Tema 3. Opciones avanzadas de Express.js

3.4. Desarrollo de aplicaciones con Nest.js

Nest.js es un framework de desarrollo web en el servidor que, apoyándose en el framework **Express**, permite construir aplicaciones robustas y escalables utilizando una terminología muy similar a la que se emplea en el framework de cliente *Angular*. Al igual que Angular, utiliza lenguaje TypeScript para definir el código, aunque también es compatible con JavaScript.

Nest.js proporciona la mayoría de características que cualquier framework de desarrollo en el servidor proporciona, tales como mecanismos de autenticación, uso de ORM para acceso a datos, desarrollo de servicios REST, enrutado, etc.

3.4.1. Instalación y creación de proyectos

Nest.js se instala como un módulo global al sistema a través del gestor de paquetes `npm`, con el siguiente comando:

```
npm i -g @nestjs/cli
```

Una vez instalado, para crear un proyecto utilizamos el comando `nest`, con la opción `new`, seguida del nombre del proyecto.

```
nest new nombre_proyecto
```

> **NOTA**: en la creación del proyecto, puede que el asistente pregunte qué gestor de paquetes vamos a utilizar. Lo normal es seleccionar `npm`.

Esto creará una carpeta con el nombre del proyecto en nuestra ubicación actual, y almacenará dentro toda la estructura básica de archivos y carpetas de los proyectos Nest. Podemos consultar la información del proyecto generado con el comando `i` (o `info`):

```
nest i
```

Obtendremos una salida similar a esta (variando los números de versión):

```
[System Information]
OS Version     : Linux 5.15
NodeJS Version : v18.12.1
NPM Version    : 8.19.2

[Nest CLI]
Nest CLI Version : 9.1.5

[Nest Platform Information]
platform-express version : 9.2.0
schematics version       : 9.0.3
testing version          : 9.2.0
common version           : 9.2.0
core version             : 9.2.0
cli version              : 9.1.5
```

3.4.1.1. Estructura de un proyecto Nest.js

La estructura de carpetas y archivos creada por el comando `nest` tiene una serie de elementos clave que conviene resaltar. La mayor parte de nuestro código fuente se ubicará en la carpeta `src`. Entre otras cosas, podemos encontrar:

- El módulo principal `app.module.ts`, ya definido
- Un controlador de ejemplo, llamado `app.controller.ts`, con una ruta definida hacia la raíz de la aplicación.
- El archivo `main.ts`, que define la inicialización de la aplicación Crea una instancia del módulo principal `AppModule`, y se queda escuchando por un puerto determinado (que se puede modificar en este mismo archivo). Define para ello una función asíncrona, que luego se lanza para poner en marcha todo:

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}
bootstrap();
```

3.4.1.2. Poner en marcha el proyecto

El asistente de creación del proyecto lo habrá dejado todo preparado, con las dependencias ya instaladas, e incluso el archivo `package.json` ya generado, para poder poner en marcha el proyecto. Sólo tenemos que ejecutar el siguiente comando:

```
npm run start
```

o con :dev que nos permite recompilar automáticamente cuando guardamos un archivo.

```
npm run start --watch
```

Si intentamos acceder a `http://localhost:3000` veremos un mensaje de bienvenida proporcionado por el módulo principal ("Hello World!").

3.4.1.3. Conexión con Express

Como hemos comentado, Nest.js utiliza internamente el framework Express para trabajar sobre él. Esto hace que no tengamos por qué acceder directamente a ciertos elementos que tenemos disponibles en dicho framework, como la petición (`req`), respuesta (`res`), parámetros de la URL (`req.params`), cuerpo de la petición (`req.body`), etc. En su lugar, Nest.js proporciona una serie de decoradores que iremos viendo más adelante, y que internamente se comunican con estas propiedades de Express. Por ejemplo, el decorador `@Param` lo emplearemos para acceder a parámetros de la URL, y el decorador `@Body` para acceder al cuerpo de la petición.

3.4.2. Estructurando la aplicación: módulos, controladores y servicios

Cuando creamos una aplicación Nest, inicialmente ya tenemos algo de código generado en la carpeta `src` . En concreto, disponemos del módulo principal de la aplicación, `app.module.ts` , que se encargará de coordinar al resto de módulos que definamos. Cada módulo debe encargarse de encapsular y gestionar un conjunto de características sobre un concepto de la aplicación. Por ejemplo, en una aplicación de una tienda online podemos tener un módulo que gestione los clientes (listados, altas, bajas), otro para el stock de productos, otro para los pedidos, etc.

Cada uno de los módulos de nuestra aplicación puede (suele) disponer de una serie de elementos adicionales que le ayuden a dividir el trabajo. Así, en cada uno de los módulos podemos tener:

- **Controladores** (*controllers*), que se encargarán de atender las peticiones relacionadas con dicho módulo. Por ejemplo, en un módulo de clientes, tendremos un controlador que se encargará de atender peticiones de listados de clientes, altas, bajas, etc.
- **Servicios** (*services*), que se encargarán de gestionar el acceso a los datos para un determinado módulo. Así, volviendo al ejemplo de los clientes, podremos tener un servicio que se encargue de realizar efectivamente las búsquedas, inserciones, borrados, etc, en la colección de datos correspondiente. En realidad, los servicios son un tipo especial de **proveedores** (*providers*), elementos que utiliza Nest para realizar tareas específicas y relativamente complejas, descargando así de trabajo a los controladores, que se limitan a atender peticiones.

De hecho, nuestro módulo principal `app.module.ts` cuenta con un servicio asociado `app.service.ts` y un controlador, `app.controller.ts`, ya creados. Inicialmente no hacen gran cosa, ya que el controlador sólo dispone de una ruta para cargar una página de bienvenida con un saludo simple, y el servicio se encarga de proporcionar ese mensaje de saludo. Pero es un punto de partida para comprender cómo se estructura el reparto de tareas en aplicaciones Nest.

3.4.2.1. Definiendo módulos, controladores y servicios

Volvamos al tema de los módulos. Un módulo básicamente es una clase TypeScript anotada con el decorador `@Module`, que proporciona una serie de metadatos para construir la estructura de la aplicación. Como ya hemos visto, toda aplicación Nest tiene al menos un módulo raíz o *root*, el archivo `app.module.ts` explicado anteriormente, que sirve de punto de entrada a la aplicación, de forma similar a como funciona Angular.

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module ({
    imports: [],
    controllers: [AppController],
    providers: [AppService],
})

export class AppModule {}
```

