Desenvolupament d'aplicacions amb Nest.js (I) Primers passos amb Nest


Nest.js és un framework de desenvolupament web en el servidor que, secundant-se en el framework **Express**, permet construir aplicacions robustes i escalables utilitzant una terminologia molt similar a la que s'empra en el framework de client *Angular*. Igual que Angular, utilitza llenguatge TypeScript per a definir el codi, encara que també és compatible amb JavaScript. A diferència d'*Express*, *Nest* és un framework *opinionated*, és a dir, definix una forma concreta d'estructurar el projecte, i nomenar i situar els diferents arxius que el componen.

Nest.js proporciona la majoria de característiques que qualsevol framework de desenvolupament en el servidor proporciona, com ara mecanismes d'autenticació, ús d'ORM per a accés a dades, desenvolupament de servicis REST, enrutado, etc.

1. Instal·lació i creació de projectes

Nest.js s'instal·la com un mòdul global al sistema a través del gestor de paquets `npm`, amb el següent comando:

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

Podem comprovar la versió instal·lada amb la comanda:

```
nest --version
```

Una vegada instal·lat, per a crear un projecte utilitzem el comando `nest`, amb l'opció `new`, seguida del nom del projecte.

```
nest new nom_projecte
```

> **NOTA**: en la creació del projecte, pot ser que l'assistent pregunte quin gestor de paquets utilitzarem. El normal és seleccionar `npm`.

Això crearà una carpeta amb el nom del projecte en la nostra ubicació actual, i emmagatzemarà dins tota l'estructura bàsica d'arxius i carpetes dels projectes Nest. Podem consultar la informació del projecte generat amb el comando `i` (o `info`):

```
nest i
```

Obtindrem una eixida similar a esta (variant els números de versió):

```
[System Information]
US 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 d'un projecte Nest.js

L'estructura de carpetes i arxius creada pel comando `nest` té una sèrie d'elements clau que convé ressaltar. La major part del nostre codi font se situarà en la carpeta `src`. Entre altres coses, podem trobar:

- El mòdul principal `app.module.ts`, ja definit
- Un controlador d'exemple, anomenat `app.controller.ts`, amb una ruta definida cap a l'arrel de l'aplicació.
- L'arxiu `main.ts`, que definix la inicialització de l'aplicació. Crea una instància del mòdul principal `AppModule`, i es queda escoltant per un port determinat (que es pot modificar en este mateix arxiu). Definix per a això una funció asíncrona, que després es llança per a posar en marxa tot:

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

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

Fora de la carpeta *src* existixen alguns arxius de configuració del projecte:

- Fitxers que gestionen la correcta estructura i codificació del projecte d'acord amb els estàndards (*eslintrc*, *prettierrc*)
- Fitxers que gestionen les dependències del projecte, els scripts de gestió o paràmetres de configuració (*package.json*, *nest-cli.json*, *tsconfig.json*, *tsconfig.build.json*)
- Altres fitxers, com *.gitignore* per a la connexió amb repositoris Git

1.2. Posar en marxa el projecte

L'assistent de creació del projecte l'haurà deixat tot preparat, amb les dependències ja instal·lades, i fins i tot l'arxiu `package.json` ja generat, per a poder posar en marxa el projecte. Només hem d'executar la següent comanda:

```
npm run start
```

Si intentem accedir a `http://localhost:3000` veurem un missatge de benvinguda proporcionat pel mòdul principal ("Hello World!").

La següent comanda:

