10 NEST Autenticación

- En esta sección vamos a hacer decoradores personalizados
- Las rutas GET serán públicas, crear, actualizar y borrar si necesitarán autenticación de admin
- Vamos a hacer modificaciones en el SEED para crear usuarios automáticamente en la db y revalidar tokens (en realidad generar uno nuevo basado en el anterior)
- Van a haber varios endpoints nuevos como login, create user, check auth status
- También veremos encriptación de contraseñas
- Hay mucho concepto nuevo en esta sección

Entidad de usuarios

- Voy a proteger rutas. Habrá rutas que solo las podrán ver usuarios con el rol de administrador, por ejemplo
- El objetivo de la entidad es tener una relación entre la db y la aplicación de Nest
- Corresponde a una tabla en la db
- La renombro a user.entity
- Le coloco el decorador **@Entity** de , le paso el nombre 'users'
- No se recomienda usar el mail de id, porque este puede cambiar y dar dolores de cabeza
- Para decirle que es un identificador único uso el decorador **@PrimaryGeneratedColumn**
    - Si no le coloco nada será un numero autoincremental, vamos a manejarlo con uuid
- El isActive servirá para un borrado suave, donde permaneceran los datos pero con el isActive en false
- En el rol le pongo user como valor por defecto
- user.entity

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

@Entity('users')
export class User{

    @PrimaryGeneratedColumn('uuid')
    id: string

    @Column('text',{
        unique: true
    })
    email: string


    @Column('text')
    password: string

    @Column('text')
    fullName: string

    @Column('bool',{
        default: true
```

```
    })
    isActive: boolean

    @Column('text',{
        array: true,
        default:['user']
    })
    roles: string[]
}
```

- Para usar la entidad debo especificar en el módulo en imports con **TypeOrmModule** y **forFeature** las entidades que quiero utilizar
- Lo exporto por si lo quiero usar en otro módulo
- En auth.module

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';

@Module({
  controllers: [AuthController],
  providers: [AuthService],
  imports: [
    TypeOrmModule.forFeature([User])
  ],
  exports: [TypeOrmModule]
})
export class AuthModule {}
```

Crear Usuario

- Para crear el usuario voy a usar el endpoint register

> http://localhost:3000/api/auth/register

- Lo añado al controlador
- Borro los dtos y creo CreateUserDto (actualizo también el servicio borrando todo menos el create)

```
import { Controller, Get, Post, Body, Patch, Param, Delete } from
'@nestjs/common';
import { AuthService } from './auth.service';
import { CreateUserDto } from './dto/create-user.dto';


@Controller('auth')
export class AuthController {
```

```
  constructor(private readonly authService: AuthService) {}

  @Post('register')
  create(@Body() createUserDto: CreateUserDto) {
    return this.authService.create(createUserDto);
  }
}
```

- En el dto necesito el email, password y fullName
- Usaré una expresión regular para validar el password

```
import { IsEmail, IsString, Matches, MaxLength, MinLength } from "class-validator"
import { Unique } from "typeorm"

export class CreateUserDto{


    @IsEmail()
    email: string

    @IsString()
    @MinLength(1)
    fullName: string

    @IsString()
    @MinLength(6)
    @MaxLength(50)
    @Matches(
    /(?:(?=.*\d)|(?=.*\W+))(?![.\n])(?=.*[A-Z])(?=.*[a-z]).*$/, {
    message: 'The password must have a Uppercase, lowercase letter and a number'
    })
    password: string;
}
```

- Falta implementar la lógica en el servicio
- Siempre en un try catch, async
- El create **no hace la inserción**

```
import { Injectable } from '@nestjs/common';
import { CreateUserDto } from './dto/create-user.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { Repository } from 'typeorm';


@Injectable()
export class AuthService {

  constructor(
```

```
    @InjectRepository(User)
    private readonly userRepository: Repository<User>
  ){}

  async create(createUserDto: CreateUserDto) {

    try {
     const user=  this.userRepository.create(createUserDto)

     await this.userRepository.save(user)

     return user

    } catch (error) {
      console.log(error)
    }
  }
}
```

- Evidentemente falta encriptar el password
- Si vuelvo a enviar el mismo usuario salta un error en la terminal, código 23505
- Manejemos la excepción

```
import { BadRequestException, Injectable, InternalServerErrorException } from
'@nestjs/common';
import { CreateUserDto } from './dto/create-user.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { Repository } from 'typeorm';


@Injectable()
export class AuthService {

  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>
  ){}

  async create(createUserDto: CreateUserDto) {

    try {
     const user=  this.userRepository.create(createUserDto)

     await this.userRepository.save(user)

     return user

    } catch (error) {
      this.handleDBErrors(error)
    }
```

```
  }

  private handleDBErrors(error: any):void{    //jamás regresa un valor
    if(error.code === '23505'){
      throw new BadRequestException(error.detail)
    }
    console.log(error)

    throw new InternalServerErrorException("Check logs") //no hace falta poner el
return

  }
}
```

Encriptar contraseña

- No debio regresar la contraseña y por supuesto, debo guardarla encriptada
- Usaremos encriptación de una sola vía con bcrypt. Instalo los tipos

> npm i bcrypt npm i -D @types/bcrypt

- Importo todo como bcrypt (es una manera ligera de hacer el patrón adaptador)
- Uso la desestructuración para extraer el password
- hashSync me pide la data y el número de vueltas de encriptación, se lo paso en un objeto

```
import { BadRequestException, Injectable, InternalServerErrorException } from
'@nestjs/common';
import { CreateUserDto } from './dto/create-user.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { Repository } from 'typeorm';
import * as bcrypt from 'bcrypt'


@Injectable()
export class AuthService {

  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>
  ){}

  async create(createUserDto: CreateUserDto) {

    try {

    const {password, ...userData} = createUserDto

    const user=  this.userRepository.create({
```

```
      ...userData,
      password: bcrypt.hashSync(password, 12)
      })

     await this.userRepository.save(user)

     return user

    } catch (error) {
      this.handleDBErrors(error)
    }
  }

  private handleDBErrors(error: any):void{
    if(error.code === '23505'){
      throw new BadRequestException(error.detail)
    }
    console.log(error)

    throw new InternalServerErrorException("Check logs")
  }
}
```

- No debería regresar la contraseña
- Hay varias tecnicas
- Cuando ya se ha grabado el usuario extraigo el password
- Uso **delete**

```
async create(createUserDto: CreateUserDto) {

try {

const {password, ...userData} = createUserDto

const user=  this.userRepository.create({
    ...userData,
    password: bcrypt.hashSync(password, 12)
    })

    await this.userRepository.save(user)

    delete user.password

    return user
    //TODO: retornar JWT de acceso

} catch (error) {
    this.handleDBErrors(error)
    }
}
```

- Luego se mejorará este delete!