El decorador `@Module` toma un objeto como parámetro, donde se definen los controladores, proveedores de servicios y otros elementos que iremos viendo más adelante.

Para **crear un nuevo módulo** para nuestro proyecto, escribimos el siguiente comando desde la raíz del proyecto:

```
nest g module nombre_modulo
```

Se creará una carpeta `nombre_modulo` dentro de la carpeta `src`, y se añadirá la correspondiente referencia en la sección `imports` del módulo principal `AppModule`. Por ejemplo, si creamos un módulo llamado `contacto`, se creará la carpeta `contacto`, y la sección `imports` del módulo principal quedará así (notar que a la clase que se genera se le añade el sufijo "Module" automáticamente):

```
@Module({
    imports: [ContactoModule],
    ...
})
```

Así, el módulo principal ya incorpora al módulo `contacto`, y todo lo que éste contenga a su vez. En la carpeta correspondiente (`contacto`, siguiendo el ejemplo anterior) se generará un archivo TypeScript (`contacto.module.ts`, en nuestro ejemplo) con la nueva clase del módulo generada.

Del mismo modo, podemos generar controladores y servicios, con estos comandos:

```
nest g controller nombre_controlador
nest g service nombre_servicio
```

Otra posibilidad para generar los controladores y servicios es usando su alias:

```
nest g co nombre_controlador
nest g s nombre_servicio
```

Si seguimos con el caso anterior, podemos crear un controlador llamado `contacto` y un servicio con el mismo nombre. Esto generará respectivamente los archivos `src/contacto/contacto.controller.ts` y `src/contacto/contacto.service.ts` en nuestro proyecto.

Al seguir estos pasos, el propio módulo `contacto.module.ts` tendrá ya registrados su controlador y servicio, con lo que está ya todo conectado para poder empezar a trabajar:

```
import { Module } from '@nestjs/common';
import { ContactoController } from './contacto.controller';
import { ContactoService } from './contacto.service';

@Module({
  controllers: [ContactoController],
  providers: [ContactoService]
})
export class ContactoModule {}
```

Observemos la jerarquía de dependencias que se está creando: el módulo principal incorpora en su bloque `imports` al módulo `contacto`, y éste a su vez contiene en su interior las dependencias con el controlador y el servicio propios.

Es importante definir los elementos en este orden (primero el módulo, y luego sus controladores y servicios), ya que de lo contrario tendríamos que definir estas dependencias a mano en el código.

> **Ejercicios propuestos**
>
> **1.** Instala Nest si todavía no lo has hecho, y crea un proyecto llamado `tareas-nest` con el siguiente comando: `nest new tareas-nest`.
>
> Después, crea desde la carpeta principal del proyecto un módulo llamado `tarea`, con el comando `nest g module tarea`. Se habrá creado una carpeta `src/tarea` en el proyecto, con el módulo `tarea.module.ts` en su interior.
>
> De forma similar, crea también el controlador y servicio con el mismo nombre `tarea`.

3.4.2.2. Preparando el modelo de datos

A la hora de almacenar en la aplicación los datos con los que vamos a trabajar, es recomendable definir dos elementos:

- Una interfaz (*interface*) que defina los campos de cada objeto a tratar
- Un DTO (*Data Transfer Object*) que defina los datos que se van a enviar entre cliente y servidor, especialmente en las peticiones de inserción (POST) y modificación (PUT).

La interfaz simplemente almacenará los campos a tratar. Básicamente es una definición de clase, pero en TypeScript esto se suele hacer definiendo interfaces. La creamos con el siguiente comando desde la raíz del proyecto (suponiendo el ejemplo de `contacto` que hemos venido contando):

```
nest g interface contacto/interfaces/contacto
```

Esto creará un archivo `src/contacto/interfaces/contacto.interface.ts`. Podemos editarlo para definir qué campos va a tener un objeto de este tipo:

```
export interface Contacto {
  id: string;
  nombre: string;
  edad: number;
  telefono: string;
}
```

> **Nota**: Si queréis aplicar alguna validación a los campos creados en la interfaz de arriba, podéis usar la clase class-validator que nos permite verificar campos como @IsEmail(), @IsNotEmpty(), etc. Ver más info aquí.