```
npm run start:dev
```

Posa en marxa el servidor i el deixa observant futurs canvis en l'aplicació. Davant qualsevol canvi, es recompilarà i tornarà a posar en marxa.

El resultat de la compilació s'emmagatzema en la carpeta `dist` del projecte, que es crearà la primera vegada que el posem en marxa. Depenent de si compilem per a desenvolupament (*dev*) o per a producció, el contingut de la carpeta serà diferent. Normalment les dependències necessàries per a producció són menors, i la grandària d'esta carpeta és més reduït.

1.3. Connexió amb Express

Com hem comentat, Nest.js utilitza internament el framework Express per a treballar sobre ell. Això fa que no tinguem per què accedir directament a uns certs elements que tenim disponibles en dita framework, com la petició (`req`), resposta (`res`), paràmetres de la URL (`req.params`), cos de la petició (`req.body`), etc. En el seu lloc, Nest.js proporciona una sèrie de decoradors que anirem veient més endavant, i que internament es comuniquen amb estes propietats d'Express. Per exemple, el decorador `@Param` l'emprarem per a accedir a paràmetres de la URL, i el decorador `@Body` per a accedir al cos de la petició.

> **Exercici 1:**
>
> Instal·la Nest si encara no ho has fet, i crea un projecte anomenat `tasques-nest` amb la següent comanda: `nest new tasques-nest`. Posa-ho en marxa i comprova que tot funciona correctament.

2. Estructurant l'aplicació: mòduls, controladors i servicis

Quan creem una aplicació Nest, inicialment ja tenim una mica de codi generat en la carpeta `src`. En concret, disposem del mòdul principal de l'aplicació, `app.module.ts`, que s'encarregarà de coordinar a la resta de mòduls que definim. Ja hem vist abans que el fitxer principal `main.ts` s'encarrega de crear una instància d'este mòdul i deixar-la escoltant per un port específic, així que tota l'aplicació es canalitza d'eixe mode (*app.module* coordina a la resta de mòduls, i *main* inicialitza *app.module*).

Cada mòdul ha d'encarregar-se d'encapsular i gestionar un conjunt de característiques sobre un concepte de l'aplicació. Per exemple, en una aplicació d'una botiga en línia podem tindre un mòdul que gestione els clients (llistats, altes, baixes), un altre per a l'estoc de productes, un altre per a les comandes, etc.

Associats a cadascun dels mòduls de la nostra aplicació sol haver-hi una sèrie d'elements addicionals que l'ajuden a dividir el treball. Així, al costat de cadascun dels mòduls podem tindre:

- **Controladors** (*controllers*), que s'encarregaran d'atendre les peticions relacionades amb este mòdul. Per exemple, en un mòdul de clients, tindrem un controlador que s'encarregarà d'atendre peticions de llistats de clients, altes, baixes, etc.
- **Servicis** (*services*), que s'encarregaran de gestionar l'accés a les dades per a un determinat mòdul, implementant la lògica de negoci. Així, tornant a l'exemple dels clients, podrem tindre un servici que s'encarregue de realitzar efectivament les cerques, insercions, esborraments, etc, en la col·lecció de dades corresponent. En realitat, els servicis són un tipus especial de **proveïdors** (*providers*), elements que utilitza Nest per a fer tasques específiques i relativament complexes, descarregant així de treball als controladors, que es limiten a atendre peticions. Això fa que els servicis siguen injectables en altres components que els necessiten per a accedir a les dades.

De fet, el nostre mòdul principal `app.module.ts` compta amb un servici associat `app.service.ts` i un controlador, `app.controller.ts`, ja creats. Inicialment no fan gran cosa, ja que el controlador només disposa d'una ruta per a carregar una pàgina de benvinguda amb una salutació simple, i el servici s'encarrega de proporcionar eixe missatge de salutació. Però és un punt de partida per a comprendre com s'estructura el repartiment de tasques en aplicacions Nest.

2.1. Definint mòduls, controladors i servicis

Tornem al tema dels mòduls. Un mòdul bàsicament és una classe TypeScript anotada amb el decorador `@Module`, que proporciona una sèrie de metadades per a construir l'estructura de l'aplicació. Com ja hem vist, tota aplicació Nest té almenys un mòdul arrel o *root*, l'arxiu `app.module.ts` explicat anteriorment, que servix de punt d'entrada a l'aplicació, de manera similar a com 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` presa un objecte com a paràmetre, on es definixen els controladors, proveïdors de servicis i altres elements que anirem veient més endavant.

Per a **crear un nou mòdul** per al nostre projecte, escrivim la següent comanda des de l'arrel del projecte:

