Desenvolupament d'aplicacions amb Nest.js (I) Implementació d'una API REST


En este document continuarem amb el vist en l'anterior sobre el framework *Nest.js*, i desenvoluparem una API REST que utilitze MongoDB i Mongoose.

1. Connexió amb la base de dades MongoDB

Anem ara a connectar des de Nest amb una base de dades MongoDB. Utilitzarem Mongoose, com hem fet en temes anteriors, però esta vegada ho farem a través d'una llibreria pont de Nest, anomenada `@nestjs/mongoose`. Per tant, hem d'instal·lar totes dues llibreries en el nostre projecte:

```
npm install @nestjs/mongoose mongoose
```

En el mòdul principal del projecte (`app.module.ts`), importem `@nestjs/mongoose` i connectem a la base de dades emprant el mètode `forRoot` en la secció de `imports`:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ContacteModule } from './contacte/contacte.module';
// Afegim esta nova dependència també
import { MongooseModule } from '@nestjs/mongoose';

@Module({
 imports: [ContacteModule,
    MongooseModule.forRoot('mongodb://127.0.0.1/contactes')],
 controllers: [AppController],
 providers: [AppService],
})
export class AppModule {}
```

1.1. Definir esquemes i models

A l'hora de definir l'esquema associat a una col·lecció de la base de dades farem ús de l'entitat (element *entity*) generat per a l'element en qüestió. Típicament se situa en una subcarpeta `entities` dins de la carpeta del recurs (per exemple, `src/contacte/entities/contacte.entity.ts`). El que hem de fer és que eixa classe herete de la classe `Document` de *mongoose*, perquè es comporte com els documents de les

col·leccions. A més, li afegim el decorador `@Schema` per a indicar que definirem un esquema de *mongoose* en ella, i dins definim els camps amb els seus tipus de dades. Els validadors s'afigen en un decorador `@Prop`, per a cada camp. Així podria quedar l'esquema per als contactes:

```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { Document } from 'mongoose';

@Schema()
export class Contacte extends Document {

    @Prop({
        required: true,
        minlength: 5
    })
    nom: string;

    @Prop({
        required: true,
        min: 0,
        max: 120
    })
    edat: number;

    telefon: string;
}

export const ContacteSchema = SchemaFactory.createForClass(Contacte);
```

En el mòdul associat a l'entitat (el mòdul `src/contacte/contacte.module.ts` en l'exemple anterior) hem d'incloure l'esquema, juntament amb el nom de col·lecció associada:

```
...
import { MongooseModule } from '@nestjs/mongoose';
import { ContacteSchema } from './entities/contacte.entity';

@Module({
  imports: [
    MongooseModule.forFeature([{
        name: 'contactes',
        schema: ContacteSchema
    }])
  ],
  ...
})
export class ContacteModule {}
```

Associem l'esquema amb un nom de model (`contactes`, en este cas), mitjançant el mètode `forFeature`. Este nom de model s'associarà a un nom de col·lecció en MongoDB. Recorda que les col·leccions en Mongo sempre es nomenen en plural, per la qual cosa convé que li assignem el nom en plural directament nosaltres, encara que també podríem haver usat `Contacto.name` i que generara automàticament el plural a partir del nom de l'entitat.

> **Exercici 1**
>
> A partir del projecte `tasques-nest` de la sessió anterior, instal·la els paquets necessaris per a treballar amb *Mongoose* i connectar a una base de dades anomenada `tasques-nest`. Definix també l'esquema per a l'entitat *tasca* (fitxer `src/tasca/entities/tasca.entity.ts`): inclourem com a camps un nom, una prioritat (numèrica) i una data. Tots seran obligatoris, el nom ha de tindre una longitud mínima de 5 caràcters, i la prioritat ha de tindre uns valors entre 1 i 5 (inclusivament). Recorda incorporar este esquema al mòdul de tasques. Posa en marxa després l'aplicació (i el servidor MongoDB) i comprova com es crea la base de dades i la col·lecció corresponent.

2. Insercions i validació de dades

Passem ara a definir la inserció de documents, i com validar les dades que ens puguen arribar en la petició.

2.1. Les dades de la petició: el DTO de creació

Si recordes de la sessió anterior, quan generem els elements d'un recurs determinat (com els del contacte, o la tasca) es generava una carpeta `dto` amb un parell de classes dins. DTO són les sigles de *Data Transfer Object*, i són els elements que empra Nest per a encapsular la informació que s'envia en una petició client-servidor. En concret, es genera un DTO per a gestionar les dades de la creació d'objectes, i un altre per a la modificació o actualització.

Centrem-nos ara en el DTO de creació (per exemple `src/contacte/dto/create-contacte-dto.ts`). És una classe simple en la qual hem de definir quins camps s'enviaran en la petició d'inserció, i de quin tipus és cadascun. Típicament ací posarem els mateixos camps que definim en l'esquema corresponent:

