Documentación NestJS

Controladores.-

La funcionalidad primordial del controlador es la de responder las solicitudes que se realicen al servidor.

- Creación.-

Para la creación de controladores podemos usar el comando propio de Nest que nos permitirá a partir de las especificaciones realizadas con subcomandos crearlo sin archivos de prueba.

Para realizarlo de la forma normal debemos utilizar el comando:

```
nest generate controller products
```

Una ves ingresado el comando nos mostrara en consola lo que hemos realizado, así:

```
midesweb@MacBook-Pro ~/sites/sandbox/hola-mundo-nestjs  master nest g co products
CREATE src/products/products.controller.spec.ts (506 bytes)
CREATE src/products/products.controller.ts (105 bytes)
UPDATE src/app.module.ts (338 bytes)
```

Para evitar que se generen archivos de pruebas, los cuales vienen con la extensión ".spec" debemos introducir el comando anterior con un tipo de subcódigo, tal que así:

```
nest generate controller products --no-spec
```

- Registro del controlador.-

Una ves ya creado nuestro controlador nuestro ambiente de programación debe importarlo de forma automática dentro del módulo principal del proyecto, caso contrario debemos ingresar a nuestro app.module.ts y debemos ingresar el siguiente comando para que nuestro controlador sea reconocible para el proyecto:

```
import { ProductsController } from './products/products.controller';
```

Esto también hace que se agregue nuestro controlador al array de los mismos dentro del decorador @Module.

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

- Cuerpo del controlador.-

El código interno de nuestro controlador esta conformado primeramente con el import del decorador que nos ayuda a convertir clases en controladores, esta se importa desde

“@nestjs/common”, siguiente podemos ver que se ha generado una clase vacía por lo cual no esta relacionada con ninguna ruta de momento.

En la clase vacía se ha añadido con decorador con la sintaxis de “@Controller(´products´), este decorador nos servirá para condicionar como controlador a lo que asignemos a en esa clase.

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

@Controller('products')
export class ProductsController {}
```

- Rutas en el controlador.-

Para poder acceder a las funcionalidades del controlador debemos utilizar métodos HTTP para ello, estos métodos van desde un GET a otros ejemplos tales como: POST, PUT, UPDATE y DELETE.

Estos métodos serán introducidos en el decorador para que nuestro proyecto pueda reconocerlo.

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

@Controller('products')
export class ProductsController {

  @Get()
  getHelloInProducts(): string {
    return "Estamos en productos";
  }

}
```

Módulos.-

Los módulos son clases en las cuales su funcionalidad es la de servir como contenedor de otras clases o artefactos, tales como los controladores y servicios. Así que podemos decir que sirven para agrupar elementos para que se puedan relacionar entre sí.

En los módulos existe el decorador @Module() que nos sirve al igual que el decorador @Controller para que las clases que vayamos a utilizar se comporten como un módulo, dentro de esto existen 4 elementos importantes:

- **Controladores:** Estas son las clases controlados que el módulo debe definir.
- **Providers:** Aquí se encuentran los servicios y elementos inyectables que deberán ser instanciados para que puedan ser usados.

- **Exports:** Es una lista de providers que pueden ser usados por otros módulos externos.
- **Imports:** Es una lista de providers externos que podemos usar dentro de este módulo actual.

A continuación, explicare la forma de creación de los módulos utilizando comandos.

- Creación.-

Al igual que en los controladores, los módulos los crearemos a partir del comando “generate” tal que así:

```
nest generate module products
```

Una ves ejecutado el comando nos mostrara por consola los cambios realizados y se actualizara el modulo principal de nuestro proyecto.

```
midesweb@MacBook-Pro ~/sites/sandbox/hola-mundo-nestjs master nest g mo products
CREATE src/products/products.module.ts (85 bytes)
UPDATE src/app.module.ts (499 bytes)
```

El módulo, el cual fue creado con el nombre “ProductsModule” vendrá con un código predeterminado tal como este:

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

@Module({})
export class ProductsModule {}

// [...]
import { ProductsModule } from './products/products.module';

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

Servicios.-

Los servicios dentro de nestjs son principalmente para que los puedan ser accesados además nos facilita con la conectividad a la base de datos que estemos utilizando, dentro del servicio se pueden realizar algunas inyecciones de dependencias para que el servidor realice más funcionalidades.