```
nest g module nom_modul
```

Es crearà una carpeta `nom_modul` dins de la carpeta `src`, i s'afegirà la corresponent referència en la secció `imports` del mòdul principal `AppModule`. Per exemple, si creem un mòdul anomenat `contacte`, es crearà la carpeta `contacte`, i la secció `imports` del mòdul principal quedarà així (notar que a la classe que es genera se li afig el sufix "Module" automàticament):

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

Així, el mòdul principal ja incorpora al mòdul `contacte`, i tot el que este continga al seu torn. En la carpeta corresponent (`contacte`, seguint l'exemple anterior) es generarà un arxiu TypeScript (`contacte.module.ts`, en el nostre exemple) amb la nova classe del mòdul generada.

De la mateixa manera, podem generar controladors i servicis, amb estos comandos:

```
nest g controller nom_controlador
nest g service nom_servici
```

Si seguim amb el cas anterior, podem crear un controlador anomenat `contacte` i un servici amb el mateix nom. Això generarà respectivament els arxius `src/contacte/contacte.controller.ts` i `src/contacte/contacte.service.ts` en el nostre projecte.

En seguir estos passos, el propi mòdul `contacte.module.ts` tindrà ja registrats el seu controlador i servici, amb el que està ja tot connectat per a poder començar a treballar:

```
import { Module } from '@nestjs/common';
import { ContacteController } from './contacte.controller';
import { ContacteService } from './contacte.service';

@Module({
    controllers: [ContacteController],
    providers: [ContacteService]
})
export class ContacteModule {}
```

Observem la jerarquia de dependències que s'està creant: el mòdul principal incorpora en el seu bloc `imports` al mòdul `contacte`, i este al seu torn conté en el seu interior les dependències amb el controlador i el servici propis.

És important definir els elements en este ordre (primer el mòdul, i després els seus controladors i servicis), ja que en cas contrari hauríem de definir estes dependències a mà en el codi.

2.1.1. Eliminar components

Si hem creat algun d'estos components per error i volem eliminar-ho, el procés és manual:

- Eliminar els arxius corresponents (o la carpeta completa que els continga)
- Actualitzar el fitxer principal `app.module.ts` eliminant qualsevol referència als arxius suprimits
- Recompilar i posar en marxa el projecte per a verificar que no queda gens pendent d'esborrar

2.1.2. Generació automàtica del CRUD

Nest posa a la nostra disposició un comando a través de la CLI per a generar de manera ràpida tot l'esquelet CRUD de la nostra API REST (és a dir, el mòdul, controlador, servici i esquelet dels mètodes per a llistar, inserir, esborrar i modificar):

```
nest g res contacte
```

En executar-ho ens permetrà triar el tipus de recurs a crear, ja que, a més de per a API REST (que és per al que ho usaríem ací) es poden crear esquemes GraphQL, o WebSockets, entre altres coses. Aplicat a una API REST, este comando (bé amb l'abreviatura `res` o amb la paraula completa `resource`) ens crearà:

- El mòdul, controlador i servici de contacte, tots ells enllaçats i incorporat el mòdul en `app.module.ts`
- Un DTO específic per a creació de contactes, i un altre per a actualització (per si volem afegir camps o validacions diferents). Veurem més endavant què és això dels *DTO*.

- Una entitat (*entity*) associada al contacte. Ens permetrà encapsular la informació que extraguem d'este mòdul des de la base de dades, proporcionant mecanismes per a associar-se a una taula o col·lecció en concret d'esta.

Completarem estos dos últims punts més endavant. Però, com veiem, és un comando útil per a generar de colp tota l'estructura bàsica d'un mòdul determinat.

> **Exercici 2:**
>
> Sobre el projecte `tasques-nest` creat abans, crea des de la carpeta principal un conjunt de recursos amb el name `tasca`, amb la comanda `nest g res tasca`. S'haurà creat una carpeta `src/tasca` en el projecte.

Després de completar l'exercici anterior, observa el contingut del controlador de tasca `src/tasca/tasca.controller.ts`:

