Authorizations

Table of Contents

Nest g mo auth
Nest g gu auth/guards/apikey

- Crear un Guard

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

@Injectable()
export class ApikeyGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(
    context: ExecutionContext,
```

```
  ): boolean | Promise<boolean> | Observable<boolean> {
    const isPublic = this.reflector.get('isPublic', context.getHandler());
    if (isPublic) {
      return true;
    }

    const request = context.switchToHttp().getRequest<Request>();
    const authHeader = request.header('Auth');
    const isAuth = authHeader === '123';
    if (!isAuth) {
      throw new UnauthorizedException(`You don't have permission `);
    }
    return isAuth;
  }
}
```

- Como usar

```
import { ApikeyGuard } from './auth/guards/apikey.guard';
```

```
@UseGuards(ApikeyGuard)
@Controller()
```

Para excluir la validacion de Guard

```
  @SetMetadata('isPublic', true)
```

- Crear un decorador personalizado

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

export const IS_PUBLIC_KEY = 'isPublic';

export const Public = () => SetMetadata(IS_PUBLIC_KEY, true);
```

- Usar un decorador

```
  @Get('nuevo')
 // @SetMetadata('isPublic', true)
 //decorador
```

```
@Public()
 newEndpoint() {
   return 'yo soy nuevo';
 }
```

- Crear un guard con variable de entorno

```
import databaseConfig from '../../config/database.config';
```

```
@Inject(databaseConfig.KEY)
  private config: ConfigType<typeof databaseConfig>,
```

```
canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    const isPublic = this.reflector.get('isPublic', context.getHandler());
    if (isPublic) {
      return true;
    }

    const request = context.switchToHttp().getRequest<Request>();
    const authHeader = request.header('Auth');
    const isAuth = authHeader === this.config.apiKey;
    if (!isAuth) {
      throw new UnauthorizedException(`You don't have permission `);
    }
    return isAuth;
  }
}
```

- Hashing de contraseñas en TypeORM

```
   "bcrypt": "^5.0.1",
   "@types/bcrypt": "^3.0.0",
```

```
import * as bcrypt from 'bcrypt';
```

1. const hashPassword = await bcrypt.hash(newModel.password, 10);
2. newModel.password = hashPassword;
3. const model = await newModel.save();
4. const { password, ...rta } = model.toJSON(); elimina el password para ser enviado a la respuesta

```
async create(data: CreateUserDto) {
  const newModel = new this.userModel(data);
  const hashPassword = await bcrypt.hash(newModel.password, 10);
  newModel.password = hashPassword;
  const model = await newModel.save();
  const { password, ...rta } = model.toJSON();
  return rta;
}
```

- **Hashing de contraseñas en TypeORM Mongo**

```
npm i bcrypt
npm i @types/bcrypt
```

```
  "bcrypt": "^5.0.1",
  "@types/bcrypt": "^3.0.0",
```

pueden instalar el paquete:
`npm i nestjs-mongoose-exclude` luego en su Entity utilizar el decorador `@ExcludeProperty()` y por último utilizar un Interceptor a nivel de ruta o del controlador completo

- Passport

1. https://docs.nestjs.com/security/authentication
    a. https://docs.nestjs.com/recipes/passport

```
$ npm install --save @nestjs/passport passport passport-local


$ npm install --save-dev @types/passport-local
```

- **Ruta de login**

Local strategy

```
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';

import { AuthService } from './../services/auth.service';
import { Strategy } from 'passport-local';

@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy, 'local') {
  constructor(private authService: AuthService) {
    super({
      usernameField: 'email',
      passwordField: 'password',
    });
  }

  async validate(email: string, password: string) {
    const user = await this.authService.validateUser(email, password);
    if (!user) {
      throw new UnauthorizedException('not allow');
    }
    return user;
  }
}
```

Como lo usas, sabes que la Strategy ‘local’

```

@Controller('auth')
export class AuthController {
  @UseGuards(AuthGuard('local'))
```

```
  @Post()
  login(@Req() req: Request) {
    return req.user;
  }
}
```