Login de usuario

- Creo el dto login-user.dto

```
import { IsEmail, IsString, Matches, MaxLength, MinLength } from "class-validator"


export class LoginUserDto{

    @IsEmail()
    email: string

    @IsString()
    @MinLength(6)
    @MaxLength(50)
    @Matches(
    /(?:(?=.*\d)|(?=.*\W+))(?![.\n])(?=.*[A-Z])(?=.*[a-z]).*$/, {
    message: 'The password must have a Uppercase, lowercase letter and a number'
    })
    password: string
}
```

- En el controlador creo el endpoint 'login'

```
@Post('login')
loginUser(@Body() loginUserDto: LoginUserDto){
  return this.authService.loginUser(loginUserDto)
}
```

- Creo el servicio
- Si uso esto

```
 const user = await this.userRepository.findOneBy({email})
```

- Me devuelve el objeto completo, incluido el password y yo no quiero eso
- El problema es que cuando haga relaciones y mostremos la relación con el usuario también va a venir la contraseña y otras cosas
- Para evitarlo, voy a la entidad y en la propiedad contraseña le coloco select: false
- user.entity

```
@Column('text',{
    select: false
```

```
})
password: string
```

- Cuando se haga un find no aparecerá, pero yo ahora necesito el password para validar, por lo que usaré el **where** con **findOne**
- Le paso el mail (solo puede haber 1 y está indexado)
- Le digo que seleccione los campos email y password

```
async login(loginUserDto: LoginUserDto){

    const {email, password} = loginUserDto

    const user = await this.userRepository.findOne({
      where: {email},
      select: {email: true, password: true}
    })

    return user
  }
```

- Hago la validación de si existe usuario y la comparación del password con bcrypt. Si no concuerda devuelvo un error

```
async login(loginUserDto: LoginUserDto){

  const {email, password} = loginUserDto

  const user = await this.userRepository.findOne({
    where: {email},
    select: {email: true, password: true}
  })

  if(!user){
    throw new UnauthorizedException('Credenciales no válidas (email)')
  }

  if(!bcrypt.compareSync(password, user.password)){
    throw new UnauthorizedException('Password incorrect')
  }

  return user
  //TODO: retornar JWT
}
```

Nest Authentication - Passport

- Instalación necesaria

> npm i @nestjs/passport passport @nestjs/jwt passport-jwt npm i -D @types/passport-jwt

- Hay varias estrategias para autenticarse
- En authModule debo definir 2 cosas:
  - **PassportModule**: debo decirle la estrategia que voy a usar. Empleo register (registerAsync es para modulos asíncronos)
    - registerAsync se suele usar para asegurarse que las variables de entorno estan previamente configuradas
    - También si mi configuracion del módulo depende de un servicio externo, un endpoint, etc
  - **JwtModule**: para la palabra secreta usaré una variable de entorno. Expirará en 2 horas

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';

@Module({
  controllers: [AuthController],
  providers: [AuthService],
  imports: [
    TypeOrmModule.forFeature([User]),
    PassportModule.register({defaultStrategy: 'jwt'}),
    JwtModule.register({
      secret: process.env.JWT_SECRET,
      signOptions:{
        expiresIn: '2h'
      }
    })
  ],
  exports: [TypeOrmModule]
})
export class AuthModule {}
```

- Sería mejor usar la manera asíncrona de carga del módulo para asegurarme de que la variable de entorno estará cargada

---

Modulos asíncronos

- En registerAsync tengo opciones como useClass y useExisting muy utiles en la parte del testing
- Voy a usar useFactory, es la función que voy a llamar cuando se intente registrar de manera asincrona el módulo
  - En el return envío el objeto con las opciones del jwt

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';

@Module({
  controllers: [AuthController],
  providers: [AuthService],
  imports: [
    TypeOrmModule.forFeature([User]),
    PassportModule.register({defaultStrategy: 'jwt'}),
    JwtModule.registerAsync({
        imports: [],
        inject: [],
        useFactory: ()=>{

          return {
            secret: process.env.JWT_SECRET,
            signOptions:{
              expiresIn: '2h'
            }
          }
        }
    })
  ],
  exports: [TypeOrmModule]
})
export class AuthModule {}
```

- Puedo inyectar el configService como hice anteriormente para trabajar con las variables de entorno
- Para ello importo el módulo **ConfigModule** e inyecto el servicio en **injects**
- Hago la inyección del servicio igual que lo haría en cualquier clase solo que aqui estoy en una funcion
- ConfigService me da la posibilidad de recibir el dato que yo espero, poder evaluarlo, establecer valores por defecto, etc

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';
import { ConfigModule, ConfigService } from '@nestjs/config';

@Module({
  controllers: [AuthController],
  providers: [AuthService],
```

```
  imports: [
    TypeOrmModule.forFeature([User]),
    PassportModule.register({defaultStrategy: 'jwt'}),

    JwtModule.registerAsync({

        imports: [ConfigModule],
        inject: [ConfigService],
        useFactory: (configService: ConfigService)=>{

          return {
            secret: configService.get('JWT_SECRET'),
            signOptions:{
              expiresIn: '2h'
            }
          }
        }
    })
  ],
  exports: [TypeOrmModule]
})
export class AuthModule {}
```

- Falta saber qué información voy a guardar en el jwt, como validarlo y a qué usuario de la db le corresponde

---

JwtStrategy

- Es recomendable guardar en el jwt algun campo que esté indexado patra que identifique rapidamente al usuario
- Añadir también en qué momento fue creado y la fecha de expiración
- Nunca guardar info sensible: cadenas de conexión, tarjetas de crédito, passwords, etc
- La firma encriptada asegura que el valor no haya sido modificado y que haga match
- Me interesa saber que el usuario esté activo, el rol y el id a través de su correo
- **Solo guardaré el correo en el jwt**
- Vamos a emplear una estrategia personalizada
- En auth creo un nuevo directorio llamado strategies con jwt.strategy.ts
- Esta clase extiende de PassportStrategy (@nestjs/passport) y le paso la estrategia de passport-jwt

```
import {PassportStrategy} from '@nestjs/passport'
import { Strategy } from 'passport-jwt';


export class JwtStrategy extends PassportStrategy(Strategy){


}
```

- Quiero implementar una forma de expandir la validación de jwt
- El passportStrategy va a revisar el jwt basado en la secret_key, tambien si ha expirado o no y la Strategy me va a decir si el token es válido, pero hasta ahí
- Si yo necesito saber si el usuario está activo y todo lo demás, lo haré en base a un método (lo llamaré validate)
- El payload momentaneamente lo pondré de tipo any (lo cambiaré más adelante)
- Devuelve una promesa que va a devolverme una instancia de Usuario de mi db
- Si el jwt es válido y no ha expirado, voy a recibir este payload y puedo validarlo como yo quiera