```
export class CreateContacteDto {
    readonly nom: string;
    readonly edat: number;
    readonly telefon: string;
}
```

Definim els camps com `readonly` per a evitar que es modifiquen accidentalment, ja que, en principi, no són dades modificables (es reben de la petició i s'afigen a la base de dades).

2.2. Validació de dades

En moltes ocasions ens interessarà validar les dades de l'objecte que estem rebent abans d'inserir-lo o actualitzar-lo en la base de dades. Per a això tenim disponible un validador anomenat `ValidationPipe`. Per a poder utilitzar-ho hem d'instal·lar el paquet `class-validator`, que disposa d'un conjunt de decoradors per a indicar criteris de validació. Podem consultar-los ací. També hem d'instal·lar un paquet addicional anomenat `class-transformer`, que és utilitzat pel primer, així que instal·lem tots dos:

```
npm install class-validator class-transformer
```

Hem d'indicar quines condicions han de complir les dades del DTO perquè siguen vàlids. Editarem la classe DTO corresponent per a afegir els diferents validadors que necessitem:

```
import { IsString, MinLength, IsNotEmpty, IsInt, Min, Max } from 'class-validator';

export class CreateContacteDto {

    @IsString({message: "El nom ha de ser un text"})
    @MinLength(3, {message: "El nom ha de contindre almenys 3 lletres"})
    readonly nom: string;

    @IsNotEmpty({message:"L'edat és obligatòria"})
    @IsInt({message: "L'edat ha de ser un nombre enter"})
    @Min(0, {message:"L'edat mínima és 0"})
    @Max(120, {message: "L'edat màxima és 120"})
    readonly edat: number;

    readonly telefon: string;
}
```

Observa com podem afegir diversos validadors en cada camp, cadascun amb el seu propi missatge d'error en el cas que no es complisca eixa validació. Alguns validadores com `IsString` o `IsNotEmpty` ens permeten obligar al fet que un camp siga obligatori. En el nostre cas, tant el nom com l'edat són obligatoris, i el telèfon no ho seria.

2.3. El mètode d'inserció en el controlador

Anem ara al controlador, al seu servici `@post`. Veuràs que està ja preparat (o, en cas contrari, hem de crear-ho d'esta manera) per a rebre el DTO del cos de la petició:

```
@Post()
create(@Body() createContacteDto: CreateContactoDto) {
    return this.contacteService.create(createContacteDto);
}
```

El que farem simplement és afegir-li el `ValidationPipe` amb el decorador `@UsePipes`:

```
// Importem ValidationPipe de @nestjs/common al costat de la resta
import { ...., ValidationPipe, UsePipes } from '@nestjs/common';

...

// POST /contacte
@Post()
@UsePipes(ValidationPipe)
create(@Body() createContacteDto: CreateContacteDto) {
    ...
}
```

Això farà que, automàticament, es genere una resposta amb codi 400 (*Bad request*) si les dades que arriben en el DTO no són vàlides.

2.4. Altres nivells de validació

En el cas que vulguem validar dades en més d'una ruta, podem repetir el decorador `UsePipes` en cadascuna d'elles o pujar-lo a nivell de controlador, perquè ho utilitzen les rutes que ho necessiten:

```
@Controller('contactes')
@UsePipes(ValidationPipe)
export class ContacteController {
    ...
}
```

Anant un pas més enllà, podem incloure'l a nivell d'aplicació, perquè qualsevol element de la mateixa que treballe amb DTOs els aplique la validació corresponent. Afegiríem llavors esta configuració de validació en `main.ts`, de la manera següent:

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

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

    // Configuració dels validadors
    app.useGlobalPipes(
        new ValidationPipe({
            whitelist: true,
            forbidNonWhitelisted: true
        })
    );

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

El paràmetre `whitelist` a *true* indica que en tots els validadors s'ignoren camps addicionals que no estiguen especificats en el DTO (per si accidentalment s'envia algun paràmetre més que no es contempla en l'especificació). El paràmetre `forbidNonWhitelisted` a *true* va un pas més enllà, i genera un error si s'envia algun element no contemplat.

2.5. El servici d'inserció

Ens queda completar el codi del servici d'inserció (fitxer `src/contacte/contacte.service.ts`). Per a poder accedir a la col·lecció corresponent i fer insercions (i posteriorment cerques, esborraments, etc) necessitem *injectar* el model en el servici. Això es fa amb el decorador `@Injectmodel` de `@nestjs/mongoose`:

