NEST 14

Entidad de usuarios

- Necesito la tabla de usuarios: quien creó el producto, que id tiene el usuario, etc
- Creo el resource de auth

> nest g res auth --no-spec

- El objetivo de las entidades es tener una relación entre las tablas de la DB y la aplicación
- Una entidad y una tabla es una relación de 1:1, 1 entidad = 1 tabla
- Creo en la entity de auth User

```
import { IsArray, IsBoolean, IsString } from "class-validator";
import { Column, Entity, PrimaryGeneratedColumn } 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 decirle a NEST que está esta entidad y crear la tabla debo ir al auth.module.
- Es un módulo, por lo que va en los imports en el TypeOrmModule.forFeature(User)

- Para que User pueda usarse fuera del módulo hay que exportar el TypeOrmModule

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

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

Crear Usuario

- Mi endpoint para la petición Post será:

> localhost:3000/api/auth/register

- Voy al controlador, que es quien escucha las request y emite una respuesta
- Borro todo y dejo solo el Post
- Borro los Dtos y creo create-user.dto.ts
- Para el password copio y pego una expresión regular para que se cree una contraseña robusta

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

export class CreateUserDto{

    @IsString()
    @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;


    @IsString()
    @MinLength(1)
```

```
    fullName: string
}
```

- controller:

```
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);
  }

}
```

- service:

```
import { Injectable } from '@nestjs/common';
import { CreateUserDto } from './dto/create-user.dto';


@Injectable()
export class AuthService {

  create(createUserDto: CreateUserDto) {
    return 'This action adds a new auth';
  }

}
```

- Ahora si hago una petición Post saltan todos los warnings de debe de ser un string, etc
- Hago la petición Post llenando el body en ThunderClient en formato JSON con email, password y fullName
- Todavía no estoy guardando nada, pero funciona
- Para crear el usuario lo meto en un try y un catch
- Para usar el modelo debo inyectarlo como un repositorio
- En el constructor, le añado el InjectRepository, importo User, importo el Repository de ORM y lo pongo de tipo User
- Ahora ya puedo usarlo para crear el usuario con el dto y guardar en la DB

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


@Injectable()
export class AuthService {

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

  async create(createUserDto: CreateUserDto) {

    try {
        const user= this.userRepository.create( createUserDto ) //esto no es la
inserción

        await this.userRepository.save(user)

        return user

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

}
```

- Obviamente falta hashear la contraseña
- Si le vuelvo a dar SEND en ThunderClient da error de usuario duplicado, hay que manejar esa excepción
- Me creo un metodo en el .service
- En el error de consola obtengo el code, me sirve para la validación (ojo que el número de código es un string)

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


@Injectable()
export class AuthService {

  constructor(
```

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

  async create(createUserDto: CreateUserDto) {

    try {
        const user= this.userRepository.create( createUserDto ) //esto no es la
inserción

        await this.userRepository.save(user)

        return user

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

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

      console.log(error)
      throw new InternalServerErrorException("Please check server logs")
  }

}
```

Encriptar la contraseña

- La encriptación será de una sola vía, con lo cual la contraseña no podrá recuperarse, habrá que resetear password

> npm i bcrypt @types/bcript

- Importo todo de bcrypt

> import * as bcrypt from 'bcrypt'

- En .services, primero necesito la contraseña que el usuario está mandando, la desestructuro del dto
- Le mando 10 rondas de encriptación

```
 async create(createUserDto: CreateUserDto) {

    try {

      const {password, ...userData} = createUserDto

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

```
            password: bcrypt.hashSync(password, 10)
          }) //esto no es la inserción

          await this.userRepository.save(user)

          return user

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

- Estoy regresando el password, el isActive. En lugar de retornar el usuario lo que voy a querer es retornar el JWT de acceso

Login

- Creo otra petición Post en el controller
- Creo el LoginUserDto, copio todo lo de create-user.dto y lo pego, no necesito el fullName
- No lo extiendo de create-user porque haría que todos los campos fueran opcionales y no me sirve
- login-user.dto:

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

export class LoginUserDto{

    @IsString()
    @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;
}
```

- .controller:

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

- Desestructuro la info del loginUserDto
- Hago una búsqueda por email del usuario
- .service:

```
async login(loginUserDto: LoginUserDto){
  const {password, email} = loginUserDto
  const user= this.userRepository.findOneBy({email})

  return user
 }
```

- Yo no quiero que al retornar el usuario me retorne el password.
- Para evitar esto voy a la entidad de User y en el campo password pongo select: false