Esta interfaz se puede incorporar al servicio o servicios que vayan a hacer uso de estos datos. Por ejemplo, en el servicio de contactos (archivo `src/contacto/contacto.service.ts`) podemos definir un atributo que sea un array de objetos de la interfaz `Contacto`, importando previamente dicha interfaz:

```
import { Injectable } from '@nestjs/common';
import { Contacto } from './interfaces/contacto/contacto.interface';

@Injectable()
export class ContactoService {
    contactos: Contacto[] = [];
}
```

De forma similar, creamos el DTO como una clase. Podemos crearla en la misma subcarpeta `interfaces` donde hemos definido la interfaz, o en una subcarpeta `dto` propia:

```
nest g class contacto/dto/ContactoDto
```

Esto creará la clase `ContactoDto` en el archivo `src/contacto/dto/contacto-dto.ts`. Podemos definir dentro campos similares a los de la interfaz, ya que en principio se enviarán cliente y servidor los mismos campos aproximadamente que luego se van a almacenar para cada objeto. En el caso del DTO, podemos definir los campos como `readonly`, ya que son de sólo lectura (se envían al servidor para que los recoja y almacene):

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

Ejercicios propuestos

2. Sobre el proyecto anterior, crea ahora la interfaz `tarea` en la subcarpeta `interfaces`, con el comando `nest g itf tarea/interfaces/tarea`. Define los siguientes campos para cada tarea que vamos a gestionar:

- Un `id` de tipo texto
- El `nombre` de la tarea (texto)
- La `prioridad` de la tarea (entero)
- La `fecha` tope de finalización de la tarea (fecha)

Añade un array de tareas (objetos de la interfaz `Tarea` que acabas de crear) en el servicio de la tarea (`src/tarea/tarea.service.ts`).

> Después, crea un DTO llamado `TareaDto` con el comando `nest g cl tarea/dto/TareaDto`. Define dentro los mismos campos que hemos definido para la interfaz (salvo el *id*), de tipo *sólo lectura*.

3.4.3. Más sobre los controladores

Los controladores en Nest.js se encargan de gestionar las peticiones y respuestas a los clientes. Internamente, son clases con el decorador `@Controller`, que añade metainformación para crear un mapa de enrutado. Gracias este mapa, Nest sabe cómo enviar las peticiones que lleguen al controlador adecuado.

Ya hemos visto cómo crear controladores en Nest, y que queden asociados a un módulo previamente creado. Suponiendo el controlador de `contacto` del ejemplo anterior, su estructura básica al crearse es la siguiente:

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

@Controller('contacto')
export class ContactoController {
    ...
}
```

El parámetro que tiene el controlador es el prefijo en la URL para acceder a él. Así, cualquier ruta que vaya a ser recogida por este controlador tendrá la estructura `http://localhost:3000/contacto` (suponiendo la ruta y puerto por defecto definido en `main.ts`).

3.4.3.1. Definir *handlers* para recoger las peticiones

Para gestionar estas peticiones, se deben definir unos métodos en el controlador, llamados manejadores o *handlers*. Estos métodos utilizan decoradores que son verbos HTTP, que indican a qué tipo de método responder (`@Get`, `@Post`, `@Put`, `@Delete` ...). Entre paréntesis, podemos indicar una ruta adicional al prefijo del controlador. Si no especificamos ninguna, se entiende que responden a la ruta raíz del controlador.

@Get

Por ejemplo, así podríamos definir *handlers* de tipo `@Get` para obtener un listado general, y un dato a partir de su *id*, respectivamente. Deberemos importar el decorador junto con el resto de elementos necesarios del paquete `@nestjs/common`.

```
import { Controller, Get, Param } from '@nestjs/common';

@Controller('contacto')
export class ContactoController {

    // GET /contacto
    @Get()
    listar() {
        ...
    }

    // GET /contacto/buscar/:id
    @Get('buscar/:id')
    buscarPorId(@Param('id') id: string) {
        ...
    }
}
```

En el caso del segundo *handler*, utilizamos un decorador `@Param` para acceder al parámetro que queramos de la URL (en este caso, el parámetro `id`), y asociarlo a un nombre de variable, que será el que utilicemos en el código del *handler*. En este caso, la variable se llama igual que el parámetro, pero podría tener un nombre diferente si quisiéramos.

A la hora de emitir una respuesta, deberemos devolver (`return`) un resultado. Nest.js serializa automáticamente objetos JavaScript a formato JSON, mientras que si enviamos un tipo simple (por ejemplo, un entero, o una cadena de texto), lo envía como texto plano. Por lo tanto, normalmente no tendremos que preocuparnos por esta tarea. Podemos devolver algo como esto:

```
// GET /contacto
@Get()
listar() {
    return {
        resultado: ... // Datos buscados
    };
}
```

@Post

Para trabajar con peticiones de tipo POST, utilizaremos el decorador `@Body` para recoger los datos del cuerpo de la petición. Sin embargo, para que estos datos puedan ser procesados, necesitamos definir un **DTO** (*Data Transfer Object*) en TypeScript. Como ya hemos visto antes, un DTO es básicamente un objeto que define cómo se enviarán los datos por la red. Podemos hacerlo con interfaces o con clases, aunque Nest.js recomienda la segunda opción, ya que dichas clases se mantienen al transpilar el código a JavaScript (los interfaces no), y esto proporciona alguna funcionalidad añadida, como poder trabajar con *pipes*.

Lo más habitual es crear una subcarpeta *dto* dentro del controlador que lo vaya a utilizar, con la clase dentro. En nuestro caso, ya hemos creado previamente nuestro DTO en `src/contacto/dto/contacto.dto.ts`, con la información que se enviará del contacto. Recordemos su estructura:

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

Podemos importar este DTO desde el controlador, y utilizarlo en los *handlers* de tipo POST que lo requieran:

```
import { Controller, Get, Post, Body, Param }
    from '@nestjs/common';
import { ContactoDto } from './dto/contacto-dto/contacto-dto';

@Controller('contacto')
export class ContactoController {

    ...