```
import {PassportStrategy} from '@nestjs/passport'
import { Strategy } from 'passport-jwt';
import { User } from '../entities/user.entity';


export class JwtStrategy extends PassportStrategy(Strategy){

    async validate(payload: any): Promise<User>{

        return
    }
}
```

- Creo el directorio interfaces en /auth para hacer la interfaz del payload
- No voy a incluir la fecha de creación ni expiración

```
export interface JwtPayloadInterface{

    email: string
    //TODO: añadir todo lo que se quiera grabar
}
```

- Se procura que el jwt no lleve mucha info porque viaja de aquí para allá, que sea liviano
- Ahora puedo desestructurar el email del payload

JwtStrategy II

- Añado la lógica para validar el payload
- El método validate **solo se va a llamar si el jwt es válido (la firma hace match) y no ha expirado**
- Necesito ir a la tabla de usuarios y buscar el correo
- Ya tengo importado el módulo user en auth.module por lo que solo debo inyectar el repositorio de usuario
- PassportStrategy me pide invocar el **constructor padre**
- Como tengo que pasarle la secret_key como variable de entorno al constructor padre inyecto el **ConfigService**
- Importo el **ConfigModule en auth.module**

- Le debo indicar también al constructor padre en qué posición voy a esperar que me manden el jwt
    - Lo puedo mandar en los headers, o como un **header de autenticación** de tipo **Bearer Token**

```
import {PassportStrategy} from '@nestjs/passport'
import { ExtractJwt, Strategy } from 'passport-jwt';
import { User } from '../entities/user.entity';
import { JwtPayloadInterface } from '../interfaces/jwt-payload.interface';
import { Repository} from 'typeorm';
import { InjectRepository } from '@nestjs/typeorm';
import { ConfigService } from '@nestjs/config';

export class JwtStrategy extends PassportStrategy(Strategy){


    constructor(

        @InjectRepository(User)
        private readonly userRepository: Repository<User>,

        private readonly configService: ConfigService
    ){
        super({
            secretOrKey: configService.get('JWT_SECRET'),
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken()
        })
    }

    async validate(payload: JwtPayloadInterface): Promise<User>{

        const {email} = payload

         const user = await this.userRepository.findOneBy({email})

        return
    }
}
```

- **Debo importar ConfigModule en imports del auth.module** (no solo en el JwtModule) ya que lo uso en este módulo
- Ahora ya puedo implementar la lógica, validar el usuario, etc
- No tengo el password. Si el token existe significa que el usuario se autenticó en su momento
- Retorno el usuario. Cuando la validación lo que yo retorne se va a añadir en la Request
    - Pasa por interceptores, por los servicios, controladores, **todo lugar donde tenga acceso a la Request**
    - Después se usarán decoradores personalizados para extraer info de la Request y hacer lo que hago en los controladores
- Todavía no he implementado el JwtStrategy, es un archivo flotando en mi app

```
async validate(payload: JwtPayloadInterface): Promise<User>{

      const {email} = payload

        const user = await this.userRepository.findOneBy({email})

        if(!user) throw new UnauthorizedException('Token not valid')

        if(!user.isActive) throw new UnauthorizedException('User is inactive')

      return user
  }
```

- Todas las estrategias son **providers**. Le añado el decorador **@Injectable**
- Como es un provider, debo indicarlo en **el módulo auth.module dónde providers**
- También lo exporto por si quiero usarlo en otro lugar. Exporto los otros módulos
- Después lo vamos a mejorar para que todo sea automático

```
@Module({
  controllers: [AuthController],
  providers: [AuthService, JwtStrategy],
  imports: [
    ConfigModule,
    TypeOrmModule.forFeature([User]),
    PassportModule.register({defaultStrategy: 'jwt'}),

    JwtModule.registerAsync({

        imports: [ConfigModule],
        inject: [ConfigService],
        useFactory: (configService: ConfigService)=>{

          return {
            secret: configService.get('JWT_SECRET'),
            signOptions:{
              expiresIn: '2h'
            }
          }
        }
    })
  ],
  exports: [TypeOrmModule, JwtStrategy, PassportModule, JwtModule]
})
export class AuthModule {}
```

Generar un JWT

- Como voy a crear un jwt en varios lugares voya crear un método en auth.service.ts

- Debo recibir **el payload** con la **info** que quiero en el jwt del tipo **JetPayloadInterface**
- Para generar el token necesito usar el servicio de jwt de nest, hago la inyección de dependencias
- Este servicio lo proporciona el JwtModule
- Uso el servicio con el método sign. Aquí podría pasarkle parámetros pero si no queda por defecto como definí en el módulo
- Esparzo con el spread mi user en el return, y añado el token
    - Si coloco directamente donde el payload user.email se me queja porque un string no cumple con el objeto de jwtPayloadInterface, así que lo meto como un objeto *{token: user.email}*
    - Hago lo mismo en el login

```
import { BadRequestException, Injectable, InternalServerErrorException,
NotFoundException, UnauthorizedException } from '@nestjs/common';
import { CreateUserDto } from './dto/create-user.dto';
import { InjectRepository } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { Repository } from 'typeorm';
import * as bcrypt from 'bcrypt'
import { LoginUserDto } from './dto/login-user.dto';
import { NotFoundError } from 'rxjs';
import { JwtPayloadInterface } from './interfaces/jwt-payload.interface';
import { JwtService } from '@nestjs/jwt';


@Injectable()
export class AuthService {

  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>,

    private readonly jwtService: JwtService
  ){}

  async create(createUserDto: CreateUserDto) {

    try {

    const {password, ...userData} = createUserDto

    const user=  this.userRepository.create({
      ...userData,
      password: bcrypt.hashSync(password, 12)
      })

     await this.userRepository.save(user)

     delete user.password

       return {
      ...user,
      token: this.getJwt({email: user.email})
```

```
    }

    } catch (error) {
      this.handleDBErrors(error)
    }
  }

  async login(loginUserDto: LoginUserDto){

    const {email, password} = loginUserDto

    const user = await this.userRepository.findOne({
      where: {email},
      select: {email: true, password: true}
    })

    if(!user){
      throw new UnauthorizedException('Credenciales no válidas (email)')
    }

    if(!bcrypt.compareSync(password, user.password)){
      throw new UnauthorizedException('Password incorrect')
    }

       return {
      ...user,
      token: this.getJwt({email: user.email})
    }
  }

  //generar JWT
  private getJwt(payload: JwtPayloadInterface){
      const token = this.jwtService.sign(payload)
      return token
  }


  private handleDBErrors(error: any):void{
    if(error.code === '23505'){
      throw new BadRequestException(error.detail)
    }
    console.log(error)

    throw new InternalServerErrorException("Check logs")
  }
}
```

- Voy al login y coloco usuario y contraseña correctos, en consola me devuelve email, password y el token!
- Quiero guardar todo en minúsculas
- Lo hago en la entidad directamente con **@BeforeInsert**
- Como en el **@BeforeUpdate** es el mismo código llamo al método anterior

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

@Entity('users')
export class User{

    @PrimaryGeneratedColumn('uuid')
    id: string

    @Column('text',{
        unique: true
    })
    email: string


    @Column('text',{
        select: false
    })
    password: string

    @Column('text')
    fullName: string

    @Column('bool',{
        default: true
    })
    isActive: boolean

    @Column('text',{
        array: true,
        default:['user']
    })
    roles: string[]

    @BeforeInsert()
    checkFieldsBeforeInsert(){
        this.email = this.email.toLowerCase().trim()
    }

    @BeforeUpdate()
    checkFieldsBeforeUpdate(){
        this.checkFieldsBeforeInsert()
    }
}
```