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

- Si coloco el cursor encima me dice que tiene que ser seleccionado por el QueryBuilder y find operations
- Entonces, tengo que hacer un poco de carpintería para extraer la información
- Uso el findOne y selecciono del email, los campos que me interesan que son el email y el password

```
async login(loginUserDto: LoginUserDto){
  const {password, email} = loginUserDto
  const user= await this.userRepository.findOne({
   where: {email},
   select: {email: true, password: true}
  })

  return user
 }
```

- Ahora si hago una petición Post al endpoint y en el body un JSON con el mail y el password, me responde el mail con el password hasheado

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

- Evalúo si existe el usuario y mando un unauthorized exception diciendo que el mail esta mal
- Ahora necesito hacer el match del password. Para ello usaré bcrypt

```
async login(loginUserDto: LoginUserDto){
  const {password, email} = loginUserDto
  const user= await this.userRepository.findOne({
```

```
      where: {email},
      select: {email: true, password: true}
    })

    if(!user) throw new UnauthorizedException('Credentials are not valid (email) ')

    if(!bcrypt.compareSync(password, user.password))
     throw new UnauthorizedException('Credentials are not valid (password)')

    return user
   }
```

NEST Authentication- Passport

- Hay que hacer un par de instalaciones para trabajar con JWT (JSON WEB TOKEN)

> npm i --save @nestjs/passport passport @nestjs/jwt passport-jwt

> npm i --save-dev @types/passport-jwt

- Hay que decirle al módulo de auth cómo quiero que evalúe el método de autenticación
- En auth.module, debajo del ORM ocupo el PassportModule y le digo que tipo de estrategia voy a seguir
- El registerAsync es para asegurarme que las variables de entorno estén previamente configuradas o si mi módulo depende de algún servicio externo
- Usaré el register con la defaultStrategy: jsonwebtoken
- Ahora hay que configurar ese modulo de JWT
- Hay que definirle la palabra secreta para firmar y revisar los tokens, la expiración de este
- Creo la variable de entorno 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/auth.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 {}
```

- Para asegurarme de que la variable de entorno esté configurada usaré un módulo asíncrono

Módulo Asíncrono

- useFactory es la función que yo voy a llamar cuando se intente registrar de manera asíncrona en el módulo
    - Le paso una función de flecha y le retorno el objeto anterior.
    - Pongo un console.log para comprobar que la variable de entrono está cargada

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/auth.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: ()=>{
        console.log(process.env.JWT_SECRET)
        return{
          secret: process.env.JWT_SECRET,
          signOptions: {
            expiresIn: "2h"
          }
        }
      }
    })
  ],
  exports:[TypeOrmModule]
})
export class AuthModule {}
```

- Así funciona, pero puedo usar el configModule y el configService para la variable de entorno

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/auth.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)=>{
      console.log(configService.get('JWT_SECRET'))

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

- Falta todavía saber qué información grabo en el JWT, cómo validarlo, y a qué usuario le corresponde ese JWT

JWTStrategy

- Con el payload del JWT, lo que quiero hacer es saber que usuario es y tener la info a través del email, si el user esta activo y que roles tiene
- Si no está activo no va a poder autenticarse
- Creo una carpeta dentro de /auth llamada strategies, y dentro un archivo que se llama jwt.strategy.ts
- Necesito dos importaciones:

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

export class JwtStrategy extends PassportStrategy(Strategy ){

}
```

- Lo que quiero es implementar una forma de expandir la validación del jwt
- El PassportStrategy va a revisar el jwt basado en la palabra secreta y también si ha expirado o no, y la strategy si el token es válido o no
- Si el usuario está activo y todo lo demás lo puedo verificar mediante un método, lo llamaré validate
- Va a recibir el payload del JWT de tipo any, y va a ser una promesa que va a regresarme una instancia de mi User
- Si pasa con la palabra secreta y no ha a expirado ya puedo recibir el payload
- Creo una carpeta en auth llamada interfaces con un archivo llamado jwt-payload.interfaces.ts
    - Es una interface para cómo quiero que luzca ese payload

```
export class JwtStrategy extends PassportStrategy(Strategy ){

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

        return;
    }
}
```

- interface:

```
export interface JwtPayload{
    email: string

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

- Extraigo el email del payload
- Se busca de que el payload del JWT sea un objeto porque así es más fácil añadir porpiedades o quitarlas
- Hay que consultar si este correo existe y analizar como luce el usuario en la DB (si está activo) y regresar la info

```
import { PassportStrategy } from "@nestjs/passport"
import { Strategy } from "passport-jwt";
import { User } from "../entities/auth.entity";
import { JwtPayload } from "../interfaces/jwt-payload.interface";
```

```
export class JwtStrategy extends PassportStrategy(Strategy ){

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

        const {email}= payload

        return;
    }
}
```

