Desarrollo de aplicaciones con Nest.js
Implementación de una API REST


En este documento vamos a continuar con lo visto en el anterior sobre el framework *Nest.js*, y desarrollaremos una API REST que utilice MongoDB y Mongoose.

1. Conexión a la base de datos Mongo

Vamos ahora a conectar desde Nest con una base de datos MongoDB. Utilizaremos Mongoose, como hemos venido haciendo en temas anteriores, pero esta vez lo haremos a través de una librería puente de Nest, llamada `@nestjs/mongoose`. Por lo tanto, debemos instalar ambas librerías en nuestro proyecto:

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

En el módulo principal del proyecto (`app.module.ts`), importamos `@nestjs/mongoose` y conectamos a la base de datos empleando el método `forRoot` en la sección de `imports`:

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ContactoModule } from './contacto/contacto.module';
// Añadimos esta nueva dependencia también
import { MongooseModule } from '@nestjs/mongoose';

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

1.1. Definir esquemas y modelos

A la hora de definir el esquema asociado a una colección de la base de datos haremos uso de la entidad (elemento *entity*) generado para el elemento en cuestión. Típicamente se ubica en una subcarpeta `entities` dentro de la carpeta del recurso (por ejemplo, `src/contacto/entities/contacto.entity.ts`). Lo que debemos hacer es que esa clase herede de la

clase `Document` de *mongoose*, para que se comporte como los documentos de las colecciones. Además, le añadimos el decorador `@Schema` para indicar que definiremos un esquema de *mongoose* en ella, y dentro definimos los campos con sus tipos de datos. Los validadores se añaden en un decorador `@Prop`, para cada campo. Así podría quedar el esquema para los contactos:

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

@Schema()
export class Contacto extends Document {

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

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

    telefono: string;

}

export const ContactoSchema = SchemaFactory.createForClass(Contacto);
```

En el módulo asociado a la entidad (el módulo `src/contacto/contacto.module.ts` en el ejemplo anterior) debemos incluir el esquema, junto con el nombre de colección asociada:

```
...
import { MongooseModule } from '@nestjs/mongoose';
import { ContactoSchema } from './entities/contacto.entity';

@Module({
  imports: [
    MongooseModule.forFeature([{
        name: 'contactos',
        schema: ContactoSchema
    }])
  ],
  ...
})
export class ContactoModule {}
```

Asociamos el esquema con un nombre de modelo (`contactos`, en este caso), mediante el método `forFeature`. Este nombre de modelo se asociará a un nombre de colección en MongoDB. Recuerda que las colecciones en Mongo siempre se nombran en plural, por lo que conviene que le asignemos el nombre en plural directamente nosotros, aunque también podríamos haber usado `Contacto.name` y que generara automáticamente el plural a partir del nombre de la entidad.

> **Ejercicio 1**
>
> A partir del proyecto `tareas-nest` de la sesión anterior, instala los paquetes necesarios para trabajar con *Mongoose* y conectar a una base de datos llamada `tareas-nest`. Define también el esquema para la entidad *tarea* (fichero `src/tarea/entities/tarea.entity.ts`): incluiremos como campos un nombre, una prioridad (numérica) y una fecha. Todos serán obligatorios, el nombre debe tener una longitud mínima de 5 caracteres, y la prioridad debe tener unos valores entre 1 y 5 (inclusive). Recuerda incorporar este esquema al módulo de tareas. Pon en marcha luego la aplicación (y el servidor MongoDB) y comprueba cómo se crea la base de datos y la colección correspondiente.

2. Inserciones y validación de datos

Pasemos ahora a definir la inserción de documentos, y cómo validar los datos que nos puedan llegar en la petición.

2.1. Los datos de la petición: el DTO de creación

Si recuerdas de la sesión anterior, cuando generamos los elementos de un recurso determinado (como los del contacto, o la tarea) se generaba una carpeta `dto` con un par de clases dentro. DTO son las siglas de *Data Transfer Object*, y son los elementos que emplea Nest para encapsular la información que se envía en una petición cliente-servidor. En concreto, se genera un DTO para gestionar los datos de la creación de objetos, y otro para la modificación o actualización.

Centrémonos ahora en el DTO de creación (por ejemplo `src/contacto/dto/create-contacto-dto.ts`). Es una clase simple en la que tenemos que definir qué campos se van a enviar en la petición de inserción, y de qué tipo es cada uno. Típicamente aquí pondremos los mismos campos que definimos en el esquema correspondiente:

```
export class CreateContactoDto {
    readonly nombre: string;
    readonly edad: number;
    readonly telefono: string;
}
```

Definimos los campos como `readonly` para evitar que se modifiquen accidentalmente, ya que, en principio, no son datos modificables (se reciben de la petición y se añaden a la base de datos).

2.2. Validación de datos

En muchas ocasiones nos va a interesar validar los datos del objeto que estamos recibiendo antes de insertarlo o actualizarlo en la base de datos. Para esto tenemos disponible un validador llamado `ValidationPipe`. Para poder utilizarlo debemos instalar el paquete `class-validator`, que dispone de un conjunto de decoradores para indicar criterios de validación. Podemos consultarlos aquí. También debemos instalar un paquete adicional llamado `class-transformer`, que es utilizado por el primero, así que instalamos ambos:

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

Debemos indicar qué condiciones deben cumplir los datos del DTO para que sean válidos. Editaremos la clase DTO correspondiente para añadir los distintos validadores que necesitemos:

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

export class CreateContactoDto {
    @IsString({message: "El nombre debe ser un texto"})
    @MinLength(3, {message: "El nombre debe contener al menos 3 letras"})
    readonly nombre: string;

    @IsNotEmpty({message:"La edad es obligatoria"})
    @IsInt({message: "La edad debe ser un número entero"})
    @Min(0, {message:"La edad mínima es 0"})
    @Max(120, {message: "La edad máxima es 120"})
    readonly edad: number;

    readonly telefono: string;
}
```

Observa cómo podemos añadir varios validadores en cada campo, cada uno con su propio mensaje de error en el caso de que no se cumpla esa validación. Algunos validadores como `IsString` o `IsNotEmpty` nos permiten obligar a que un campo sea obligatorio. En nuestro caso, tanto el nombre como la edad son obligatorios, y el teléfono no lo sería.

2.3. El método de inserción en el controlador

Vamos ahora al controlador, a su servicio `@Post`. Verás que está ya preparado (o, en caso contrario, debemos crearlo de este modo) para recibir el DTO del cuerpo de la petición:

```
@Post()
create(@Body() createContactoDto: CreateContactoDto) {
    return this.contactoService.create(createContactoDto);
}
```

Lo que haremos simplemente es añadirle el `ValidationPipe` con el decorador `@UsePipes`:

```
// Importamos ValidationPipe de @nestjs/common junto a lo demás
import { ...., ValidationPipe, UsePipes } from '@nestjs/common';

...

// POST /contacto
@Post()
@UsePipes(ValidationPipe)
create(@Body() createContactoDto: CreateContactoDto) {
    ...
}
```

Esto hará que, automáticamente, se genere una respuesta con código 400 (*Bad request*) si los datos que llegan en el DTO no son válidos.

