Desenvolupament d'aplicacions amb Nest.js
Més opcions del framework


En este document veurem algunes opcions addicionals que oferix *Nest.js* en el desenvolupament d'aplicacions.

1. Vistes i contingut estàtic en Nest.js

Des de Nest.js també podem emprar el nostre motor de plantilles preferit i renderitzar les vistes que vulguem. En el nostre cas, tornarem a utilitzar Nunjucks, com en sessions anteriors. El primer que hem de fer és instal·lar la llibreria. També podem instal·lar Bootstrap de pas, si tenim pensat utilitzar-ho:

```
npm install nunjucks bootstrap
```

Després, editem l'arxiu principal `main.ts`. Hem d'importar d'una banda la llibreria `nunjucks`, i per una altra, l'objecte `NestExpressApplication`, ja que ara necessitem especificar que la nostra aplicació Nest.js es recolzarà en Express per a utilitzar els mètodes associats per a la gestió del motor de plantilles.

En el codi del mètode `bootstrap` d'este arxiu `main.ts`, crearem ara una aplicació que serà un subtipus de `NestExpressApplication` i, abans de posar-la en marxa, configurarem Nunjucks com ho féiem en sessions prèvies, i emprarem els mètodes `useStaticAssets` i `setViewEngine` de l'aplicació `app` per a especificar el/les carpeta(s) on haurà contingut estàtic, i el motor de plantilles a utilitzar, respectivament. En el nostre cas, pot quedar una cosa així:

```
...
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();
```

Les carpetes `public` i `views` hauran de situar-se, d'acord amb el codi anterior, en l'arrel del projecte Nest. Després, per a renderitzar qualsevol vista des d'un *handler* (mètode d'un controlador), n'hi ha prou que li passem la resposta com a paràmetre amb el decorador `@Res()`, per a poder accedir al seu mètode `render`, com hem fet en sessions prèvies:

```
@Get()
async prova(@Res() res) {
    return res.render('index');
}
```

> **Exercici 1**:
>
> Fes una còpia del projecte `tasques-nest` de sessions anteriors i canvia'l de nom a `tasques-nest-web`. Instal·la Nunjucks i Bootstrap en este nou projecte, i configura l'aplicació principal `main.ts` perquè use Nunjucks com a motor de plantilles, i carregue el contingut estàtic de Bootstrap.
>
> Definix una carpeta `views` per a les vistes, i una vista `base.njk` de la qual heretarà la resta, amb un bloc per a poder definir el seu títol en la capçalera (*title*), i un altre bloc per al seu contingut. Fes que la vista base incorpore els estils de Bootstrap.
>
> Crea un nou mòdul anomenat `web`, amb el seu controlador associat. Abans de seguir, hauràs d'exportar el servici `TascaService` en el mòdul de tasques per a poder-lo utilitzar en este altre mòdul:

```
@module({
   imports: ...
   controllers: ...
   providers: ...
   exports: [TascaService]
})
```

Després, hauràs d'importar el mòdul de tasques sencer des del nou mòdul web:

```
@module({
   imports: [TascaModule],
   controllers: [WebController]
})
```

Ara, definix un parell de *handlers* en el controlador de web (arxiu `src/web/web.controller.ts`) per a respondre a les rutes `/web/tasques` i `/web/tasques/:id`. El primer haurà de renderitzar la vista `tasques_llistat.njk`, que hauràs de crear, amb un llistat amb els noms de les tasques. En fer clic en cadascuna d'elles es dirà al segon *handler*, que renderitzarà la vista `tasques_fitxa.njk`, que també hauràs d'implementar, amb la fitxa de cada tasca, indicant el seu nom, prioritat i data.

Finalment, fes que la ruta arrel redirigisca al llistat de tasques.

1.1. Limitacions en l'ús de Nest.js per a *frontend*

A pesar que, com hem vist, és possible utilitzar el framework *Nest.js* per a desenvolupar plantilles i vistes que conformen nostre *frontend*, el seu ús és molt més habitual per a desenvolupar API REST en el costat del *backend*, a les quals accedir des de qualsevol altra aplicació *frontend*, com a Angular, React, etc.

En eixe sentit, configurar autenticació per sessions, o uns certs tipus de formularis, pot complicar el codi a desenvolupar. A pesar que *Nest* se secunda o basa en Express, la comunicació o connexió entre tots dos frameworks no és tan senzilla com podria en algunes d'estes situacions, i la documentació a la qual acudir per a poder fer-ho és realment escassa. Per este motiu, en estos apunts ens limitarem a aprofundir en l'ús de *Nest.js* per a desenvolupament d'API REST.

2. Altres opcions addicionals

Per a finalitzar este tutorial d'ús del framework *Nest.js* vegem a continuació algunes qüestions addicionals que s'han quedat en el tinter.