Priovate Route - General

- Creo mi primera ruta privada que su único objetivo va a asegurar de que hay un jwt, que el usuario esté activo y el token no haya expirado (más adelante se evaluará tambien el rol)
- Voy a usar Get y la llamaré testingPrivateRoute

- auth.controller

```
  @Get('private')
  testingPrivateRoute(){
    return {
      ok: true
    }
  }
```

- Los **Guards** son usados para permitir o prevenir el acceso a una ruta
- **Es dónde se debe de autorizar una solicitud**
- Autenticación y autorización **no son lo mismo**
- Autenticado es cuando el usuario está validado y autorizado es que tiene permiso para acceder
- Para usar el **guard** uso el decorador **@UseGuards** de @nestjs/common (por el momento, se hará un guard personalizado)
- Uso AuthGuard de @nestjs/passport, que usa la estrategia que yo definí por defecto, la configuración que definí, etc
- Para probarlo en Postman/ThunderClient debo añadir el token proporcionado en el login en Auth donde dice Bearer
- Si le cambio el isActive a **FALSE** y le paso el token adecuado, me devuelve un error controlado diciendo que no estoy autorizado porque mi usuario está inactivo
- **Pero de dónde sale eso?**
- Recuerda que en la estrategia, en el validate hago la verificación
- Es la estrategia que está usando por defecto el **Guard**

```
import {PassportStrategy} from '@nestjs/passport'
import { ExtractJwt, Strategy } from 'passport-jwt';
import { User } from '../entities/user.entity';
import { JwtPayloadInterface } from '../interfaces/jwt-payload.interface';
import { Repository} from 'typeorm';
import { InjectRepository } from '@nestjs/typeorm';
import { ConfigService } from '@nestjs/config';
import {UnauthorizedException, Injectable} from '@nestjs/common'


@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy){


    constructor(

        @InjectRepository(User)
        private readonly userRepository: Repository<User>,

        private readonly configService: ConfigService
    ){
        super({
            secretOrKey: configService.get('JWT_SECRET'),
```

```
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken()
        })
    }

    async validate(payload: JwtPayloadInterface): Promise<User>{

        const {email} = payload

         const user = await this.userRepository.findOneBy({email})

         if(!user) throw new UnauthorizedException('Token not valid')

         if(!user.isActive) throw new UnauthorizedException('User is inactive')

        return user
    }
}
```

- Si cambio la secret_key de la variable de entorno, el mismo token va a dar un error de autenticación "Unauthorized"
- Esto esta bien que sea asi

Cambiar el email por el id en el payload

- El email puede cambiar, por lo que conviene usar el uuid
- En el payload del jwt en lugar del email debe ir el uuid. Va a haber que actualizar la estrategia
- Primero, cuando hago el user de retorno en el login debo pedir también el id
- Cambio el email por el id en la generación del token en el return
    - Me marca error porque la interfaz me pide el email. Cambio la interfaz

```
async login(loginUserDto: LoginUserDto){

    const {email, password} = loginUserDto

    const user = await this.userRepository.findOne({
      where: {email},
      select: {email: true, password: true, id: true}
    })

    if(!user){
      throw new UnauthorizedException('Credenciales no válidas (email)')
    }

    if(!bcrypt.compareSync(password, user.password)){
      throw new UnauthorizedException('Password incorrect')
    }

    return {
      ...user,
```

```
      token: this.getJwt({id: user.id})
    }
  }
```

- Hago lo mismo en el método create(en lugar del {mail: user.email},{id: user.id}
- Cambio la interfaz

```
export interface JwtPayloadInterface{

    id: string
}
```

- Falta cambiar la estrategia, ya que desestructuro el email y ya no lo tengo
- Cambio email por id

```
import {PassportStrategy} from '@nestjs/passport'
import { ExtractJwt, Strategy } from 'passport-jwt';
import { User } from '../entities/user.entity';
import { JwtPayloadInterface } from '../interfaces/jwt-payload.interface';
import { Repository} from 'typeorm';
import { InjectRepository } from '@nestjs/typeorm';
import { ConfigService } from '@nestjs/config';
import {UnauthorizedException, Injectable} from '@nestjs/common'


@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy){


    constructor(

        @InjectRepository(User)
        private readonly userRepository: Repository<User>,

        private readonly configService: ConfigService
    ){
        super({
            secretOrKey: configService.get('JWT_SECRET'),
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken()
        })
    }

    async validate(payload: JwtPayloadInterface): Promise<User>{

        const {id} = payload

         const user = await this.userRepository.findOneBy({id})
```

```
        if(!user) throw new UnauthorizedException('Token not valid')

        if(!user.isActive) throw new UnauthorizedException('User is inactive')

      return user
    }
}
```

- Vuelvo a generar un token, lo pruebo y debería ver la respuesta
- Cuando me autentique en una ruta siempre va a pasar por el JwtStrategy. Ahi ya tengo el usuario, puedo hacer un console.log
- Ahora, si cambia el correo no tengo problema
- Veamos cómo puedo obtener el usuario en los controladores y dónde necesite

---

Custom Property Decorator - GetUser

- Puedo extraer el usuario del **Guard**
- Si se me olvidara que tengo implementado el Guard y quisiera extraer el usuario, debería lanzar un error propio.
    - Es un problema que yo como desarrollador del backend debo resolver
- Hay varias formas. Puedo escribir nest -h para ver la ayuda y usar el CLI

> nest g d nombre_decorador

- Pero este decorador funciona de manera global, por clase y por controlador
- No funciona para propiedad
- **Para extraer el usuario** usaré **@Request** de @nestjs/common
- Si hago un console.log de la request me manda un montón de info en consola

```
  @Get('private')
  @UseGuards( AuthGuard())
  testingPrivateRoute(
    @Request() request: Express.Request
  ){
    return {
      ok: true
    }
  }
```