- **JWT**

```
npm install --save @nestjs/jwt passport-jwt
npm install --save-dev @types/passport-jwt
```

```
  generateJWT(user: User) {
    const payload: PayloadToken = { role: user.role, sub: user._id };
    return {
      access_token: this.jwtService.sign(payload),
      user,
    };
  }
```

Registrar el Jwt para que lea el secreto desde un variable de entorno

```
import { JwtModule } from '@nestjs/jwt';
import databaseConfig from './../config/database.config';
import { ConfigType } from '@nestjs/config';
```

```
    JwtModule.registerAsync({
      inject: [databaseConfig.KEY],
      useFactory: (configService: ConfigType<typeof databaseConfig>) => {
        return {
          secret: configService.jwtSecret,
          signOptions: {
            expiresIn: '10d',
          },
        };
      },
    }),
```

1.- Primero creamos el archivo jwt.strategy.ts en el cual agregaremos lo siguiente

```
import { ExtractJwt, Strategy } from 'passport-jwt';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable } from '@nestjs/common';
import { jwtConstants } from './constants';
import { TokenPayloadModel } from './models/token.model';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
    constructor() {
        super({
            jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
            // OBTENDREMOS EL TOKEN LOS HEADERS COMO 'Bearer token'
            ignoreExpiration: false,
            // IGNORA LA EXPIRACION, EN TU CASO EL TIEMPO QUE LE HAYAS
PUESTO
          // EJE.  signOptions: { expiresIn: '24h' }, YO LE PUSE 1 DIA
            secretOrKey: jwtConstants.secret,
            // LA LLAVE SECRETA CON LA QUE FIRMAMOS EL TOKEN AL HACER LOGIN
        });
    }

   // ESTA FUNCION LO QUE HARA SERA RECIBIR EL TOKEN DECODIFICADO
   // CON LA CARGA DE DATOS QUE LE PUSIMOS AL HACER LOGIN
    async validate( payload: TokenPayloadModel ) {

        return payload;

    }
}
```

LA FUNCION VALIDATE ES IMPORTANTE YA QUE SI EL TOKEN ES VALIDO POR NUESTRO SERVIDOR, ESTE RETURN DE LA FUNCION MISMA DEVOLVERA EL PAYLOAD AL REQUEST DE NUESTRA RUTA A LA QUE VAYAMOS A PROTEGER, ESTO TE PUEDE SERVIR PARA A BUSCAR EN TU BD AL USUARIO O ALGUN PROCESO QUE REQUIERAS

2.- Lo siguiente es importar nuestra estrategia en nuestro modulo de autenticación en los providers
“auth.module.ts”

```
import { JwtStrategy } from './jwt.strategy';

@Module({
    imports: [
        ...Tus imports
    ],
    controllers: [Tus Controllers...],
    providers: [Tus demas providers..., JwtStrategy]
})
export class AuthModule {}
```

3.- Ya casi acabamos!, solo falta crear el guardián que protegerá nuestrs rutas, creamos un archivo llamado "jwt.auth.guard.ts" dentro de nuestor modulo de autenticacion

```
import { HttpException, HttpStatus, Injectable, UnauthorizedException }
from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {

    // CON ESTA FUNCION DE PODEMOS PERSONALIZAR SI
    //  QUEREMOS LANZAR ERRORES PERZONALIZADOS
    handleRequest(err, user, info) {
        if (err || !user) {

            throw new HttpException({
                status: HttpStatus.UNAUTHORIZED,
                error: 'Usuario no autorizado',
            }, HttpStatus.UNAUTHORIZED);

        }

        return user;

    }

}
```

