Desarrollo de aplicaciones con Nest.js
Más opciones del framework

En este documento vamos a ver algunas opciones adicionales que ofrece *Nest.js* en el desarrollo de aplicaciones.

1. 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 install 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:

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

Ejercicio 1:

Haz una copia del proyecto `tareas-nest` de sesiones anteriores y renómbralo a `tareas-nest-web`. Instala Nunjucks y Bootstrap en este nuevo proyecto, y 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.

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.

1.1. Limitaciones en el uso de Nest.js para *frontend*

A pesar de que, como hemos visto, es posible utilizar el framework *Nest.js* para desarrollar plantillas y vistas que conformen nuestro *frontend*, su uso es mucho más habitual para desarrollar API REST en el lado del *backend*, a las que acceder desde cualquier otra aplicación *frontend*, como Angular, React, etc.

En ese sentido, configurar autenticación por sesiones, o ciertos tipos de formularios, puede complicar el código a desarrollar. A pesar de que *Nest* se apoya o basa en Express, la comunicación o conexión entre ambos frameworks no es todo lo sencilla que podría en algunas de estas situaciones, y la documentación a la que acudir para poder hacerlo es realmente escasa. Por este motivo, en estos apuntes nos limitaremos a profundizar en el uso de *Nest.js* para desarrollo de API REST.

2. Otras opciones adicionales

Para finalizar este tutorial de uso del framework *Nest.js* veamos a continuación algunas cuestiones adicionales que se han quedado en el tintero.

2.1. Limpieza del proyecto recién creado

Cuando creamos un proyecto Nest, como hemos visto, se añaden muchos ficheros de configuración y código inicial. Algunos de estos ficheros no son esenciales, y puede que no los lleguemos a utilizar en nuestro desarrollo, con lo que podemos eliminarlos sin problemas. Aquí indicamos algunos ejemplos:

- El controlador y servicio principal `src/app.controller.ts` y `src/app.service.ts`, junto con la especificación `src/app.controller.spec.ts` rara vez se suelen utilizar. Podemos eliminarlos, y quitar las referencias que hay de ellos en el módulo `src/app.module.ts`.
- Las librerías sobre *prettier* pueden causar errores de compilación simplemente por una mala estructuración del código. Si nos resulta molesto podemos desinstalar (`npm uninstall`) los paquetes relacionados, como *prettier*, *eslint-config-prettier* o *eslint-plugin-prettier*.

2.2. Despliegue de la aplicación en producción

Lo normal cuando estamos desarrollando una aplicación es ejecutarla en modo desarrollo (*dev*), lo que hacemos por ejemplo con el comando *npm run start:dev*. Sin embargo, una vez la aplicación está lista conviene recompilarla y distribuirla para producción, ya que el contenido compilado es significativamente más pequeño, porque no se incorporan ciertas librerías que sólo necesitamos durante el desarrollo, como por ejemplo *prettier* o *lint* para formateo de código.

Para construir nuestra app para producción disponemos de un script llamado `npm build`, que internamente ejecuta el comando `nest build` para regenerar la carpeta *dist* con los contenidos propios para producción. Una vez regenerada la carpeta, podemos poner en marcha la aplicación en modo producción con el comando `npm run start:prod`, que básicamente ejecuta el archivo `main` de la carpeta `dist` generada en el paso anterior.

2.3. Uso de *pipes* para validación de datos

Además del uso de *ValidationPipe* para establecer mecanismos de validación personalizados, Nest incorpora una serie de *pipes* predefinidos que se encargan de comprobar ciertas validaciones simples. Un ejemplo es `ParseIntPipe`, que podemos emplear para forzar a que, por ejemplo, el *id* que recibimos en una petición sea un número entero válido:

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

...

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

Esto provocará una excepción y una respuesta de error si no proporcionamos un dato numérico válido en la URL. Podéis consultar otros *pipes* alternativos en la documentación de Nest.

2.4. 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` .

2.5. Conexiones entre esquemas Mongoose en Nest

En ejemplos anteriores hemos visto cómo definir esquemas sencillos en Nest. Pero también existe la posibilidad de conectar datos de un esquema con otro:

- A través del *id* de un documento en otro (relaciones entre colecciones)
- Mediante subdocumentos

2.5.1. Conexiones entre colecciones

Si queremos conectar los documentos de una colección con los de otra, basta con que añadamos como campo de un esquema el *id* de la otra. Por ejemplo, si tenemos un esquema asociado al modelo *usuarios* y estamos definiendo un esquema con comentarios de esos usuarios, podríamos hacer algo así:

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

@Schema()
export class Comentario extends Document {

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

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

export const ComentarioSchema =
    SchemaFactory.createForClass(Comentario);
```

2.5.2. Conexiones mediante subdocumentos

Si lo que queremos es hacer subdocumento(s) de un esquema dentro de otro, basta con que indiquemos un nuevo campo que sea un array del tipo del otro esquema. Por ejemplo, a partir del esquema de comentarios anterior, podemos hacer que un *post* tenga embebidos un array de comentarios:

```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { Document } from 'mongoose';
import { ComentarioSchema } from ... // Añadir la ruta adecuada;

@Schema()
export class Post extends Document {

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

    ...

    comentarios: [ComentarioSchema]
}

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