Para su creación utilizaremos el comando generate enfocado al servicio el cual es:

```
nest generate service products
```

Para poder brindar una conectividad del servicio con una base de datos debemos utilizar la inyección de dependencias enfocada a ello, la cual es conocida como @InjectRepository y su implementación vendría a ser esta:

```
@Injectable()
export class SubjectsService {
  constructor(
    @InjectRepository(SubjectEntity)
    private subjectRepository: Repository<SubjectEntity>,
  ) {}
```

Además, dentro de los servicios los métodos HTTP que fueron establecidos dentro del controlador también pueden ser utilizado aquí gracias a la inyección de dependencias tal como el método "repository.create", cabe recalcar que utilizamos el repository porque estamos utilizando esa dependencia.

```
async create(payload: CreateSubjectDto) {
    const newSubject = this.subjectRepository.create(payload);

    return await this.subjectRepository.save(newSubject);
  }
```

Y esta misma lógica se puede utilizar para los otros métodos que se requieran dentro del proyecto o depende de los métodos HTTP que se vayan a utilizar.

DTO.-

Los DTO o Data Transfer Object se utilizan principalmente para la comunicación entre cliente y servidor, por lo que podemos decir que un DTO es un transporte de datos.

para la creación de los DTO también los podemos crear con el comando generate y utilizaremos la siguiente estructura para que los DTO sean generados dentro de una carpeta especifica, esto nos ayudara con la organización de carpetas, aunque podemos crearlas en donde nos sea necesario para nuestro proyecto.

```
nest generate class products/dto/product.dto
```

Dentro de los DTO podemos establecer las validaciones que van a ser utilizadas por los campos que hayamos establecido en el controlador, para que esto sea posible dentro de nestjs existe el "class-validator" en donde podemos encontrar todos los tipos de validaciones.

Para que sean funcionales dentro del DTO deben ser importadas todas las validaciones a ser utilizadas, además también podemos personalizar mensajes propios que serán visualizados por el cliente al realizar la petición, aso como condicionar a que sean un tipo de dato en especial o la cantidad de caracteres máximo y mínimo.

A continuación, se definirán algunos ejemplos de validaciones y su funcionalidad dentro de los DTO.

Validación	Funcionalidad
@IsEmpty()	Revisa si el valor dado es vacío, null o indefinido.
@IsIn(values: any [])	Revisa si los valores dentro de un array son valores permitidos.
@IsBoolean()	Revisa si los datos son booleanos (verdadero o falso).
@IsArray()	Revisa si los valores dados están es un array.
@IsPositive()	Revisa si los valores dados son números superiores a 0.
@IsNegative()	Revisa si los valores dados son números inferiores a 0.
@Contains(seed: string)	Revisa si la cadena de caracteres tiene una semilla especificada
@IsAscii()	Revisa que la cadena de caracteres posea caracteres ASCII únicamente.
@IsBase64()	Revisa si la cadena de caracteres esta codificada en base64.
@IsHexColor()	Revisa si la cadena de caracteres coincide con un color hexadecimal.

ENTITIES.-

Las entities son utilizadas como recursos para el trabajo de forma conjunta con la base de datos ya que en ella podemos realizar directamente el mapeo de los objetos y convertirlo a registros internos en las tablas de las bases de datos.

Para poder definir las entidades debemos utilizar el decorador @Entity, lo cual va a generar el siguiente código.

```
import { Entity } from "typeorm";


@Entity()
export class User {


}
```

Al tener comunicación directa con la base de datos, al momento de mapear las tablas serán necesario establecer los campos que sean llaves primarias las que sirven como identificador de relaciones entre tablas.

Para poder definir las llaves primarias utilizaremos el decorador @PrimaryGeneratedColumn, con esto el código de la entity se vería de esta forma.

```ts
import { Column, Entity, PrimaryGeneratedColumn } from "typeorm";

@Entity()
export class Product {

  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @Column()
  description: string;

  @Column()
  stock: number;
}
```

Dentro de las columnas en las entidades se pueden cambiar los tipos de datos, esto se puede hacer utilizando los parámetros exclusivos para definir los condicionales de datos, tal que así:

```ts
import { Column, Entity, PrimaryGeneratedColumn } from "typeorm";

@Entity()
export class Product {

  @PrimaryGeneratedColumn()
  id: number;

  @Column('int')
  stock: number;

  @Column('varchar', { length: 50 })
  name: string;

  @Column('int', { width: 5 })
  stock: number;
}
```