- Para hacer uso de mi entidad voy a inyectar en el constructor mi modelo
- Cuando mando llamar al constructor, el PassportStrategy necesita mandar a llamar el constructor padre (super)
- Necesito pasarle la variable de entorno con configService
- Para ello primero debo importarlo en el módulo de auth (no solo en el useFactory del jwt)
- 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/auth.entity';
import {PassportModule} from '@nestjs/passport';
import {JwtModule} from '@nestjs/jwt'
import { ConfigModule, ConfigService } from '@nestjs/config';



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

```
})
export class AuthModule {}
```

- jwt.strategy:

```
import { ConfigService } from "@nestjs/config";
import { PassportStrategy } from "@nestjs/passport"
import { InjectRepository } from "@nestjs/typeorm";
import { Strategy } from "passport-jwt";
import { Repository } from "typeorm";
import { User } from "../entities/auth.entity";
import { JwtPayload } from "../interfaces/jwt-payload.interface";

export class JwtStrategy extends PassportStrategy(Strategy ){

    constructor(

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

        ){
            super({
                secretOrKey: configService.get('JWT_SECRET')

            })
        }

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

        const {email}= payload



        return;
    }
}
```

- Necesito definir cuando mande una petición http **dónde espero el token**, en los headers? Es muy tradicional mandarlo como un header de autenticación, Bearer Token
- Lo especifico debajo de secretOrKey en el super

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

- Ahora si, voy buscar el usuario por el email

- No necesito validar el password, de todos modos no lo tengo
- Si el token existe es que se autenticó el usuario en su momento, porque solo con el usuario y password voy a generar ese token
- Si el usuario no existe lanzo una excepción
- jwt-strategy:

```
    async validate(payload: JwtPayload): 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 todo va bien retorno el usuario. Que gano retornando el usuario? se va a añadir en la Request
- En la Request yo voy a tener acceso en todo el camino desde que entra, pasa por los interceptores, servicios, donde yo tenga acceso a la request, voy a tener acceso al usuario
- Después se usarán decoradores personalizados para extraer info de la Request y hacer lo mismo que se hace en los controladores
- Crear un decorador para extraer ese usuario
- Ahora, esta clase JwtStrategy que he creado es una clase flotando, todavía no hay ninguna relación con ningún módulo
- Las estrategias son providers
- **Le añado el decorador de @Injectable** en jwt.strategy.ts porque la clase es un provider
- Voy a auth.module y **en providers añado el JwtStrategy**. También **lo voy a exportar** junto a TypeOrmModule
    - Por si acaso quiero validar el token manualmente, **exporto también PassportModule, JwtModule**
- Ya está listo elmódulo de autenticación. Ahora falta ver cómo se usa para proteger las rutas
- Voy a usar la estrategia jwt para bloquear y validar una ruta

Generar un JWT

- Cuando un usuario se crea, automáticamente lo voy a autenticar (generar JWT que identifica a este usuario), con lo cual en el servicio es un buen sitio para crear un JWT
- También cuando alguien hace el login yo quiero regresar el JWT (que vive por dos horas). Una vez expire tengo que pensar en generar un nuevo JWT antes de que el otro expire
- Cómo lo voy a usar en diferentes sitios, me creo un método para hacerlo en .service llamado getJwt
- Voy a necesitar el payload que será de tipo JwtPayload (asegurarse de importarlo de interfaces)
- Uso el servicio de @nest/jwt que es proporcionado por el JwtModule
- este JwtModule ya le va a decir a este servicio la fecha de expiración, la palabra secreta y todo

- Firmo el JWT
- auth.service:

```
import { BadRequestException, Injectable, InternalServerErrorException,
UnauthorizedException } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { CreateUserDto } from './dto/create-user.dto';
import { User } from './entities/auth.entity';
import * as bcrypt from 'bcrypt'
import { LoginUserDto } from './dto/login-user.dto';
import { JwtPayload } 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, 10)
        }) //esto no es la inserción

        await this.userRepository.save(user)

        return user

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

 async login(loginUserDto: LoginUserDto){
   const {password, email} = loginUserDto
   const user= await this.userRepository.findOne({
    where: {email},
    select: {email: true, password: true}
   })
```

