Desarrollo de aplicaciones con Nest.js (I) Primeros pasos con Nest

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. A diferencia de *Express*, *Nest* es un framework *opinionated*, es decir, define una forma concreta de estructurar el proyecto, y nombrar y ubicar los diferentes archivos que lo componen.

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.

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
```

Podemos comprobar la versión instalada con el comando

```
nest --version
```

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     : Windows 10
NodeJS Version : v18.17.1
NPM Version    : 9.6.7

[Nest CLI]
Nest CLI Version : 10.2.1

[Nest Platform Information]
platform-express version : 10.3.0
schematics version       : 10.0.3
testing version          : 10.3.0
common version           : 10.3.0
core version             : 10.3.0
cli version              : 10.2.1
```

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();
```

Fuera de la carpeta *src* existen algunos archivos de configuración del proyecto:

- Ficheros que gestionan la correcta estructura y codificación del proyecto de acuerdo a los estándares (*eslintrc*, *prettierrc*)
- Ficheros que gestionan las dependencias del proyecto, los scripts de gestión o parámetros de configuración (*package.json*, *nest-cli.json*, *tsconfig.json*, *tsconfig.build.json*)
- Otros ficheros, como *.gitignore* para la conexión con repositorios Git

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
```

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

El siguiente comando:

```
npm run start:dev
```

Pone en marcha el servidor y lo deja observando futuros cambios en la aplicación. Ante cualquier cambio, se recompilará y volverá a poner en marcha.

El resultado de la compilación se almacena en la carpeta `dist` del proyecto, que se creará la primera vez que lo pongamos en marcha. Dependiendo de si compilamos para desarrollo (*dev*) o para producción, el contenido de la carpeta será diferente. Normalmente las dependencias necesarias para producción son menores, y el tamaño de esta carpeta es más reducido.

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.

> **Ejercicio 1:**

> Instala Nest si todavía no lo has hecho, y crea un proyecto llamado `tareas-nest` con el siguiente comando: `nest new tareas-nest`. Prueba a ponerlo en marcha y comprobar que todo funciona correctamente.

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. Ya hemos visto antes que el fichero principal `main.ts` se encarga de crear una instancia de este módulo y dejarla escuchando por un puerto específico, así que toda la aplicación se canaliza de ese modo (*app.module* coordina al resto de módulos, y *main* inicializa *app.module*).

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.

Asociados a cada uno de los módulos de nuestra aplicación suele haber una serie de elementos adicionales que le ayuden a dividir el trabajo. Así, junto a 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, implementando la lógica de negocio. 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. Esto hace que los servicios sean inyectables en otros componentes que los necesiten para acceder a los datos.

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.

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
```

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.

2.1.1. Eliminar componentes

Si hemos creado alguno de estos componentes por error y queremos eliminarlo, el proceso es manual:

- Eliminar los archivos correspondientes (o la carpeta completa que los contenga)
- Actualizar el fichero principal `app.module.ts` eliminando cualquier referencia a los archivos suprimidos
- Recompilar y poner en marcha el proyecto para verificar que no queda nada pendiente de borrar

2.1.2. Generación automática del CRUD

Nest pone a nuestra disposición un comando a través de la CLI para generar de forma rápida todo el esqueleto CRUD de nuestra API REST (es decir, el módulo, controlador, servicio y esqueleto de los métodos para listar, insertar, borrar y modificar):