    // POST /contacto
    @Post()
    crear(@Body() crearContactoDto: ContactoDto) {
        // Aquí podemos utilizar
        // crearContactoDto.nombre, o edad, etc.
    }
}
```

@Put y @Delete

Del mismo modo se definen los *handlers* para las operaciones PUT y DELETE. En el caso de PUT, también será necesario utilizar un DTO para recoger los datos de la petición. Puede ser el mismo que para la inserción, si se van a enviar los mismos datos.

```
// PUT /contacto/:id
@Put(':id')
actualizar(@Param('id') id: string,
           @Body() actualizarContactoDto: ContactoDto) {
    ...
}

// DELETE /contacto/:id
@Delete(':id')
borrar(@Param('id') id: string) {
    ...
}
```

3.4.3.2. Códigos de estado, cabeceras de respuesta y redirecciones

Por defecto, los manejadores o *handlers* en Express devuelven automáticamente un código 200 junto con la respuesta, salvo en el caso de peticiones POST, donde se devuelve un estado 201. Si queremos devolver otro estado diferente, podemos utilizar el decorador `@HttpCode` en el encabezado del *handler*, indicando el código a devolver:

```
@Post
@HttpCode(204)
crear() {
    ...
}
```

Además, también podemos emplear el decorador `@Header` para enviar cabeceras de respuesta (una vez por cada cabecera), indicando en cada caso el nombre de la cabecera y su valor asociado.

```
@Post
@HttpCode(204)
@Header('Cache-Control', 'none')
crear() {
    ...
}
```

Finalmente, podemos utilizar el decorador `@Redirect` para hacer que un *handler* redirija a otra URL.

```
@Get('prueba')
@Redirect('http://....', 302)
prueba() {
    ...
}
```

Por defecto, la redirección genera un código 301, pero podemos cambiarlo en el segundo parámetro. De hecho, también podemos hacer que tanto la ruta a la que redirigir como el código de estado cambien, devolviendo desde el *handler* un objeto con los campos `url` y `statusCode` establecidos con los valores indicados (ambos campos son opcionales):

```
@Get('prueba')
@Redirect('http://....', 302)
prueba() {
    if (...)
        return { statusCode: 300 };
}
```

> **NOTA**: deberemos importar en la instrucción `import` correspondiente estos decoradores desde `@nestjs/common`.

> **Ejercicios propuestos**:
>
> **3.** Sobre el proyecto que venimos desarrollando de tareas, vamos a rellenar el contenido del controlador de tareas creado en ejercicios anteriores, y que debería estar ubicado en `src/tarea/tarea.controller.ts`. El controlador responderá al prefijo `/tarea`. Define en el controlador los métodos (vacíos, de momento) para:
>
> - Listar todas las tareas (GET)
> - Buscar una tarea por su *id* (GET)
> - Insertar una tarea (POST)
> - Borrar una tarea (DELETE)
> - Modificar una tarea (PUT)
>
> Haz que cada método, de momento, simplemente devuelva una cadena de texto con la información del método al que se ha accedido. Por ejemplo, para el listado de tareas, podemos devolver "Listado de tareas".
>
> Utiliza el DTO `TareaDto` creado en ejercicios anteriores para las operaciones de POST y PUT.
>
> Crea una colección en Postman llamada `TareasNest` y define una petición de prueba para cada uno de los servicios implementados.

3.4.4. Conexión con una base de datos MongoDB

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 i @nestjs/mongoose mongoose
```

3.4.4.1. Conectando a la base de datos

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';
import { MongooseModule } from '@nestjs/mongoose';

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

3.4.4.2. Definiendo esquemas y modelos

Igual que hemos hecho en temas previos, necesitamos definir los esquemas y modelos de nuestra base de datos MongoDB. En este caso, es conveniente ubicar los esquemas junto al módulo que los vaya a utilizar. En nuestro caso, definiríamos el esquema de los contactos en una subcarpeta `schemas` dentro de la carpeta `src/contacto`. Crearíamos un archivo `src/contacto/schema/contacto.schema.ts` con la definición del esquema, de forma similar a como hemos hecho en temas previos:

```
import * as mongoose from 'mongoose';

export const ContactoSchema = new mongoose.Schema({
  nombre: {
      type: String,
      required: true,
      minlength: 3
  },
  edad: {
      type: Number,
      required: true,
      min: 0,
      max: 120
  },
  telefono: {
      type: String,
      required: true,
      minlength: 9
  }
});
```

El siguiente paso será incluir el esquema en el módulo asociado (archivo `src/contacto/contacto.module.ts`):

```
...
import { MongooseModule } from '@nestjs/mongoose';
import { ContactoSchema } from './schemas/contacto.schema';

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

Asociamos el esquema con un nombre de modelo (`Contacto`, en este caso), mediante el método `forFeature`.

Ahora que ya tenemos definido el esquema, lo importamos en el servicio asociado (el de contactos, en este caso), utilizando el mismo nombre que usamos en el módulo principal (`Contacto`). Deberemos importar el decorador `@InjectModel` para poder inyectar el modelo en el servicio:

```
...
import { Model } from 'mongoose';
import { InjectModel } from '@nestjs/mongoose';
import { Contacto } from './interfaces/contacto.interface';

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

> **NOTA**: el modelo se asocia a la interfaz `Contacto` que hemos creado en pasos previos, a través del genérico `Model<Contacto>`, de manera que se transforman los objetos del modelo para acoplarlos al interfaz.

