
    status: 201,
    description: "Se ha subido correctamente"
  })
  @UseInterceptors(FileInterceptor('file'))
  uploadFile(@UploadedFile() file: Express.Multer.File) {
    return this.uploadFileService.uploadFile(file);
  }

  @Post('upload-files')
  @ApiOperation({
    description: "Sube varios ficheros"
  })
  @ApiResponse({
    status: 201,
    description: "Se han subido correctamente"
  })
  @UseInterceptors(FilesInterceptor('files'))
```

```
  uploadFiles(@UploadedFiles() files: Express.Multer.File[]) {
    return this.uploadFileService.uploadFiles(files);
  }

  @Get('download')
  @ApiOperation({
    description: "Descarga archivo"
  })
  @ApiResponse({
    status: 200,
    description: "Se ha descargado correctamente"
  })
  @ApiResponse({
    status: 409,
    description: "No existe el archivo"
  })
  download(@Response() res, @Body() body:any ) {
    console.log(body)
    console.log(res)
    return this.uploadFileService.download(res, body.filename);
  }

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

  @Patch(':id')
  update(@Param('id') id: string, @Body() updateUploadFileDto:
UpdateUploadFileDto) {
    return this.uploadFileService.update(+id, updateUploadFileDto);
  }

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

- Falta poder seleccionar un fichero en la documentación de swagger

---

Subir archivos en swagger

- En el upload colocamos un nuevo decorador llamado **@ApiConsumes**
- Le indicamos que tipo de "body" va a captar
- En **@ApiBody** le indico que es de tipo objeto, con la propiedadd de nombre file (como puse en FileInterceptor)

```
@Post('upload')
  @ApiOperation({
```

```
    description: "Sube un fichero"
  })
  @ApiResponse({
    status: 201,
    description: "Se ha subido correctamente"
  })
  @ApiConsumes('multipart/form-data')
  @ApiBody({
    schema: {
      type: 'object',
      properties:{
        file: {
          type: 'string',
          format: 'binary'
        }
      }
    }
  })
  @UseInterceptors(FileInterceptor('file'))
  uploadFile(@UploadedFile() file: Express.Multer.File) {
    return this.uploadFileService.uploadFile(file);
  }
```

- Con varios archivos es un poco diferente. Sería la propiedad files y sería de tipo array

```
@Post('upload-files')
  @ApiOperation({
    description: "Sube varios ficheros"
  })
  @ApiResponse({
    status: 201,
    description: "Se han subido correctamente"
  })
  @ApiConsumes('multipart/form-data')
  @ApiBody({
    schema: {
      type: 'object',
      properties:{
        files: {
          type: 'array',
          items:{
            type: 'string',
            format: 'binary'
          }
        }
      }
    }
  })
  @UseInterceptors(FilesInterceptor('files'))
  uploadFiles(@UploadedFiles() files: Express.Multer.File[]) {
    return this.uploadFileService.uploadFiles(files);
  }
```

- el download la propiedad es filename, y es de tipo string

```
@Get('download')
  @ApiOperation({
    description: "Descarga archivo"
  })
  @ApiResponse({
    status: 200,
    description: "Se ha descargado correctamente"
  })
  @ApiResponse({
    status: 409,
    description: "No existe el archivo"
  })
  @ApiConsumes('multipart/form-data')
  @ApiBody({
    schema: {
      type: 'object',
      properties:{
        filename: {
          type: 'string',
        }
      }
    }
  })
  download(@Response() res, @Body() body:any ) {
    console.log(body)
    console.log(res)
    return this.uploadFileService.download(res, body.filename);
  }
```

NEST Microservicios

- Este módulo trata de comunicar dos backends mediante mensajes
- Vamos a controlar desde ambos si hay o no conexión
- Qué es un microservicio?
    - Es un tipo de arquitectura que nos permite conectarnos a diferentes sitios y obtener info4
    - Sirven para comunicar a los backends entre si
    - Desde el microservicio, a través de TCP se enviará y recibirá información entre uno y otro backend
    - Es el microservicio quien realmente se comunica, no el backend en si (aunque haga las tareas)
    - Puedes dividir el peso de la aplicación en diferentes backends, así si falla algo no se cae toda la app

Creando el proyecto

- Crearemos dos proyectos con nest new microservicios-1 y 2
- Me abro dos vscode, uno con el backend1 y otro con el 2
- Instalo las dependencias necesarias en ambos backends por separado

> npm i @nestjs/microservices

- Creamos los módulos necesarios en ambos backends. Un módulo con controlador y servicio
- NOTA: Lo hago con res, prefiero borrar lo que no necesito

> nest g res example-communication nest g res microservice-connection

- En el controller de example-communication pongo 'api/v1/microservices-b1' y 'api/v1/microservices-b2' en el del segundo backend
- Borro los .spec y borro tambien app.service, y app.controller, los quito del app.module
- Me aseguro de que ha importado los módulos example-communication y microservice-connection en el imports de app.module

---

Creando nuestros microservicios

- Para crear el microservicio voy al main.ts
- Voy a crear un microservicio que se conectará a mi mismo, que me servirá para que el otro conecte con su respectivo microservicio
- Para algo unidireccinal me puede valer el método .createtMicroservice
- Cuando quiero hacerlo bidireccional, como es el caso, es mejor esta manera
- Le indico el tipo de transporte, y en options el host (a si mismo) y un puerto aleatorio (en este caso 3032)
- Este 3032, cuando hagamos el otro microservicio en el backend 2, **apuntaremos a este 3032**
- No estamos conectando microservicios, de momento. Apunta a si mismo