```
    if(!user) throw new UnauthorizedException('Credentials are not valid (email) ')

    if(!bcrypt.compareSync(password, user.password))
     throw new UnauthorizedException('Credentials are not valid (password)')

    return user
   }

   private getJwt(payload: JwtPayload){
     const token = this.jwtService.sign( payload )
     return token
   }

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

       console.log(error)
       throw new InternalServerErrorException("Please check server logs")
   }

}
```

- En el login, esparzo el usuario con el rest, y genero el token. De payload le paso el objeto del user.email que responde a la interfaz

```
async login(loginUserDto: LoginUserDto){
   const {password, email} = loginUserDto
   const user= await this.userRepository.findOne({
    where: {email},
    select: {email: true, password: true}
   })

   if(!user) throw new UnauthorizedException('Credentials are not valid (email) ')

   if(!bcrypt.compareSync(password, user.password))
    throw new UnauthorizedException('Credentials are not valid (password)')

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

- Repito lo mismo en el método create
- Ya puedo enviar las peticiones Post desde POSTMAN o ThunderClient con los datos en el body y me devuelve el Token con los datos
- Resolvamos el asunto de la capitalización.
- Voy a la entity y uso el BeforeInsert

```
import { IsArray, IsBoolean, IsString } from "class-validator";
import { BeforeInsert, BeforeUpdate, Column, Entity, PrimaryGeneratedColumn } 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()
    }
}
```

Private Route - General

- Voy a crearme mi primera ruta privada en el auth.controller, cuyo único objetivo es asegurarse de que el JWt esté presente en los Headers como un Bearer Token.
- Que pueda encontrar un usuario, que el usuario esté activo y que el token no haya expirado
- Para crear una ruta privada están los **Guards**.

- Autenticación es saber que el usuario tiene lo necesario para estar autenticado
- Autorización es que tiene los permisos y acceso al recurso
- Cada uno de estos Guards debe de tener el CanActivate
- AuthGuard (de nestjs/passport) ya tiene toda la info de la estrategia de jwt

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


@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}

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

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

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

}
```

- Coloco el Bearer Token en Thunder Client (en Auth-Bearer) que obtengo del Login y voy al Get del endpoint de private

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

Cambiar el email por id en el payload

- En el Login pido también el id con el select

```
 async login(loginUserDto: LoginUserDto){
   const {password, email} = loginUserDto
   const user= await this.userRepository.findOne({
```

```
      where: {email},
      select: {email: true, password: true, id: true}
     })

     if(!user) throw new UnauthorizedException('Credentials are not valid (email) ')

     if(!bcrypt.compareSync(password, user.password))
      throw new UnauthorizedException('Credentials are not valid (password)')

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

- Cambio la interface de email a id

```
export interface JwtPayload{
    id: string
}
```

- Corrijo los errores, donde pone email: user.email pongo id: user.id en el getJwt

```
import { BadRequestException, Injectable, InternalServerErrorException,
UnauthorizedException } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { CreateUserDto } from './dto/create-user.dto';
import { User } from './entities/auth.entity';
import * as bcrypt from 'bcrypt'
import { LoginUserDto } from './dto/login-user.dto';
import { JwtPayload } 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, 10)
        }) //esto no es la inserción

        await this.userRepository.save(user)

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

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

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

   if(!user) throw new UnauthorizedException('Credentials are not valid (email) ')

   if(!bcrypt.compareSync(password, user.password))
    throw new UnauthorizedException('Credentials are not valid (password)')

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

  private getJwt(payload: JwtPayload){
    const token = this.jwtService.sign( payload )
    return token
  }

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

      console.log(error)
      throw new InternalServerErrorException("Please check server logs")
  }

}
```

- En el jwt.strategy cambio el email por el id

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

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

    constructor(

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

        ){
            super({
                secretOrKey: configService.get('JWT_SECRET'),
                jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken()

            })
        }

    async validate(payload: JwtPayload): 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;
    }
}
```

- Cómo obtener el usuario en los controladores? en la proxima lección!

Custom Property- Decorator GetUser

- Voy a obtener el usuario del as rutas que está autenticado. Si se me olvidara que tengo implementado el Guard de la autenticación y trato de sacar al usuario me tendría que devolver un error.
- El cli no crea decoradores por propiedad, los hace globales, por clase y por controlador
- Tomo la Request del testingPrivateRoute en el controlador

