01- NEST MICROSERVICIOS - Gateway y Manejo de Errores

- El primer microservicio lo trabajaremos con SQLite (productos)
- products/entities/product.entity

```
export class Product {


  public id: number;

  public name: string;

  public price: number;

}
```

- Hay que instalar class-validator y class-transformer
- También hay que configurarlo en el main

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

- products/dtos/create-product.dto

```
import { Type } from 'class-transformer';
import { IsNumber, IsString, Min } from 'class-validator';

export class CreateProductDto {

  @IsString()
  public name: string;

  @IsNumber({
    maxDecimalPlaces: 4, //máximo de decimales
  })
  @Min(0)
  @Type(() => Number ) //lo casteo a número
  public price: number;

}
```

Configurar variables de entorno

- Instalo dotenv y joi
- /config/envs.ts

```
import 'dotenv/config';
import * as joi from 'joi';

interface EnvVars {
  PORT: number;
  DATABASE_URL: string;
}

const envsSchema = joi.object({
  PORT: joi.number().required(),
  DATABASE_URL: joi.string().required(),
})
.unknown(true); //hay muchas más variables de entorno flotando en mi aplicación
(el path de node, etc)


//hago la validación extrayendo los valores con desestructuración
const { error, value } = envsSchema.validate( process.env );


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

//En enVars de tipo Envars (interface) guardo los valores que desestructuré del
Schema
const envVars:EnvVars = value;

//Exporto las variables en un objeto
export const envs = {
  port: envVars.PORT,
  databaseUrl: envVars.DATABASE_URL,
}
```

- Es recomendable crear un snippet con este código (usar easySnippet)
- Creo el .env y el .env.template

```
PORT=3001
DATABASE_URL="file:./dev.db"
```

- Uso las variables donde corresponde (en el main, uso await app.listen(envs.port))

- Cuando configuremos el microservicio lo pondremos **en otro lugar**

Prisma SQLite

- Instalo prisma como dependencia de desarrollo (prisma crea un cliente)
- Para inciar prisma

> npx prisma init

- Produjo una cadena de conexión para postgres en .env
- Lo borro y coloco la de SqLite

```
DATABASE_URL="file:./dev.db"
```

- En schema.prisma (instalar extensión de sintaxis de prisma)
- Luce como js (o ts) pero no lo es! Es sintaxis propia de prisma

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

// Looking for ways to speed up your queries, or scale easily with your serverless
or edge functions?
// Try Prisma Accelerate: https://pris.ly/cli/accelerate-init

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"  //La db!!
  url      = env("DATABASE_URL") //le paso la variable de entorno
}

model Product {
  id    Int    @id @default(autoincrement()) //Agrego un id autoincrementado, el
Int va a ser el identificador de id (@id)
  name  String
  price Float

  available Boolean @default(true)

  createdAt DateTime @default(now()) //uso now para usar la fecha del momento
  updatedAt DateTime @updatedAt

  @@index([available]) //indexo avaliable para que no aparezcan en las búsquedas
}
```

- Ejecuto la migración

> npx prisma migrate dev --name init

- Me crea la db
- Instalo el @prisma/client
- Voy al servicio y lo extiendo de PrismaCLient e implementaré OnModuleInit

```
@Injectable()
export class ProductsService extends PrismaClient implements OnModuleInit {

    onModuleInit() {
    this.$connect();  //Database connected!!
  }
}
```

- Ya podemos empezar a trabajar con la DB
- Puedo crear un logger para mejorar los logs

```
@Injectable()
export class ProductsService extends PrismaClient implements OnModuleInit {

  private readonly logger = new Logger('ProductsService'); //creo un logger con el
cabezal de ProductsService

  onModuleInit() {
    this.$connect();
    this.logger.log('Database connected'); //mejoro el log!
  }
}
```

- También lo uso en el main de la misma forma

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { Logger, ValidationPipe } from '@nestjs/common';
import { envs } from './config';
import { MicroserviceOptions, Transport } from '@nestjs/microservices';

async function bootstrap() {

  const logger = new Logger('Main');

  //const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    //AppModule,
    //{
      //transport: Transport.TCP,
      //options: {
        //port: envs.port
      //}
```

```
    //}
  //);

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

  await app.listen();
  logger.log(`Products Microservice running on port ${ envs.port }`); //el logger!

}
bootstrap();
```

Insertar y comprobar la DB

- createdAt ya tioene un valor por defecto
- Creo el método create en el servicio
- Es this.product porque la clase (el servicio) extiende de PrismaClient y el modelo se llama Product
- Uso el método create para insertar. regresa una promesa con el registro insertado
- El producto espera un name y un price

```
create(createProductDto: CreateProductDto) {

return this.product.create({
    data: createProductDto
});

}
```

- Para ver la DB puedo usar Abrir con.. "TablePlus"
- Para añadir data en Table PLus crear la nueva conexión con SQLite e importar el archivo desde la interfaz

---

Obtener productos y paginarlos

- Creo el dto de paginación en common/dto/pagination.dto
- Hago ambos opcionales y casteo el tipo a number
- Les pongo valores por defecto

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

export class PaginationDto {
```

```
  @IsPositive()
  @IsOptional()
  @Type(() => Number)
  page?: number = 1;

  @IsPositive()
  @IsOptional()
  @Type(() => Number)
  limit?: number = 10;

}
```

- En el servicio

```
async findAll( paginationDto: PaginationDto ) {

    const { page, limit } = paginationDto;

    const totalProducts = await this.product.count({ where: { available: true } 
}); //para contar los productos disponibles
    const lastPage = Math.ceil( totalProducts / limit ); //divido el total de 
páginas (número de productos disponibles) por el limite
                                                         //.ceil redondea al siguiente 
número positivo

    return {
      data: await this.product.findMany({
        //skip = 0 * (limit = 10) = 0 primera posición del arreglo páginas tengo 
1,2,3
        //si estoy en la página 2 = (2-1) == 1 * limit ===10, skip 10 registros

        skip: ( page - 1 ) * limit,
        take: limit,
        where: {
          available: true //listo los que están disponibles
        }
      }),
      //meta de metadata
      meta: {
        total: totalProducts, //el resultado de .count de la cantidad de productos 
disponibles
        page: page,
        lastPage: lastPage, //el total de páginas en el documento
      }
    }
  }
```

Retornar producto por id

```
async findOne(id: number) {
    const product =  await this.product.findFirst({
      where:{ id, available: true }
    });

    if ( !product ) {
      throw new BadRequestException(`No hay ningún producto con el id ${id}`);
    }

    return product;

  }
```

Actualizar

- El dto
- Uso PATCH /:id

```
  async update(id: number, updateProductDto: UpdateProductDto) {

    await this.findOne(id);

    return this.product.update({
      where: { id },
      data: updateProductDto,
    });
  }
```

Eliminación

- Por lo general no voy a querer borrar un producto porque no sé que microservicios pueden tener relaciones con ese producto
- Esto podría generar una serie de errores en cascada
- Hago un borrado lógico. Cambio el avaliable a false

```
async remove(id: number) {

    await this.findOne(id);

    // return this.product.delete({
    //   where: { id }
    // });

    const product = await this.product.update({
      where: { id },
```

```
      data: {
        available: false
      }
    });

    return product;


  }
```

Transformar a microservicio

- Instalo @nestjs/microservices
- Para crear el microservicio, en el main creo app con NestFactory
- Le mando el AppModule y el objeto de configuración

```
import { MicroserviceOptions, Transport } from '@nestjs/microservices';

async function bootstrap() {

  const logger = new Logger('Main');


const app = await NestFactory.createMicroservice<MicroserviceOptions>(

    AppModule,
    {
      transport: Transport.TCP, //Elijo que tipo de transporte quiero usar
      options: {
        port: envs.port
      }
    }
  );
}
```

- Puedo usar **app.startAllMicroservices()** para inciar todos los microservicios
- En este momento esto haría mi aplicación híbrida entre REST y microservicios (**ES COMPATIBLE**)
- Pero yo no quiero que esto sea un híbrido por lo que no usaré este comando
- Si te fijas en consola ya no aparacen los endpoints GET POST PATCH DELETE que se habían incializado
- Ya no estamos escuchando peticiones HTTP en ese puerto (pese a que ahi está el microservicio)
- Para comunicarnos tenemos los **eventos** y la **mensajería**
- *@MessagePattern* es "**te envío la pelota, regrésame la pelota con la información, ya puedo seguir con mi tarea**"
- *@EventPattern* es "**yo te mando el evento, y lo que suceda ahí ya es cosa tuya a mi me importa poco, sigo con mi vida**"
- En el controlador

```
import { Controller, ParseIntPipe } from '@nestjs/common';
import { ProductsService } from './products.service';
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';
import { PaginationDto } from 'src/common';
import { MessagePattern, Payload } from '@nestjs/microservices';

@Controller('products')
export class ProductsController {
  constructor(private readonly productsService: ProductsService) {}

  // @Post()
  @MessagePattern({ cmd: 'create_product' }) //lo que hay en cmd es el string que
me servirá para comunicar los microservicios
  create(@Payload() createProductDto: CreateProductDto) { //en el Payload está la
información
    return this.productsService.create(createProductDto);
  }

  // @Get()
  @MessagePattern({ cmd: 'find_all_products' })
  findAll(@Payload() paginationDto: PaginationDto) {
    return this.productsService.findAll(paginationDto);
  }

  // @Get(':id')
  @MessagePattern({ cmd: 'find_one_product' })
  findOne(@Payload('id', ParseIntPipe) id: number) {
    // { id: 1
    return this.productsService.findOne(id);
  }

  // @Patch(':id')
  @MessagePattern({ cmd: 'update_product' })
  update(
    // @Param('id', ParseIntPipe) id: number,
    // @Body() updateProductDto: UpdateProductDto,
    @Payload() updateProductDto: UpdateProductDto,
  ) {
    return this.productsService.update(updateProductDto.id, updateProductDto);
  }

  // @Delete(':id')
  @MessagePattern({ cmd: 'delete_product' })
  remove(@Payload('id', ParseIntPipe) id: number) { //puedo usar el ParseIntPipe
con el id del payload!!
    return this.productsService.remove(id);
  }
}
```

- Para la actualización ya no tengo **@Params**, viene todo en el **@Payload**
- Modifico esto, hago un dto para el update

```
import { PartialType } from '@nestjs/mapped-types';
import { CreateProductDto } from './create-product.dto';
import { IsNumber, IsPositive } from 'class-validator';

                                    //PartialType hace todas las propiedades del
dto padre opcionales
export class UpdateProductDto extends PartialType(CreateProductDto) {

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

}
```

- Cambio el método en el servicio

```
async update(id: number, updateProductDto: UpdateProductDto) {

    //ya tengo el id en id:updateProduct.id desde el controlador en la variable id
    //extraigo la data y le quito el id pq no me interesa
    const { id: __, ...data } = updateProductDto; //el id lo renombro a guión bajo
pq no me interesa


    await this.findOne(id);

    return this.product.update({
      where: { id },
      data: data,
    });
  }
```

- Cuando el microservicio A quiera comunicarse con el microservicio B va a tener que usar el mismo objeto **{cmd:'lo que sea'}**
- Ahora uso **@Payload** para la información
- **Podría mantener el @POST o el @GET si fuera un híbrido**, iniciando desde el main con **.startAllMicroservices**
- Esto será útil con la autenticación, ya que puede ser una API propia o un microservicio
- No tengo porqué mandar el mensaje como un cmd:"string", puedo colocar solo un string, pero el standard es el objeto con cmd
    - **Ya está el microservicio implementado. Cómo lo vamos a probar?**
    - **Lo vamos a probar mediante un GATEWAY, va a ser el punto intermedio. Crearemos un API REST donde mis clientes se van a conectar y este GATEWAY se va a encargar de comunicarse mediante los microservicios usando TCP**

Github Organization

- Podemos crear una organización para agrupar todos los microservicios
- En Github Your Organizations /New Organization / Free Organization
- No invito a nadie (skip this step)
- Creo un nuevo repo

02- NEST MICROSERVICES - CLIENT-GATEWAY

- Instalo dotenv joi @nestjs/microservices
- Tenemos que crear algo muy parecido al REST API de productos
- Es el cliente quien se va aq conectar al microservicio de productos
- Configuro las variables de entorno (uso el snippet)
- config/envs.ts

```
import 'dotenv/config';

import * as joi from 'joi';

//hago la interfaz
interface EnvVars {
  PORT: number;
  PRODUCTS_MICROSERVICE_HOST: string;
  PRODUCTS_MICROSERVICE_PORT: number;
}

//creo el Schema
const envsSchema = joi.object({
  PORT: joi.number().required(),
  PRODUCTS_MICROSERVICE_HOST: joi.string().required(),
  PRODUCTS_MICROSERVICE_PORT: joi.number().required(),

})
.unknown(true); //para el resto de variables en process.env

//desestructuro el error y value
const { error, value } = envsSchema.validate( process.env );


//si hay algún error lo lanzo
if ( error ) {
  throw new Error(`Config validation error: ${ error.message }`);
}

//guardo el valor en una variable que cumple con la interfaz
const envVars:EnvVars = value;


//exporto las variables en un objeto
export const envs = {
  port: envVars.PORT,
```

```
  productsMicroserviceHost: envVars.PRODUCTS_MICROSERVICE_HOST,
  productsMicroservicePort: envVars.PRODUCTS_MICROSERVICE_PORT,
};
```

- .env

```
PORT=3000


PRODUCTS_MICROSERVICE_HOST=localhost
PRODUCTS_MICROSERVICE_PORT=3001
```

- Creemos las rutas

> nest g res products

- El cliente **SI ES UN RESTFULL API**
- No voy a necesitar el servicio
- Pongo a correr el microservicio de products
- Creo la conexión en products.module del CLIENTE_GATEWAY, registro el microservicio en imports

```
import { Module } from '@nestjs/common';
import { ProductsController } from './products.controller';
import { ClientsModule, Transport } from '@nestjs/microservices';
import { PRODUCT_SERVICE, envs } from 'src/config';

@Module({
  controllers: [ProductsController],
  providers: [],
  imports: [
    ClientsModule.register([

      {
        name: PRODUCT_SERVICE, //variable en /config/services (injection token)
        transport: Transport.TCP, //uso el mismo canal de comunicación que usa el
microservicio de productos
        options: {
          host: envs.productsMicroserviceHost, //localhost
          port: envs.productsMicroservicePort //3001, el de products-microservice
        }
      },

    ]),
  ]
})
export class ProductsModule {}
```

- Fijarse que .register abre un arreglo en el que puedes registrar todos los microservicios que necesites

- En app.module (del client-gateway) solo tengo ProductsModule
- Creo el fichero /config/services.ts donde coloco la variable (o injection token) que va a identificar a el transport que voy a colocar. Viene a ser la definición de mi microservicio

```
export const PRODUCT_SERVICE = 'PRODUCT_SERVICE';
```

- Es lo que vamos a necesitar para inyectar el (micro)servicio en los controladores
- Lo guardamos en una variable para no tener problemas de errores con el string
- Creo un archivo de barril en config/index.ts

```
export * from './envs';
export * from './services';
```

Obtener todos los productos

- Si estuviera trabajando las variables de entorno con ConfigModule tendria que usar r .registerAsync() en products.module del gateway, inyectar el ConfigModule...
- Para conectar con findProducts inyectamos el microservicio de products en el controlador
- client-gateway-products.controller

```
import { ClientProxy} from '@nestjs/microservices';