```
...
import { Model } from 'mongoose';
import { InjectModel } from '@nestjs/mongoose';
// Importem l'entitat si no ho hem fet abans
import { Contacte } from './entities/contacte.entity';

@Injectable()
export class ContactoService {
    constructor(@Injectmodel('contactes')
    private readonly contacteModel: Model<Contacte>) {}
}
```

El model s'associa a l'entitat `Contacte` que hem creat en passos previs, a través del genèric `Model<Contacte>`, de manera que es transformen els objectes del model per a acoblar-los a l'entitat.

A partir del constructor que hem definit, ja podem fer referència a l'objecte `this.contacteModel` en la resta de mètodes que definim, i així podrem realitzar les corresponents insercions, cerques, etc, sobre el model. El nostre mètode per a crear podria ser així:

```
async create(crearContacteDto: CreateContactoDto) {
    const nouContacte = await this.contacteModel.create(crearContacteDto);
    return nouContacte;
}
```

> **NOTA**: observa que hem definit el mètode com a asíncron per a gestionar les operacions amb la base de dades, que són asíncrones. Haurem de fer el mateix amb altres mètodes del servici.

> **Exercici 2:**
>
> Sobre l'exercici anterior, definix la inserció de tasques. Establix els mecanismes de validació en el DTO de creació (complint les mateixes regles indicades en l'esquema), amb els missatges d'error apropiats. Prova de fer diferents insercions des de *ThunderClient* o *Postman*, tant correctes com incorrectes, i verifica que els missatges d'error que es generen i les dades que es guarden en la col·lecció són adequats.

3. Altres operacions sobre el model

Realitzarem a continuació la resta d'operacions sobre el model: cerques, actualitzacions i esborraments. Les **cerques** són senzilles: n'hi ha prou amb utilitzar el corresponent mètode *find* en el servici i retornar el resultat. Per exemple, per a buscar per *id* podríem fer una cosa així:

```
async findOne(id: string) {
    const resultat = await this.contacteModel.findById(id);
    return resultat;
}
```

L'esborrament també resulta senzill:

```
async remove(id: string) {
    const resultat = await this.contacteModel.findByIdAndDelete(id);
    return resultat;
}
```

> **NOTA**: depenent de la versió de *mongoose* que estigues utilitzant, pots utilitzar diferents mètodes d'esborrament, com *findByIdAndDelete*, *findByIdAndRemove* (no existix en versions més recents), etc.

3.1. L'operació d'actualització

Per a abordar l'operació d'actualització hem de tindre en compte algunes qüestions addicionals. En primer lloc, en generar els recursos de l'element s'ha creat un DTO específic per a actualitzacions (per exemple, `src/contacte/dto/update-contacte.dto.ts` per al cas dels contactes). En principi podríem descartar este DTO i utilitzar el mateix que per a les insercions, però emprar este DTO alternatiu té els seus avantatges com que, per exemple, podem permetre actualitzacions parcials, sense obligar que des del client se'ns envien totes les dades del document (fins i tot les que no canvien). Això és gràcies a que la classe per al DTO d'actualització hereta de la de creació usant `PartialType`, la qual cosa permet que tinga un contingut parcial:

```
...

export class UpdateContacteDto extends PartialType(CreateContacteDto) {}
```

El controlador d'actualització respon per defecte al mètode `Patch`, encara que podem canviar-lo a `Put` fàcilment si el preferim:

```
@Patch(':id')
update(@Param('id') id: string, @Body() updateContacteDto: UpdateContacteDto) {
    return this.contacteService.update(id, updateContacteDto);
}
```

El mètode del servici que s'encarrega de fer l'actualització pot simplement buscar l'element pel seu *id* i cridar al mètode d'actualització passant-li el DTO amb les dades a actualitzar:

```
async update(id: string, updateContacteDto: UpdateContacteDto) {
    const contacteActualitzat = await
        this.contacteModel.findByIdAndUpdate.(id,
            {$set: updateContacteDto}, {new: true});
    return contacteActualitzat;
}
```

3.2. Gestionant errors

Evidentment, les operacions que hem realitzat abans (insercions, esborraments, cerques...) poden produir un error, i fins ara no hem vist com donar una resposta adequada a uns certs errors que es produïsquen, més enllà d'errors en la validació.

3.2.1. Ús d'*exception filters*

Nest incorpora una sèrie d'excepcions predefinides que generen automàticament un codi d'estat associat. Per exemple, la classe `BadRequestException` genera automàticament un codi *400 - Bad request* en la resposta

al client. Ací detallem alguns dels *exception filters* més habituals juntament amb el seu codi d'estat associat:

- `BadRequestException` : codi 400 (*Bad request*, per a errors en les dades de la petició)
- `UnauthorizedException` : codi 401 (per a intents de *login* incorrectes)
- `ForbiddenException` : codi 403 (per a accés no permés a recursos protegits)
- `NotFoundException` : codi 404 (*Not found*, per a URLs incorrectes)
- `RequestTimeoutException` : codi 408 (per a peticions que s'ha tardat massa temps a respondre)
- `InternalServerErrorException` : codi 500 (per a errors interns del servidor)

Podem trobar molts altres en la documentació oficial. Però... com utilitzar estos *exception filters* per a generar la resposta adequada davant un error? N'hi ha prou amb llançar l'excepció corresponent (important-la prèviament de `@nestjs/common`), afegint de manera opcional el missatge personalitzat que volem enviar en la resposta. Per exemple, així podríem tractar la cerca d'un contacte que no existix:

```
import { ... NotFoundException } from '@nestjs/common';

...

async findOne(id: string) {
    const resultat = await this.contacteModel.findById(id);
    if (!resultat)
        throw new NotFoundException(`ID '${ id }' de contacte no trobat`);
    return resultat;
}
```

3.2.2. Control manual de la validació

Si volem tindre un control total sobre el que s'envia al client en cada cas podem gestionar nosaltres l'excepció que es genera i el missatge d'error i codi d'estat que s'envia. De manera addicional, també podem encapsular el codi que pot fallar (per exemple, l'intent d'inserció), en un bloc `try..catch` i, en la clàusula `catch`, decidir quin missatge enviar i amb quin codi d'estat. Així podríem modificar el mètode d'inserció en el servici en qüestió:

```
async create(createContacteDto: CreateContacteDto) {
    try
    {
        const nouContacte =
            await this.contacteModel.create(createContactoDto);
        return { ok: true, resultat: nouContacte };
    } catch(error) {
        if(error.name == 'ValidationError')
        {
            throw new HttpException({
                ok: false,
                error: 'Error: dades de tasca no vàlids'
            }, HttpStatus.BAD_REQUEST);
        } else {
            throw new HttpException({
                ok: false,
                error: 'Error inserint tasca'
            }, HttpStatus.BAD_REQUEST);
        }
    }
}
```

> **NOTA**: els elements `HttpException` i `HttpStatus` pertanyen a `@nestjs/common`.

En el cas que no tinguem especial interés a personalitzar fins a este punt la resposta emesa, podem fer ús dels mecanismes vistos abans (*ValidationPipe*, *exception filters*) i delegar en Nest la gestió d'eixe error.

> **Exercici 3**:
>
> Sobre el projecte anterior completa els servicis i rutes restants, definint missatges d'error apropiats en cada cas:
>
> - Llistar totes les tasques (GET)
> - Buscar una tasca per la seua *aneu* (GET)
> - Esborrar una tasca (DELETE)
> - Modificar una tasca (PUT o PATCH)
>
> Crea una col·lecció en *Thunder Client*, *Postman* o una ferramenta similar i definix una petició de prova per a cadascun dels servicis implementats.

4. Autenticació basada en tokens

Com a últim pas en nostra API REST, vegem com afegir autenticació basada en tokens en el nostre projecte Nest. Els passos que seguirem són:

- Definir el mòdul de gestió d'usuaris

- Definir el mòdul d'autenticació
- Incorporar l'estàndard JWT al projecte
- Definir l'accés a recursos protegits

4.1. El mòdul de gestió d'usuaris

Començarem definint un mòdul per a gestionar els usuaris registrats en l'aplicació. Usarem un servici auxiliar que ens permetrà buscar si un usuari i password determinats existixen en el sistema.