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.

Por ejemplo, así podemos definir un método (asíncrono, en este caso) para obtener un listado de todos los contactos:

```
async listar(): Promise<Contacto[]> {
    return await this.contactoModel.find().exec();
}
```

Y este otro método permite insertar un contacto a partir de su DTO:

```
async insertar(crearContactoDto: ContactoDto): Promise<Contacto> {
    const nuevoContacto = new this.contactoModel(crearContactoDto);
    return await nuevoContacto.save();
}
```

Lo que nos queda es utilizar estos métodos del servicio desde los correspondientes *handlers* del controlador. Los métodos del controlador también podemos definirlos como asíncronos si queremos:

```
...

@Controller('contacto')
export class ContactoController {

    constructor(private readonly contactoService: ContactoService) {}

    // GET /contacto
    @Get()
    async listar() {
        return this.contactoService.listar();
    }

    // GET /contacto/buscar/:id
    @Get('buscar/:id')
    async buscarPorId(@Param('id') id: string) {
        try {
            let resultado = await this.contactoService.buscarPorId(id);
            if (resultado) return {resultado: resultado};
            throw new Error();
        } catch(Error) {
            return { error: 'Error buscando al contacto' };
        }
    }

    // POST /contacto
    @Post()
    async crear(@Body() crearContactoDto: ContactoDto) {
        return this.contactoService.insertar(crearContactoDto);
    }

    // PUT /contacto/:id
    @Put(':id')
    actualizar(@Param('id') id: string,
        @Body() actualizarContactoDto: ContactoDto) {
        return this.contactoService.actualizar(id, actualizarContactoDto);
    }

    // DELETE /contacto/:id
    @Delete(':id')
    borrar(@Param('id') id: string) {
        return this.contactoService.borrar(id);
    }
    ...
}
```

Notar que en el constructor del controlador inyectamos el servicio, y luego podemos hacer uso de él, y de sus métodos, en cualquier *handler*.

Ejercicios propuestos:

4. Sobre el proyecto anterior de tareas, vamos a añadir las siguientes funcionalidades:

- Instalaremos los módulos `@nestjs/mongoose` y `mongoose` para conectar desde el módulo principal con una base de datos llamada `tareas-nest`.
- Definiremos un esquema llamado `TareaSchema` en el archivo `src/tarea/schemas/tarea.schema.ts`, con el esquema de cada tarea. Incluiremos los tres campos que ya tenemos (nombre, prioridad y 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 tanto al módulo de tareas como al servicio, de forma similar a como hemos hecho con el ejemplo de contactos.
- Implementaremos el servicio de tareas, definiendo métodos para realizar las cinco operaciones básicas sobre la base de datos anterior: listado general, búsqueda por *id*, inserción, borrado y modificación. Define todos los métodos como asíncronos.
- Modificaremos el controlador de tareas para hacer uso de los métodos del servicio en cada *handler*. Haz también asíncronos los métodos del controlador.

Prueba el funcionamiento adecuado de todos los servicios desde la colección de Postman que habrás creado previamente.

3.4.5. Vistas y contenido estático en Nest.js

Desde Nest.js también podemos emplear nuestro motor de plantillas preferido y renderizar las vistas que queramos. En nuestro caso, volveremos a utilizar Nunjucks, como en sesiones anteriores. Lo primero que debemos hacer es instalar la librería. También podemos instalar Bootstrap de paso, si tenemos pensado utilizarlo:

```
npm i nunjucks bootstrap
```

Después, editamos el archivo principal `main.ts`. Debemos importar por un lado la librería `nunjucks`, y por otra, el objeto `NestExpressApplication`, ya que ahora necesitamos especificar que nuestra aplicación Nest.js se va a apoyar en Express para utilizar los métodos asociados para la gestión del motor de plantillas.

En el código del método `bootstrap` de este archivo `main.ts`, crearemos ahora una aplicación que será un subtipo de `NestExpressApplication` y, antes de ponerla en marcha, configuraremos Nunjucks como lo hacíamos en sesiones previas, y emplearemos los métodos `useStaticAssets` y `setViewEngine` de la aplicación `app` para especificar el/las carpeta(s) donde habrá contenido estático, y el motor de plantillas a utilizar, respectivamente. En nuestro caso, puede quedar algo así:

```
import { NestExpressApplication } from '@nestjs/platform-express';
import * as nunjucks from 'nunjucks';

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

  nunjucks.configure('views', {
    autoescape: true,
    express:app
  });

  app.useStaticAssets(__dirname + '/../public', {prefix: 'public'});
  app.useStaticAssets(__dirname + '/../node_modules/bootstrap/dist');
  app.setViewEngine('njk');

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

Las carpetas `public` y `views` deberán ubicarse, de acuerdo al código anterior, en la raíz del proyecto Nest. Después, para renderizar cualquier vista desde un *handler* (método de un controlador), basta con que le pasemos la respuesta como parámetro con el decorador `@Res()`, para poder acceder a su método `render`, como hemos hecho en sesiones previas. Por ejemplo, si quisiéramos obtener el listado de contactos sería:

```
@Get('contactos')
async listar(@Res() res) {
    const resultado = await this.contactoService.listar();
    return res.render('contactos_listado', { contactos: resultado });
}
```

Del mismo modo, podríamos usar el decordador `@Param` para buscar la información de un contacto en concreo y mostrar su ficha, o rellenar los campos de un formulario para poder modificar la información de dicho contacto.