@Controller('products')
export class ProductsController {
  constructor(
    @Inject(PRODUCT_SERVICE) private readonly productsClient: ClientProxy,
  ) {}
}
```

- En el controlador findAllProducts llamo al microservicio
- Si espero una respuesta uso **.send**
- Si no espero una respuesta uso **.emit**
- Le paso exactamente lo mismo que puse entre paréntesis del @MessagePattern
- En este caso de segundo argumento le paso un objeto vacío
- Es porque está esperando el payload, que en este caso es el paginationDto

```
@Get()
  findAllProducts() {
    return this.productsClient.send(
      { cmd: 'find_all_products' }, {}
    );
  }
```

- En products-micro-service.controller

```
@MessagePattern({ cmd: 'find_all_products' })
  findAll(@Payload() paginationDto: PaginationDto) {
    return this.productsService.findAll(paginationDto);
  }
```

- Los valores del paginationDTo son opcionales
- Para incluir los query parameters que introduzco en la url "products?page=2&limit=10" uso **@Query**
- client-gateway.products.controller

```
@Get()
findAllProducts(@Query() paginationDto: PaginationDto) {
return this.productsClient.send(
    { cmd: 'find_all_products' },
    paginationDto,
);
}
```

- Resumiendo:
    - En el products.module del cliente registro el microservicio con **ClientsModule.register** que abre un arreglo
    - Le paso el **token de inyección** que usaré en el controlador para inyectar el microservicio (no es más que una variable que me sirve para identificar el microservicio que estoy registrando en products.module del cliente-gateway), el mismo tipo de transporte que usa dicho microservicio, y dentro de options le paso el puerto del microservicio y el host (en este caso localhost)
    - Inyecto el microservicio en el controlador del cliente usando **@Inject(token de inyección)** y el tipado es client: **ClientProxy**
    - Para comunicarme con el controlador de products y poder usarlo, usaré **.send** si espero una respuesta, y cómo argumento primero el objeto literal que hay en **@MessagePattern** y lo que sea que me pide (**Payload**)

Manejo de excepciones

- Para buscar por id desde el controlador del cliente-gateway debo pasarle el mismo objeto que en el @MessagePattern del servicio de productsy el @Payload, en este caso el id
- Uso con **productsClient: ClientProxy** al que le he inyectado el token de PRODUCT_SERVICE, con **.send** porque espero respuesta
- Este .send es un **Observable** (devuelve un Observable). No es más que **un flujo de información**
- Para escuchar los Observables necesito el **.subscribe()** y mandarle la respuesta y retornarla

```
.subscribe(res=>{
  //return res
})
```

- Esa como se trabajan comunmente los Observables
- Estoy lanzando un NotFoundException desde el servicio de products.microservice, pero los errores son atrapados en los RpcException cuando usamos .send y microservicios
- Vamos a crear un ExceptionFilter para atrapar todos los Rpc
- Primero solucionémoslo de manera empírica
- Lo meto en un try catch y uso el .firstValueFrom de rxjs que me permite trabajar como una promesa el Observable
- Le estoy diciendo "espera el primer valor que este Observable va a emitir"
- Puedo usar .pipe con catchError o puedo capturarlo en el catch -client-gateway.products.controller

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

    try{
      const product= await firstValuefrom(
        this.productsClient.send({ cmd: 'find_one_product' }, { id })
      )

      return product

    }catch(error){
      throw new BadRequestException(error)
    }


  }
```

- Pero así no estoy atrapando la RpcException

ExceptionFilter

- Las excepciones van a estar continuamente manejadas como RpcException, a veces puede que manden un string y no un objeto
- Vamos a hacer que las excepciones siempre sean manejadas como objetos y no solo como strings
- Uso el decorador **@Catch()** y le paso el RpcException de @nestjs/microservices
- Implemento la clase **ExceptionFilter**
- En el metodo **catch** le paso la **RpcException** y el **host: ArgumentsHost**
- Creo el **context** usando el **host** y **.switchToHttp()**
- Obtengo la **response** (la respuesta) con el context **.getResponse()**
- **Guardo el error** usando la RpcException que pasé como parámetro al catch con **.getError()**
- Si el error es un objeto y contiene status y message **me aseguro de que el status es un número** y si no lo casteo
- **Retorno la response que obtuve del context** con **.status** y **envío el error con .json**
- Si no pasa las validaciones **genero yo la respuesta como un objeto** pasándole **status y messsage**
- **El global exceptionFilter no está disponible en aplicaciones híbridas**

- El exceptionFilter esta **fuera del Exception zone**
- /common/exceptions/roc-custom-exception.filter.ts

```
import { Catch, ArgumentsHost, ExceptionFilter } from '@nestjs/common';

import { RpcException } from '@nestjs/microservices';

@Catch(RpcException)
export class RpcCustomExceptionFilter implements ExceptionFilter {
  catch(exception: RpcException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse();

    const rpcError = exception.getError();

    if (
      typeof rpcError === 'object' &&
      'status' in rpcError &&
      'message' in rpcError
    ) {
      const status = isNaN(+rpcError.status) ? 400 :+rpcError.status;
      return response.status(status).json(rpcError);
    }

    response.status(400).json({
      status: 400,
      message: rpcError,
    });
  }
}
```

- Lo coloco en el main par aplicarlo

```
app.useGlobalFilters(new RpcCustomExceptionFilter())
```

- En el controlador del cliente estaba atrapando la excepción con un try catch y mandando un BadRequest
- Si hago un console.log del error no tengo una instancia de RpcException si no un objeto con status y message
- Debo enviar el RpcException para poder mandarlo en la respuesta
- client-gateway.products.controller

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

    try{
      const product= await firstValuefrom(
```

```
        this.productsClient.send({ cmd: 'find_one_product' }, { id })
      )

      return product

    }catch(error){
      throw new RpcException(error)
    }
  }
```

- En product-microservice.service lanzo el RpcException con el message y el status

```
async findOne(id: number) {
  const product =  await this.product.findFirst({
    where:{ id, available: true }
  });

  if ( !product ) {
    throw new RpcException({
      message: `Product with id #${ id } not found`,
      status: HttpStatus.BAD_REQUEST //HttpStatus de @nestjs/common
    });
  }

  return product;

}
```

- En el cliente puedo usar .pipe con catchError para atrapar la RpcException que ha pasado por el ExceptionFilter
- .pipe viene en los observables, catchError viene de rxjs
- client-gateway.products.controller

```
@Get(':id')
async findOne(@Param('id') id: string) {
  return this.productsClient.send({ cmd: 'find_one_product' }, { id }).pipe(
    catchError((err) => {
      throw new RpcException(err);
    }),
  );
}
```

- Se puede trabajar como Observable con .pipe o como promesa con try catch

---

Implementar métodos faltantes

- Creación, borrado y actualizacion en el client-gateway.products.controller

```
import {
  BadRequestException,
  Body,
  Controller,
  Delete,
  Get,
  Inject,
  Param,
  ParseIntPipe,
  Patch,
  Post,
  Query,
} from '@nestjs/common';
import { ClientProxy, RpcException } from '@nestjs/microservices';
import { catchError, firstValueFrom } from 'rxjs';
import { PaginationDto } from 'src/common';
import { PRODUCT_SERVICE } from 'src/config';
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';

@Controller('products')
export class ProductsController {
  constructor(
    @Inject(PRODUCT_SERVICE) private readonly productsClient: ClientProxy,
  ) {}

  @Post()
  createProduct(@Body() createProductDto: CreateProductDto) {
    return this.productsClient.send(
      { cmd: 'create_product' },
      createProductDto,
    );
  }

  @Get()
  findAllProducts(@Query() paginationDto: PaginationDto) {
    return this.productsClient.send(
      { cmd: 'find_all_products' },
      paginationDto,
    );
  }

  @Get(':id')
  async findOne(@Param('id') id: string) {
    return this.productsClient.send({ cmd: 'find_one_product' }, { id }).pipe(
      catchError((err) => {
        throw new RpcException(err);
      }),
    );

    // try {
```

```
    //   const product = await firstValueFrom(
    //     this.productsClient.send({ cmd: 'find_one_product' },{ id })
    //   );
    //   return product;

    // } catch (error) {
    //   throw new RpcException(error);
    // }
  }

  @Delete(':id')
  deleteProduct(@Param('id') id: string) {
    return this.productsClient.send({ cmd: 'delete_product' }, { id }).pipe(
      catchError((err) => {
        throw new RpcException(err);
      }),
    );
  }

  @Patch(':id')
  patchProduct(
    @Param('id', ParseIntPipe) id: number, //casteo el id
    @Body() updateProductDto: UpdateProductDto,
  ) {
    return this.productsClient
      .send(
        { cmd: 'update_product' },
        {
          id,
          ...updateProductDto,
        },
      )
      .pipe(
        catchError((err) => {
          throw new RpcException(err);
        }),
      );
  }
}
```

- Paso el servicio de products.microservices

```
import { HttpStatus, Injectable, Logger, OnModuleInit } from '@nestjs/common';
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';
import { RpcException } from '@nestjs/microservices';
import { PrismaClient } from '@prisma/client';

import { PaginationDto } from 'src/common';

@Injectable()
export class ProductsService extends PrismaClient implements OnModuleInit {
```

```
  private readonly logger = new Logger('ProductsService');

  onModuleInit() {
    this.$connect();
    this.logger.log('Database connected');
  }

  create(createProductDto: CreateProductDto) {

    return this.product.create({
      data: createProductDto
    });

  }

  async findAll( paginationDto: PaginationDto ) {

    const { page, limit } = paginationDto;

    const totalPages = await this.product.count({ where: { available: true } });
    const lastPage = Math.ceil( totalPages / limit );

    return {
      data: await this.product.findMany({
        skip: ( page - 1 ) * limit,
        take: limit,
        where: {
          available: true
        }
      }),
      meta: {
        total: totalPages,
        page: page,
        lastPage: lastPage,
      }
    }
  }

  async findOne(id: number) {
    const product =  await this.product.findFirst({
      where:{ id, available: true }
    });

    if ( !product ) {
      throw new RpcException({
        message: `Product with id #${ id } not found`,
        status: HttpStatus.BAD_REQUEST
      });
    }

    return product;

  }
```

```
  async update(id: number, updateProductDto: UpdateProductDto) {

    const { id: __, ...data } = updateProductDto;


    await this.findOne(id);

    return this.product.update({
      where: { id },
      data: data,
    });


  }

  async remove(id: number) {

    await this.findOne(id);

    // return this.product.delete({
    //   where: { id }
    // });

    const product = await this.product.update({
      where: { id },
      data: {
        available: false
      }
    });

    return product;


  }
}
```

- Paso también el products-microservice.controller

```
import { Controller, ParseIntPipe } from '@nestjs/common';
import { ProductsService } from './products.service';
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';
import { PaginationDto } from 'src/common';
import { MessagePattern, Payload } from '@nestjs/microservices';

@Controller('products')
export class ProductsController {
  constructor(private readonly productsService: ProductsService) {}

  // @Post()
  @MessagePattern({ cmd: 'create_product' })
```

```
  create(@Payload() createProductDto: CreateProductDto) {
    return this.productsService.create(createProductDto);
  }

  // @Get()
  @MessagePattern({ cmd: 'find_all_products' })
  findAll(@Payload() paginationDto: PaginationDto) {
    return this.productsService.findAll(paginationDto);
  }

  // @Get(':id')
  @MessagePattern({ cmd: 'find_one_product' })
  findOne(@Payload('id', ParseIntPipe) id: number) {
    // { id: 1
    return this.productsService.findOne(id);
  }

  // @Patch(':id')
  @MessagePattern({ cmd: 'update_product' })
  update(
    // @Param('id', ParseIntPipe) id: number,
    // @Body() updateProductDto: UpdateProductDto,
    @Payload() updateProductDto: UpdateProductDto,
  ) {
    return this.productsService.update(updateProductDto.id, updateProductDto);
  }

  // @Delete(':id')
  @MessagePattern({ cmd: 'delete_product' })
  remove(@Payload('id', ParseIntPipe) id: number) {
    return this.productsService.remove(id);
  }
}
```

- Resumen de la comunicación:
  - En el products-microservice.controller creo el objeto de @MessagePattern que me servirá para comunicarme con el cliente
  - Le paso al servicio los parámetros que necesita que vendrán del cliente
  - En el cliente inyectando el ClientProxy en el controlador, uso .send y le paso en un objeto el objeto (literal) del MessagePattern
    - Y en otro objeto el Payload del products.microservice.controller (no disponemos de @Params, @Body, etc) con lo que necesita
  - Cuando trabajo con .send trabajo con Observables, para manejar los errores debo usar RpcException
  - Puedo trabajar los Observables como promesas con try catch y async await
- Evidentemente hay más pasos con la conexión, el registro del cliente, etc
- Pero la comunicación viene a ser esa

03- NEST MICROSERVICIOS - ORDERS

- Trabajaremos con PostgreSQL
- Las órdenes solo será el header de las órdenes
- En otro microservicio con Mongo tendremos el detalle
- Lo quiero manetener independiente para que puedan escalar sin manetener una relación entre si
- Creo un nuevo proyecto de Nest con **nest new**
- Levanto el gateway (da error porque no tengo Productos levantado)
- Configuro las variables de entorno de Orders, instalo **joi y dotenv**
- /config/envs.ts

```
import 'dotenv/config';

import * as joi from 'joi';

interface EnvVars {
  PORT: number;
}

const envsSchema = joi.object({
  PORT: joi.number().required(),
})
.unknown(true);

const { error, value } = envsSchema.validate( process.env );


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

const envVars:EnvVars = value;


export const envs = {
  port: envVars.PORT,
};
```

- Coloco el puerto en el main para que corra en el 3002 y no de error
- Creo el logger en el main con **new Logger**

Configuración

- Necesito crear los canales de comunicación para poder crear4 una orden similar a un CRUD pero con el MessagePattern y falta la instalación de los microservicios
- Instalo @nestjs/microservices
- Configuro el main con .createMicroservice al que le paso el AppModule y el objeto de configuración del microservicio con el tipo de transporte y el puerto en el objeto options

```
import { Logger, ValidationPipe } from '@nestjs/common';
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { envs } from './config';
import { MicroserviceOptions, Transport } from '@nestjs/microservices';

async function bootstrap() {
  const logger = new Logger('OrdersMS-Main');

  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      transport: Transport.TCP,
      options: {
        port: envs.port,
      },
    },
  );

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

  await app.listen();
  logger.log(`Microservice running on port ${envs.port}`);
}
bootstrap();
```

- Usamos el **CLI** con **nest g res orders**
  - En lugar de crear un CRUD API REST, elegimos **Microservices (non-HTTP)**
  - Creo los endpoints (le digo que si)
  - La entity no la voy a ocupar
  - En lugar de usar @Get, @Post usará @MessagePattern
  - Los de actualizar y borrar no los necesito
- De forma automática coloco aun string en el MessagePattern
  - Aunque usamos un objeto anteriormente como {cmd: 'mensaje'} en orders usaremos un string
  - De todas maneras trata de estandarizar los MessagePattern y enviar siempre el mismo formato
- Evidentemente quiero llegar a orders desde el cliente-gateway

---

Conectar Gateway con Orders

- Para conectar orders.microservice con el cliente-gateway el microservicio tiene que estar levantado
- **En el cliente** genero un RESTFUL API con nest g res orders
  - La entity no la voy a ocupar
  - El servicio tampoco (lo borro del controlador y del módulo)
- En el ApppModule del cliente-gateway ahora tengo (en imports) a ProductsModule y OrdersModule

- Los endpoints de actualizar y borrar no los necesito
- Seguimos en el cliente-gateway, en /config/services.ts declaro el token de inyección para identificar el microservicio