```
  @Get('private')
  @UseGuards(AuthGuard())
  testingPrivateRoute(
    @Req() request: Express.Request
  ){
    console.log(request)

    return {
      ok: true
    }
  }
```

- Gracias al console.log puedo ver que en la extensa respuesta del request en consola está el usuario
- Puedo hacer un console.log({request: request.user})
- Si no paso por el Guard (lo muteo para hacer la prueba) **NO TENGO EL USUARIO** por lo que voy a tener que hacer unas validaciones
- Para ello voy a crear un custom decorator, no quiero el usuario a través del request
- Creo una carpeta en auth/ decorators/ get-user.decorator.ts
- Los decoradores son funciones

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

export const GetUser= createParamDecorator()
```

- El decorador espera como argumento un callback

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

export const GetUser= createParamDecorator(
    ()=>{
        return "Hola mundo"

})
```

- Si yo ahora voy al auth.controller y uso el @GetUser como decorador, y voy a decorar una propiedad user de tipo User (entity) (aunque yo se que está regresando un string ahora para la prueba), si hago un console.log del user me devuelve el "Hola mundo"

```
  @Get('private')
  @UseGuards(AuthGuard())
  testingPrivateRoute(
    @GetUser() user: User
  ){
    console.log({user})
  }
```

- Lo que sea que retorne el decorator es lo que va a regresar como propiedad
- Una de las ventajas de este createParamDecorator es que cuando NEST lo llame voy a tener la data y el contexto de tipo ExecutionContext
- Si hago un console.log de la data me da undefined

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

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

        return "Hola mundo"
})
```

- Si en el controlador le paso de argumento 'email' al @GetUser() (GetUser('email')) me devuelve 'email' en la data (cuando llamo al endpoint 'private')
- Si quiero mandar varios argumentos en el @GetUser lo hago como un array

```
@GetUser(['email', 'password'])
```

- El ExecutionContext es el contexto en el que se está ejecutando la función en este momento. Tiene acceso a la request
- La extraigo y extraigo el user.
- Si el user no se encuentra va a ser un fallo mio. Si solicito un usuario y no estoy dentro de una ruta autenticada es fallo mío
- Si todo sale bien voy a regresa al usuario

```
import { createParamDecorator, ExecutionContext, InternalServerErrorException }
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
(request)')

        return user
})
```

- en el controller

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

- Esto me regresa un usuario cuando en ThunderClient hago una petición Get al endpoint con un BearerToken válido

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

Tarea Custom Decorator

- Del getUser sólo quiero el email
- Si le mando el GetUser sin argumento me interesa todo el usuario
- Si le mando 'email' como argumento me interesa el email

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

- Hago un ternario y le paso la data computada
- get-user.decorator

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

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

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

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

```
(request)')

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

- Vuelvo a poner la Request en el controller, para imprimir en consola la request porque quiero hacer un custom decorator de los rawHeaders
- controller:

```
  @Get('private')
  @UseGuards(AuthGuard())
  testingPrivateRoute(
    @Req() request: Express.Request,
    @GetUser() user: User,
    @GetUser('email') userEmail: string
  ){

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

- Creo el archivo raw-headers.decorator.ts

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

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

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

        const rawHeaders= req.rawHeaders

        return rawHeaders
    }
)
```

- Lo añado al controller

```
  @Get('private')
  @UseGuards(AuthGuard())
  testingPrivateRoute(
    @Req() request: Express.Request,
    @GetUser() user: User,
```

```
    @RawHeaders() rawHeaders: string,
    @GetUser('email') userEmail: string,
  ){

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

- Hay un decorador que ya hacer esto
- Importar IncomingHttpHeaders de 'http'

> @Headers() headers: IncomingHttpHeaders

Custom Guard y Custom Decorator

- Podría hacer la validación en el controlador

> user.roles.includes('admin')....

- Pero voy a crear un Guard y un custom decorator que implemente esa lógica
- Creo otro Get en el controlador
- Recuerda: para tener el user disponible uso el UseGuards(AuthGuard())

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

- La idea es crear un decorador que sirva para que este Get necesite tener ciertos roles. Un decorador que me ayude a validar si tengo esos roles
- Puedo usar @SetMetadata(), sirve para añadir información extra al controlador

```
  @Get('private2')
  @SetMetadata('roles', ['admin', 'super-user'])
  @UseGuards(AuthGuard())
  privateRoute2(
    @GetUser() user: User
```

```
  ){
    return{
      ok: true,
      user
    }
  }
```

- Ahora necesito un Guard para evaluar esta data. Voy a crear un custom Guard con el CLI en la carpeta auth

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

- Me devuelve esto (le coloco un console.log)