```
@Get('contactos/:id')
async buscarPorId(@Res() res, @Param('id') id: string) {
    const resultado = await this.contactoService.buscarPorId(id);
    return res.render('contactos_ficha', { contacto: resultado });
}
```

Para gestionar una petición post, usaremos el manejador `@Post` acompañado de su ruta (sino se indica ruta se entiende que es la raíz del controlador) y con el decorador `@Body` podemos acceder a la información que se le envía de un formulario.

```
@Post('contactos')
async insertarContacto(@Res() res, @Body() body) {
    try {
        const resultado = await this.contactoService.insertar(body);
        return res.render('contactos_ficha', { contacto: resultado });
    } catch (error) {
        res.render('error', { error: "Error al insertar el contacto" });
    }

}
```

En las sesiones previas vimos que usando la librería method-override podíamos implementar los métodos put y delete. En este caso no podremos usar eso, teniendo que usar para la parte web el servicio Post.

```
// Modificar contacto
@Post('contactos/:id')
async modificarContacto(@Res() res, @Param('id') id: string, @Body() body) {
    try {
        const resultado = await this.contactoService.actualizar(id, body);
        return res.render('contactos_ficha', { contacto: resultado });
    } catch (error) {
        res.render('error', { error: "Error al modificar el contacto" });
    }
}

// Borrar contacto
@Post('contactos/borrar/:id')
async borrarContacto(@Res() res, @Param('id') id: string) {
    const resultado = await this.contactoService.borrar(id);
    this.listar(res);
}
```

Ejercicios propuestos:

5. Instala Nunjucks y Bootstrap en el proyecto `tareas-nest` que hemos venido desarrollando. Configura la aplicación principal `main.ts` para que use Nunjucks como motor de plantillas, y cargue el contenido estático de Bootstrap.

Define una carpeta `views` para las vistas, y una vista `base.njk` de la que heredará el resto, con un bloque para poder definir su título en la cabecera (*title*), y otro bloque para su contenido. Haz que la vista base incorpore los estilos de Bootstrap.

6. Crea un nuevo módulo llamado `web`, con su controlador asociado. Antes de seguir, deberás exportar el servicio `TareaService` en el módulo de tareas para poderlo utilizar en este otro módulo:

```
@Module({
  imports: ...
  controllers: ...
  providers: ...
  exports: [TareaService]
})
```

Después, deberás importar el módulo de tareas entero desde el nuevo módulo web:

```
@Module({
  imports: [TareaModule],
  controllers: [WebController]
})
```

Ahora, define un par de *handlers* en el controlador de web (archivo `src/web/web.controller.ts`) para responder a las rutas `/web/tareas` y `/web/tareas/:id`. El primero deberá renderizar la vista `tareas_listado.njk`, que deberás crear, con un listado con los nombres de las tareas. Al hacer click en cada una de ellas se llamará al segundo *handler*, que renderizará la vista `tareas_ficha.njk`, que también deberás implementar, con la ficha de cada tarea, indicando su nombre, prioridad y fecha.

Finalmente, haz que la ruta raíz redireccione al listado de tareas. **7.** Dentro del módulo `web`, implementa las rutas que nos permitan modificar y borrar una tarea. Para ello, previamente dentro de la vista `tareas_listado.njk` añade un botón a cada tarea que te permita seleccionar la tarea a borrar o modificar. Una vez creado el botón renderiza la vista que te permita modificarla `tareas_editar.njk`.

3.4.6. Autenticación en Nest.js

En este apartado, vamos a ver los dos mecanismos vistos anteriormente para autenticar a los usuarios que acceden a nuestros servicios.

3.4.6.1 Autenticación basada en tokens usando JWT.

En el tema 3.2 vimos cómo usar la autenticación para loguearnos a nuestras aplicaciones. En este punto vamos a ver cómo usar la autenticación en Nestjs en el lado del servidor usando JSON Web Token, mecanismo de autenticación que permita ser aplicado a aplicaciones cliente (no de navegador), como por ejemplo las aplicaciones de escritorio, o móviles.

La autenticación basada en tokens es un método mediante el cual nos aseguramos de que cada petición a un servidor viene acompañada por un token firmado, que contiene los datos necesarios para verificar que el cliente ya ha sido validado previamente.

Para poder generar un token utilizaremos la librería *passport-jwt*:

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

El paquete @nestjs/jwt que hemos instalado es el que tiene las utilidades para manipular JWT. El paquete passport-jwt es el encargado de implementar la estrategia JWT y @types/passport-jwt proporciona las definiciones de tipos de TypeScript.

Con el objetivo de encapsular las operaciones de autenticación, vamos a generar el controlador, un módulo y su servicio:

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

En el fichero `auth.service.ts` añadiremos el método `login`, importando el servicio `JwtService`. Los usuarios los tendremos por ahora en un array predefinido.