```
export const PRODUCT_SERVICE = 'PRODUCT_SERVICE';
export const ORDER_SERVICE = 'ORDER_SERVICE';
```

- Registro el microservicio en el módulo de la REST API de Orders del cliente-gateway

```
import { Module } from '@nestjs/common';
import { OrdersController } from './orders.controller';
import { ClientsModule, Transport } from '@nestjs/microservices';
import { ORDER_SERVICE, envs } from 'src/config';

@Module({
  controllers: [OrdersController],
  imports: [
    ClientsModule.register([

      {
        name: ORDER_SERVICE,
        transport: Transport.TCP,
        options: {
          host: envs.ordersMicroserviceHost,
          port: envs.ordersMicroservicePort
        }
      },

    ]),
  ]
})
export class OrdersModule {}
```

- Por supuesto añado las variables de entrono a .env (y .env.template) y a .envs
- Recuerda que el puerto tiene que ser un número

```
import 'dotenv/config';

import * as joi from 'joi';

interface EnvVars {
  PORT: number;
  PRODUCTS_MICROSERVICE_HOST: string;
  PRODUCTS_MICROSERVICE_PORT: number;

  ORDERS_MICROSERVICE_HOST: string;
  ORDERS_MICROSERVICE_PORT: number;
}
```

```
const envsSchema = joi.object({
  PORT: joi.number().required(),
  PRODUCTS_MICROSERVICE_HOST: joi.string().required(),
  PRODUCTS_MICROSERVICE_PORT: joi.number().required(),

  ORDERS_MICROSERVICE_HOST: joi.string().required(),
  ORDERS_MICROSERVICE_PORT: joi.number().required(),

})
.unknown(true);

const { error, value } = envsSchema.validate( process.env );


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

const envVars:EnvVars = value;


export const envs = {
  port: envVars.PORT,

  productsMicroserviceHost: envVars.PRODUCTS_MICROSERVICE_HOST,
  productsMicroservicePort: envVars.PRODUCTS_MICROSERVICE_PORT,

  ordersMicroserviceHost: envVars.ORDERS_MICROSERVICE_HOST,
  ordersMicroservicePort: envVars.ORDERS_MICROSERVICE_PORT,


};
```

- En .env.template

```
PORT=3000


PRODUCTS_MICROSERVICE_HOST=localhost
PRODUCTS_MICROSERVICE_PORT=3001

ORDERS_MICROSERVICE_HOST=localhost
ORDERS_MICROSERVICE_PORT=3002
```

- Inyecto el microservicio en el OrdersController del cliente-gateway usando el token de inyección, de tipo **ClientProxy**
- Con **.send** (porque espero una respuesta) le mando el string del @MessagePattern en orders-microservice.controller para conectar, y el dto o lo que sea que necesite si lo requiere
- En el código hay cosas que se irán explicando sobre la marcha

- cliente-gateway.controller

```
import { Controller, Get, Post, Body, Param, Inject, ParseUUIDPipe, Query, Patch }
from '@nestjs/common';


import { ORDER_SERVICE } from 'src/config';
import { ClientProxy, RpcException } from '@nestjs/microservices';
import { CreateOrderDto, OrderPaginationDto, StatusDto } from './dto';
import { firstValueFrom } from 'rxjs';
import { PaginationDto } from 'src/common';

@Controller('orders')
export class OrdersController {

  constructor(
    @Inject(ORDER_SERVICE) private readonly ordersClient: ClientProxy,
  ) {}

  @Post()
  create(@Body() createOrderDto: CreateOrderDto) {
    return this.ordersClient.send('createOrder', createOrderDto);
  }

  @Get()
  findAll( @Query() orderPaginationDto: OrderPaginationDto ) {
    return this.ordersClient.send('findAllOrders', orderPaginationDto);
  }

  @Get('id/:id')
  async findOne(@Param('id', ParseUUIDPipe ) id: string) {
    try {
      const order = await firstValueFrom(
        this.ordersClient.send('findOneOrder', { id })
      );

      return order;

    } catch (error) {
      throw new RpcException(error);
    }
  }

  @Get(':status')
  async findAllByStatus(
    @Param() statusDto: StatusDto,
    @Query() paginationDto: PaginationDto,
  ) {
    try {

      return this.ordersClient.send('findAllOrders', {
        ...paginationDto,
        status: statusDto.status,
```

```
      });

    } catch (error) {
      throw new RpcException(error);
    }
  }


  @Patch(':id')
  changeStatus(
    @Param('id', ParseUUIDPipe ) id: string,
    @Body() statusDto: StatusDto,
  ) {
    try {
      return this.ordersClient.send('changeOrderStatus', { id, status:
statusDto.status })
    } catch (error) {
      throw new RpcException(error);
    }
  }
}
```

- En el orders-microservice.controller

```
import { Controller, NotImplementedException, ParseUUIDPipe } from
'@nestjs/common';
import { MessagePattern, Payload } from '@nestjs/microservices';
import { OrdersService } from './orders.service';
import { CreateOrderDto } from './dto/create-order.dto';
import { OrderPaginationDto } from './dto/order-pagination.dto';
import { ChangeOrderStatusDto } from './dto';

@Controller()
export class OrdersController {
  constructor(private readonly ordersService: OrdersService) {}

  @MessagePattern('createOrder')
  create(@Payload() createOrderDto: CreateOrderDto) {
    return this.ordersService.create(createOrderDto);
  }

  @MessagePattern('findAllOrders')
  findAll(@Payload() orderPaginationDto: OrderPaginationDto ) {
    return this.ordersService.findAll(orderPaginationDto);
  }

  @MessagePattern('findOneOrder')
  findOne(@Payload('id', ParseUUIDPipe ) id: string) { //Si recojo el id, debo
pasarlo en el payload (y lo parseo)
    return this.ordersService.findOne(id);
  }
```

```
  @MessagePattern('changeOrderStatus')
  changeOrderStatus(@Payload() changeOrderStatusDto: ChangeOrderStatusDto ) {
    return this.ordersService.changeStatus(changeOrderStatusDto)

  }
}
```

- Una vez probado que los endpoints se comunican corectamente, Orders debe crear la base de datos y hay que desarrollar la lógica en el servicio

---

Docker - Levantar PostgreSQL

- Si no quieres usar Docker puedes usar neon.tch, ofrecen un ServerLess Postgres (tienes un espacio gratuito para trabajar con PostgreSQL con limitaciones)
- Aqui lo haremos con Docker
- En la raíz de orders-microservice creo el docker-compose.yml- Para obtener la info de como generar volúmenes, puedes consultar en Docker-hub, en la imagen de la DB PostgreSQL
- Le estoy diciendo que enlace mi carpeta postgres de mi fileSystem con la ruta del fs del contenedor
- Pongo un puerto que no esté ocupado
- No uso variables de entorno porque en producción no voy a usar Docker, voy a usar algún servicio

```
 version: '3'


services:
  orders-db:
    container_name: orders_database
    image: postgres:16.2
    restart: always
    volumes:
      - ./postgres:/var/lib/postgresql/data
    ports:
      - 5434:5432
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=123456
      - POSTGRES_DB=ordersdb
```

- Ejecuto (con Docker abierto corrriendo) en la raiz del proyecto
- Con el -d el container no se bajará si se cierra la terminal

```
docker-compose -d
```

- En .gitignore coloco postgres/
- Me puedo conectar con Table Plus creando la conexión
- Vamos con Prisma

Modelo y conexión

- Está en la documentación de Nest cómo generar una cosa y otra usando Prisma
- Básicamente **instalo prisma** con **npm**, uso npx prisma, npx prisma init,
- Esto crea una conexión postgres automática en .env
- La cambio y coloco la mía, con el password y user que le puse en el docker file
- Al ser una base de datos de desarrollo coloco pública la cadena de conexión

```
DATABASE_URL="postgresql://postgres:123456@localhost:5434/ordersdb?schema=public"
```

- Instalo el cliente de prisma con npm i @prisma/client
- Creo el schema en Prisma
- Creo un enum con los status
- Creo el modelo en orders-microservice/prisma/schema.prisma

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

// Looking for ways to speed up your queries, or scale easily with your serverless
or edge functions?
// Try Prisma Accelerate: https://pris.ly/cli/accelerate-init

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

enum OrderStatus {
  PENDING
  DELIVERED
  CANCELLED
}

model Order {
  id          String @id @default(uuid()) //uso uuid com id, @id crea el índice la
llave primaria,etc
  totalAmount Float  //El total de la orden
  totalItems  Int   //cantidad de items

  status OrderStatus  //status de la orden
  paid   Boolean     @default(false)  //está pagada o no
  paidAt DateTime?                    //pagada cuando

  createdAt DateTime @default(now())  //valor por defecto now()
```

```
  updatedAt DateTime @updatedAt       //@updatedAt es una función propia
}
```

- Quizá sería más conveniente crear otra tabla con las pagadas y añadir el paidAt
- Ejecuto el comando **npx prisma migrate dev --name init**
- Se crea el cliente y se actualiza la DB
- Si todo va bien, aparecen los campos de Orders en tablePlus (!)
- Para terminar creamos un Logger en orders-microservice.service y conectamos con Prisma usando la herencia con PrismaCLient (que lo acabamos de instalar) e implementamos la interfaz de OnModuleInit que nos obliga al método onModuleInit para conectarnos con la DB

```
import { HttpStatus, Injectable, Logger, OnModuleInit } from '@nestjs/common';
import { CreateOrderDto } from './dto/create-order.dto';
import { PrismaClient } from '@prisma/client';
import { RpcException } from '@nestjs/microservices';
import { OrderPaginationDto } from './dto/order-pagination.dto';
import { ChangeOrderStatusDto } from './dto';


@Injectable()
export class OrdersService extends PrismaClient implements OnModuleInit {

  private readonly logger = new Logger('OrdersService');


  async onModuleInit() {
    await this.$connect();
    this.logger.log('Database connected');
  }
}
```

Crear una nueva orden

- 'orders' está en el decorador @Controller('orders') de orders-microservice.controller y también en el decorador del cliente-gateway.orders.controller. Los dos apuntan a orders
- Me creo el dto en orders-microservice (debo instalar class-validator y class-transformer)
- Debo configurar tambien el useGlobalPipes en el main
- Para utilizar el enum uso @IsEnum, le paso el listado y envío un mensaje dentro de un objeto

```
import { IsBoolean, IsEnum, IsNumber, IsOptional, IsPositive } from 'class-
validator';
import { OrderStatus, OrderStatusList } from '../enum/order.enum';

export class CreateOrderDto {


  @IsNumber()
```

```
  @IsPositive()
  totalAmount: number;

  @IsNumber()
  @IsPositive()
  totalItems: number;

  @IsEnum( OrderStatusList, {
    message: `Possible status values are ${ OrderStatusList }`
  })
  @IsOptional()
  status: OrderStatus = OrderStatus.PENDING //por defecto pongo PENDING

  @IsBoolean()
  @IsOptional()
  paid: boolean = false; //por defecto pongo false
}
```

- Uso de useGlobalPipes

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

- El cliente-gateway tiene el mismo dto, con el status y el paid con valores por defecto
- En cliente-gateway.orders.enum (order.enum.ts) tengo el OrderStatus (que también creé sin valores como string en el schema de prisma de orders.microservice)
- Y creo también el array con la OrderStatusList usando OrderStatus (estoy en el cliente-gateway)

```
export enum OrderStatus {
  PENDING = 'PENDING',
  DELIVERED = 'DELIVERED',
  CANCELLED = 'CANCELLED',
}


export const OrderStatusList = [
  OrderStatus.PENDING,
  OrderStatus.DELIVERED,
  OrderStatus.CANCELLED,
]
```

- En el enum de orders-microservice , ya tengo el OrderStatus en el PrismaClient (desde el schema)

```
import { OrderStatus } from '@prisma/client';


export const OrderStatusList = [
  OrderStatus.PENDING,
  OrderStatus.DELIVERED,
  OrderStatus.CANCELLED,
]
```

- El schema en orders-microservice (extracto)

```
enum OrderStatus {
  PENDING     //no tiene valores (strings) asignados pero si lo están en el
cliente-gateway
  DELIVERED
  CANCELLED
}

model Order {
  id          String @id @default(uuid())
  totalAmount Float
  totalItems  Int

  status OrderStatus //se lo estoy pasando aqui
  paid   Boolean     @default(false)
  paidAt DateTime?

  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}
```

- Ya tenemos el dto para el cliente y tenemos el otro dto para el microservicio (que son iguales)
- Importo el dto en el controlador del cliente-gateway para pasarselo al @Post
- Vamos con el orders-microservice.service
- No hace falta que inyecte nada en el servicio porque trabajo con el cliente de Prisma, y mi servicio hereda de PrismaClient y ya está conectado a la DB con onModuleInit
- Uso this.order, order porque en el schema le pusimos Order al modelo
- Le paso en la data el dto para hacer la isnerción con .create

```
import { HttpStatus, Injectable, Logger, OnModuleInit } from '@nestjs/common';
import { CreateOrderDto } from './dto/create-order.dto';
import { PrismaClient } from '@prisma/client';
import { RpcException } from '@nestjs/microservices';
import { OrderPaginationDto } from './dto/order-pagination.dto';
import { ChangeOrderStatusDto } from './dto';


@Injectable()
```

```
export class OrdersService extends PrismaClient implements OnModuleInit {

  private readonly logger = new Logger('OrdersService');


  async onModuleInit() {
    await this.$connect();
    this.logger.log('Database connected');
  }

  create(createOrderDto: CreateOrderDto) {

    return this.order.create({
      data: createOrderDto
    })

  }
```

- Para insertar apunto a http://localhost:3000/api/orders y le paso lo que me pide el dto

```
{
  "totalAmount":135,
  "totalItems":4

}
```

- Esto devuelve algo como esto

```
{
  "id": "34fc8622-44c0-4775-a491-8f8db607967e",
  "totalAmount": 135,
  "totalItems": 4,
  "status": "PENDING",
  "paid": false,
  "paidAt": null, //puede ser que yo no quiera valores null en mi tabla, por lo
que tendría que crear una para paid y paidAt
  "createdAt": "2024-05-02T13:09:57.905Z",
  "updatedAt": "2024-05-02T13:09:57.905Z"
}
```

Obtener orden por ID

- Copio un uuid de alguna order en TablePlus
- Lo paso en el endpoint de THUNDERCLIENT/POSTMAN orders/UUId-6456500uu-65656i-DUU-665656IDuuiD
- Primero nos aseguramos que desde cliente-gateway le mandemos el id

- Si en lugar de tratar como un Observable lo que devuelve .send y usar el catchError lo quiero tratar como una promesa con async await, debo usar firstValueFrom en un try catch, y atrapar la RpcException en el catch

```
@Get('id/:id')
async findOne(@Param('id', ParseUUIDPipe ) id: string) {
  try {
    const order = await firstValueFrom(
      this.ordersClient.send('findOneOrder', { id })
    );

    return order;

  } catch (error) {
    throw new RpcException(error);
  }
}
```

- En el orders-microservice.service busco que exista y si no existe envio la RpcException.
- Si existe la order retorno
- *NOTA*: sigo el mismo patrón siempre en las RpcException de incluir message y status para que todo vaya bien con el ExceptionFilter

```
async findOne(id: string) {
  const order = await this.order.findFirst({
    where: { id }
  });

  if ( !order ) {
    throw new RpcException({
      status: HttpStatus.NOT_FOUND,
      message: `Order with id ${ id } not found`
    });
  }

  return order;

}
```