```
nest g module usuari
nest g service usuari
```

> **NOTA**: en este cas no definim un controlador (*controller*) perquè no hi haurà cap ruta o *endpoint* específic per a accés als usuaris.

Per a tractar amb objectes de tipus *Usuari* en el sistema definirem una interfície i un DTO:

```
nest g interface usuari/interfaces/usuari
nest g class usuari/dto/UsuariDto
```

Emplenem el contingut de la interfície `src/usuari/interfaces/usuari/usuari.interface.ts` amb els camps que utilitzarem de cada usuari:

```
export interface Usuari {
    login: string;
    password: string;
}
```

Quant al DTO, el contingut és similar a l'anterior, encara que amb propietats *readonly*:

```
export class UsuarioDto {
    readonly login: string;
    readonly password: string;
}
```

Definirem el contingut del servici d'usuaris (fitxer `src/usuaris/usuari.service.ts`):

```
import { Injectable } from '@nestjs/common';
import { Usuari } from './interfaces/usuari/usuari.interface';

@Injectable()
export class UsuariService {
 // Llistat predefinit d'usuaris per a simplificar el procés
    private readonly usuaris: Usuari[] = [
        {
            login: 'usuari1',
            password: 'password1'
        },
        {
            login: 'usuari2',
            password: 'password2'
        },
    ];

    async buscar(login: string, password: string): Promise<Usuari | undefined> {
        return this.usuaris.find(o => o.login === login && o.password == password);
    }
}
```

Com veiem, hem definit un array predefinit d'usuaris on buscar a qui intente accedir, i un mètode `buscar` que ens retornarà l'usuari en qüestió (o *undefined* si no es troba).

Quant al mòdul d'usuaris, l'única cosa que hem d'afegir és l'exportació (propietat *exports*) del servici d'usuaris perquè el puga utilitzar qualsevol altre component de l'aplicació:

```
import { Module } from '@nestjs/common';
import { UsuariService } from './usuari.service';

@Module({
    providers: [UsuariService],
    exports: [UsuariService]
})
export class UsuarioModule {}
```

4.2. El mòdul d'autenticació

Definim ara un mòdul d'autenticació juntament amb el seu controlador i servici associat:

```
nest g module auth
nest g controller auth
nest g service auth
```

En este cas començarem pel servici d'autenticació `src/auth/auth.service.ts`. Este servici farà ús del servici d'usuaris per a buscar si l'usuari que ens arriba està registrat.

```
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { UsuariService } from '../usuari/usuari.service';

@Injectable()
export class AuthService {
    constructor(private usuariService: UsuariService) {}

    async login(login: string, password: string): Promise<any> {
        const usuari = await this.usuariService.buscar(login, password);
        if (!usuari) {
            throw new UnauthorizedException();
        }
        // Queda pendent la gestió del token ací
        return usuari;
    }
}
```

Ara actualitzem el mòdul d'autenticació per a importar el d'usuaris, ja que fem ús d'ell.

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

@Module({
    imports: [UsuariModule],
    controllers: [AuthController],
    providers: [AuthService]
})
export class AuthModule {}
```

Finalment, definim una ruta *login* en el controlador (de tipus POST) per a rebre les credencials en el cos de la petició i comprovar si són correctes.

```
import { Controller, Body, Post } from '@nestjs/common';
import { AuthService } from './auth.service';
import { UsuariDto } from 'src/usuari/dto/usuari-dto/usuari-dto';

@Controller('auth')
export class AuthController {

    constructor(private authService: AuthService) {}

    @Post('login')
    async login(@Body() usuariDto: UsuariDto) {
        return this.authService.login(usuariDto.login, usuariDto.password);
    }
}
```

4.3. Incorporar tokens JWT

Va arribar el moment d'incorporar tokens JWT a l'aplicació. Instal·larem per a això el mòdul `@nestjs/jwt` propi de Nest:

```
npm install @nestjs/jwt
```