4.- Por ultimo solo importamos nuestro guardián en nuestro controlador que queramos proteger
Yo usare mi controlador llamado Pages
y listo, nuestra ruta quedo protegida, para acceder a ella abría que enviar por los headers nuestro token

```
import { AuthGuard } from '@nestjs/passport';
import { JwtAuthGuard } from 'src/auth/guards/jwt-auth.guard';

@ApiTags('pages')
@UseGuards(JwtAuthGuard) // colocamos nuestro guardián
@Controller('pages')
export class PagesController {
  constructor(private pageService: PageService) { }

  )
  @Get()
  getPages() {

    return this.pageService.getAllPages();

  }

}
```

Responder

En la clase anterior vimos como puedes hacer para identificar un rol y de acuerdo a eso poder darle acceso a un endpoint, ahora te pongo el siguiente escenario:

¿Como crear un endpoint que retorne todas las órdenes relacionados con el usuario que tiene sesión?

Una solución válida a este problema es que puedes crear un endpoint que enviándole el ID de un usuario te retorne dichas órdenes, algo como esto:

`http://localhost:3000/users/1/orders`
Este endpoint me retornaría todas las órdenes del usuario con ID 1, sin embargo puede ser algo peligroso, es decir, este endpoint solo debería estar disponible para los roles administrativos, pero no para los usuarios porque si conoces el ID de algún cliente podrías obtener la información de órdenes de compra de un usuario.

La solución más práctica para este caso es crear un endpoint de este tipo:

`http://localhost:3000/profile/my-orders`

Como ves en el endpoint anterior no estás colocando de forma explícita el ID de un usuario. ¿Entonces como sabe a qué usuario le pertenecen las órdenes? Sencillo, sí nuestro usuario ya tiene una sesión, esta información ya está en el JWT que se le otorgó a ese usuario al autentificarse en el sistema así con eso podemos inferirlo de acuerdo a la sesión

Ok, ok, cerebrito, suena bien pero, ¿y el código…? Vamos a iniciar con ello. Lo primero es que vamos a hacer es crear un nuevo controlador para todos las solicitudes que sean de este tipo. Lo vamos a crear con el nombre `ProfileController`.

```
nest g co users/controllers/profile --flat
```

Allí vamos a exponer un nuevo endpoint que va a recibir la solicitud y gracias al decorador de `@UseGuards(JwtAuthGuard, RolesGuard)` nos aseguramos que tenga un token válido, así que en ese método recogemos la información de usuario que tiene esa sesión así:

```
  @Roles(Role.CUSTOMER)
  @Get('my-orders')
```

```
  getOrders(@Req() req: Request) {
    const user = req.user as PayloadToken;
    return this.orderService.findOne(user.sub);
  }
```

Simplemente con decirle que dentro del método que queremos la información del Request podemos obtener la información del usuario que tiene sesión, luego simplemente obtenemos el ID y ya con eso podemos crear un método en nuestro servicio de órdenes que las retorne, algo así:

```
  ordersByCustomer(customerId: number) {
    return this.orderRepo.find({
      where: {
        customer: customerId,
      },
    });
  }
```

Este sería el código completo del nuevo controlador `ProfileController`:

```
import { Controller, Get, UseGuards, Req } from '@nestjs/common';
import { ApiTags } from '@nestjs/swagger';

import { Request } from 'express';

import { JwtAuthGuard } from '../../auth/guards/jwt-auth.guard';
import { RolesGuard } from '../../auth/guards/roles.guard';
import { Roles } from '../../auth/decorators/roles.decorator';
import { Role } from '../../auth/models/roles.model';
import { PayloadToken } from 'src/auth/models/token.model';
import { OrdersService } from '../services/orders.service';

@UseGuards(JwtAuthGuard, RolesGuard)
@ApiTags('profile')
@Controller('profile')
@Controller('profile')
export class ProfileController {
  constructor(private orderService: OrdersService) {}

  @Roles(Role.CUSTOMER)
  @Get('my-orders')
  getOrders(@Req() req: Request) {
    const user = req.user as PayloadToken;
    return this.orderService.ordersByCustomer(user.sub);
  }
}
```

Puedes ver toda esta solución en la [rama 14.]