- EN THUNDERCLIENT apunto con un GET a http://localhost:3000/api/orders/id/34fc8622-44c0-4775-a491-8f8db607967e (un UUID válido)

Paginación y Filtro (findAll)

- Crea varias órdenes
- Algunas ponlas canceladas y entregadas desde TablePlus
- En el controlador del cliente-gateway recojo de las Query y valido la data con orderPaginationDto

- En el .send primero le paso el string que es el mismo que me conecta desde MessagePattern con el orders-microservice.controller

```
@Get()
  findAll( @Query() orderPaginationDto: OrderPaginationDto ) {
    return this.ordersClient.send('findAllOrders', orderPaginationDto);
  }
```

- El dto de orderPaginationDto quiero filtrar a través del status
- Extiendo de PaginationDto para tener las mismas propiedades opcionales de PaginationDto disponibles

```
import { IsEnum, IsOptional } from 'class-validator';
import { PaginationDto } from 'src/common';
import { OrderStatus, OrderStatusList } from '../enum/order.enum';


export class OrderPaginationDto extends PaginationDto {


  @IsOptional()
  @IsEnum( OrderStatusList, {
    message: `Valid status are ${ OrderStatusList }`
  })
  status: OrderStatus;

}
```

- En el controlador de orders-microservice tenemos el mismo dto (este alojado en el microservicio Orders/src/orders/dto)
- Me interesa filtrar por el status
- Como el dto extiende de paginationDto, tengo disponibles pages y limit que son opcionales

```
@MessagePattern('findAllOrders')
  findAll(@Payload() orderPaginationDto: OrderPaginationDto ) {
    return this.ordersService.findAll(orderPaginationDto);
  }
```

- En el servico de orders-microservice en el que me comunico con el cliente de Prsima para trabajar con la db a través del modelo y el cliente
- Con .count tengo el total de elementos, pasándole la condicion en el objeto de where
- Establezco la currentPage y el limite por página
- En el return, en el objeto data, con un await utilizo .findMany y hago la paginación
- En meta coloco la info que me parece interesante
    - Coloco la cantidad total de elementos filtrados por el status
    - Coloco la página donde estoy

- Coloco el total de páginas (Math.ceil sirve para redondear)

```
async findAll(orderPaginationDto: OrderPaginationDto) {

  //para tener el total de elementos según el status
  const totalPages = await this.order.count({
    where: {
      status: orderPaginationDto.status
    }
  });

  const currentPage = orderPaginationDto.page; //como orderPaginationDto extiende
de paginationDto, tengo disponible page
  const perPage = orderPaginationDto.limit; //también limit


  return {
    //hago la paginación, que siempre es la misma cosa
    data: await this.order.findMany({
      skip: ( currentPage - 1 ) * perPage,
      take: perPage,
      where: {
        status: orderPaginationDto.status
      }
    }),
    meta: {
      total: totalPages,
      page: currentPage,
      lastPage: Math.ceil( totalPages / perPage )
    }
  }
}
```

Cambiar estado de la orden

- En el controlador del cliente-gateway , en orders, uso @Patch
- De los parámetros extraemos el id, lo casteamos a un UUID, viene a ser un id de tipo string
- En el body tenemos el status (que debe coincidir con alguno de los del enum)
- Pasamos el id y el status que lo guardamos de statusDto.status
- metemos el .send en un try catch
- Lanzamos la RpcException en el catch

```
  @Patch(':id')
  async changeStatus(
    @Param('id', ParseUUIDPipe ) id: string,
    @Body() statusDto: StatusDto,
  ) {
    try {
```

```
      return this.ordersClient.send('changeOrderStatus', { id, status:
statusDto.status })
    } catch (error) {
      throw new RpcException(error);
    }
  }
```

- El statusDto está alojado en cliente-gateway/src/orders/dto

```
import { IsEnum, IsOptional } from 'class-validator';
import { OrderStatus, OrderStatusList } from '../enum/order.enum';



export class StatusDto {


  @IsOptional()
  @IsEnum( OrderStatusList, {
    message: `Valid status are ${ OrderStatusList }`
  })
  status: OrderStatus;

}
```

- En el orders-microservice.controller dentro de MessagePattern tenemos el mismo string que hemos usado en el cliente-gateway.orders.controller, y en el Payload le pasamos el dto

```
@MessagePattern('changeOrderStatus')
changeOrderStatus(@Payload() changeOrderStatusDto: ChangeOrderStatusDto ) {
  return this.ordersService.changeStatus(changeOrderStatusDto)

}
```

- El changeOrderStatusDto de orders-microservice.change-order-status.dto contiene un id (tipo UUID versión 4) y el status

```
import { OrderStatus } from '@prisma/client';
import { IsEnum, IsUUID } from 'class-validator';
import { OrderStatusList } from '../enum/order.enum';



export class ChangeOrderStatusDto {

  @IsUUID(4)
  id: string;
```

```
  @IsEnum( OrderStatusList, {
    message: `Valid status are ${ OrderStatusList }`
  })
  status: OrderStatus;
}
```

- En orders-microservice.service extraigo el id y el status con desestructuración
- Busco por id con findOne. Si el status a actualizar es el mismo que el de la orden encontrada, devuelvo la orden tal cual
- Si no uso el .update, busco con el where por el id y en el objeto data cambio el status
- Puedo colocarlo en el return directamente

```
async changeStatus(changeOrderStatusDto: ChangeOrderStatusDto) {

   const { id, status } = changeOrderStatusDto;

   const order = await this.findOne(id);
   if ( order.status === status ) {
     return order;
   }

   return this.order.update({
     where: { id },
     data: { status: status }
   });
 }
```

04- NEST MICROSERVCIIOS - DETALLES

- Vamos a conectar Ordenes con Productos directamente para comprobar que los productos existen
- Debemos cambiar el dto de creacion de la orden para aceptar los tems, en orders igual
- Hay que crear en Products algún método para recibir el id de los productos y verificar que existen
- No voy a poder crear una orden si un producto no existe
- Es conveniente que inttroduzcamos algun tipo de middleman como NATS o RabbitMQ, algun sistema que mantenga el orden en este caos de tanto microservicio

---

Orders-microservice

- Ordenes y detalle van a estar en el mismo microservicio
- Ambos están altamente acoplados, uno no va a existir sin el otro
- Comunicaremos ordenes y productos mediante TCP para validar
- Despues de esta sección implementaremos un middleman entre el cliente y los microservicios (un servidor NATS)
- Habrá otro microservicio de autenticación con MONGO

- Vamos con el desarrollo. Levantamos Docker, Orders y el cliente
- Uso docker-compose up -d en orders-microservice

OrderItems - detalles de la orden

- En orders-microservice
- Para entender la comunicación que vamos a establecer, lo mejor es entender la estructura de la DB
- Practicamente, excepto totalAmount y totalItems, el resto de campos se crean automáticamente
- Voy a pedir siempre una cantidad de hijos (items) y esos items los voy a contar y sumar su valor para el totalAmount
- Una orden en la vida real podría tener más cosas, como un cupón de descuento
- Creo otro modelo como OrderItem
- productId no tiene una relación directa con SQLite de Products
    - Hay quien trabaja todo en un amisma DB, pero no son buenas practicas en microservicios
    - Se puede hacer, pero no permitiría escalar cada microservicio de manera independiente
- Si yo coloco esto en Order

```
OrderItem OrderItem[]
```

- y PRESIONO CTRL+aLT+SHIFT me creará la relación automáticamente

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

// Looking for ways to speed up your queries, or scale easily with your serverless
or edge functions?
// Try Prisma Accelerate: https://pris.ly/cli/accelerate-init

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

enum OrderStatus {
  PENDING
  DELIVERED
  CANCELLED
}

model Order {
  id          String @id @default(uuid())
  totalAmount Float
  totalItems  Int
```

```
  status OrderStatus @default(PENDING) //establezco PENDING por defecto
  paid   Boolean     @default(false)   //false por defecto
  paidAt DateTime?

  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt

  OrderItem OrderItem[]  //
}

model OrderItem {
  id        String @id @default(uuid())
  productId Int  //no hay una relación (física) directa con SQLite
  quantity  Int  //cantidad de este producto
  price     Float  //los precios pueden variar. Este precio se queda aqui en el
momento que se creó la orden

  Order   Order?  @relation(fields: [orderId], references: [id]) //establezco la
relación
  orderId String?
}
```

- Desde orders-microservice impacto la db con una migracion

> npx prisma migrate dev --name order-item

- Debes tener Docker corriendo y poder establecer conexión con el puerto correcto y la autenticación
- Puedo mirar en TablePlus
- Borro las ordenes anteriores porque estan mal creaqdas, les falta el OrderItem

---

DTOs de creación de orden

- Para orders-microservice.create-order.dto pido un Array de minimo 1 elemento de tipo item: orderItemDto

```
import { ArrayMinSize, IsArray, ValidateNested } from 'class-validator';

import { OrderItemDto } from './order-item.dto';
import { Type } from 'class-transformer';

export class CreateOrderDto {
  @IsArray()
  @ArrayMinSize(1) //por lo menos un item
  @ValidateNested({ each: true }) //valida internamente los objetos en el array
  @Type(() => OrderItemDto) //ojo! no pongo que es un arreglo
  items: OrderItemDto[]; //aquí si indico que es un arreglo
}
```

- En order-item.dto tengo productId, quantity y price

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

export class OrderItemDto {
  @IsNumber()
  @IsPositive()
  productId: number; //En la DB de productos los productos tienen id de tipo
numérico

  @IsNumber()
  @IsPositive()
  quantity: number;

  @IsNumber()
  @IsNumber()
  price: number;
}
```

- Esto vendría a pedir algo asi en POSTMAN

```
{
    "items":[
        {
            "productId": 4 ,
            "quantity": 3,
            "price": 20
        }
    ]
}
```

- Pero esto solo valida la petición en orders-microservice
- **Hay que hacer lo mismo en cliente-gateway**
- En cliente-gateway.orders.create-oder.dto es el mismo código
- En cliente-gateway.orders.order-item.dto también es el mismo código
- En el método create de orders-microservice.service voy a tener un error porque prsima valida el dto
- Coloco un return {service: "orders-service create", createOrderDto} devolviendo el dto
- **Falta validar que los productos de la orden existan**

Products-microservice - Validar productos por ID

- En products-microservice estoy trabajando con SQLite (la abro en TablePlus) donde tengo la data
- Abro el products-microservice.controller y creo el controlador

```
@MessagePattern({ cmd: 'validate_products' })
validateProduct( @Payload() ids: number[] ) {
return this.productsService.validateProducts(ids);
}
```

- En products-microservice.service creo un array de id's y utilizo Set para eliminar los id's repetidos
- Hago la búsqueda con findMany le digo que el id debe estar en el arreglo ids con **where:{id:{in: ids}}**
- Si la cantidad de productos encontrados no coincide con la cantidad del array de ids es que algunos productos no los ha encontrado
- Retorno products

```
async validateProducts(ids: number[]) {
ids = Array.from(new Set(ids));

const products = await this.product.findMany({
    where: {
    id: {
        in: ids
    }
    }
});

if ( products.length !== ids.length ) {
    throw new RpcException({
    message: 'Some products were not found',
    status: HttpStatus.BAD_REQUEST,
    });
}


return products;

}
```

Comunicar orders-microservice con products-microservice

- En orders-microservice tengo que registrar el products-microservice con ClientsModule.register
- Voy a necesitar añadir las variables de entorno en .env de orders-microservice

```
import { Module } from '@nestjs/common';
import { OrdersService } from './orders.service';
import { OrdersController } from './orders.controller';
import { ClientsModule, Transport } from '@nestjs/microservices';
import { PRODUCT_SERVICE, envs } from 'src/config';

@Module({
  controllers: [OrdersController],
  providers: [OrdersService],
  imports: [
    // ClientsModule.register......
    ClientsModule.register([
```

```
      {
        name: PRODUCT_SERVICE,
        transport: Transport.TCP, //después usaremos NATS!!!
        options: {
          host: envs.productsMicroserviceHost,
          port: envs.productsMicroservicePort,
        }
      }


    ])
  ]
})
export class OrdersModule {}
```

- orders-microservice .env

```
PORT=3002

PRODUCTS_MICROSERVICE_HOST=localhost
PRODUCTS_MICROSERVICE_PORT=3001

DATABASE_URL="postgresql://postgres:123456@localhost:5434/ordersdb?schema=public"
```

- También debo añadirlas en el archivo de configuración de envs.ts de orders-microservice!!
- envs.ts

```
import 'dotenv/config';

import * as joi from 'joi';

interface EnvVars {
  PORT: number;

  PRODUCTS_MICROSERVICE_HOST: string;
  PRODUCTS_MICROSERVICE_PORT: number;
}

const envsSchema = joi.object({
  PORT: joi.number().required(),

  PRODUCTS_MICROSERVICE_HOST: joi.string().required(),
  PRODUCTS_MICROSERVICE_PORT: joi.number().required(),
})
.unknown(true);

const { error, value } = envsSchema.validate( process.env );
```

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

const envVars:EnvVars = value;


export const envs = {
  port: envVars.PORT,

  productsMicroserviceHost: envVars.PRODUCTS_MICROSERVICE_HOST,
  productsMicroservicePort: envVars.PRODUCTS_MICROSERVICE_PORT,
};
```

- También creo el archivo /config/services.ts con el token de inyección
- Lo coloco en el archivo de barril

```
export const PRODUCT_SERVICE = 'PRODUCT_SERVICE';
```

- En orders-microservice.service necesito hacer la inyección usando **@Inject** pasándole el token de inyección **PRODUCT_SERVICE**
- Constructores de clases derivadas (herencia) deben llamar a **super**

```
import {
  HttpStatus,
  Inject,
  Injectable,
  Logger,
  OnModuleInit,
} from '@nestjs/common';
import { CreateOrderDto } from './dto/create-order.dto';
import { PrismaClient } from '@prisma/client';
import { ClientProxy, RpcException } from '@nestjs/microservices';
import { OrderPaginationDto } from './dto/order-pagination.dto';
import { ChangeOrderStatusDto } from './dto';
import { PRODUCT_SERVICE } from 'src/config';
import { firstValueFrom, throwError } from 'rxjs';

@Injectable()
export class OrdersService extends PrismaClient implements OnModuleInit {
  private readonly logger = new Logger('OrdersService');

  constructor(
    @Inject(PRODUCT_SERVICE) private readonly productsClient: ClientProxy,
  ) {
    super(); //llamo a super
  }

  async onModuleInit() {
```

```
    await this.$connect();
    this.logger.log('Database connected');
  }
}
```

- Estoy en orders-microservice.service
- Con el método create necesito llegar a products con .send si quiero trabajarlo como un Observable
- Si quiero trabajarlo con una **promesa** debo usar **firstValueFrom**
- Coloco todo dentro de un try catch
- Con .map extraigo el arreglo de ids del dto
- Uso el await con firstValueFrom, como parámetro le paso el .send usando el productsClient (el servicio inyectado de PRODUCTS_SERVICE)
    - Para comunicarme con el método validateProducts le paso en el objeto el cmd con el string validate_products (del MessagePattern) y el arreglo de ids que he sacado haciendo un map del dto.id
- Para calcular el total a pagar hago uso del reducer
    - acc es el acumulador. Uso el createOrderDto porque es donde están las orders con los items (y el precio)
    - Uso el arreglo de products para encontrar los productos que coincidan con los ids de cada orderItem y obtener el precio
    - Retorno el precio * la cantidad en cada orderItem
    - El acumulador empieza en 0
- Uso un reducer también para el total de items
    - Sumo el acumulador a la cantidad de items por orden
- Para crear la orden necesito insertar la orden y los items, ambas inserciones deben ser exitosas
- Esto suele hacerse con una **.$transaction** porque si una falla tengo que hacer un rollback
- Vamos a crearlo todo en una sola orden
- *NOTA*: el reducer va acumulando en el acumulador el numero de iteraciones y los guarda en acc.
    - El segundo parámetro del callback es el objeto que voy a iterar. El 0 es el valor inicial del acumulador
    - Por ejemplo:

```
const reducidor = [1,2,3,4,5].reduce((acc, el)=> acc + el, 0)
console.log(reducidor) //Esto devuelve 15

//En la primera iteracion acc vale 0 y el vale 1, 0+1 == 1
//En la segunda acc vale 1 y el 2, 1+2 == 3
//En la tercera acc vale 3 y el 3 == 6
//En la cuarta acc vale 6 y el 4 == 10
//en la quinta acc vale 10 y el 5 == 15
```

```
 async create(createOrderDto: CreateOrderDto) {
    try {
      //1 Confirmar los ids de los productos
      const productIds = createOrderDto.items.map((item) => item.productId);
```

```
//extraigo los ids en un arreglo


      //llamo al microservicio para validar que existan los productos
      const products: any[] = await firstValueFrom(
        this.productsClient.send({ cmd: 'validate_products' }, productIds),
      );

      //2. Cálculos de los valores                          //en orderItem tengo
el precio
      const totalAmount = createOrderDto.items.reduce((acc, orderItem) => {

        //necesito encontrar orderItem dentro del arreglo de productos
        //no quiero confiar en el precio del dto, por eso uso el de los productos
través del id
        const price = products.find(
          (product) => product.id === orderItem.productId).price;
        return price * orderItem.quantity + acc;
      }, 0);

      const totalItems = createOrderDto.items.reduce((acc, orderItem) => {
        return acc + orderItem.quantity; //Si tengo x cantidad, necesito contarlo
por cada uno de los elementos del arreglo
      }, 0); //Para la suma de todos los elementos de un arreglo, ene el acc voy
guardando la suma de las iteraciones

      //3. Crear una transacción de base de datos
      const order = await this.order.create({
        data: {
          totalAmount: totalAmount,
          totalItems: totalItems,
          OrderItem: {
            createMany: {  //uso createMany
              data: createOrderDto.items.map((orderItem) => ({
                price: products.find(  //no puedo tomar directamente el
orderItems.price porque no lo hemos validado, no sabemos si es el correcto
                  (product) => product.id === orderItem.productId, //uso los
precios del arreglo de products que viene de la tabla de Products
                ).price,
                productId: orderItem.productId,
                quantity: orderItem.quantity,
              })),
            },
          },
        },
        //que incluya el OrderItem. Si pongo solo OrderItem: true me devuelve todo
        include: {
          OrderItem: {
            select: { //puedo seleccionar los campos que quiero devolver
              price: true,
              quantity: true,
              productId: true,
            },
          },
```

```
        },
      });

      return {
        ...order, //me quedo con todo lo de order menos OrderItem, qdel que me
aseguro que el nombre coincida con la tabla de products
        OrderItem: order.OrderItem.map((orderItem) => ({
          ...orderItem, //me quedo con todo de orderItem, agrego el nombre de la
tabla de Products desde el arreglo de products
          name: products.find((product) => product.id ===
orderItem.productId).name,
        })),
      };
    } catch (error) {
      throw new RpcException({
        status: HttpStatus.BAD_REQUEST,
        message: 'Check logs',
      });
    }
  }
```

- De hecho, podríamos ignorar el precio en OrderItem porque lo vamos a usar de Products
- En findOrderById quiero que aparezca el detalle

Buscar order por Id con su detalle

- Quiero saber el detalle de esa orden y los ids de los productos
- Uso el include para retornar el OrderItem con los campos indicados en el select

```
  async findOne(id: string) {
    const order = await this.order.findFirst({
      where: { id },
      include: {
        OrderItem: {
          select: {
            price: true,
            quantity: true,
            productId: true,
          },
        },
      },
    });

    if (!order) {
      throw new RpcException({
        status: HttpStatus.NOT_FOUND,
        message: `Order with id ${id} not found`,
      });
    }
    const productIds = order.OrderItem.map((orderItem) => orderItem.productId);
```

```
//extraigo los ids, un arreglo de números

    //valido comunicándome con el microservicio de products que los products
existan
    //ESTO EN ESTE MOMENTO NO DEBERÍA FALLAR
    const products: any[] = await firstValueFrom(
      this.productsClient.send({ cmd: 'validate_products' }, productIds),
    );

    return {
      ...order, //retorno la order y trabajo con OrderItem
      OrderItem: order.OrderItem.map((orderItem) => ({
        ...orderItem, //retorno orderItem y del arreglo de products validado
obtengo el nombre
        name: products.find((product) => product.id === orderItem.productId)
          .name,
      })),
    };
  }
```

Problemas y soluciones

- Esto, organizado así, es muy probable que se salga de control
- Orders está conectado directamente con Products
- En algún momento (cuando implementemos autenticación) orders va a tener que validar un token con el microservicio de auth
- Habría que conectar orders con auth, implica cambios, el cliente-gateway, etc
- Esto va a crear un anidamiento dificil de leer y gestionar
- La solución pasa por un **SERVICE BROKER**, un middleman que se encargue de procesar ese montón de paquetes y pedidos entre los microservicios
- **RabbitM**Q es muy popular. Se crea una cola de procesos y mensajería, es cómo una oficina postal que se va a dedicar a mandar las cartas a los destinatarios, eso queda en el file system y hasta que el destinatario lo confirma no se borra de la cola
- También se puede, basado en alguna transacción o evento que suceda, notificar a dos o tres microservicios de manera simultanea
- Por ejemplo, si alguien paga una orden: yo quiero notificar a orders que la orden fue pagada, quiero notificar al cliente que el pago se recibió. Son dos cosas independientes relacionadas a microservicios distintos que reaccionan ante un mismo evento
- COn la arquitectura actual, significaría que desde el microservicio de pagos llamar al microservicio de notificaciones y esto sería ineficiente, significa acoplamiento, y si las notificaciones se caen fallan podría hacer fallar el servicio de pagos porque esa parte nos va a fallar
- Podemos mandar un evento desde orders a notificaciones y si lo recibe bien y si no también, pero no tiene mucho sentido
- Por todo esto vamos a implementar una arquitectura diferente, con un servidor de **NATS** que estará en medio del cliente-gateway y mis microservicios
- Vamos a centralizar la comunicación entre microservicios
- NATS server se va a encargar de **notificar a todos los microservicios que les interese un mensaje**

- Esto va a **eliminar la comunicación directa entre microservicios**
- NATS va a crear unos **TOPICS** y estos se notificarán a mis microservicios
    - Cuando se cree una orden, NATS se lo notificará al cliente-gateway
    - Como el cliente-gateway está suscrito a la respuesta de la creación de la orden, va a notificar al cliente
- En la práctica es más fácil trabajar con este servidor de NATS!

05- NEST MICROSERVICIOS - NATS

Problema / Solución

- La orden se crea con el detalle, están fuertemente acopladas, las hicimos en el mismo microservicio
- Orders se comunica directamente con Products para confirmar los productos
- Con la arquitectura actual, cuando añadamos autenticación y queramos modificar una orden, vamos a tener que añadir otra dependencia a ordenes (el microservicio de auth) En lugar de eso, nos encargaremos de hacer las comunicaciones mediante un Service Broker, un middleman situado entre el cliente-gateaway y mis microservicios
- Cuando NATS recibe la solicitud de creación de una orden desde el cliente-gateway mandada por el cliente, orders que está suscrita al tópico de creación de la orden dará una respuesta
- Orders necesita saber si los ids de los productos que lleva el detalle existen en la db de productos
- El servidor de NATS servirá de intermediario entre orders-microservice y productws-microservice
- Bueno, esto es una forma simplificada de lo que ocurre
- **NATS BROKER**
    - Es open source, ligero y facil de configurar
    - Comunicará mis microservicios (estos pueden mantener interconexiones sin necesidad de pasar por NATS)
    - NATS se encarga de hacer balanceo de carga
        - Es decir, si tengo varios microservicios y creo que todos respondan al mismo tiempo se puede configurar con NATS
        - Cuando implementemos los microservicios de pagos y notificaciones vamos a querer hacer uso de esto
    - Trabaja con mensajería tipo **publicar y suscribir**
    - Hay temas **topics/subjects** a los cuales **se escucha**
    - Puedes tener **múltiples escuchas (listeners)** al mismo topic
    - Pensado para **escalamiento horizontal**
    - Seguridad, balanceo de carga inlcuido
    - **Payload agnóstico**, pueden ser strings, numeros, lo que sea necesario
    - Rápido y eficiente, y open source
- Por ejemplo, una vez realizado un pago voy a querer comunicarme con tres microservicios de manera instantánea
    - Con auth, email notification y ordenes
- Con la arquitectura que hemos implementado hasta ahora se convertirían en dependencias del microservicio de pagos
- Crearemos una **Docker Network** para tenerlo todo en un mismo lado y que solo mediante el puerto 3000 a través de una API REST se acceda a esta red interna

- Primero vamos a establecer la arquitectura con NATS y luego crearemos la Docker Network

---

- En orders-microservice debo ejecutar docker compose up -d para levantar el microservicio con Docker corriendo
- Si aparece el error de prisma usar **npx prisma generate**
- Para que no haya que instalar NATS físicamente en el host usaremos Docker
- Para levantar el servidor de NATS usar
    - 4222: Nuestros microservicios van a estar hablando con NATS por este puerto
    - 8222: ofrece una comunicación HTTP para monitorear los clientes y ver quien se conecta, quien se cae, se levanta, etc
    - 6222: Puerto utilizado para el clustering. En estecaso no lo usaremos
    - nats al final esla imagen de nats:latest por defecto
    - Le llamaremos nats-sever

> docker run -d --name nats-server -p 4222:4222 -p 8222:8222 nats npm run start:dev

- En .env de orders tengo

```
PORT=3002

PRODUCTS_MICROSERVICE_HOST=localhost
PRODUCTS_MICROSERVICE_PORT=3001

DATABASE_URL="postgresql://postgres:123456@localhost:5434/ordersdb?schema=public"

# NATS_SERVERS="nats://localhost:4222,nats://localhost:4223"
NATS_SERVERS="nats://localhost:4222"
```

- En .env de products tengo
- El :4223 no existe. Servirá luego parahacer las pruebas de validación de las variables de entorno, donde validaremos que sea un array y separaremos por comas los strings para porder validarlos

```
PORT=3001

DATABASE_URL="file:./dev.db"


NATS_SERVERS="nats://localhost:4222,nats://localhost:4223"
```

- En .env de cliente-gateway tengo

```
PORT=3000


# PRODUCTS_MICROSERVICE_HOST=localhost
```

```
# PRODUCTS_MICROSERVICE_PORT=3001

# ORDERS_MICROSERVICE_HOST=localhost
# ORDERS_MICROSERVICE_PORT=3002


# NATS_SERVERS="nats://localhost:4222,nats://localhost:4223"
NATS_SERVERS="nats://localhost:4222"
```

- Una vez ya ha descargado la imagen de Docker, para iniciar el microservicio usar el mismo prompt sin el --name
- Con crear la conexión con NATS en orders es suficiente

> docker run -d -p 4222:4222 -p 8222:8222 nats npm run start:dev

- En localhost:8222 puedo monitorear el NATS (tiene una interfaz gráfica)
- **RESUMEN**:
    - uso el comando de docker run con el nombre del nats-server y los puertos + la imagen para que descargue la imagen de NATS si no la tengo

---

Products-microservice - Cambiar de TCP a NATS

- Para trabajar con nats en NEST debo instalar nats con npm i nats
- Es muy similar a la conexión con TCP, solo que en lugar de los puertos tengo en options un arreglo con los servidores (puede ser uno, pueden ser más)
- Debo añadir la variable de entorno a envs.ts

```
import 'dotenv/config';
import * as joi from 'joi';

interface EnvVars {
  PORT: number;
  DATABASE_URL: string;

  NATS_SERVERS: string[]; //añado el string del NATS
}

const envsSchema = joi.object({
  PORT: joi.number().required(),
  DATABASE_URL: joi.string().required(),
                                                    //en este momento
todavía no es un arreglo, pero hago la validación
  NATS_SERVERS: joi.array().items( joi.string() ).required(), //valido que el NATS
sea un array y que contenga un string ( yque sea obligatorio)
})
.unknown(true);

const { error, value } = envsSchema.validate({
```

```
  ...process.env,         //hago que sea un arreglo con .split
  NATS_SERVERS: process.env.NATS_SERVERS?.split(',') //si tengo varios strings,
los separo por la coma para validarlos
});

//explicación: ES AL APLICAR SPLIT QUE DEVUELVE UN ARREGLO

const loquesea ="loalaoaoala, pwepejdoiejhd"

const arrayLoquesea = loquesea.split(',')

console.log(arrayLoquesea) //["loalaoaoala", " pwepejdoiejhd"]


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

const envVars:EnvVars = value;


export const envs = {
  port: envVars.PORT,
  databaseUrl: envVars.DATABASE_URL,

  natsServers: envVars.NATS_SERVERS, //exporto la variable
}
```

- En el main

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { Logger, ValidationPipe } from '@nestjs/common';
import { envs } from './config';
import { MicroserviceOptions, Transport } from '@nestjs/microservices';

async function bootstrap() {

  const logger = new Logger('Main');

  const app = await NestFactory.createMicroservice<MicroserviceOptions>(
    AppModule,
    {
      transport: Transport.NATS,
      options: {
        servers: envs.natsServers //'nats://localhost:4222,
nats://localhosts:4223'
      }
    }
  );

  app.useGlobalPipes(
```

```
    new ValidationPipe({
      whitelist: true,
      forbidNonWhitelisted: true,
    }),
  );

  await app.listen();
  logger.log(`Products Microservice running on port ${ envs.port }`);

}
bootstrap();
```

- Puedo seguir usando el **MessagePattern**, puedo usar comodines

```
@MessagePattern('time.*') //escuchará cualquier mensaje que venga de time
getDate(@Payload() data: number[], @Ctx() context: NatsContext){
  console.log(`Subject: ${context.getSubject()}`)
  return new Date().toLocaleString(...)
}
```

- Para poder ver los productos debemos cambiar tambien el canal de comunicación en cliente-gateway
- **RESUMEN**: instalo nats, configuro y valido la variable de entorno, me aseguro de que sea un arreglo, coloco la variable en el main y cambio el transporte

---

Cliente-gateway - Cambiar TCP a NATS

- Debemos instalar nats tambien con npm i nats
- Vamos a crear un módulo centralizado para poder importar la comunicación
- Podemos hacer el cambio en cliente-gateway.products.module de esta forma

```
imports:[
    ClientsModule.register([
    {
      name: NATS_SERVICE,
      transport: Transport.NATS,
      options: {
        servers: envs.natsServers,
      },
    },
  ])
]
```