2.4. Otros niveles de validación

En el caso de que queramos validar datos en más de una ruta, podemos repetir el decorador `UsePipes` en cada una de ellas o subirlo a nivel de controlador, para que lo usen las rutas que lo necesiten:

```
@Controller('contactos')
@UsePipes(ValidationPipe)
export class ContactoController {
    ...
}
```

Yendo un paso más allá, podemos incluirlo a nivel de aplicación, para que cualquier elemento de la misma que trabaje con DTOs les aplique la validación correspondiente. Añadiríamos entonces esta configuración de validación en `main.ts`, de la siguiente forma:

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

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

  // Configuración de los validadores
  app.useGlobalPipes(
    new ValidationPipe({
        whitelist: true,
        forbidNonWhitelisted: true
    })
  );

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

El parámetro `whitelist` a *true* indica que en todos los validadores se ignoren campos adicionales que no estén especificados en el DTO (por si accidentalmente se envía algún parámetro más que no se contempla en la especificación). El parámetro `forbidNonWhitelisted` a *true* va un paso más allá, y genera un error si se envía algún dado no contemplado.

2.5. El servicio de inserción

Nos queda completar el código del servicio de inserción (fichero `src/contacto/contacto.service.ts`). Para poder acceder a la colección correspondiente y hacer inserciones (y posteriormente búsquedas, borrados, etc) necesitamos *inyectar* el modelo en el servicio. Esto se hace con el decorador `@InjectModel` de `@nestjs/mongoose`:

```
...
import { Model } from 'mongoose';
import { InjectModel } from '@nestjs/mongoose';
// Importamos la entidad si no lo hemos hecho antes
import { Contacto } from './entities/contacto.entity';

@Injectable()
export class ContactoService {
  constructor(@InjectModel('contactos')
    private readonly contactoModel: Model<Contacto>) {}
}
```

El modelo se asocia a la entidad `Contacto` que hemos creado en pasos previos, a través del genérico `Model<Contacto>`, de manera que se transforman los objetos del modelo para acoplarlos a la entidad.

A partir del constructor que hemos definido, ya podemos hacer referencia al objeto `this.contactoModel` en el resto de métodos que definamos, y así podremos realizar las correspondientes inserciones, búsquedas, etc, sobre el modelo. Nuestro método para crear podría ser así:

```
async create(createContactoDto: CreateContactoDto) {
    const nuevoContacto = await this.contactoModel.create(createContactoDto);
    return nuevoContacto;
}
```

> **NOTA**: observa que hemos definido el método como asíncrono para gestionar las operaciones con la base de datos, que son asíncronas. Tendremos que hacer lo mismo con otros métodos del servicio.

> **Ejercicio 2:**
>
> Sobre el ejercicio anterior, define la inserción de tareas. Establece los mecanismos de validación en el DTO de creación (cumpliendo las mismas reglas indicadas en el esquema), con los mensajes de error apropiados. Prueba a hacer distintas inserciones desde *ThunderClient* o *Postman*, tanto correctas como incorrectas, y verifica que los mensajes de error que se generan y los datos que se guardan en la colección son adecuados.

3. Otras operaciones sobre el modelo

Realizaremos a continuación el resto de operaciones sobre el modelo: búsquedas, actualizaciones y borrados. Las **búsquedas** son sencillas: basta con utilizar el correspondiente método *find* en el servicio y devolver el resultado. Por ejemplo, para buscar por *id* podríamos hacer algo así:

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

El borrado también resulta sencillo:

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

> **NOTA**: dependiendo de la versión de *mongoose* que estés utilizando, puedes utilizar distintos métodos de borrado, como *findByIdAndDelete*, *findByIdAndRemove* (no existe en versiones más recientes), etc.

3.1. La operación de actualización

Para abordar la operación de actualización tenemos que tener en cuenta algunas cuestiones adicionales. En primer lugar, al generar los recursos del elemento se ha creado un DTO específico para actualizaciones (por ejemplo, `src/contacto/dto/update-contacto.dto.ts` para el caso de los contactos). En principio podríamos descartar este DTO y utilizar el mismo que para las inserciones, pero emplear este DTO alternativo tiene sus ventajas como que, por ejemplo, podemos permitir actualizaciones parciales, sin obligar que desde el cliente se nos envíen todos los datos del documento (incluso los que no cambian). Esto es gracias a que la clase para el DTO de actualización hereda de la de creación usando `PartialType`, lo que permite que tenga un contenido parcial:

```
...

export class UpdateContactoDto extends PartialType(CreateContactoDto) {}
```