```
import { HttpStatus, Injectable, UnauthorizedException } from '@nestjs/common
import { JwtService } from '@nestjs/jwt';
import { LoginDto } from './dto/login-dto';

// Este array de usuarios debería estar en la BBDD con la contraseña CIFRADA
const usuarios = [
  { usuario: 'rosa', password: 'rosa', rol: 'admin' },
  { usuario: 'pepe', password: 'pepe111', rol: 'normal' }
];

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

  async login(user: LoginDto) {
    const usuario = usuarios.find(u => u.usuario === user.usuario && u.passwo
    if (usuario) {
      const payload = { sub: usuario.usuario };
      return {
        access_token: this.jwtService.sign(payload),
      };
    } else {
      throw new UnauthorizedException({
        status: HttpStatus.UNAUTHORIZED,
        error: 'Usuario o password incorrecto',
      });
    }
  }
}
```

Como vemos, estamos usando la librería @nestjs/jwt que nos permite usar el método sign() para generar nuestro token JWT, devolviendolo en la respuesta. Hemos elegido la propiedad sub (subject) donde el estándar JWT define que se guardará aquello que identifique (en la base de datos) de nuestro usuario. El siguiente paso será actualizar AuthModule para importar las nuevas dependencias y configurar JwtModule.

Primero, crearemos un archivo de constantes (constants.ts) dentro de auth, donde almacenaremos la clave secreta para firmar el token digitalmente y que no pueda ser alterado por terceros:

```
export const jwtConstants = {
    secret: 'WEr34fwGter',
};
```

El siguiente paso será abrir auth.module.ts en el directorio auth para configurar `JwtModule`, llamando al método `register()` pasándole las opciones del token:

```
import { Module } from '@nestjs/common';
import { JwtModule } from '@nestjs/jwt';
import { PassportModule } from '@nestjs/passport';
import { AuthController } from './auth.controller';
import { AuthService } from './auth.service';
import { jwtConstants } from './constants';
import { JwtStrategy } from './jwt.strategy';

@Module({
  controllers: [AuthController],
  imports: [
    PassportModule,
    JwtModule.register({
      secret: jwtConstants.secret,
      signOptions: { expiresIn: '30d' } // El token expira en 30 días
    })
  ],
  providers: [AuthService, JwtStrategy]
})
export class AuthModule { }
```

Una vez hecho esto, debemos actualizar la ruta de login para que devuelva el resultado de la llamada al servicio de autenticación:

```
import { Body, Controller, HttpStatus, Post, UnauthorizedException, Validatio
import { AuthService } from './auth.service';
import { LoginDto } from './dto/login-dto';

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

    @Post('login')
    async login(
      @Body(new ValidationPipe({ whitelist: true }))
      userDto: LoginDto
    ) {
        return await this.authService.login(userDto);
    }
}
```

Como se puede observar, se ha utilizado una clase (LoginDto) que representa los datos del login que se van a recibir: *usuario* y *password*. Con `ValidationPipe` hacemos también una validación de los campos que nos llegan. Los decoradores para validar los campos se encuentran en la clase **LoginDto**. Para usar

validaciones debemos tener instalada la librería **class_validator** `npm i class-validator class-transformer`. En nuestro caso, verificamos que los 2 campos tengan valores de tipo string y que no estén vacíos.

```
import {
    IsNotEmpty, IsString
} from 'class-validator';

export class LoginDto {
    @IsNotEmpty()
    @IsString()
    usuario: string;
    @IsNotEmpty()
    @IsString()
    password: string;
}
```

Si algún campo no cumple la validación, Nest devuelve automáticamente una respuesta de error 400 (Bad Request) indicando los campos que no han sido validados.

Validación de rutas

Para proteger las rutas que requieren credenciales de login para acceder (token), debemos crear un guardian derivado de la clase **AuthGuard** que utilice 'jwt', es decir, la clase JWTStrategy que hemos creado antes para validar el token y dejarnos acceder al servicio en caso de ser válido (o devolver un error 401).

`nest g guard auth/jwt-auth --flat` (la opción --flat es para no crear una carpeta)

```
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

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

Una vez creado esto, debemos indicar qué métodos de los controladores necesitarán token para acceder a ellos. Esto se hace con el decorador **@UseGuards(JwtAuthGuard)**:

```
import { Controller, Get, Param, Put, Body, Delete, Post, UseGuards } from '@
import { JwtAuthGuard } from 'src/auth/jwt-auth.guard';
import { ContactoDto } from './dto/contacto-dto/contacto-dto';
import { ContactoService } from './contacto.service';

@Controller('contacto')
export class ContactoController {
    ...
    // POST /contacto
    @Post()
    @UseGuards(JwtAuthGuard)
    async crear(@Body() crearContactoDto: ContactoDto) {
        return this.contactoService.insertar(crearContactoDto);
    }
    ...
}
```

Lo último que nos queda, es crear un strategy para poder exigir en aquellas rutas que queramos proteger un token. Para ello, nos crearemos el fichero `jwt.strategy.ts` dentro de `auth` con el siguiente contenido:

```
import { Injectable } from "@nestjs/common";
import { PassportStrategy } from "@nestjs/passport";
import { Strategy, ExtractJwt } from "passport-jwt";
import { jwtConstants } from "./constants";

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: jwtConstants.secret,
    });
  }

  async validate(payload: any) {
    return { usuario: payload.sub };
  }
}
```

A partir de este momento solo se podrá acceder a estas rutas incluyendo el token generado en el login en la cabecera **Authorization** de la petición, con el prefijo **Bearer**: `Authorization: Bearer TOKEN_JWT`.