- Pero podemos crear un módulo que podamos copiar y pegar a los otros clientes (orders, en este caso)
- Creo la carpeta en cliente-gateway/src/products/transports/nats.module.ts

```
import { Module } from '@nestjs/common';
import { ClientsModule, Transport } from '@nestjs/microservices';
import { NATS_SERVICE, envs } from 'src/config';

@Module({
  imports: [
    ClientsModule.register([
      {
        name: NATS_SERVICE,
        transport: Transport.NATS,
        options: {
          servers: envs.natsServers,
        },
      },
    ]),
  ],
  exports: [
    ClientsModule.register([
      {
        name: NATS_SERVICE,
        transport: Transport.NATS,
        options: {
          servers: envs.natsServers,
        },
      },
    ]),
  ],
})
export class NatsModule {}
```

- El token de inyección NATS_SERVICE está alojado en cliente-gateway/src/config/services.ts

```
export const PRODUCT_SERVICE = 'PRODUCT_SERVICE';
export const ORDER_SERVICE = 'ORDER_SERVICE';


export const NATS_SERVICE = 'NATS_SERVICE';
```

- Importo el módulo en cliente-gateway.app.module

```
import { Module } from '@nestjs/common';
import { ProductsModule } from './products/products.module';
import { OrdersModule } from './orders/orders.module';
import { NatsModule } from './transports/nats.module';


@Module({
  imports: [ProductsModule, OrdersModule, NatsModule],
```

```
})
export class AppModule {}
```

- Y también en cliente-gateway.products.module

```
import { Module } from '@nestjs/common';
import { ProductsController } from './products.controller';
import { NatsModule } from 'src/transports/nats.module';

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

- Hago lo mismo en orders! copio la carpeta transports y hago la importación del módulo
- Para usarlo en el controller de cliente-gateway.products.controller debo inyectarlo con el token de inyección y usar CLientProxy nuevamente
- ya no lo nombro productsClient, simplemente **client**
- cliente-gateway.products.controller

```
import { ClientProxy, RpcException } from '@nestjs/microservices';
import { catchError, firstValueFrom } from 'rxjs';
import { PaginationDto } from 'src/common';
import { NATS_SERVICE } from 'src/config';
import { CreateProductDto } from './dto/create-product.dto';
import { UpdateProductDto } from './dto/update-product.dto';

@Controller('products')
export class ProductsController {
  constructor(
    @Inject(NATS_SERVICE) private readonly client: ClientProxy, //inyecto el token
y uso Clientproxy
  ) {}

  @Post()
  createProduct(@Body() createProductDto: CreateProductDto) {
    return this.client.send(
      { cmd: 'create_product' },
      createProductDto,
    );
  }
}
```

- Lo mismo en el módulo de orders de cliente-gateway

- Tengo que hacer la misma configuración en orders-microservice, copiar el modulo transports, importarlo, validar la variable de entorno, inyectar el token en el controlador, renombrar el servicio a client...

Docker Network - problema y necesidad

- Tenemos varias terminales corriendo, el Docker, tenemos que levantar el NATS....
- Un poco complicado (y tedioso de levantar y subir) para alguien que venga de fuera desarrollar así
- Crearemos una red que se encargue de comunicarse con mis servidores, que mediante un solo comando levante toda la infraestructura
- Puedo hacer que no se levanten los microservicios si NATS no está arriba
- Lo mismo con la DB, si no está arriba el microservicio no se va a levantar
- Crearemos un monorepo. No es más que un repositorio que tiene varios repositorios de nuestra app
- Nest ofrece una manera un poco acoplada
- Usaremos otra metodología

```
------
# NEST MICROSERVICIOS - LAUNCHER (DOCKER)

- Un comando para levantar todo y tenerlo todo en una terminal
- Es bastante común trabajar con monorepos que contienen referencias a otros
repositorios
----

## Crear red y levantar todo con un solo comando

- En mi código no habrá la relación con los submódulos para conectar con los
microservicios, pero si dejaré la documentación aquí
- Creo el docker-compose.yml en la raíz del launcher

~~~yml
version: '3'


services:

  # levanto el nats
  nats-server:
    image: nats:latest # descargo la última verisión de la imagen de nats
    ports:
      - "8222:8222" # el 8222 es el puerto por defecto junto al 4222, mirar la
docu
                    # exponer estos puertos es para que el mundo exterior pueda
llegar a NATS, no la red interna
                    # 8222 me ofrece el servicio de poder monitorear quien se
conecta, etc
                    # por lo que estaría exponiendo el puerto en la zona externa
entre el cliente y el gateway
                    # en la vida real podría quitarlo, no necesito exponerlo
```

```
  client-gateway: #nombro mi servicio, necesito crear una imagen para montarla en
un contenedor
    build: ./client-gateway # vendrá a esta ruta a buscar el dockerfile
    ports:
      - ${CLIENT_GATEWAY_PORT}:3000 #comunico el puerto de mi computadora con el
del contenedor
    volumes:
      - ./client-gateway/src:/usr/src/app/src #puedo enfocarme solo en el src, lo
mapeo a usr/src/app/src (node tiene este path)
    command: npm run start:dev
    environment: # definimos las variables de entorno (es como tener mi .env aqui,
las validaciones que hice aplican aqui)
      - PORT=3000
      - NATS_SERVERS=nats://nats-server:4222 # coloco nats-server en lugar de
localhost porque asi se llama el servicio y le pone nombre al contenedor

  products-ms: # este es el nombre del server(imagen de Docker)
    build: ./products-ms
    volumes:
      - ./products-ms/src:/usr/src/app/src # mapeo el src
    command: npm run start:dev
    environment:
      - PORT=3001
      - NATS_SERVERS=nats://nats-server:4222
      - DATABASE_URL=file:./dev.db # products está en el filesystem porque uso
SQLite


  # Orders MS
  orders-ms:
    depends_on:
      - orders-db #este microservicio no se debe levantar hasta que orders-db se
levante (levantar, no construir)
    build: ./orders-ms
    volumes:
      - ./orders-ms/src:/usr/src/app/src
    command: npm run start:dev
    environment:
      - PORT=3002                                    # apuntando a docker, no tengo
localhost, tego orders-db (así llamé al servicio)
      - DATABASE_URL=postgresql://postgres:123456@orders-db:5432/ordersdb?
schema=public # lo conecto al puerto de la imagen
      - NATS_SERVERS=nats://nats-server:4222



  # Orders DB     también descargoi la imagen de postgres!!
  orders-db:
    container_name: orders_database
    image: postgres:16.2
    restart: always
    volumes:
      - ./orders-ms/postgres:/var/lib/postgresql/data
    ports:
```

```
      - 5434:5432
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=123456
      - POSTGRES_DB=ordersdb
```

- No se recomienda usar el tag :latest porque luego se despliega y se hacen versiones incompatibles. No hacer en producción (se suben manualmente)
- NATS ya está dentro de la red, detrás del Gateway que comunica con el exterior
- La idea de exponer los puertos 4222, 8222 es para que el mundo exterior pueda llegar a ellos, no la red interna
- El puerto que me interesa es el 8222 porque me facilita el servicio para la monitorización de NATS desde el navegador
- El nombre del servidor es exactamente igual al nombre del servicio donde levanto la imagen de NATS en el archivo de docker
- En lugar de localhost en el string de conexión, pondré nats-server, lo mismo en las variables de entorno del .ym en environment
- Para la DATABASE_URL En el caso de products coloco el archivo del filesystem de la carpeta de prisma
- Para postgres la dirección de la imagen de Docker
- Empezando por el client-gateway, voy a necesitar configurar el dockerfile en la raíz

```
FROM node:20-alpine3.19 /*tomo la imagen de NODE*/

WORKDIR /usr/src/app    /*desde aqui trabajaremos, es donde colocaremos la app*/

COPY package.json ./    /*copiamos el json*/
COPY package-lock.json ./


RUN npm install          /*instalamos las dependencias porque en mi maquina tengo
un linux, no son las mismas*/

COPY . .                 /*copiamos todo lo que no está ignorado en el
dockerignore*/

EXPOSE 3000              /*expongo el puerto de client-gateway*/
```

- Creo el .dockerignore (es el mismo para prodcuts y orders)

```
dist/

node_modules/

.env

.vscode/
```

- Debo crear el dockerfile en products y orders
- En products dockerfile llamo **npx prisma generate**, pero esto no va a funcionar en la vida real
- Aqui funciona porque uso SQLite y tengo la Db en el fileSystem ya creada, en la vida real usaría postgres o mongo
- Si no tenemos la db ni las migraciones, lo que tengo en mi Schema debería ser suficiente para crear mi db
- Pero desde el dockerfile estamos construyendo la imagen
- Mi Schema lo ocupo para ejecutarlo
- npx prisma generate solo es útil **si la DB YA EXISTE** por lo que con products si va a funcionar, pero con postgres no
- Si no existiera crear con **npx prisma migrate dev --name init**
- Me aseguro de tener data en la DB
- products dockerfile

```
FROM node:20-alpine3.19

WORKDIR /usr/src/app

COPY package.json ./
COPY package-lock.json ./


RUN npm install

COPY . .

RUN npx prisma generate


EXPOSE 3001
```

- Para solucionar este problema en products y en orders creo un nuevo script en el json y lo coloco en el start:dev
- No hace falta usar npx porque ya hemos creado el cliente desde el dockerfile products-ms

```
{
  "docker:start":"prisma migrate dev && prisma generate",
  "start:dev":"npm run docker:start && nest start --watch"
}
```

- orders dockerfile

```
FROM node:20-alpine3.19

WORKDIR /usr/src/app
```

```
COPY package.json ./
COPY package-lock.json ./


RUN npm install

COPY . .

EXPOSE 3002
```

- En las .envs de orders-microservice coloco el string de conexión

```
PORT=3002


PRODUCTS_MICROSERVICE_HOST=localhost
PRODUCTS_MICROSERVICE_PORT=3001

DATABASE_URL="postgresql://postgres:123456@orders-db:5432/ordersdb?schema=public"

# NATS_SERVERS="nats://localhost:4222,nats://localhost:4223"
NATS_SERVERS="nats://localhost:4222"
```

- Si observamos en Docker, nos podemos conectar a orders-db porque estamos mapeando los puertos
- En la vida real esto no sería necesario, exponer la db de esta manera con 5432:5432
- Si lo quito del docker-compose sigue funcionando igual pero tengo un error en TablePlus porque ya no tengo el puerto
- Es genial, porque vamos a crear **una red encapsulada** para que los servicios puedan comunicarse entre si basado en los nombres de los servidores
- **ESTO ES INCREIBLE**
- Vamos a dejar el puerto porque me interesa seguir trabajando con TablePlus

---

Expandir nuestro Custom Exception Filter

- Cuando un microservicio no se levanta nos manda un error de Empty response. There are no subscribers listening to that message "string del message pattern"
- Para centralizar las excepciones en client-gateway/src/exception/custom-esception.filter
- En este momento, si voy al client-gateway.orders.controller **no estoy disparando el exceptionFilter**
- Si todo lo estoy manejando mediante un try y un catch, lo coherente es usarlo también en findAll

```
@Get()
findAll( @Query() orderPaginationDto: OrderPaginationDto ) {
  try{
    const orders = await this.client.send('findAllOrders', orderPaginationDto);
    return orders
  }catch(error){
```

```
    throw new RpcException(error)
  }
}
```

- No hay muchas opciones para manejar el error. exception.name devuelve 'Error' y getError().toString tampoco resuelve mucho
- Tengo una manera usando el .includes con el string que devuelve el error del microservicio no conectado "Empty response etc"
- client-gateway.microservice/common/exceptions/rpc-custom-exception.filter.ts

```
import { Catch, ArgumentsHost, ExceptionFilter } from '@nestjs/common';

import { RpcException } from '@nestjs/microservices';

@Catch(RpcException)
export class RpcCustomExceptionFilter implements ExceptionFilter {
  catch(exception: RpcException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse();

    const rpcError = exception.getError();

    if(rpcError.toString().includes('Empty response')){
      return response.status(500).json({
        status: 500,
        //no me interesa mandar el string del messagePattern en el manejo de la
excepción. Es info valiosa
        //uso toSubstring para quedarme con el error desde el principio hasta el
paréntesis (donde aparece el string del controlador del microservicio) y le resto
1 para que no incluya el paréntesis
        message: rpcError.toString().substring(0, rpcError.toString().indexOf('(',
-1))
      })
    }

    if (
      typeof rpcError === 'object' &&
      'status' in rpcError &&
      'message' in rpcError
    ) {
      const status = isNaN(+rpcError.status) ? 400 :+rpcError.status;
      return response.status(status).json(rpcError);
    }

    response.status(400).json({
      status: 400,
      message: rpcError,
    });
  }
}
```

Monorepo o no monorepo

- Para usar submódulos paso el README

````
## Dev

1. Clonar el repositorio
2. Crear un .env basado en el .env.template
3. Ejecutar el comando `git submodule update --init --recursive` para reconstruir los sub-módulos
4. Ejecutar el comando `docker compose up --build`


### Pasos para crear los Git Submodules

1. Crear un nuevo repositorio en GitHub
2. Clonar el repositorio en la máquina local
3. Añadir el submodule, donde `repository_url` es la url del repositorio y `directory_name` es el nombre de la carpeta donde quieres que se guarde el sub-módulo (no debe de existir en el proyecto)
```
git submodule add <repository_url> <directory_name>
```
4. Añadir los cambios al repositorio (git add, git commit, git push)
Ej:
```
git add .
git commit -m "Add submodule"
git push
```
5. Inicializar y actualizar Sub-módulos, cuando alguien clona el repositorio por primera vez, debe de ejecutar el siguiente comando para inicializar y actualizar los sub-módulos
```
git submodule update --init --recursive
```
6. Para actualizar las referencias de los sub-módulos
```
git submodule update --remote
```



## Importante
Si se trabaja en el repositorio que tiene los sub-módulos, **primero actualizar y hacer push** en el sub-módulo y **después** en el repositorio principal.

Si se hace al revés, se perderán las referencias de los sub-módulos en el repositorio principal y tendremos que resolver conflictos.
````

- En la raíz del Launcher que alberga orders, products, etc coloco el .gitmodules con las direcciones de los repos (en este caso de Herrera)

-.gitmodules

```
[submodule "client-gateway"]
    path = client-gateway
    url = https://github.com/Nest-Microservices-DevTalles/client-gateway.git
[submodule "products-ms"]
    path = products-ms
    url = https://github.com/Nest-Microservices-DevTalles/products-
microservice.git
[submodule "orders-ms"]
    path = orders-ms
    url = https://github.com/Nest-Microservices-DevTalles/orders-microservice.git
```

Trabajar basado en el Launcher

- Aquí fernando ha dado seguimiento a los archivos fuera de los microservicios como el docker-compose.yml y demás, y en submodulos los microservicios y el client-gateway
- Puedo borrar el launcher de Docker y volverlo a levantar sin problemas
- Si quiero trabajar solo con products-ms puedo comentar el orders-ms y la db en el docker-compose.yml

---

NEST MICROSERVICIOS - PAYMENTS

- Primero trabajaré con payments-ms fuera de la red Docker
- Será a través de una petición POST que realizaré la transacción
- Estableceré y configuraré la conexión con Stripe, la secret-key en una variable de entorno que genraré en la web de Stripe y le pasaré a la nueva instancia de Stripe en el servicio tras validarla en el archivo envs.
- Stripe hará lo que tenga que hacer (puede realizar el pago o cancelar) y mediante un webhook lo confirmaré