Ara configurarem la gestió de tokens en el mòdul `auth.module.ts`:

```
...
import { JwtModule } from '@nestjs/jwt';

@Module({
    imports: [
        UsuariModule,
        JwtModule.register({
            // Per a no haver d'importar el mòdul en cada component
            global: true,
            // Paraula secreta
            // Podem guardar-la en fitxer .env extern, per exemple
            secret: 'la_meua_paraula_secreta',
            signOptions: {expiresIn: '2h'}
    })],
    providers: [AuthService],
    exports: [AuthService]
})
export class AuthModule {}
```

En el servici d'autenticació `auth.service.ts` generem un *payload* amb les credencials de l'usuari (*login*) i retornem un token amb eixes credencials. Modifiquem, per tant, l'anterior codi d'este fitxer perquè quede així:

```
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { UsuariService } from '../usuari/usuari.service';
import { JwtService } from '@nestjs/jwt';

@Injectable()
export class AuthService {
    constructor(
        private usuariService: UsuariService,
        private jwtService: JwtService
    ) {}

    async login(login: string, password: string): Promise<any> {
        const usuari = await this.usuariService.buscar(login, password);
        if (!usuari) {
            throw new UnauthorizedException();
        }
        // Generem un token amb el login de l'usuari
        let token = await this.jwtService.signAsync({login: login});
        return token;
    }
}
```

Emprarem este servici ara en el controlador `auth.controller.ts` per a recollir el token i enviar-ho en la resposta JSON al client, una vegada es valide correctament:

```
import { Controller, Body, Post } from '@nestjs/common';
import { AuthService } from './auth.service';
import { UsuariDto } from 'src/usuari/dto/usuari-dto/usuari-dto';

@Controller('auth')
export class AuthController {

    constructor(private authService: AuthService) {}

    @Post('login')
    async login(@Body() usuariDto: UsuariDto) {
        let token = await this.authService.login(usuariDto.login, usuariDto.password
        return {ok: true, resultat: token};
    }
}
```

4.4. Definir l'accés a recursos protegits

Finalment implementarem la gestió de l'accés a recursos protegits. Per a això definirem un *guard*, un component de l'aplicació que estarà pendent de si una petició podrà processar-se o no sobre la base d'unes certes condicions. En este cas la condició serà estar debitamente autenticat.

Creem per a això un arxiu `auth/auth.guard.ts`, amb el següent codi:

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

import { JwtService } from '@nestjs/jwt';
import { Request } from 'express';

@Injectable()
export class AuthGuard implements CanActivate {

    constructor(private jwtService: JwtService) {}

    async canActivate(context: ExecutionContext): Promise<boolean> {
        const request = context.switchToHttp().getRequest();
        const token = this.obtindreToken(request);
        if (!token) {
            throw new UnauthorizedException();
        }
        try {
            const payload = await this.jwtService.verifyAsync(token);
            request['usuari'] = payload;
        } catch {
            throw new UnauthorizedException();
        }
        return true;
    }

    private obtindreToken(request: Request): string | undefined {
        const [type, token] = request.headers.authorization?.split(' ') ?? [];
        return type === 'Bearer' ? token : undefined;
    }
}
```

En cada fitxer on hi haja rutes que protegir, haurem d'afegir el *guard* en cadascuna d'eixes rutes mitjançant el decorador `UseGuards` (que haurem d'afegir)

```
...
import { UseGuards } from '@nestjs/common';
import { AuthGuard } from './auth.guard';

...
@Controller('xxx')
export class XXXController {

    ...
    @UseGuards(AuthGuard)
    @Get('unaRuta')
    metode(...) {
        return ...;
    }
```

Exercici 4:

Fes una còpia del projecte `tasques-nest` i canvia-la de nom a `tasques-nest-jwt`. Realitza els passos següents en este nou projecte:

1. Crea un mòdul de gestió d'usuaris `usuaris`, com el que hem definit en els apunts, amb el servici associat per a buscar un usuari en una llista predefinida.
2. Crea un mòdul d'autenticació `auth`, amb el corresponent controlador i servici, com hem explicat abans en els apunts.
3. Incorpora el mòdul JWT de Nest i configura'l amb la duració i paraula secreta que vulgues, perquè retorne un token en cas d'èxit. Seguix els mateixos passos explicats en l'apartat corresponent d'estos apunts.
4. Afig un *guard* que s'encarregue de vigilar l'accés a recursos protegits en els controladors (en este cas, en el de tasques). Hauràs de protegir les operacions d'inserció, esborrament i modificació.
5. Prova l'accés als recursos públics i protegits des d'una col·lecció *Thunder Client* o similar.