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

@Injectable()
export class UerRoleGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {

    console.log('useRoleGuard')
    return true;
  }
}
```

- Para que un Guard sea válido tiene que implementar el canActivate que tiene que regresar un valor booleano. Si es true lo deja pasar
- Lo uso en el useGuards del controller

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

- Porqué no lleva paréntesis?
- Podría hacer una nueva instancia con new UserRoleGuard() y poner los paréntesis

- Pero el AuthGuard de 'passport' ya crea una nueva instancia, por lo que los custom guards que le siguen van sin paréntesis
- Ahora si hago un Get al endpoint 'private2' me imprime en consola el 'useRoleGuard'
- Los Guards entran dentro del ciclo de vida de NEST. Al estar dentro del exception zone, yo puedo lanzar un BadRequest al comprobar si tiene o no tiene los roles
- Antes de evaluar al usuario, necesito obtener la metadata
- Para obtenerla hay que hacer una inyección
- La idea del reflector es que me ayuda a ver información de los decoradores y otra info de la metadata
- Me pide la metadata y el target

```
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)

    return true;
  }
}
```

- Ahora lo que tengo que hacer es ver si hay match con los roles

Verificar Role Usuario

- En tablePlus puedo ver que ninguno de los usuarios que tengo tiene el rol de admin ni super-user
- Para hacer la validación, si voy al GetUser, mi custom decorator, ya había extraído el user del request, es el mismo código sólo que aqui no tengo ctx si no context
- Le digo que el user es de tipo User (entidad)

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

    console.log({user: user.roles})

    return true;
  }
}
```

- Hago la solicitud, miro en la terminal y me dice el rol que tiene ese usuario
- Para hacer la evaluación usaré un for of

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

    console.log({user: user.roles})

    for(const role of user.roles){
      if(validRoles.includes(role)){
        return true
      }
    }

    throw new ForbiddenException(
      `User ${user.fullName} needs a valid role ${validRoles}`
    )
  }
}
```

- Si ahora cambio el role a admin desde tablePlus del usuario con el cual estoy apuntando al endpoint con el token me da acceso
- Hay muchas cosas que memorizar aquí: tengo que poner el SetMetadata
- Crearé un custom decorator que me ayude a establecer la metadata de una manera controlada
- NOTA: el SetMetadata se usa muy poco porque cualquier error afecta al resultado. Si le pongo role, en lugar de roles me deja pasar, por ejemplo

Custom Decorator - RoleProtected

- Si no es un decorador de propiedad, se puede usar el nest CLI
- El decorador está amarrado a auth? si, porque es para validar los roles. Entonces lo coloco en auth y no en common

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

- Me devuelve esto:

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

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

- Recibe un arreglo de strings y los establece en la metadata
- Establezco una variable para el roles

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

export const META_ROLES= "roles"

export const RoleProtected = (...args: string[]) => SetMetadata(META_ROLES, args);
```

- Lo establezco también en el user-role.guard.ts:

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

- Si al día de mañana tengo que cambiar el nombre solo lo cambio en un lugar
- Voy a crear una interface que me permita validar los roles que voy a permitir
- Bueno, en lugar de una interface va a ser una enumeración (enum)

```
export enum ValidRoles{
    admin,
    superuser,
    user
}
```

- Por defecto typescript los indexa como 0, 1, 2 pero tienen que ser strings, así que voy a nombrarlos como strings

```
export enum ValidRoles{
    admin = "admin",
    superuser= "superuser",
    user= "user"
}
```

- Ahora en lugar de decirle a RoleProtected que los ...args son un arreglo de tipo string (que va a permitirme usar cualquier cosa) va a ser un arreglo de ValidRoles

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

export const META_ROLES= "roles"

export const RoleProtected = (...args: ValidRoles[]) => SetMetadata(META_ROLES,
args);
```

- Ahora tengo que usar este decorador

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

```
      ok: true,
      user
    }
  }
```

- Voy a crear un decorador que unifique autenticación y autorización, ya que es un poco lioso y así uso metadata de forma controlada

Composición de decoradores

- Cuando quiero componer un decorador basado en otros decoradores uso applyDecorators de @nestjs/common
- Bien útil para agruparlos y hacer refactorizaciones
- Creo otro endpoint llamado private3, copia del private2
- Creo el auth.decorator.ts en /decorators
- Importo la interface de ValidRoles
- Importo RoleProtected sin la @rroba, los decoradores importados van sin la arroba
- Importo el UseGuards y dentro el AuthGuard, lo ejecuto porque así lo pide

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

export function Auth(...roles: ValidRoles[]){
    return applyDecorators(
        RoleProtected(...roles),
        UseGuards(AuthGuard(), UserRoleGuard)
    )
}
```