- **request.user** me devuelve el usuario
- Esto así funcionaría pero no es muy bonito
- Además necesito pasar por el Guard, por lo que habría que hacer un par de validaciones también
- Mejor creemos un **Custom Property Decorator**
- auth/decorators/get-user.decorator.ts
- El createParamDecorator es una función que usa un callback que debe retornar algo

```
import { createParamDecorator } from "@nestjs/common";

export const GetUser = createParamDecorator(
    ()=>{

        return 'Hola mundo'
    }
)
```

- En el controller

```
  @Get('private')
  @UseGuards( AuthGuard())
  testingPrivateRoute(
        @GetUser() user: User
  ){
    console.log({user})  //imprime en consola Hola mundo

    return {
      ok: true
    }
  }
```

- Lo que sea que retorne createParamDecorator es lo que voy a poder extraer
- En el callback de createParamDecorator dispongo de la data y el context (lo importo de @nestjs/common)

```
import { ExecutionContext, createParamDecorator } from "@nestjs/common";

export const GetUser = createParamDecorator(
    (data, ctx: ExecutionContext)=>{
        console.log({data})

    }
)
```

- La consola me devuelve data: undefined.
- Si voy al controlador y escribo 'email' en el decorador **@GetUser('email)** la consola me devuelve data: 'email'
- Puedo pasarle todos los argumentos que quiera en un arreglo

```
  @Get('private')
  @UseGuards( AuthGuard())
  testingPrivateRoute(
        @GetUser(['email', 'role', 'fullName']) user: User
  ){
```

```
    console.log({user})
    return {
      ok: true,
      user
    }
  }
```

- El **ExecutionContext** es el contexto en el que se está ejecutando la función en la app
- Tengo, entre otras cosas, **la Request** (tambien la Response)
- Uso **switchToHttp.getRequest** para extraer la Request. Usaría getResponse para la Response
- Lanzo un error 500 si no está el usuario porque es un error mío ya que debería haber pasado por el Guard

```
import { ExecutionContext, InternalServerErrorException, createParamDecorator }
from "@nestjs/common";

export const GetUser = createParamDecorator(
    (data, ctx: ExecutionContext)=>{

        const req = ctx.switchToHttp().getRequest()

        const user = req.user

        if(!user) throw new InternalServerErrorException('User not found')

        return user
    }
)
```

Tarea Custom Decorators

- Quiero usar el @GetUser dos veces en el mismo endpoint en el controller
- Una sin pasarle ningún argumento que me devuelva el User completo
- Otra pasándole solo el email como parámetro a @GetUser para que me devuelva el email
- Podría usar los Pipes para validar/transformar la data pertfectamente, pero no es el caso

```
  @Get('private')
  @UseGuards( AuthGuard())
  testingPrivateRoute(
        @GetUser() user: User,
        @GetUser('email') email: string
  ){
    console.log({user})
    return {
      ok: true,
      user
```

```
    }
  }
```

- Uso un ternario para devolver si no hay data el user, y si la hay user[propiedad_computada]
- get-user.decorator.ts

```
import { ExecutionContext, InternalServerErrorException, createParamDecorator }
from "@nestjs/common";

export const GetUser = createParamDecorator(
    (data, ctx: ExecutionContext)=>{

        const req = ctx.switchToHttp().getRequest()

        const user = req.user

        if(!user) throw new InternalServerErrorException('User not found')

        return (!data) ? user : user[data]
    }
)
```

- Si hago un console.log de la Request usando el decorador @Request y lo imprimo en consola, puedo crear un decorador que me devuelva lo que yo quiera de ella, por ejemplo los rawHeaders
- Aunque es un decorador que iría más bien en el módulo common, lo pondré junto al otro decorador por tenerlos agrupados
- get-rawheaders.decorator.ts

```
import { ExecutionContext, createParamDecorator } from "@nestjs/common";


export const GetRawHeaders = createParamDecorator(
    (data, ctx: ExecutionContext)=>{

        const req = ctx.switchToHttp().getRequest()


        return  req.rawHeaders
    }
)
```

- auth.controller

```
  @Get('private')
  @UseGuards( AuthGuard())
  testingPrivateRoute(
```

```
        @GetUser() user: User,
        @GetUser('email') email: string,
        @GetRawHeaders() rawHeaders: string[]
  ){
    return {
      ok: true,
      user,
      email,
      rawHeaders
    }
  }
```

- Nest ya tiene su propio decorador **@Headers** para los headers (de @nestjs/common)
- El tipo de headers es IncomingHttpHeaders (importar de http)

Custom Guard y Custom Decorator

- En este momento, si yo quisiera validar el rol podría hacerlo en el controlador con user.roles.includes('admin), por ejemplo
- Pero voy a crear un Guard y un Custom Decorator para esta tarea
- Creo otro Get en el auth.controller

```
@Get('private2')
@UseGuards(AuthGuard())
privateRoute2(
  @GetUser() user: User,
){
  return{
    ok: true,
    user
  }
}
```

- Este Get necesita tener ciertos roles, y quiero crear un decorador que los valide
- Puedo usar **@SetMetaData**

```
@Get('private2')
@UseGuards(AuthGuard())
@SetMetadata('roles', ['admin'])
privateRoute2(
  @GetUser() user: User,
){
  return{
    ok: true,
    user
  }
}
```

- Con esto no es suficiente, debo crear un Guard para que lo evalue
- Puedo hacerlo con el CLI usando gu

```
nest g gu auth/guards/userRole --no-spec
```

- Esto genera por mi

```
import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class UserRoleGuard implements CanActivate {

  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {

    console.log('UserGuard')

    return true;
  }
}
```

- Para que un Guard sea válid tiene que implementar canActivate
- Tiene que retornar un boolean o una Promesa que sea un boolean, si es true lo deja pasar si no no
- También puede devolver un Observable que emita un boolean
- Los Guards por defecto son async
- Coloco el userRoleGuard en el controlador

```
@Get('private2')
@UseGuards(AuthGuard(), UserRoleGuard)
@SetMetadata('roles', ['admin'])
privateRoute2(
  @GetUser() user: User,
){
  return{
    ok: true,
    user
  }
}
```

- Porqué no lleva paréntesis?
- Podría generar una nueva instancia con new
- **AuthGuard ya devuelve la instancia**, por lo que los Guards personalizados no llevan paréntesis, **para usar la misma instancia**
- Se puede hacer usando el new pero eso lo que haría es generar una nueva instancia, y lo que queremos es usar la misma