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

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

  app.connectMicroservice({
    transport: Transport.TCP,
    options: {
      host: '0.0.0.0',
      port: 3032
    }
  })

  await app.startAllMicroservices()
  await app.listen(3000);
```

```
}
bootstrap();
```

- Copio y pego en el backend2, pongo de puerto 3030
- No se pueden hacer peticiones a este puerto. Lo usará el microservicio para conectarse y pasarse datos

---

Creando la conexión de nuestros microservicios

- Creo el cliente de tipo ClientProxy. Es quién va a hacer la mágia de la conexión
- ClientProxyFactory es el que crea el transporte. Le indico lo mismo que en el main y lo conecto al puerto del microservicio del backend2
- Creo un get para obtener el cliente

```
import { Injectable } from '@nestjs/common';
import { ClientProxy, ClientProxyFactory, Transport } from
'@nestjs/microservices';

@Injectable()
export class MicroserviceConnectionService {
    private client: ClientProxy

    constructor(){
        this.client = ClientProxyFactory.create({
            transport: Transport.TCP,
            options:{
                host: '0.0.0.0',
                port: 3030
            }
        })
    }

    getClient(){
        return this.client
    }
}
```

- Hago lo mismo en el service del backend2 y lo conecto al microservicio del backend1 (al puerto 3032)

Creando el endpoint

- En ExampleCommunicationController

```
import { Controller, Get } from '@nestjs/common';
import { ExampleCommunicationService } from './example-communication.service';
import { CreateExampleCommunicationDto } from './dto/create-example-
communication.dto';
import { UpdateExampleCommunicationDto } from './dto/update-example-
```

```
communication.dto';

@Controller('api/v1/microservices-b1')
export class ExampleCommunicationController {
  constructor(private readonly exampleCommunicationService:
ExampleCommunicationService) {}

  @Get('send-message')
  sendMessage(){
    return this.exampleCommunicationService.sendMessage('hola')
  }

}
```

- En el service

```
import { Injectable } from '@nestjs/common';
import { CreateExampleCommunicationDto } from './dto/create-example-
communication.dto';
import { UpdateExampleCommunicationDto } from './dto/update-example-
communication.dto';

@Injectable()
export class ExampleCommunicationService {

  sendMessage(msg: string){
    return msg
  }
}
```

- Hago lo mismo en el backend2

Patterns 2

- Son los mensajes que tienen que cuadrar en un backend y otro para que sepan que evento están recibiendo
- Sirve para saber adónde tiene que ir (con indicarle el puerto en el Transport no es suficiente)
- Es como una dirección, con la ciudad (el puerto) no es suficiente, necesito saber en que calle
- Creo el archivo example-communication.constants.ts
- Puedo usar cualquier palabra en el objeto, uso controller

```
export const PATTERNS ={
    MESSAGES:{
        SEND_MESSAGE: { controller: 'sendMessage'}
    },
    EVENTS:{
        RECEIVE_MESSAGE: { controller: 'receiveMessage'}
```

```
    }
}
```

- Copio el archivo en el backend2

MESSAGE PATTERN Backend1 al Backend2

- Usaremos **MessagePattern** en el service en el ExampleCommunicationService
- En el constructor inyecto el MicroserviceConnectionService
- Debo **importar el módulo** de *MicroserviceConnection* **en el imports** de example-communication.module.ts y **colocar el servicio en provider**
- Uso el servicio **.getClient** con el **send**
- El send es lo que tenemos para comunicar eventos
- Con el send le voy a mandar un pattern y un objeto al controlador del backend2, que tiene que estar escuchando ese patrón, y luego vuelve al backend1
- Uso **firstValueFrom**

```
import { Injectable } from '@nestjs/common';
import { MicroserviceConnectionService } from 'src/microservice-
connection/microservice-connection.service';
import { PATTERNS } from './example-communication.constants';
import { firstValueFrom } from 'rxjs';

@Injectable()
export class ExampleCommunicationService {

  constructor(private microServiceConnection:MicroserviceConnectionService){ }

  sendMessage(msg: string){
    return firstValueFrom(
      this.microServiceConnection
      .getClient()
      .send(
        PATTERNS.MESSAGES.SEND_MESSAGE,
        {
          msg
        }
      )
      )
  }
}
```

- Ahora en el controller del backend2 usamos **@MessagePattern** y le paso el mismo pattern que he puesto en el send
- En el objeto data lo tipo con el mismo nombre (msg)
- controller backend2

```
import { Controller, Get, Post, Body, Patch, Param, Delete } from
'@nestjs/common';
import { ExampleCommunicationService } from './example-communication.service';
import { MessagePattern } from '@nestjs/microservices';
import { PATTERNS } from './example-communication.constants';

@Controller('api/v1/microservices-b2')
export class ExampleCommunicationController {
  constructor(private readonly exampleCommunicationService:
ExampleCommunicationService) {}

  @Get('send-message')
  sendMessage(){
    return this.exampleCommunicationService.sendMessage('adiós')
  }

  @MessagePattern(PATTERNS.MESSAGES.SEND_MESSAGE)
  receivemessageFromMessagePatternB1(data: {msg: string}){
    console.log("Mensaje de B1 recibido", data.msg)
    return true
  }
}
```

- Si apunto al endpoint del b1 'send-message' enviará el mensaje

> http://localhost:3000/api/v1/microservices-b1/send-message