3.4.6.2 Autenticación basada en sesiones

Como sabemos, la autenticación basada en sesiones permite autenticar usuarios en aplicaciones web basadas en navegadores, y "recordar" el usuario que se validó en sus sucesivas visitas. Para ello, utilizan las **sesiones**, que comprenden un conjunto de interacciones de un cliente con un servidor en un determinado período. Cuando abrimos un navegador y accedemos a una web, automáticamente se inicia la sesión en dicha web, y mientras no cerremos el navegador o la sesión manualmente, la aplicación recuerda (o puede recordar, si quiere) que ya hemos accedido, y los pasos que hemos ido dando en la actual sesión.

En este apartado seguiremos utilizando el mismo ejemplo de contactos, sobre el que probaremos dicha autenticación.

3.4.6.2.1. Definición de sesiones en Nest.js

Para poder trabajar con sesiones en Nest, instalaremos el módulo *express-session* que vimos anteriormente. Es un *middleware* que permite, en cada petición que requiera una comprobación, determinar si el usuario ya se ha validado y con qué credenciales, antes de dejarle acceder a lo que busca o no.

Así que lo primero que haremos será instalar el módulo:

```
npm install express-session
```

Después, lo incorporamos a nuestro servidor Express junto con el resto de módulos:

```
const session = require('express-session');
...
```

A continuación, configuramos la sesión dentro de la aplicación Express:

```
let app = express();
...
app.use(session({
    secret: '1234', // Clave para cifrar la sesión
    resave: true, // Refresca la sesión en cada nuevo acceso
    saveUninitialized: false, // Guarda las sesiones aun sin haberse completa
    expires: new Date(Date.now() + (30 * 60 * 1000)) // La sesión expirará en
}));
```

3.4.6.2.2. Validación

En todo proceso de autenticación debe haber una validación previa, donde el usuario envíe sus credenciales y se cotejen con las existentes en la base de datos, antes de dejarle acceder. Por simplicidad, vamos a cargar

los usuarios en un array, con su nombre de usuario y su password, pero sabéis que los usuarios deberían estar en una tabla de nuestra base de datos con la contraseña cifrada:

```
const usuarios = [
    { usuario: 'nacho', password: '12345' },
    { usuario: 'pepe', password: 'pepe111' }
];
```

Ahora tendríamos que definir el servicio de login:

```
@Post('login')
async login(@Res() res, @Req() req, @Body() body) {
    let usu = body.usuario;
    let pass = body.password;
    let existe = aUsuarios.filter(usuario => usuario.usuario == usu && usuari

    if (existe.length > 0) {
        req.session.usuario = existe[0].usuario;
        this.listar(res);
    } else {
        res.render('iniciarSesion', { error: "Error usuario o contraseña inco
    }
}
```

3.4.6.2.3. Autenticación

Una vez validado el usuario, debemos comprobar en cada una de las rutas que queramos proteger, si el usuario ha sido logueado o no. Para ello, a cada una de esos recursos deberemos comprobar si existe la sesión haciendo uso del decorador `@Session`. Veamos un ejemplo donde sólo aquellos usuarios que han sido logueados podrán crear un contacto:

```
@Get('contactos/nuevo')
async crearContacto(@Res() res, @Session() session) {
    if(!session.usuario) return res.render('login', {error: "El usuario debe e
    return res.render('contactos_nuevo');
}
```

Para poder acceder a la sesión desde las vistas, debemos definir un middleware que asocie la sesión con los recursos de la vista:

```
app.use((req, res, next) => {
    res.locals.session = req.session;
    next();
});
```

Después, podemos acceder a esta sesión desde las vistas, a través de la variable `session` que hemos definido en la respuesta (`res.locals`). Por ejemplo, así podríamos ver si un usuario está ya logueado, para mostrar o no el botón de "Login" en el menú:

```
{% if (session and session.usuario) %}
    <a class="btn btn-dark" href="/web/logout">Logout</a>
{% else %}
    <a class="btn btn-dark" href="/web/login">Login</a>
{% endif %}
```

Y con esto, ya sólo nos quedaría el apartado de cerrar sesión, que consistirá en destruir la sesión de dicho usuario, redirigiendo a posteriori a otro recurso.

```
@Get('logout')
async cerrarSession(@Res() res, @Req() req) {
    req.session.destroy();
    this.listar(res);
}
```

Ejercicios propuestos:

1. A partir del proyecto creado de "**tareas-nest**", vamos a añadir ahora autenticación basada en sesiones. Instala el *middleware express-session* en el proyecto, y configúralo como en el ejemplo visto antes. Define a mano en el servidor principal un array con nombres y passwords de usuarios autorizados, y protege las rutas que permitan hacer cualquier modificación sobre el catálogo de tareas. En concreto, sólo los usuarios validados podrán:

- Ver el formulario de inserción de tareas e insertar tareas (enviar el formulario)
- Borrar tareas
- Ver el formulario de edición de tareas y editar tareas (enviar el formulario)

Añade para ello una vista `login.njk` al conjunto de vistas de la aplicación. Puedes emplear el mismo formulario de login que en el ejemplo, y también añade las dos rutas para mostrar el formulario y para recoger los datos y validar el usuario. En caso de validación exitosa, se renderizará la vista del listado de tareas. En caso contrario, el formulario de login con un mensaje de error, como en el ejemplo proporcionado.

> Finalmente, añade también una función de *logout* al menú de la aplicación, que sólo será visible si el usuario ya está validado, y que permitirá destruir su sesión y redirigir al listado de tareas.