2.1. Neteja del projecte recentment creat

Quan creem un projecte Nest, com hem vist, s'afigen molts fitxers de configuració i codi inicial. Alguns d'estos fitxers no són essencials, i pot ser que no els arribem a utilitzar en el nostre desenvolupament, amb el que podem eliminar-los sense problemes. Ací indiquem alguns exemples:

- El controlador i servici principal `src/app.controller.ts` i `src/app.service.ts`, juntament amb l'especificació `src/app.controller.spec.ts` rares vegades se solen utilitzar. Podem eliminar-los, i llevar les referències que hi ha d'ells en el mòdul `src/app.module.ts`.
- Les llibreries sobre *prettier* poden causar errors de compilació simplement per una mala estructuració del codi. Si ens resulta molest podem desinstal·lar (`npm uninstall`) els paquets relacionats, com *prettier*, *eslint-config-prettier* o *eslint-plugin-prettier*.

2.2. Desplegament de l'aplicació en producció

El normal quan estem desenvolupant una aplicació és executar-la en mode desenvolupament (*dev*), la qual cosa fem per exemple amb *npm run start:dev*. No obstant això, una vegada l'aplicació està llesta convé recompilar-la i distribuir-la per a producció, ja que el contingut compilat és significativament més xicotet, perquè no s'incorporen unes certes llibreries que només necessitem durant el desenvolupament, com per exemple *prettier* o *lint* per a formatejar de codi.

Per a construir la nostra app per a producció disposem d'un script anomenat `npm build`, que internament executa `nest build` per a regenerar la carpeta *dist* amb els continguts propis per a producció. Una vegada regenerada la carpeta, podem posar en marxa l'aplicació en mode producció amb `npm run start:prod`, que bàsicament executa l'arxiu `main` de la carpeta `dist` generada en el pas anterior.

2.3. Ús de *pipes* per a validació de dades

A més de l'ús de *ValidationPipe* per a establir mecanismes de validació personalitzats, Nest incorpora una sèrie de *pipes* predefinits que s'encarreguen de comprovar unes certes validacions simples. Un exemple és `ParseIntPipe`, que podem emprar per a forçar al fet que, per exemple, el *aneu* que rebem en una petició siga un nombre enter vàlid:

```
// Importem ParseIntPipe de @nestjs/common al costat de la resta
import { ...., ParseIntPipe } from '@nestjs/common';

...

// DELETE /contacte/:id
@Delete(':id')
esborrar(@param('id', ParseIntPipe) id: number) {
    ...
}
```

Això provocarà una excepció i una resposta d'error si no proporcionem una dada numèrica vàlida en la URL. Podeu consultar altres *pipes* alternatius en la documentació de Nest.

2.4. Codis d'estat, capçaleres de resposta i redireccions

Per defecte, els *handlers* en Express retornen automàticament un codi 200 juntament amb la resposta, excepte en el cas de peticions POST, on es retorna un estat 201. Si volem retornar un altre estat diferent, podem utilitzar el decorador `@Httpcode` en l'encapçalat del *handler*, indicant el codi a retornar:

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

A més, també podem emprar el decorador `@Header` per a enviar capçaleres de resposta (una vegada per cada capçalera), indicant en cada cas el nom de la capçalera i el seu valor associat.

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

Finalment, podem utilitzar el decorador `@Redirect` per a fer que un *handler* redirigisca a una altra URL.

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

Per defecte, la redirecció genera un codi 301, però podem canviar-lo en el segon paràmetre. De fet, també podem fer que tant la ruta a la qual redirigir com el codi d'estat canvien, retornant des del *handler* un objecte amb els camps `url` i `statusCode` establits amb els valors indicats (tots dos camps són opcionals):

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

> **NOTA**: haurem d'importar en la instrucció `import` corresponent estos decoradors des de `@nestjs/common`.

2.5. Connexions entre esquemes Mongoose en Nest

En exemples anteriors hem vist com definir esquemes senzills en Nest. Però també existix la possibilitat de connectar dades d'un esquema amb un altre:

- A través de l'*id* d'un document en un altre (relacions entre col·leccions)
- Mitjançant subdocumentos

2.5.1. Connexions entre col·leccions

Si volem connectar els documents d'una col·lecció amb els d'una altra, n'hi ha prou que afegim com a camp d'un esquema l'*id* de l'altra. Per exemple, si tenim un esquema associat al model *usuaris* i estem definint un esquema amb comentaris d'eixos usuaris, podríem fer una cosa així:

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

@Schema()
export class Comentari extends Document {

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

    @Prop({
        required: true,
        type: mongoose.Schema.Types.ObjectId,
        ref: 'usuaris'
    })
    usuari: string;
}

export const ComentariSchema =
    SchemaFactory.createForClass(Comentari);
```

2.5.2. Connexions mitjançant subdocumentos

Si el que volem és fer subdocument(s) d'un esquema dins d'un altre, n'hi ha prou que indiquem un nou camp que siga un array del tipus de l'altre esquema. Per exemple, a partir de l'esquema de comentaris anterior, podem fer que un *post* tinga embeguts un array de comentaris:

```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { Document } from 'mongoose';
import { ComentariSchema } from ... // Afegir la ruta adequada;

@Schema()
export class Post extends Document {

    @Prop({
        required: true,
        minlength: 3
    })
    titol: string;

    ...

    comentaris: [ComentariSchema]
}

export const PostSchema =
    SchemaFactory.createForClass(Post);
```