- Si ejecuto el endpoint private2 con el token en consola imprime el console.log, con lo que ha pasado por el Guard
- Los Guards se encuentran dentro del ciclo de vida de Nest
    - Están dentro de la **Exception Zone**
    - Significa que si devolviera un error en lugar del true va a ser controlado por Nest (BadRequestException o lo que fuera)
- Este Guard se va a encargar de verificar los roles.
- Para ello primero debo extraer la metadata del decorador **@SetMetadata**
- Aquí **no se pone fácil la cosa**. **Tirando de documentación**
- Inyecto Reflector en el constructor
- Lo uso para guardar en la variable roles con el .get('roles') (lo que pone en **@SetMetadata**) y el target es context.getHandler()

```
import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Observable } from 'rxjs';

@Injectable()
export class UserRoleGuard implements CanActivate {

  constructor(
    private readonly reflector: Reflector
  ){}


  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {

    const validRoles: string[] = this.reflector.get('roles', context.getHandler()
)

    console.log({validRoles}) //para testear que los haya extraído bien

    return true;
  }
}
```

- Ahora lo que debo hacer es comparar si existen en el arreglo de roles de mi entidad
- Si no existe niguno voy a devolver un error

Verificar Rol del usuario

- Para obtener el usuario es el mismo código de **ctx.switchToHttp().getrequest()**
- Tipo el usuario con **as User** así obtengo el completado también
- Verifico que venga el usuario para asegurarme de que se usa el Guard de autenticación
- Uso un ciclo for para recorrer el array y verificar el rol
- Si no es un role valido lanzaré un ForbiddenException

```
import { BadRequestException, CanActivate, ExecutionContext, ForbiddenException,
Injectable } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Observable } from 'rxjs';
import { User } from 'src/auth/entities/user.entity';

@Injectable()
export class UserRoleGuard implements CanActivate {

  constructor(
    private readonly reflector: Reflector
  ){}


  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {

    const validRoles: string[] = this.reflector.get('roles', context.getHandler()
)

    const req = context.switchToHttp().getRequest()
    const user = req.user as User

    if(!user) throw new BadRequestException('User not found')

    for(const role of user.roles){
      if(validRoles.includes(role)){
        return true
      }
    }
    throw new ForbiddenException(`User ${user.fullName} needs a valid role`)
  }
}
```

- Para que lo deje pasar añado en TablePlus el role de admin al usuario
- Para usar esta lógica que estoy implementando tengo que memorizar muchas cosas. Establecer el SetMetadata, etc
- Si me olvidara del SetMetadata, al extraer los validRoles mi app reventaría. Debería validarlo
- También es muy volatil el arreglo de roles, me puedo equivocar. El SetMetadata se usa muy poco como decorador directamente
- Mejor crear un **Custom Decorator**

Custom Decorator RoleProtected

- Si no son decoradores de propiedades, perfectamente puedo usar el CLI
- ¿El decorador que voy a crear esta fuertemente ligado al módulo auth o es algo general que podría ir en common?
    - Me va aservir para establecer los roles que el usuario ha de tener para poder ingresar a la ruta

    - Por lo que SI está amarrado al módulo de auth

> nest g d auth/decorators/roleProtected --no-spec

- Esto me genera este código

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

export const RoleProtected = (...args: string[]) => SetMetadata('role-protected',
args);
```

- Cambio 'role-protected' en el SetMetadata por 'roles'
- Defino el string con una variable para tenerla en un solo lugar, por si hubiera cambios
- Importo META_ROLES en el UserRoleGuard para añadirlo en el this.reflector.get

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

export const META_ROLES= 'roles'

export const RoleProtected = (...args: string[]) =>{

    SetMetadata(META_ROLES, args);
}
```

- Creo una enum en la carpeta de interfaces para especificar los roles que voy a permitir
- Tienen que ser strings, Typescript les asigna un número 0,1,2

```
export enum ValidRoles{

    admin= 'admin',
    superUser= 'super-user',
    user= 'user'
}
```

- Le paso el enum como tipo como parámetro del decoradorrole-protected

```
import { SetMetadata } from '@nestjs/common';
import { ValidRoles } from '../interfaces/valid-roles';

export const META_ROLES= 'roles'

export const RoleProtected = (...args: ValidRoles[]) =>{

   return  SetMetadata(META_ROLES, args);
}
```

- Uso el **@RoleProtected** en el controller
- Si lo pusiera sin parámetros, cualquier usuario tendría acceso a la ruta
- Uso el **enum**

```
@Get('private2')
@UseGuards(AuthGuard(), UserRoleGuard)
@RoleProtected(ValidRoles.admin)
privateRoute2(
  @GetUser() user: User,
){
  return{
    ok: true,
    user
  }
}
```

- Puedo pasarle varios valores separados por comas, **@RoleProtected(ValidRoles.admin, ValidRoles.user)**
- Es fácil que me olvide de implementar el AuthGuard (autenticación), o el RoleProtected(autorización)
- Podemos crear un único decorador que lo haga todo

Composición de decoradores

- Con **applyDecorators de @nestjs/common** podemos hacer composición de decoradores
- Muy útil para agrupar varios decoradores en uno
- Creo un tercer endpoint privateRoute3
    - Va a funcionar igual solo que en lugar de tener tantos decoradores tendrñe uno que haga todo el trabajo
- controller

```
@Get('private3')
@UseGuards(AuthGuard(), UserRoleGuard)
@RoleProtected(ValidRoles.admin)
privateRoute3(
  @GetUser() user: User,
){
  return{
    ok: true,
    user
  }
}
```

- Creo el auth.decorator.ts en /auth/decorators/
- En lugar de usar el SetMetadata puedo usar el RoleProtected
- AuthGuard de @nestjs/passport hay que ejecutarlo porque así funciona
- Le paso jwt

```
import { UseGuards, applyDecorators } from "@nestjs/common";
import { META_ROLES, RoleProtected } from "./role-protected.decorator";
import { AuthGuard } from "@nestjs/passport";
import { ValidRoles } from "../interfaces/valid-roles";
import { UserRoleGuard } from "../guards/user-role/user-role.guard";



export function Auth(...roles: ValidRoles[]){

    return applyDecorators(
        RoleProtected(...roles),
        UseGuards(AuthGuard('jwt'), UserRoleGuard)
    )
}
```

- Lo uso en el controller
- Si lo envio sin nada entre paréntesis debe querer decir que no necesita ningún role especial y pasar

```
  @Get('private3')
  @Auth()
  privateRoute3(
    @GetUser() user: User,
  ){
    return{
      ok: true,
      user
    }
  }
```

