01 Autenticación Nest

Creando el proyecto

> nest new authentication

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

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

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

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


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

Configuración

> npm i @nestjs/config dotenv

- Importo el ConfigModule en app.module

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

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

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

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

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

}))
```

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

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

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

- Creo el archivo .env en la raiz del proyecto

```
NODE_ENV=development
```

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

Conexión MongoDB

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

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

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

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

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

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

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

@Injectable()
export class MongoConnectionService {


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

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

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

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

```
         this.dbConnection = await createConnection(DB_URI)

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

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


    }

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


}
```

- En development.env

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

- En configuration-mongo.ts

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

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

NOTA: El comando mongo para la terminal esta deprecado. Ir a "Editar Variables de entorno.." /Configuración/Variables de entorno/Path + Editar

(añadir la ruta de instalación de mongo (C:\Program Files\MongoDB\Server\6.0\bin)). Debes instalar mongosh de https://www.mongodb.com/try/download/shell el paquete msi

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

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

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

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

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

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

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

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

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

> MongoServerSelectionError: connect ECONNREFUSED ::1:27017

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

---

Crear módulo users

- En src/modules creo el modulo de users

> nest g mo users

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

> nest g res users

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

Creando user-dto

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

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


export class CreateUserDto {

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

```
    @IsEmail()
    email: string;

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

Creando interfaz de usuario

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

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

Creando el Schema del usuario

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

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

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

Inyectar el modelo en el servicio

- Debo añadir en providers el objeto provide. UseFactory es una función a la que le debo pasar el MongoConnectionModule que tengo een el app.module.
- Por tanto, **SACO el módulo MongoConnectionModule DE app.module** porque lo voy a importar en **users.module**
- En provide le paso un string que usaré posteriormente en el servicio (users.servcie) para inyectar el modelo con el decorador **@Inject**

- Uso la función useFactory y como parámetro le paso el servicio de Mongo, con el método getConnection le paso el modelo con el tipado IUSER, le paso el nombre del modelo (que le pongo ahora), el schema y la colección (la tabla)
- El Servicio **debe de estar exportado en exports de mongo-connection.module.ts**
- **Inject es necesario** para que el useFactory funcione, porque para hacer el .getConnection necesito inyectar el servicio

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

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

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

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

@Injectable()
export class UsersService {

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

Creando usuarios

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

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

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

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

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

```
@Injectable()
export class UsersService {

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

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

    return user.save()
  }
}
```

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

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

- En el endpoint

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

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

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

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

    await user.save()

    user.password = undefined

    return user
  }
```

- Hay otra manera que explica Herrera bastante más elegante

Encriptando contraseñas

- Con bcrypt

> npm i bcrypt

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

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

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

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

Obteniendo usuarios

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

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

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

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

Obteniendo un usario por su email

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

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

- Lo uso en createUser para comprobar que el usuario existe

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

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

  await user.save()

  user.password = undefined

  return user
}
```

Poblar usuarios

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

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

@Injectable()
export class UsersService {

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

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

```
      }
    ]

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

- Si refresco ya tengo los usuarios en la db

Creando auth module

- Uso el CLI

> nest g res auth

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

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

Archivo de configuración para auth

- Añado una nueva variable en development.env

```
SECRETKEY_AUTH=secretkey
```

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

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

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

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

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

Creando el dto de auth

- Copio el dto de User, es el mismo

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


export class AuthCredentialsDto {

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

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

Passport

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

```
npm i @nestjs/passport passport
```

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

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

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

Importando JwtModule usadno el secretkey

- Necesitamos colocar la secretkey en el PassportModule
- Instalamos jwt

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

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

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

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

```
      }
    },
    inject: [ConfigService]
  })
],
  controllers: [AuthController],
  providers: [AuthService]
})
export class AuthModule {}
```

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

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

- Nosotros como tal no podemos borrar tokens

Estrategia JWT

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

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

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

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

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

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

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

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

Validando el usuario

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

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

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

- Si no estás autorizado no te va a dejar pasar
- Si le pusiera un return true siempre dejaría

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


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

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

        return user;
    }
}
```

Validar credenciales

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

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

@Injectable()
```

```
export class AuthService {

  constructor(
    private usersService: UsersService
  ){}

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

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

Login

- Creo el endpoint en el controller

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

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

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

@Injectable()
export class AuthService {

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

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

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

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

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

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

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

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

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

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

---

jwt.io

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

---

Protegiendo los endpoints del método user

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

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

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

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

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

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

  @Post()
  @UseGuards(AuthGuard('jwt'))
```

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

  @Get()
  @UseGuards(AuthGuard('jwt'))
  getUsers() {
    return this.usersService.getUsers();
  }
}
```

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

Comprobando la validación de la estrategia

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

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


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

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

        return user;
```

```
    }
}
```

Devolviendo datos del usuario logueado

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

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

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

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

---

Documentar

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

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

```
  @ApiResponse({
  status: 201,
  description: "Login correcto"
})
login(@Body() authCredentials: AuthCredentialsDto){
    return this.authService.login(authCredentials)
}
```

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