```
nest g res contacto
```

Al ejecutarlo nos permitirá elegir el tipo de recurso a crear, ya que, además de para API REST (que es para lo que lo usaríamos aquí) se pueden crear esquemas GraphQL, o WebSockets, entre otras cosas. Aplicado a una API REST, este comando (bien con la abreviatura `res` o con la palabra completa `resource`) nos creará:

- El módulo, controlador y servicio de contacto, todos ellos enlazados e incorporado el módulo en `app.module.ts`
- Un DTO específico para creación de contactos, y otro para actualización (por si queremos añadir campos o validaciones diferentes). Veremos más adelante qué es esto de los *DTO*.
- Una entidad (*entity*) asociada al contacto. Nos permitirá encapsular la información que extraigamos de este módulo desde la base de datos, proporcionando mecanismos para asociarse a una tabla o colección en concreto de la misma.

Completaremos estos dos últimos puntos más adelante. Pero, como vemos, es un comando útil para generar de golpe toda la estructura básica de un módulo determinado.

> **Ejercicio 2:**
>
> En el proyecto `tareas-nest` creado anteriormente, crea desde la carpeta principal un conjunto de recursos bajo el nombre `tarea`, con el comando `nest g res tarea`. Se habrá creado una carpeta `src/tarea` con el contenido dentro.

Tras completar el ejercicio anterior, observa el contenido del controlador de tarea `src/tarea/tarea.controller.ts`:

```
import { Controller, Get, Post, Body, Patch, Param, Delete } from '@nestjs/common';
import { TareaService } from './tarea.service';
import { CreateTareaDto } from './dto/create-tarea.dto';
import { UpdateTareaDto } from './dto/update-tarea.dto';

@Controller('tarea')
export class TareaController {
  constructor(private readonly tareaService: TareaService) {}

  @Post()
  create(@Body() createTareaDto: CreateTareaDto) {
    return this.tareaService.create(createTareaDto);
  }

  @Get()
  findAll() {
    return this.tareaService.findAll();
  }

  @Get(':id')
  findOne(@Param('id') id: string) {
    return this.tareaService.findOne(+id);
  }

  ...
}
```

Como puedes comprobar, se inyecta en el constructor el servicio asociado (`TareaService`) para poderlo usar en los diferentes métodos. Cada método viene precedido por un decorador *Post*, *Get*, etc, que indica a qué comando va a responder, y se hace uso de los métodos implementados en `tareaService` para efectivamente hacer la operación. Es decir, el servicio es quien se va a encargar de acceder a los datos, y el controlador se limita a usar esas funcionalidades.

Por su parte, el servicio `src/tarea/tarea.service.ts` de momento se ha creado con este contenido:

```
import { Injectable } from '@nestjs/common';
import { CreateTareaDto } from './dto/create-tarea.dto';
import { UpdateTareaDto } from './dto/update-tarea.dto';

@Injectable()
export class TareaService {
  create(createTareaDto: CreateTareaDto) {
    return 'This action adds a new tarea';
  }

  findAll() {
    return `This action returns all tarea`;
  }

  findOne(id: number) {
    return `This action returns a #${id} tarea`;
  }
  ...
}
```

Es un elemento inyectable (`@Injectable`), es decir, se puede inyectar como dependencia en otros elementos (como se ha hecho con su controlador asociado), y tiene ya preparados los métodos para crear, buscar, modificar... las tareas correspondientes. En próximos documentos veremos cómo conectar con la base de datos y completar el código de estos métodos.

3. Más sobre los controladores

Los controladores en Nest.js se encargan de gestionar las peticiones y respuestas a los clientes. Como hemos visto, son clases con el decorador `@Controller` , que añade metainformación para crear un mapa de enrutado. Gracias a este mapa, Nest sabe cómo gestionar las peticiones para 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` , 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`).

3.1. 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, e 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.

Por ejemplo, así han quedado definidos los *handlers* de tipo `@Get` para obtener un listado general, y una tarea a partir de su *id*, respectivamente. Se debe importar el decorador junto con el resto de elementos necesarios del paquete `@nestjs/common`.

```
@Get()
findAll() {
    return this.tareaService.findAll();
}

@Get(':id')
findOne(@Param('id') id: string) {
    return this.tareaService.findOne(+id);
}
```

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()
findAll() {
    return {
        ok: true;
        resultado: ... // Datos buscados
    };
}
```