- Para darle más flexibilidad, en el UserRoleGuard voy a dejarlo pasar aunque no tenga ningún role
- Importante: validRoles con minúscula

> if(!validRoles) return true

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

@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())


    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 ${validRoles}`
    )
  }
}
```

- Coloco el @Auth en el auth.controller
- Le pongo como role válido el admin

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

- Cómo hago para validar como admin otro endpoint fuera del módulo? En el seed por ejemplo, para que solo los admin puedan ejecutarlo

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


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


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

}
```

- Esto me devuelve este error

> In order to use "defaultStrategy", please, ensure to import PassportModule in each place where AuthGuard() is being used. Otherwise, passport won't work correctly.

Auth en otros módulos

- AuthGuard está conectado a @nestjs/passport, y passport es un módulo.
- Hay que exportar ese Auth. Es el defaultStrategy lo que está demandando.
- Ya lo exporté en su momento, es lo que necesito para usar todo lo relacionado a passport del módulo auth
- Lo importo en seed.module

```
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 {}
```

- Lo importo también en products
- Ahora si uso el @Auth vacío (sin roles) en el products.controller, cómo lo hice más flexible, tiene que estar autenticado pero da igual si tiene roles o no.

```
@Get()
 @Auth()
 findAll(@Query()paginationDto: PaginationDto) {
   console.log(paginationDto)
   return this.productsService.findAll(paginationDto);
 }
```

- O puedo ponerle roles para que quien quiera crear un producto tiene que ser user

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

Usuario que creó el producto

- Sería útil saber que usuario creó el producto
- Abro la entidad de User en el módulo Auth y la entidad de Producto
- Cómo se relaciona un usuario con un producto? Un único usuario puede crear varios productos, pero varios productos pueden estar creados por un usuario.
- Del lado del usuario es uno a muchos, y del lado del producto de muchos a uno
- OneToMany, importo Product como entity
- Lo configuro: el primer valor es la otra entidad, luego voy a tener una instancia de mi producto y cómo se relaciona con esta tabla
- auth.entity: (antes del @BeforeInsert)

```
  @OneToMany(
       ()=>Product,
       (product)=> product.user
   )
   product: Product;
```

- El product.user todavía no existe en la tabla de productos. Guardo cambios y voy a la tabla de productos
- Relación ManyToOne, importo User como entidad
- product.entity

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

    )
    user: User
```

- Si ahora voy a tablePlus, en la parte de los productos, hay una nueva columna llamada userId
- TypeORM lo hizo por mi
- Para que pueda ver en la petición qué usuario cargó ese producto pongo el egaer en true

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

    )
    user: User
```

- Están todos en NULL.
- Lo siguiente es crear un producto y que especifique que usuario lo creó

Insertar userId en los productos

- Si tengo que especificar qué usuario creó el producto, obviamente tengo que estar autenticado
- Voy a productos.controller a la parte de la creación
- Pongo el Auth() vacío para manejar solo con usuarios autenticados.
- Uso el @GetUser que es dónde tengo toda la info del usuario y mando al método el user

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

- Ahora en el product.service necesito también un user de tipo User (entity)
- Añado el user después delas imágenes

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

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

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

      return {...product, images}

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

- Lo mismo con la actualización, que usuario lo actualizó. Antes de guardarlo uso el product.user = user

```
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('This product does not exists')

    const queryRunner=this.dataSource.createQueryRunner()

    product.user = user
    await queryRunner.connect()
    await queryRunner.startTransaction()

    try {
      if(images){
        await queryRunner.manager.delete(ProductImage,{product: id})

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

      }

      await queryRunner.manager.save(product) //esto puede fallar

      await queryRunner.commitTransaction()
      await queryRunner.release()
```

```
      //await this.productRepository.save(product)
      return this.findOnePlain(id)

    } catch (error) {

      await queryRunner.rollbackTransaction()
      await queryRunner.release()

      this.handleDBExceptions(error)

    }
  }
```

- Esto todavía da error en la parte del seed.service(está esperando un usuario para crearlo)
- Voy y comento el forEach del seed.service, ya no hay errores
- Si hago el Post de un producto me aparece el usuario que lo creó (gracias al eager), y en TablePlus aparece la relación de qué usuario lo creó

SEED de usuarios, productos, imágenes

- El seed.service necesita que especifique el usuario para la creación de productos
- Voy a crear un procedimiento para purgar la DB de forma manual y en el orden respectivo
- Porque si intento borrar un usuario y ese usuario está en products no me va a dejar
- Primero los productos porque mantienen la integridad referencial con los usuarios
- Puedo inyectar el repositorio del usuario como si estuviera en el servicio
- Creo el queryBuilder, delete es lo que quiero