Configuración

- Creo una API (todavía no va a ser un microservice) fuera del Launcher con nest new paymenst-ms
- Borro lo que no necesito
- Creo el archivo de config/envs, requiere dos instalaciones dotenv i joi
- De momento solo condfiguraré el puerto

```
import 'dotenv/config';
import * as joi from 'joi';

interface EnvVars {
```

```
  PORT: number;

}

const envsSchema = joi.object({
  PORT: joi.number().required(),

})
.unknown(true);

const { error, value } = envsSchema.validate( process.env );


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

const envVars:EnvVars = value;


export const envs = {
  port: envVars.PORT
}
```

- En .env

```
PORT=3003
```

- Creemos un Logger en el main

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

async function bootstrap() {

  const logger = new Logger('Payments-microservice')


  const app = await NestFactory.create(AppModule);
  await app.listen(envs.port);
  logger.log(`Server running on http://localhost:${envs.port}`)
}
bootstrap();
```

- Creo dos métodos en el controlador

```
import { Controller, Get, Post } from '@nestjs/common';
import { PaymentsService } from './payments.service';

@Controller('payments')
export class PaymentsController {
  constructor(private readonly paymentsService: PaymentsService) {}

  @Post('create-payment-session')
  createPaymentSession(){
    return this.paymentsService.createPaymentSession();
  }

  @Get('success')
  success(){
    return this.paymentsService.success();
  }

  @Get('cancel')
  cancel(){
    return this.paymentsService.cancel();
  }

  @Post('webhool')
  async stripeWebhook(){
    return this.paymentsService.stripeWebhook();
  }
}
```

- Instalo el paquete de Stripe con npm i
- En el servicio creo una nueva instancia de Stripe y declaro los métodos que llamo desde el controlador

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

@Injectable()
export class PaymentsService {

  createPaymentSession(): string {
    return 'This action adds a new payment session';
  }

  success(): string {
    return 'This action returns success';
  }

  cancel(): string {
    return 'This action returns cancel';
  }

  stripeWebhook(): string {
    return 'This action returns webhook';
  }
```

```
}
```

Configuración de Stripe

- Para obtener la secret_key debo poner Stripe en modo Test y acudir al apartado Developer una vez logeado
- Instalo stripe con npm i strip
- Tienes la documentación en stripe docs
- Valido la env STRIPE_SECRET en mi archivo de configuración /config/envs

```
import 'dotenv/config';
import * as joi from 'joi';

interface EnvVars {
  PORT: number;
  STRIPE_SECRET_KEY: string;

}

const envsSchema = joi.object({
  PORT: joi.number().required(),
  STRIPE_SECRET_KEY: joi.string().required(),
})
.unknown(true);

const { error, value } = envsSchema.validate( process.env );


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

const envVars:EnvVars = value;


export const envs = {
  port: envVars.PORT,
  stripeSecretKey: envVars.STRIPE_SECRET_KEY
}
```

- En payments.service creo la instancia de Stripe con la secret_key de la env

```
import { Injectable } from '@nestjs/common';
import { envs } from 'src/config/envs';
import Stripe from 'stripe';
```

```
@Injectable()
export class PaymentsService {

  private readonly stripe = new Stripe(envs.stripeSecretKey)

  createPaymentSession(): string {
    return 'This action adds a new payment session';
  }

  success(): string {
    return 'This action returns success';
  }

  cancel(): string {
    return 'This action returns cancel';
  }

  stripeWebhook(): string {
    return 'This action returns webhook';
  }

}
```

Crear sesión de pago

- En el método de payments.service createPaymentSession

```
import { Injectable } from '@nestjs/common';
import { envs } from 'src/config/envs';
import Stripe from 'stripe';

@Injectable()
export class PaymentsService {

  private readonly stripe = new Stripe(envs.stripeSecretKey)

  async createPaymentSession() {
    const session = await this.stripe.checkout.sessions.create({
      //colocar aquí el id de mi order
      payment_intent_data:{
        metadata:{
          order_id: 'order_123456789'
        }
      },
      //aquí van los items que la gente está comprando
      line_items:[
        {
          price_data:{
            currency: 'eur',
            product_data:{   //product_data es para crear el producto en el
```

```
momento. product es para referenciar un producto ya creado en Stripe
              name: 'T-shirt'
            },
            unit_amount: 2000 //esto equivale a 20 euros (el precio del producto).
No permite decimales como 20.00
          },
          quantity: 2 //20*2 = 40 eur
        }
      ],
      mode:'payment',
      success_url: 'http://localhost:3003/payments/success',
      cancel_url: 'http://localhost:3003/payments/cancel'

     })

     return session
  }


  success(): string {
    return 'This action returns success';
  }

  cancel(): string {
    return 'This action returns cancel';
  }

  stripeWebhook(): string {
    return 'This action returns webhook';
  }

}
```

- Si voy al endpoint localhot:3003/payments/create-payment-session con POST obtengo esto


```
{
  "id": "cs_test_a15xIlRB5KXo4TwX0rNjfm6Cw554MK22xFKxSUevuI6sqcZQcSXjq7abu4",
  "object": "checkout.session",
  "after_expiration": null,
  "allow_promotion_codes": null,
  "amount_subtotal": 4000,
  "amount_total": 4000,
  "automatic_tax": {
    "enabled": false,
    "liability": null,
    "status": null
  },
  "billing_address_collection": null,
  "cancel_url": "http://localhost:3003/payments/cancel",
  "client_reference_id": null,
  "client_secret": null,
  "consent": null,
```

```
    "consent_collection": null,
    "created": 1715056083,
    "currency": "eur",
    "currency_conversion": null,
    "custom_fields": [],
    "custom_text": {
      "after_submit": null,
      "shipping_address": null,
      "submit": null,
      "terms_of_service_acceptance": null
    },
    "customer": null,
    "customer_creation": "if_required",
    "customer_details": null,
    "customer_email": null,
    "expires_at": 1715142483,
    "invoice": null,
    "invoice_creation": {
      "enabled": false,
      "invoice_data": {
        "account_tax_ids": null,
        "custom_fields": null,
        "description": null,
        "footer": null,
        "issuer": null,
        "metadata": {},
        "rendering_options": null
      }
    },
    "livemode": false,
    "locale": null,
    "metadata": {},
    "mode": "payment",
    "payment_intent": null,
    "payment_link": null,
    "payment_method_collection": "if_required",
    "payment_method_configuration_details": null,
    "payment_method_options": {
      "card": {
        "request_three_d_secure": "automatic"
      }
    },
    "payment_method_types": [
      "card"
    ],
    "payment_status": "unpaid",
    "phone_number_collection": {
      "enabled": false
    },
    "recovered_from": null,
    "saved_payment_method_options": null,
    "setup_intent": null,
    "shipping_address_collection": null,
    "shipping_cost": null,
```

```
  "shipping_details": null,
  "shipping_options": [],
  "status": "open",
  "submit_type": null,
  "subscription": null,
  "success_url": "http://localhost:3003/payments/success",
  "total_details": {
    "amount_discount": 0,
    "amount_shipping": 0,
    "amount_tax": 0
  },
  "ui_mode": "hosted",
  "url":
"https://checkout.stripe.com/c/pay/cs_test_a15xIlRB5KXo4TwX0rNjfm6Cw554MK22xFKxSUe
vuI6sqcZQcSXjq7abu4#fidkdWxOYHwnPyd1blpxYHZxWjA0VUFUfHBGcmFNZD1cR2JxV25gNDB8cFFIdj
ZcbkdLRFJiZHZvUFBdY2l1NDddUFNvTHdkY3RuQj1TSVxUcEZ9NFRGPEY0b11PfT1VVVI3M1RBV25Of2Bn
NTV3Xz1UcFFzYCcpJ2N3amhWYHdzYHcnP3F3cGApJ2lkfGpwcVF8dWAnPyd2bGtiaWBabHFgaCcpJ2BrZG
dpYFVpZGZgbWppYWB3dic%2FcXdwYHgl"
}
```


- Si le doy a la url del final me lleva a la pantalla de stripe con la opción de pagar 40 eur y para poner los datos de la tarjeta
- Relleno los datos con datos ficticios (usar 4242 4242 4242 4242 para la tarjeta)
- Una vez hecho el pago me redirecciona al endpoint success y puedo ver el pago desde la web de stripe como pago exitoso

---

Payment Session DTO

- En lugar de poner la información en duro, los datos que van en create-payment-session vendrá desde otro microservicio
- Para trabajar con dtos hay queinstalar class-validator class-transformer
- Hay que configurar el globalPipes en el main

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

async function bootstrap() {

  const logger = new Logger('Payments-microservice')


  const app = await NestFactory.create(AppModule);

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

```
  await app.listen(envs.port);
  logger.log(`Server running on http://localhost:${envs.port}`)
}
bootstrap();
```

- En el dto

```
import { Type } from "class-transformer";
import { ArrayMinSize, IsArray, IsNumber, IsPositive, IsString, ValidateNested }
from "class-validator";

export class PaymentSessionDto {

    @IsString()
    currency: string

    @IsArray()
    @ArrayMinSize(1)
    @ValidateNested({each:true})
    @Type(()=>PaymentSessionItemDto) //transformo items en instancias de
PaymentSessionItemDto
    items: PaymentSessionItemDto[]
}


export class PaymentSessionItemDto{
    @IsString()
    name: string

    @IsNumber()
    @IsPositive()
    price: number

    @IsNumber()
    @IsPositive()
    quantity: number
}
```

- Recojo del Body en el controller y valido con el dto

```
@Post('create-payment-session')
  createPaymentSession(@Body()paymentSessionDto: PaymentSessionDto){
    return this.paymentsService.createPaymentSession(paymentSessionDto);
  }
```

- En el servicio recojo el dto
- Debo pasarle al endpoint el objeto que me pide el dto en el body de POSTMAN o similares

```
{
  "currency": "eur",
  "items":[
    {
      "name": "Preservativos",
      "price": 18.03,
      "quantity": 1
    }]
}
```

- En el servicio extraemos con desestructuración la data para poder trabajar con ella
- Hago un map de items y cojo el objeto que me pide la documentación y lo reconstruyo con mi data
- Como el precio puede venir en decimales uso Math.round para redondear y le debo añadir 2 ceros para dejarlo en el formato de stripe
- Le paso el lineItems a la session

```
  async createPaymentSession(paymentSessionDto:PaymentSessionDto) {

    const {currency, items} = paymentSessionDto

    const lineItems= items.map(({name, price, quantity})=>{
      return {
        price_data:{
          currency,
          product_data:{
            name
          },
          unit_amount: Math.round(price *100)
        },
        quantity
      }
    })

    const session = await this.stripe.checkout.sessions.create({
      //colocar aquí el id de mi order
      payment_intent_data:{
        metadata:{
          order_id: 'order_123456789'
        }
      },
      //aquí van los items que la gente está comprando
      line_items: lineItems,
      mode:'payment',
      success_url: 'http://localhost:3003/payments/success',
      cancel_url: 'http://localhost:3003/payments/cancel'

     })

     return session
  }
```

- Ahora falta configurar el webhook para ser notificados cuando el pago se haya realizado

---

Probando webhooks de stripe

- Caundo realizo un pago, Stripe mediante un POST manda a llamar el webhook y lo envia a mi endpoint
- Tengo que controlar que sea stripe, con la firma de stripe
- En la web de Stripe voy a Developers/webhooks (hay muchos!)
- Añadir punto de conexión: podemos probar test in a local environment o directamente con un endpoint
- Para el test in local environment hay que instalar el cliente de stripe (un zip, darle al exe y configrar el path). Se puede hacer a traves de Docker
- Usaré un endpoint real directamente, selecciono los eventos a escuchar
- Al lado derecho tenemos el código a implementar
- Usa express, y nosotros también (debajo de Nest)
- Dice que mandemos el body como un raw y eso puede ser un poco tedioso si queremos crear un middleware, pero Nest facilita mucho la faena
- En el main, en app, coloco el rawBody en true

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

async function bootstrap() {

  const logger = new Logger('Payments-microservice')


  const app = await NestFactory.create(AppModule,{
    rawBody: true //esto va a mandar el body como un buffer que es exactamente lo
que me piden
  });

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

  await app.listen(envs.port);
  logger.log(`Server running on http://localhost:${envs.port}`)
}
bootstrap();
```

- Va a venir la firma en los headers de la petición. Necesito la request

```
stripeWebhook(req: Request, res: Response) {
    const sig = req.headers['stripe-signature'];
    return res.status(200).json({sig})
  }
```

- En el controller ocupo tomar la request

```
@Post('webhook')
async stripeWebhook(@Req() req: Request,@Res() res:Response){
  return this.paymentsService.stripeWebhook(req,res);
}
```

Implementar el Webhook

- Si procesamos el Body vamos a tener un error, porque Stripe lo verifica
- Por eso lo tomamos directamente de la Request
- Creo un evento
- Copio el endpointSecret de la documentación
- En un try catch, de la req tomo el rawBody (hay que escribirlo asi), el signature y el endpointSecret

```
  stripeWebhook(req: Request, res: Response) {
    const sig = req.headers['stripe-signature'];


    let event: Stripe.Event
    const endpointSecret =
"whsec_c74a813338de786ac11af0b167e3b53e74f63303a960b7cc6adcc5f585cf088d";


    try {
      event = this.stripe.webhooks.constructEvent(req['rawBody'], sig,
endpointSecret);
    } catch (err) {
      res.status(400).send(`Webhook Error: ${err.message}`);
      return;
    }

    return res.status(200).json({sig})
  }
```

- Si no hace match con el endpointSecret mandará un error
- Si estoy probandolo localmente y he instalado el cliente puedo usar stripe trigger payment_intent.succeeded
- Esto me devuelve charge.succeded, payment_intent.succeded y payment_intent.created (Stripe primero crea un intento de pago)

- Charge succeded es la confirmación
- Si pruebo el webhook con un aurl, no voy a poder usar localhost
- Vamos a tener que usar un Proxy o algo intermedio

Hookdeck - EventGateway - Forwarder

- Podemos usar un switch para tomar el tipo de evento que me devuelve el webhook y realizar una acción

```
async stripeWebhook(req: Request, res: Response) {
    const sig = req.headers['stripe-signature'];


    let event: Stripe.Event
    const endpointSecret =
"whsec_c74a813338de786ac11af0b167e3b53e74f63303a960b7cc6adcc5f585cf088d";


    try {
      event = this.stripe.webhooks.constructEvent(req['rawBody'], sig,
endpointSecret);
    } catch (err) {
      res.status(400).send(`Webhook Error: ${err.message}`);
      return;
    }

    switch(event.type){
      case 'charge.succeeded':
        //llamar al microservicio
        break;

        default:
        console.log(`Evento  ${event} not handled`)
    }

    return res.status(200).json({sig})
  }
```

- Para obtener la url usaremos hoockdeck
- hay otros: smee (no funciona con stripe porque procesa el body)
- le coloco un nombre cualquiera al endpoint fuente stripe-to-localhost
- Le coloco otronombre cualquiera al endpoint destino to-localhost
- Instalo el CLI