```
import { Controller, Get, Post, Body, Patch, Param, Delete } from '@nestjs/common';
import { TascaService } from './tasca.service';
import { CreateTascaDto } from './dto/create-tasca.dto';
import { UpdateTascaDto } from './dto/update-tasca.dto';

@Controller('tasca')
export class TascaController {
    constructor(private readonly tascaService: TascaService) {}

    @Post()
    create(@Body() createTascaDto: CreateTascaDto) {
        return this.tascaService.create(createTascaDto);
    }

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

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

    ...
}
```

Com pots comprovar, s'injecta en el constructor el servici associat (`TascaService`) per a poder-ho usar en els diferents mètodes. Cada mètode ve precedit per un decorador *Post*, *Get*, etc, que indica a quina comanda respondrà, i es fa ús dels mètodes implementats en `tascaService` per a efectivament fer l'operació. És a dir, el servici és qui s'encarregarà d'accedir a les dades, i el controlador es limita a usar eixes funcionalitats.

Per part seua, el servici `src/tasca/tasca.service.ts` de moment s'ha creat amb este contingut:

```
import { Injectable } from '@nestjs/common';
import { CreateTascaDto } from './dto/create-tasca.dto';
import { UpdateTascaDto } from './dto/update-tasca.dto';

@Injectable()
export class TascaService {

    create(createTascaDto: CreateTascaDto) {
        return 'This action adds a new tasca';
    }

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

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

És un element injectable (`@Injectable`), és a dir, es pot injectar com a dependència en altres elements (com s'ha fet amb el seu controlador associat), i té ja preparats els mètodes per a crear, buscar, modificar... les tasques corresponents. En pròxims documents veurem com connectar amb la base de dades i completar el codi d'estos mètodes.

3. Més sobre els controladors

Els controladors en Nest.js s'encarreguen de gestionar les peticions i respostes als clients. Com hem vist, són classes amb el decorador `@Controller` , que afig metainformació per a crear un mapa d'enrutament. Gràcies a aquest mapa, Nest sap com gestionar les peticions perquè arriben al controlador adequat.

Ja hem vist com crear controladors en Nest, i que queden associats a un mòdul prèviament creat. Suposant el controlador de `contacte` , la seua estructura bàsica en crear-se és la següent:

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

@Controller('contacte')
export class ContacteController {
 ...
}
```

El paràmetre que té el controlador és el prefix en la URL per a accedir a ell. Així, qualsevol ruta que vaja a ser arreplegada per este controlador tindrà l'estructura `http://localhost:3000/contacte` (suposant la ruta i port per defecte definit en `main.ts`).

3.1. Definir *handlers* per a arreplegar les peticions

Per a gestionar estes peticions, s'han de definir uns mètodes en el controlador, anomenats *handlers*. Estos mètodes utilitzen decoradors que són verbs HTTP, i indiquen a quin tipus de mètode respondre (`@Get`, `@Post`, `@Put`, `@Delete`...). Entre parèntesi, podem indicar una ruta addicional al prefix del controlador. Si no especifiquem cap, s'entén que responen a la ruta arrel del controlador.

Per exemple, així han quedat definits els *handlers* de tipus `@Get` per a obtindre un llistat general, i una tasca a partir de la seua *id*, respectivament. S'ha d'importar el decorador juntament amb la resta d'elements necessaris del paquet `@nestjs/common`.

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

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

En el cas del segon *handler*, utilitzem un decorador `@Param` per a accedir al paràmetre que vulguem de la URL (en este cas, el paràmetre `id`), i associar-lo a un nom de variable, que serà el que utilitzem en el codi del *handler*. En este cas, la variable es diu igual que el paràmetre, però podria tindre un nom diferent si volguérem.

A l'hora d'emetre una resposta, haurem de retornar (`return`) un resultat. Nest.js serialitza automàticament objectes JavaScript a format JSON, mentres que si enviem un tipus simple (per exemple, un enter, o una cadena de text), ho envia com a text pla. Per tant, normalment no haurem de preocupar-nos per esta tasca. Podem retornar alguna cosa com això:

```
@Get()
findAll() {
    return {
        ok: true;
        resultat: ... // Dades buscades
    };
}
```