El controlador de actualización, por su parte, responde por defecto al método `Patch`, aunque podemos cambiarlo a `Put` fácilmente si lo preferimos:

```
@Patch(':id')
update(@Param('id') id: string, @Body() updateContactoDto: UpdateContactoDto) {
    return this.contactoService.update(id, updateContactoDto);
}
```

El método del servicio que se encarga de hacer la actualización puede simplemente buscar el elemento por su *id* y llamar al método de actualización pasándole el DTO con los datos a actualizar:

```
async update(id: string, updateContactoDto: UpdateContactoDto) {
    const contactoActualizado = await
        this.contactoModel.findByIdAndUpdate.(id,
            {$set: updateContactoDto}, {new: true});
    return contactoActualizado;
}
```

3.2. Gestionando errores

Evidentemente, las operaciones que hemos realizado antes (inserciones, borrados, búsquedas...) pueden producir un error, y hasta ahora no hemos visto cómo dar una respuesta adecuada a ciertos errores que se produzcan, más allá de fallos en la validación.

3.2.1. Uso de *exception filters*

Nest incorpora una serie de excepciones predefinidas que generan automáticamente un código de estado asociado. Por ejemplo, la clase `BadRequestException` genera automáticamente un código *400 - Bad request* en la respuesta al cliente. Aquí detallamos algunos de los *exception filters* más habituales junto con su código de estado asociado:

- `BadRequestException` : código 400 (*Bad request*, para errores en los datos de la petición)
- `UnauthorizedException` : código 401 (para intentos de *login* incorrectos)
- `ForbiddenException` : código 403 (para acceso no permitido a recursos protegidos)
- `NotFoundException` : código 404 (*Not found*, para URLs incorrectas)
- `RequestTimeoutException` : código 408 (para peticiones que se ha tardado demasiado tiempo en responder)
- `InternalServerErrorException` : código 500 (para errores internos del servidor)

Podemos encontrar muchos otros en la documentación oficial. Pero... ¿cómo utilizar estos *exception filters* para generar la respuesta adecuada ante un error? Basta con lanzar la excepción correspondiente (importándola previamente de `@nestjs/common`), añadiendo de forma opcional el mensaje personalizado que queremos enviar en la respuesta. Por ejemplo, así podríamos tratar la búsqueda de un contacto que no existe:

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

...

async findOne(id: string) {
    const resultado = await this.contactoModel.findById(id);
    if (!resultado)
        throw new NotFoundException(`ID '${ id }' de contacto no encontrado`);
    return resultado;
}
```

3.2.2. Control manual de la validación

Si queremos tener un control total sobre lo que se envía al cliente en cada caso podemos gestionar nosotros la excepción que se genera y el mensaje de error y código de estado que se envía. De forma adicional, también podemos encapsular el código que puede fallar (por ejemplo, el intento de inserción), en un bloque `try..catch` y, en la cláusula `catch`, decidir qué mensaje enviar y con qué código de estado. Así podríamos modificar el método de inserción en el servicio en cuestión:

```
async create(createContactoDto: CreateContactoDto) {
    try
    {
        const nuevoContacto =
            await this.contactoModel.create(createContactoDto);
        return { ok: true, resultado: nuevoContacto };
    } catch(error) {
        if(error.name == 'ValidationError')
        {
            throw new HttpException({
                ok: false,
                error: 'Error: datos de tarea no válidos'
            }, HttpStatus.BAD_REQUEST);
        } else {
            throw new HttpException({
                ok: false,
                error: 'Error insertando tarea'
            }, HttpStatus.BAD_REQUEST);
        }
    }
}
```

> **NOTA**: los elementos `HttpException` y `HttpStatus` pertenecen a `@nestjs/common`.

En el caso de que no tengamos especial interés en personalizar hasta este punto la respuesta emitida, podemos hacer uso de los mecanismos vistos antes (*ValidationPipe*, *exception filters*) y delegar en Nest la gestión de ese error.

> **Ejercicio 3**:
>
> Sobre el proyecto anterior completa los servicios y rutas restantes, definiendo mensajes de error apropiados en cada caso:
>
> - Listar todas las tareas (GET)
> - Buscar una tarea por su *id* (GET)
> - Borrar una tarea (DELETE)
> - Modificar una tarea (PUT o PATCH)
>
> Crea una colección en *Thunder Client*, *Postman* o una herramienta similar y define una petición de prueba para cada uno de los servicios implementados.

4. Autenticación basada en tokens

Como último paso en nuestra API REST, veamos cómo añadir autenticación basada en tokens en nuestro proyecto Nest. Los pasos que seguiremos son:

- Definir el módulo de gestión de usuarios
- Definir el módulo de autenticación
- Incorporar el estándar JWT al proyecto
- Definir el acceso a recursos protegidos

4.1. El módulo de gestión de usuarios

Comenzaremos definiendo un módulo para gestionar los usuarios registrados en la aplicación. Usaremos un servicio auxiliar que nos permitirá buscar si un usuario y password determinados existen en el sistema.

```
nest g module usuario
nest g service usuario
```

> **NOTA**: en este caso no definimos un controlador (*controller*) porque no va a haber ninguna ruta o *endpoint* específico para acceso a los usuarios.

Para tratar con objetos de tipo *Usuario* en el sistema definiremos una interfaz y un DTO:

```
nest g interface usuario/interfaces/usuario
nest g class usuario/dto/UsuarioDto
```

Rellenamos el contenido de la interfaz `src/usuario/interfaces/usuario.interface.ts` con los campos que utilizaremos de cada usuario:

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

En cuanto al DTO, el contenido es similar al anterior, aunque con propiedades *readonly*. Lo podemos crear por ejemplo en `src/usuario/dto/usuario.dto.ts`:

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

Vamos a definir el contenido del servicio de usuarios (fichero `src/usuarios/usuario.service.ts`):

```
import { Injectable } from '@nestjs/common';
import { Usuario } from './interfaces/usuario/usuario.interface';

@Injectable()
export class UsuarioService {
    // Listado predefinido de usuarios para simplificar el proceso
    private readonly usuarios: Usuario[] = [
        {
          login: 'usuario1',
          password: 'password1'
        },
        {
          login: 'usuario2',
          password: 'password2'
        },
    ];

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

Como vemos, hemos definido un array predefinido de usuarios donde buscar a quien intente acceder, y un método `buscar` que nos devolverá el usuario en cuestión (o *undefined* si no se encuentra).

En cuanto al módulo de usuarios, lo único que tenemos que añadir es la exportación (propiedad *exports*) del servicio de usuarios para que lo pueda utilizar cualquier otro componente de la aplicación:

```
import { Module } from '@nestjs/common';
import { UsuarioService } from './usuario.service';

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

4.2. El módulo de autenticación

Definimos ahora un módulo de autenticación junto con su controlador y servicio asociado:

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

En este caso vamos a comenzar por el servicio de autenticación `src/auth/auth.service.ts`. Dicho servicio hará uso del servicio de usuarios para buscar si el usuario que nos llega está registrado.

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

@Injectable()
export class AuthService {
  constructor(private usuarioService: UsuarioService) {}

  async login(login: string, password: string): Promise<any> {
    const usuario = await this.usuarioService.buscar(login, password);
    if (!usuario) {
      throw new UnauthorizedException();
    }
    // Queda pendiente la gestión del token aquí
    return usuario;
  }
}
```

Ahora actualizamos el módulo de autenticación para importar el de usuarios, ya que hacemos uso de él.