```
npm install hookdeck-cli -g
```

- En endpoint url coloco el localhost:3003/payments/webhook (solo payments/webhook)
- recibo un url. Lo debo colocar en la parte de Stripe

> https://hkdk.events/nan95cp9850tx9

- Ahora solo tengo que usar el CLI

> hookdeck login hookdeck listen 3003 stripe-to-localhost

- Si apunto con un POST a http://localhost:3003/payments/webhook debería darme un error como Webhook Error: No webhook payload was provided.
- Todavía no hemos añadido el string a stripe
- En developers/webhooks/añadir punto de conexión coloco la uRL
- Escucho el evento charge.failed y charge.succeeded
- Añadir evento
- Copio el secreto de firma de la pantalla esperando eventos... (el de ahora es el real, el anterior era de testing)
- Hago todo el proceso de crear y pagar y me manda a la pantalla de this action returns success que es lo que tengo en el endpoint de success
- Puedo hacer console.logs para chequear eventos y demás

Enviar y recibir la Id de la orden

- Necesito el orderId (UUID) de mi orders-ms
- Puedo grabar el id de la transacción en order, o el URL de receipt_url del recibo (conveniente)
- En PaymentSessionDto creo el orderId como string

```
export class PaymentSessionDto {

    @IsString()
    orderId: string

    @IsString()
    currency: string

    @IsArray()
    @ArrayMinSize(1)
    @ValidateNested({each:true})
    @Type(()=>PaymentSessionItemDto)
    items: PaymentSessionItemDto[]
}
```

- En el payments.service.create extraigo la orderId con desestructuración
- Lo coloco en la metadata de la session
- Podemos tomar la información del evento en el switch
- **NOTA**: he configurado las variables de entorno

```
import { Injectable } from '@nestjs/common';
import { envs } from 'src/config';
import Stripe from 'stripe';
```

```
import { PaymentSessionDto } from './dto/payment-session.dto';
import { Request, Response } from 'express';

@Injectable()
export class PaymentsService {
  private readonly stripe = new Stripe(envs.stripeSecret);

  async createPaymentSession(paymentSessionDto: PaymentSessionDto) {
    const { currency, items, orderId } = paymentSessionDto; /7extraigo la orderId

    const lineItems = items.map((item) => {
      return {
        price_data: {
          currency: currency,
          product_data: {
            name: item.name,
          },
          unit_amount: Math.round(item.price * 100),
        },
        quantity: item.quantity,
      };
    });

    const session = await this.stripe.checkout.sessions.create({
      // Colocar aquí el ID de mi orden
      payment_intent_data: {
        metadata: {
          orderId: orderId
        },
      },
      line_items: lineItems,
      mode: 'payment',
      success_url: envs.stripeSuccessUrl,
      cancel_url: envs.stripeCancelUrl,
    });

    return session;
  }

  async stripeWebhook(req: Request, res: Response) {
    const sig = req.headers['stripe-signature'];

    let event: Stripe.Event;

    // Real
    const endpointSecret = envs.stripeEndpointSecret;

    try {
      event = this.stripe.webhooks.constructEvent(
        req['rawBody'],
        sig,
        endpointSecret,
      );
    } catch (err) {
```

```
      res.status(400).send(`Webhook Error: ${err.message}`);
      return;
    }

    switch( event.type ) {
      case 'charge.succeeded':
        const chargeSucceeded = event.data.object;
        // TODO: llamar nuestro microservicio
        console.log({
          metadata: chargeSucceeded.metadata,
          orderId: chargeSucceeded.metadata.orderId,
        });
      break;

      default:
        console.log(`Event ${ event.type } not handled`);
    }

    return res.status(200).json({ sig });
  }
}
```

- Ahora debo hacer un pago. Recuerda tener el hoockdeck corriendo en la terminal
- **NOTA**: Estoy teniendo problemas con hookdeck, usaré ngrok
- Abrir cmd como admin

```
choco install ngrok
```

```
ngrok http 3003
```

- En stripe coloco el endpoint con la ruta que me da ngrok

```
https://03b6-2-152-179-155.ngrok-free.app/payments/webhook
```

- En la documentacion dice que debes añadir la clave secreta del webhook que te proporciona stripe con

```
ngrok http 8080 --verify-webhook=stripe --verify-webhook-secret=mySecret
```

- Necesito obtener el UUID de orders-microservices

Launcher Dockerizado NEST MS

- Vamos a crear contenedores para cada microservicio
- Cada uno de estos directorios sea una imagen de Docker, y que con hacer docker run y el nombre del microservicio se eche a correr
- Vamos a tenr que crear las imágenes basadas en una arquitectura de procesador según dónde vayan a correr
- De múltiples arquitecturas, para amd64, x86, etc
- Con CI/CD podremos construir la imagen en la nube, no será mi pc quién la construya

- Se desplegará automáticamente a través de Kubernetes (hay que habilitarlo desde DockerDesktop, reiniciar y darle a Reset Kubernetes cluster)
- En producción no voy a querer correr nest start:dev
- De hecho no necesito a nest para hacer el build, lo puedo hacer con node

---

Construir a producción

- En producción no voy a querer correr nest start:dev
- De hecho no necesito a nest para hacer el build, lo puedo hacer con node
- Puedo hacer un build con **npm run build** del client-gateway y correr con **node dist/main.js**
- Dará error porque no se está comunicando con nadie a través de NATS, pero devuelve un error 500 al llamar a un endpoint desde POSTMAN
- Para crear la imagen de Docker para producción de client-gateway, crearemos Dockerfile.prod
- Para reducir el tamaño de la imagen de Docker y eviatr usar el usuario root para la construcción de la imagen
- Haremos el multi-stage build
- Dividiremos el dockerfile en 3 etapas: dependencias, builder, crear la imagen final
- Para identificar que viene una nueva etapa se hace una refrencia a una nueva imagen, en este caso node:21, la llamo build
- Me muevo al working directory
- Se recomienda que los comandos que tienden menos a cambiar se coloquen al front, ya que pueden mantenerse en caché
- Copio de deps (así llamé a la primera imagen donde instalé las dependencias) las dependencias que hay en node_modules al ./node_modules del working directory
- La ruta es la misma que indiqué en la imagen anterior
- Puedo limpiar los módulos de node que no necesito y dejar solo los de producción. Esto reducirá el peso de mi imagen a la mitad
- Agrego una nueva etapa que llamo prod para la imagen final
- Siempre indico el working directory, el directorio donde estoy trabajando, donde trabaja la imagen de node:21
- Copio de la imagen anterior llamada build los node_modules
- Copio el codigo fuente de la carpeta dist a ./dist (la carpeta dist del working directory) que ya tiene el código de JS
- Le indico que la variable NODE_ENV será de production
- Hago uso del usuario node porque el usuario que viene por defecto en la imagen de node:21 tiene demasiados privilegios
- Expongo el puerto 3000. Los contenedores son pequeñas computadoras virtuales, no habrá ningún conflicto de puertos
- Ejecuto el comando de node dist/main.js

```
# Dependencias
FROM node:21-alpine3.19 as deps

WORKDIR /usr/src/app

COPY package.json ./
```

```
COPY package-lock.json ./

RUN npm install


# Builder - Construye la aplicación
FROM node:21-alpine3.19 as build

WORKDIR /usr/src/app

# Copiar de deps, los módulos de node
COPY --from=deps /usr/src/app/node_modules ./node_modules

# Copiar todo el codigo fuente de la aplicación
COPY . .

# RUN npm run test
RUN npm run build

# para limpiar los módulos de node que no necesito y dejar solo los de producción
(opcional)
# reducirá el peso de la imagen
RUN npm ci -f --only=production && npm cache clean --force



# Crear la imagen final de Docker
FROM node:21-alpine3.19 as prod

WORKDIR /usr/src/app


COPY --from=build /usr/src/app/node_modules ./node_modules

# Copiar la carpeta de DIST
COPY --from=build /usr/src/app/dist ./dist


ENV NODE_ENV=production

USER node


EXPOSE 3000

# El comando a ejecutar para correr la imagen
CMD [ "node", "dist/main.js" ]
```

- Uso docker build -f nombredelarchivo -t (de tag) el nombre de la aplicación y punto para que busque el archivo en el contexto dónde está

> docker build -f dockerfile.prod -t client-gateway .

- En containers le doy al play y en optional settings introduzco el nombre client-gateway-prod
- En ports coloco 3000 : 3000/tcp
- En environment variables coloco **variable=PORT value=3000 y NATS_SERVERS value=nats://nats:4222**

Launcher

- Quiero que mediante un comando pueda construirlas todas
- Es casi idéntico que el docker-compose.yml pero más simple, ya que no hay que exponer ciertos puertos ni ciertos comandos porque las imágenes ya estarán construidas
- No ocupamos los puertos del nats-server porque eran puertos de monitoreo
- Necesitamos que client-gateway apunte a mi Dockerfile.prod
- En build le indico el contexto y el dockerfile
- Le indico el nombre de la imagen con image
- El puerto y los volúmenes no son necesarios en producción. El puerto es necesario solo para ejecutar.
- El volumen es para tener un enlace entre mi src y el src del contenedor, para que cuando haya un cambio se recargue todo. Tampoco lo necesito
- El command no es necesario ni las variables de entorno tampoco, ya está especificado en el dockerfile
- Si usara el docker compose con up si necesitaría el puerto y las variables de entorno
- Indicamos el resto de imagenes en el docker-compose
- Para orders necesito el aprovisionamiento de una base de datos postgres en la nube, porque no voy a usar ni levantar una imagen de postgres dede aqui
- Usaremos Serverless Postgres, con la version de postgres 16 que me dará una cadena de conexión
- El de payments se conecta a través del puerto ya que es híbrido y se comunica por http y por NATS
- En orders uso args para mandarle un argumento en el momento de la construcción que es la variable de entorno de la db postgres en la nube
-

```
version: '3'


services:

  nats-server:
    image: nats:latest

  client-gateway:
    build:
      context: ./client-gateway
      dockerfile: dockerfile.prod
    image: client-gateway-prod
    ports:
      - ${CLIENT_GATEWAY_PORT}:${CLIENT_GATEWAY_PORT}
    environment:
      - PORT=${CLIENT_GATEWAY_PORT}
      - NATS_SERVERS=nats://nats-server:4222
```

```
  auth-ms:
    build:
      context: ./auth-ms
      dockerfile: dockerfile.prod
    image: auth-ms
    environment:
      - PORT=3000
      - NATS_SERVERS=nats://nats-server:4222
      - DATABASE_URL=${AUTH_DATABASE_URL}
      - JWT_SECRET=${JWT_SECRET}

  products-ms:
    build:
      context: ./products-ms
      dockerfile: dockerfile.prod
    image: products-ms
    environment:
      - PORT=3000
      - NATS_SERVERS=nats://nats-server:4222
      - DATABASE_URL=file:./dev.db


  # Orders MS
  orders-ms:
    build:
      context: ./orders-ms
      dockerfile: dockerfile.prod
      # uso args para mandarle la variable de entorno con el string de conexión
      args:
        - ORDERS_DATABASE_URL=${ORDERS_DATABASE_URL}
    image: orders-ms
    environment:
      - PORT=3000
      - DATABASE_URL=${ORDERS_DATABASE_URL}
      - NATS_SERVERS=nats://nats-server:4222


# ======================
# Payments Microservice
# ======================
  payments-ms:
    build:
      context: ./payments-ms
      dockerfile: dockerfile.prod
    image: payments-ms
    ports:
      - ${PAYMENTS_MS_PORT}:${PAYMENTS_MS_PORT}
    environment:
      - PORT=${PAYMENTS_MS_PORT}
      - NATS_SERVERS=nats://nats-server:4222
      - STRIPE_SECRET=${STRIPE_SECRET}
      - STRIPE_SUCCESS_URL=${STRIPE_SUCCESS_URL}
      - STRIPE_CANCEL_URL=${STRIPE_CANCEL_URL}
      - STRIPE_ENDPOINT_SECRET=${STRIPE_ENDPOINT_SECRET}
```

- El Dockerfile-prod de auth, es prácticamente el mismo que el de client-gateway
- Solo que tengo que usar prisma generate, genera el cliente y lo almacena en los módulos de node
- Se podrían aplicar migraciones en este punto, siempre y cuando la db esté definida

```
# Dependencias
FROM node:21-alpine3.19 as deps

WORKDIR /usr/src/app

COPY package.json ./
COPY package-lock.json ./

RUN npm install



# Builder - Construye la aplicación
FROM node:21-alpine3.19 as build

WORKDIR /usr/src/app

# Copiar de deps, los módulos de node
COPY --from=deps /usr/src/app/node_modules ./node_modules

# Copiar todo el codigo fuente de la aplicación
COPY . .

# RUN npm run test
RUN npm run build

RUN npm ci -f --only=production && npm cache clean --force

RUN npx prisma generate



# Crear la imagen final de Docker
FROM node:21-alpine3.19 as prod

WORKDIR /usr/src/app


COPY --from=build /usr/src/app/node_modules ./node_modules

# Copiar la carpeta de DIST
COPY --from=build /usr/src/app/dist ./dist


ENV NODE_ENV=production
```

```
USER node


EXPOSE 3000

CMD [ "node", "dist/main.js" ]
```

- En el Dockerfile.prod de orders

```
# Dependencias
FROM node:21-alpine3.19 as deps

WORKDIR /usr/src/app

COPY package.json ./
COPY package-lock.json ./

RUN npm install




# Builder - Construye la aplicación
FROM node:21-alpine3.19 as build

# necesito pasarle la variable de entorno de la db postgres en la nube con args,
la recojerá del docker-compose.prod.yml
ARG ORDERS_DATABASE_URL
# le digo que la DATABASE_URL, que es la variable de entorno que uso en
environment del docker-compose.prod.yml es esta variable de args
ENV DATABASE_URL=$ORDERS_DATABASE_URL
# todo esto es necesario para que prisma recoja la variable de entorno para que
cuando llegue a la migración tenga la url y se pueda ejecutar

WORKDIR /usr/src/app

# Copiar de deps, los módulos de node
COPY --from=deps /usr/src/app/node_modules ./node_modules

# Copiar todo el codigo fuente de la aplicación
COPY . .


# aqui se puede hacer la migración, para este punto la db debería de existir
RUN npx prisma migrate deploy
RUN npx prisma generate


# RUN npm run test
RUN npm run build
```

```
RUN npm ci -f --only=production && npm cache clean --force




# Crear la imagen final de Docker
FROM node:21-alpine3.19 as prod

WORKDIR /usr/src/app


COPY --from=build /usr/src/app/node_modules ./node_modules

# Copiar la carpeta de DIST
COPY --from=build /usr/src/app/dist ./dist


ENV NODE_ENV=production

USER node


EXPOSE 3000

CMD [ "node", "dist/main.js" ]
```

- En el Dockerfile de products

```
# Dependencias
FROM node:21-alpine3.19 as deps

WORKDIR /usr/src/app

COPY package.json ./
COPY package-lock.json ./

RUN npm install




# Builder - Construye la aplicación
FROM node:21-alpine3.19 as build

WORKDIR /usr/src/app

# Copiar de deps, los módulos de node
COPY --from=deps /usr/src/app/node_modules ./node_modules

# Copiar todo el codigo fuente de la aplicación
COPY . .
```

```
# RUN npm run test
RUN npm run build

RUN npm ci -f --only=production && npm cache clean --force

# si la variable de entorno no está definida este comando no se ejecutará
# uso args en el docker-compose.prod.yml
RUN npx prisma generate



# Crear la imagen final de Docker
FROM node:21-alpine3.19 as prod

WORKDIR /usr/src/app


COPY --from=build /usr/src/app/node_modules ./node_modules

# Copiar la carpeta de DIST
COPY --from=build /usr/src/app/dist ./dist
# Hay que clonar la carpeta de prisma desde build
COPY --from=build /usr/src/app/prisma ./prisma


ENV NODE_ENV=production

USER node


EXPOSE 3000

CMD [ "node", "dist/main.js" ]
```

> docker compose -f docker-compose.prod.yml build