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


@Injectable()
export class SeedService {

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

async  runSeed() {
    await this.insertNewProducts()
    return 'SEED EXECUTED'
  }
```

```
  private async deleteTables(){
    await this.productsService.deleteAllProducts()
    const queryBuilder = this.userRepository.createQueryBuilder()

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

  private async insertNewProducts(){
    await this.productsService.deleteAllProducts()

    const products = initialData.products

    const insertPromises = []

   //products.forEach(product=>{
   //  insertPromises.push(this.productsService.create( product ))
   //})
    await Promise.all( insertPromises)
  }

}
```

- Esto primero va a borrar todos los productos, luego los usuarios y como tengo cascade en los productos va a borrar las imágenes automáticamente
- Antes de insertar los productos voy a llamar al deleteTables()

```
async  runSeed() {
    await this.deleteTables()
    await this.insertNewProducts()
    return 'SEED EXECUTED'
  }
```

- Voy a seed.data.ts y creo una interface para SeedUser
- Añado users: SeedUser a la interface SeedData
- En SeedData también voy a tener users

```
interface SeedProduct {
    description: string;
    images: string[];
    stock: number;
    price: number;
    sizes: ValidSizes[];
    slug: string;
    tags: string[];
    title: string;
```

```
    type: ValidTypes;
    gender: 'men'|'women'|'kid'|'unisex'
}

type ValidSizes = 'XS'|'S'|'M'|'L'|'XL'|'XXL'|'XXXL';
type ValidTypes = 'shirts'|'pants'|'hoodies'|'hats';

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

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-user']
        }
    ],
    products: [.........data........]
```

- Ahora ya tengo mi initialData con un arreglo de usuarios y puedo usarlos para crearlos de manera automática
- Voy al servicio para insertar los usuarios
- Selecciono los usuarios del initialData
- Creo el arreglo de usuarios
- hago el push pero esto no los salva
- Los salvo y los guardo en una constante (para que no de error por no tener el id cuando lo inserto)
- Retorno el [0] para poder insertarlo en insertNewProducts

```
  private async insertUsers(){

    const seedUsers = initialData.users;
```

```
    const users: User[]= []

    seedUsers.forEach( user=>{
      users.push(this.userRepository.create( user ))
    })

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

    return dbUsers[0]
  }
```

- Creo una constante que se llame adminUser en runSeed y se lo mando a la función de insertProducts
- En la funciónde insertNewProducts voy a recibir un user de tipo User
- Este user se lo voy a mandar al create que está esperando el user (que daba error)

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


@Injectable()
export class SeedService {

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

async  runSeed() {
    await this.deleteTables()
    const adminUser = await this.insertUsers()
    await this.insertNewProducts(adminUser)
    return 'SEED EXECUTED'
  }

  private async deleteTables(){
    await this.productsService.deleteAllProducts()
    const queryBuilder = this.userRepository.createQueryBuilder()

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

  private async insertUsers(){
```

```
    const seedUsers = initialData.users;

    const users: User[]= []

    seedUsers.forEach( user=>{
      users.push(this.userRepository.create( user ))
    })

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

    return dbUsers[0]
  }

  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)
  }

}
```

- Quito el @Auth() del seed.controller para realizar el seed
- petición Get

> http://localhost:3000/api/seed

Encriptar contraseña de los usuarios SEED

- Las contraseñas de los usuarios Seed no están encriptadas
- Las encripto en seed-data

```
    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-user']
```

```
        }
    ],
```

> import * as bcrypt from 'bcrypt'

Check AuthStatus

- Necesito poder recibir el token mientras siga vigente y generar un nuevo token basado en este token
- Si no, si refresca la aplicación del lado del cliente no estaría autenticado
- El endpoint va a recibir el token con la información del email, password, fullName
- En el token solo tengo el id del usuario, que es todo lo que necesito para extraer la info
- auth.controller

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

- En el auth.service, para este punto tengo toda la info en el user, y el token es válido
- auth.service:

```
async checkAuthStatus(user: User){

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

- Hago un login para extraer el token, uso uno de los mails de test

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

```
{
  "email": "test1@google.com",
  "password": "Abc123"

}
```

- Copio el Bearer Token y lo introduzco en auth-bearer de ThunderClient y hago la petición Get al endpoint

> http://localhost:3000/api/auth/check-auth-status

- Me devuelve otro token con la info que hay en la DB