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

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

Finalmente, definimos una ruta *login* en el controlador (de tipo POST) para recibir las credenciales en el cuerpo de la petición y comprobar si son correctas.

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

@Controller('auth')
export class AuthController {

    constructor(private authService: AuthService) {}

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

4.3. Incorporar tokens JWT

Llegó el momento de incorporar tokens JWT a la aplicación. Instalaremos para ello el módulo `@nestjs/jwt` propio de Nest:

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

Ahora vamos a configurar la gestión de tokens en el módulo `auth.module.ts`:

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

@Module({
  imports: [
    UsuarioModule,
    JwtModule.register({
      // Para no tener que importar el módulo en cada componente
      global: true,
      // Palabra secreta
      // Podemos guardarla en fichero .env externo, por ejemplo
      secret: 'mi_palabra_secreta',
      signOptions: {expiresIn: '2h'}
    })],
  controllers: [AuthController],
  providers: [AuthService],
  exports: [AuthService]
})
export class AuthModule {}
```

En el servicio de autenticación `auth.service.ts` generamos un *payload* con las credenciales del usuario (*login*) y devolvemos un token con esas credenciales. Modificamos, por tanto, el anterior código de este fichero para que quede así:

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

@Injectable()
export class AuthService {
  constructor(
    private usuarioService: UsuarioService,
    private jwtService: JwtService
  ) {}

  async login(login: string, password: string): Promise<any> {
    const usuario = await this.usuarioService.buscar(login, password);
    if (!usuario) {
      throw new UnauthorizedException();
    }
    // Generamos un token con el login del usuario
    let token = await this.jwtService.signAsync({login: login});
    return token;
  }
}
```

Emplearemos este servicio ahora en el controlador `auth.controller.ts` para recoger el token y enviarlo en la respuesta JSON al cliente, una vez se valide correctamente:

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

@Controller('auth')
export class AuthController {

    constructor(private authService: AuthService) {}

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

4.4. Definir el acceso a recursos protegidos

Finalmente vamos a implementar la gestión del acceso a recursos protegidos. Para ello definiremos un *guard*, un componente de la aplicación que va a estar pendiente de si una petición va a poder procesarse o no en base a ciertas condiciones. En este caso la condición será estar debitamente autenticado.

Creamos para ello un archivo `auth/auth.guard.ts`, con el siguiente código:

```
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.extraerToken(request);
    if (!token) {
      throw new UnauthorizedException();
    }
    try {
      const payload = await this.jwtService.verifyAsync(token);
      request['usuario'] = payload;
    } catch {
      throw new UnauthorizedException();
    }
    return true;
  }

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

En cada fichero donde haya rutas que proteger, deberemos añadir el *guard* en cada una de esas rutas mediante el decorador `UseGuards` (que deberemos añadir)

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

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

  ...
  @UseGuards(AuthGuard)
  @Get('miRuta')
  metodo(...) {
    return ...;
  }
```

Ejercicio 4:

Haz una copia del proyecto `tareas-nest` y renómbrala a `tareas-nest-jwt`. Realiza los pasos siguientes en este nuevo proyecto:

1. Crea un módulo de gestión de usuarios `usuarios`, como el que hemos definido en los apuntes, con el servicio asociado para buscar un usuario en una lista predefinida.
2. Crea un módulo de autenticación `auth`, con el correspondiente controlador y servicio, como hemos explicado antes en los apuntes.
3. Incorpora el módulo JWT de Nest y configúralo con la duración y palabra secreta que quieras, para que devuelva un token en caso de éxito. Sigue los mismos pasos explicados en el apartado correspondiente de estos apuntes.
4. Añade un *guard* que se encargue de vigilar el acceso a recursos protegidos en los controladores (en este caso, en el de tareas). Deberás proteger las operaciones de inserción, borrado y modificación.
5. Prueba el acceso a los recursos públicos y protegidos desde una colección *Thunder Client* o similar.