- NOTA: EN USERROLEGUARD FALTABAN DOS LINEAS DE CÓDIGO PARA QUE PUEDA PASAR SIN ROLES
- UserRoleGuard

```
import { BadRequestException, CanActivate, ExecutionContext, ForbiddenException,
Injectable } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Observable } from 'rxjs';
import { META_ROLES } from 'src/auth/decorators/role-protected.decorator';
import { User } from 'src/auth/entities/user.entity';

@Injectable()
export class UserRoleGuard implements CanActivate {

  constructor(
    private readonly reflector: Reflector
  ){}
```

```
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {

    const validRoles: string[] = this.reflector.get(META_ROLES,
context.getHandler() )

    //faltaba este código!!!
    if(!validRoles) return true  //<----------------
    if (validRoles.length === 0) return true //<-----------
    //

    const req = context.switchToHttp().getRequest()
    const user = req.user as User

    if(!user) throw new BadRequestException('User not found')

    for(const role of user.roles){
      if(validRoles.includes(role)){
        return true
      }
    }
    throw new ForbiddenException(`User ${user.fullName} needs a valid role`)
  }
}
```

- Debo estar autenticado con el token
- Si quiero que deba tener algún role en particular uso el ValidRoles

```
@Get('private3')
@Auth(ValidRoles.admin)
privateRoute3(
  @GetUser() user: User,
){
  return{
    ok: true,
    user
  }
}
```

- Si no pusiera el @Auth tendría un error porque necesitamos el usuario en la Request
- Para usarlo en otros endpoints de otros módulos, como el SEED por ejemplo, que solo debería hacerlo el admin, debo **importar el PassportModule** en el módulo dónde quiera utilizar el **@Auth**

Auth en otros módulos

- Quiero usar en mi **SeedController** el decorador **@Auth**
    - **@Auth** está usando **@AuthGuard** que está asociado a **Passport**, y Passport es un **módulo**

    - En el error en consola al intentar usar **@Auth** fuera del módulo lo que está pidiendo el el **"defaultStrategy"**
- En el modulo de **Auth** tengo exportado el **JwtStrategy** y el **PassportModule**

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { PassportModule } from '@nestjs/passport';
import { JwtModule } from '@nestjs/jwt';
import { ConfigModule, ConfigService } from '@nestjs/config';
import { JwtStrategy } from './strategies/jwt.strategy';

@Module({
  controllers: [AuthController],
  providers: [AuthService, JwtStrategy],
  imports: [
    ConfigModule,
    TypeOrmModule.forFeature([User]),
    PassportModule.register({defaultStrategy: 'jwt'}),

    JwtModule.registerAsync({

        imports: [ConfigModule],
        inject: [ConfigService],
        useFactory: (configService: ConfigService)=>{

          return {
            secret: configService.get('JWT_SECRET'),
            signOptions:{
              expiresIn: '2h'
            }
          }
        }
    })
  ],
  exports: [TypeOrmModule, JwtStrategy, PassportModule, JwtModule]
})
export class AuthModule {}
```

- Es lo que necesito para exponer todo lo que está relacionado a Passport fuera de este módulo
- Importo AuthModule en el módulo de SEED. Es todo

```
import { Module } from '@nestjs/common';
import { SeedService } from './seed.service';
import { SeedController } from './seed.controller';
import { ProductsModule } from 'src/products/products.module';
import { AuthModule } from 'src/auth/auth.module';
```

```
@Module({
  controllers: [SeedController],
  providers: [SeedService],
  imports:[ProductsModule, AuthModule]
})
export class SeedModule {}
```

- Ahora puedo usar el decorador **@Auth** en el SEED controller

```
import { Controller,Get} from '@nestjs/common';
import { SeedService } from './seed.service';
import { Auth } from 'src/auth/decorators/auth.decorators';
import { ValidRoles } from 'src/auth/interfaces/valid-roles';


@Controller('seed')
export class SeedController {
  constructor(private readonly seedService: SeedService) {}

  @Get()
  @Auth(ValidRoles.admin)
  executeSeed() {
    return this.seedService.runSeed();
  }
}
```

- Si quiero proteger las rutas de productos solo tengo que repetir el procedimiento
- Si lo que quiero es que para cualquiera de las rutas el usuario **deba estar autenticado** coloco @Auth en el Controlador padre (sin ningún rol como parámetro)
- El usuario deberá tener el token de autorización (independientemente del rol)
- Falta crear en el Seed una forma de crear usuarios admin

Usuario que creó el producto

- Sería útil saber qué usuario creó el producto. Tenemos una autenticación en marcha que me lo puede decir
- Cómo se relaciona un usuario con un producto. Un usuario puede crear muchos productos
- Es una relación de uno a muchos **OneToMany**
- En productos, muchos productos pueden ser de un usuario, por lo que es una relación de muchos a uno **ManyToOne**
- En user.entity (en el módulo auth)
- El OneToMany no va a hacer que cree ningún valor nuevo en la columna, pero en Product si. Importo Product para tipar el valor
- Lo primero que debo añadir es **la relación con la otra entidad**
- Luego, como la entidad se relaciona con esta tabla, seria **product.user**, pero este user no existe todavía

```
@OneToMany(
    ()=>Product,
    (product)=> product.user //<--------este .user no existe todavía
)
product: Product
```

- En product.entity

```
@ManyToOne(
    ()=>User,
    (user)=>user.product
)
user: User
```

- Ahora en los productos, en TablePlus, hay una nueva columna que es userId. TypeOrm lo hizo por nosotros
- Lo normal es que cuando haga una consulta sobre el producto vaya a querer también el usuario que creó el producto
- Para que lo muestre en la consulta debo añadir eñl eager en true, para que cargue automáticamente esta relación

```
@ManyToOne(
    ()=>User,
    (user)=>user.product,
    {eager: true}
)
user: User
```

- Por ahora la columna de usuarios en producto solo tiene valores NULL porque en el SEED no había usuarios asignados a productos
- Esto es un error que debemos resolver.
- No debería permitir la creación de productos con el campo de usuario en NULL
- Borro toda la tabla de productos en TablePlus
- Ahora falta que al crear un producto, especifique que usuario lo creó a través de la autenticación

---

Insertar userId en los productos

- En el módulo de products **debo importar el AuthModule** para usar la autenticación con **@Auth** en el controller
- Solo los admin van a poder crear productos
- Uso el decorador **@GetUser** para extraer el usuario
- Se lo paso al servicio
- products.controller

```
@Post()
@Auth(ValidRoles.admin)
create(
  @Body() createProductDto: CreateProductDto,
  @GetUser() user: User
  ) {
        return this.productsService.create(createProductDto, user);
  }

  @Patch(':id')
  update(
    @Param('id', ParseUUIDPipe) id: string,
    @Body() updateProductDto: UpdateProductDto,
    @GetUser()user: User) {
    return this.productsService.update(id, updateProductDto, user);
  }
```

- Voy al servicio
- Actualizo create y update. Le paso el user a product, y antes de salvar en el update guardo el user en product.user

```
 async create(createProductDto: CreateProductDto, user:User) {
    try {

      const {images = [], ...productDetails} = createProductDto

      const product = this.productRepository.create({
        ...productDetails,
        images: images.map(image=> this.productImageRepository.create({url:
image})),
        user
      })

      await this.productRepository.save(product)

      return {...product, images}

    } catch (error) {

      this.handleDBExceptions(error)
    }

  }

   async update(id: string, updateProductDto: UpdateProductDto, user: User) {

    const {images, ...toUpdate} = updateProductDto


    const product = await this.productRepository.preload({id, ...toUpdate})
```

```
    if(!product) throw new NotFoundException(`Product with id : ${id} not found`)

    const queryRunner = this.dataSource.createQueryRunner()
    await queryRunner.connect()
    await queryRunner.startTransaction()

    try {

      if(images){
        await queryRunner.manager.delete(ProductImage, {product: {id}}) //con esto
borramos las imágenes anteriores

       product.images= images.map(image=> this.productImageRepository.create({url:
image}))

      }else{
          //product.images = await this.productImageRepository.findBy({product:
{id}}) puedo hacerlo así pero usaré findOnePlain
      }

      product.user= user
      await queryRunner.manager.save(product)

      await queryRunner.commitTransaction()  //commit
      await queryRunner.release() //desconexión

      return this.findOnePlane( id )

    } catch (error) {
      await queryRunner.rollbackTransaction()
      await queryRunner.release()

      this.handleDBExceptions(error)
    }
  }
```

- Tengo un error en el SEED porque llamo al productService.create y no le estoy pasando el user
- Muteo la linea de código donde está el error para poder compilar y crear un producto

```
//insertPromises.push(this.productsService.create(product))
```

NOTA: Si al crear el producto aparece un error que dice Cannot read properties of undefined(reading 'challenge') es porque en la composición del decorador @Auth, en su decorador @AuthGuard le falta pasarle 'jwt'

- Como tengo eager en true en la respuesta me carga directamente el usuario. Si no tendría que hacerlo manualmente
- Falta hacer funcional el SEED ya que le falta el usuario

SEED de usuarios, productos e imágenes

- Voy a crear un método para purgar las tablas de manera manual en el orden respectivo
    - Si intento borrar primero los usuarios, estos están siendo utilizados por los productos, la integridad referencial me va a molestar
- Borro todos los productos con el servicio de productos
- Para los usuarios debo inyectar el repositorio de usuarios
    - Uso el decorador **@InjectRepository**
    - Importo User y Repository
- Estoy exportando TypeORM y en TypeORM ya venía el usuario, por eso no da error
- Con el queryBuilder hago el delete, al no poner el nada en el where es todo, lo ejecuto
    - Recuerda que al tener el cascade en true va a borrar las imágenes también
- LLamo el método que he creado deleteTables en el runSEED
- Antes de insertar productos debo insertar usuarios
- Creo la interfaz en seed-data.ts
- Añado users al SeedData
- Añado los users a initialData

```
export interface SeedUser{
    email: string
    fullName: string
    password: string
    roles: string[]
}

export interface SeedData {
    users: SeedUser[]
    products: SeedProduct[]
}


export const initialData: SeedData = {

    users:[
        {
            email: 'test1@google.com',
            fullName: 'Test One',
            password: 'Abc123',
            roles: ['admin']
        },
        {
            email: 'test2@google.com',
            fullName: 'Test Two',
            password: 'Abc123',
            roles: ['user', 'super']
        }

    ],

    products: [ (etc...etc)
```

- Ahora puedo usar los usuarios para insertarlos masivamente desde el servicio
- Este firstUser que me retorna se lo paso a insertNewProducts (se lo paso al método para que no de error) y se lo paso al forEach, para que lo inserte en cada producto, por eso necesitaba retornar el user[0]
- seed.service

```
import { Injectable } from '@nestjs/common';
import { ProductsService } from 'src/products/products.service';
import { initialData } from './data/seed-data';
import { InjectRepository } from '@nestjs/typeorm';
import { User } from 'src/auth/entities/user.entity';
import { Repository } from 'typeorm';


@Injectable()
export class SeedService {

  constructor(
    private readonly productsService: ProductsService,

    @InjectRepository(User)
    private readonly userRepository: Repository<User>
  ){}

  private async insertUsers(){

    const seedUsers= initialData.users
    const users: User[] = []

    seedUsers.forEach(user=>{
      users.push(this.userRepository.create(user)) //esto no salva el usuario en
la db
    })

    const dbUsers = await this.userRepository.save(seedUsers)

    return dbUsers[0] //retorno el primer usuario para que le pueda mandar
insertUsers a insertProducts

  }

  async runSeed() {

    await this.deleteTables()
    const firstUser = await this.insertUsers()

    this.insertNewProducts(firstUser)

    const products = initialData.products


  }
```

```
  private async deleteTables(){

    await this.productsService.deleteAllProducts()

    const queryBuilder = this.userRepository.createQueryBuilder()

    await queryBuilder
    .delete()
    .where({})
    .execute()



  }

  private async insertNewProducts(user: User){
   await  this.productsService.deleteAllProducts()

   const products = initialData.products
   const insertPromises = []


   products.forEach(product=>{
     insertPromises.push(this.productsService.create(product, user))
   })

   await Promise.all(insertPromises)

   return `SEED EXECUTED`;
  }
}
```

- Creo los usuarios, regreso un usuario, ese usuario es el que utilizo para insertar los productos
- El password no está encriptado, por lo que necesita encriptación si quiero que haga match!!
- Para ello no hay más que usar bcrypt en la data en seed-data.ts

```
import * as bcrypt from 'bcrypt'

export const initialData: SeedData = {

    users:[
        {
            email: 'test1@google.com',
            fullName: 'Test One',
            password: bcrypt.hashSync('Abc123',10),
            roles: ['admin']
        },
        {
            email: 'test2@google.com',
```

```
            fullName: 'Test Two',
            password: bcrypt.hashSync('Abc123',10) ,
            roles: ['user', 'super']
        }

    ],

    (etc...etc)
```

Check AuthStatus

- Falta poder revalidar el token. No es revalidar exactamente
- Es usar el token suministrado y generar un nuevo token basado en el anterior
- Si no hago esto, si el usuario refresca el navegador no va a estar autenticado
- Creo un nuevo endpoint en auth.controller (con su respectivo servicio)
- Un Get que llamaré checkAuthStatus

```
@Get('check-auth')
@Auth()
checkAuthStatus(
  @GetUser() user: User,
){
  return this.authService.checkAuthStatus(user)
}
```

- Esparzo el user, genero un nuevo JWT con el id que es el user.id
- auth.service

```
async checkAuthStatus(user: User){
  return {
    ...user,
    token: this.getJwt({id: user.id})
  }
}
```

- En la respuesta regreso un nuevo JWT y la info de name, fullName, email, etc por si le sirve al frontend
- El usuario tiene que estar activo