Table of Contents

NestJs

1. Install:

node -v
npm i -g @nestjs/cli
verification
nest –version

nest –help

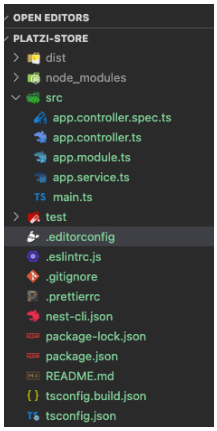
2. create

nest new name-project
start project
inside od folder
npm run start

modo Desarrollo

npm run start:dev
nest start --watch

3. Project structure :



Add
File .editorconfigo

4. Summary TypeScript

NestJS utiliza TypeScript como lenguaje de programación y conocer sus características y qué le adiciona a Javascript te convertirá en un profesional más completo de esta tecnología.
Qué es TypeScript
TypeScript es un lenguaje de programación mantenido por Microsoft. En otras palabras, es un "superconjunto" de Javascript que le agrega tipado de datos y programación orientada a objetos.
El código fuente escrito en TypeScript, se "transpila" a código Javascript que es el que finalmente entienden los intérpretes de Javascript como los navegadores web o NodeJS.
Tipado de Datos con TypeScript
Con Javascript puedes crear una variable del tipo String y posteriormente asignarle un valor del tipo Entero o Boolean. Esto es propenso a tener errores en tiempo de ejecución.
TypeScript permite tipar los datos para que estos no cambien de tipo.

```
// Tipado de datos con TypeScript
const text: string;
const num: number;
const bool: boolean:
const arr: Array<number>[];
Safe type
let name: string;
const age = 19;
const suma = (a: number, b:number) => {
  return a + b;
}
Classes
class Person {
  constructor(private age: number, private name: string) {}

  getSummary() {
    return `I'm ${this.name} and I'm ${this.age}`;
  }
}
```

POO con TypeScript

Javascript permite el desarrollo de aplicaciones utilizando POO basada en Prototipos. Puedes tomar el Curso Básico de Programación Orientada a Objetos con JavaScript para entrar en más detalle.

TypeScript llega para permitir escribir código utilizando los conceptos de la POO más fácilmente con clases, herencia, polimorfismo, etc.

```
// Programación Orientada a Objetos con TypeScript
class Alumno {
  private nombre: string;
  private apellido: string;

  constructor(nombre: string, apellido: string) {
    this.nombre = nombre;
    this.apellido = apellido;
  }

  getNombre() {
    return this.nombre;
  }
  setNombre(nuevoNombre: string) {
    this.nombre = nuevoNombre;
  }
}
const alumno = new Alumno('Freddy', 'Vega');
```

Estas son las diferencias básicas que tienes que conocer entre Javascript y TypeScript. Si quieres aprender más de esta tecnología puedes tomar el Curso de Fundamentos de TypeScript.

5. Controller

El concepto más básico para desarrollar una aplicación con NestJS son los Controladores.

Qué son los controladores en NestJS
Los Controladores manejarán las rutas o endpoints que la aplicación necesite, además de validar los permisos del usuario, filtro y manipulación de datos.

Estructura de un controlador

La aplicación de NestJS creada por defecto con el CLI con el comando nest new <project-name> trae consigo un controlador básico con el nombre app.controller.ts. Verás que dicho archivo contiene una clase que a su vez posee un decorador llamado @Controller().

Dicho decorador le indica al compilador de NestJS que esta clase tendrá el comportamiento de un controlador.

```
// app.controller.ts
import { Controller, Get } from '@nestjs/common';
```

```
import { AppService } from './app.service';

@Controller()
export class AppController {

  constructor(private readonly appService: AppService) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```

Los controladores deben ser importados en un módulo para que sean reconocidos los endpoints.

```
// app.module.ts
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';

@Module({
  imports: [],
  controllers: [
    // Imports de Controladores
    AppController
  ],
})
export class AppModule {}
```

El controlador importa un servicio que son los responsables de la lógica y obtención de datos desde una BBDD que el controlador requiere.

```
// app.service.ts
import { Injectable } from '@nestjs/common';

@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
  }
}
```

Puedes correr el servidor de NestJS con el comando npm run start:dev e ingresar a la ruta localhost:3000/ para visualizar el contenido que el controlador envía.

Si quieres crear una nueva ruta, basta con crear un método en la clase del controlador y colocarle el decorador @Get() con un nombre para el nuevo endpoint.

```
// app.controller.ts
@Controller()
export class AppController {

  constructor(private readonly appService: AppService) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }

  @Get('new-endpoint')
  newEndpoint(): string {
    return 'New endpoint';
  }
}
```

Ingresa a esta nueva ruta desde localhost:3000/new-endpoint para visualizar su respuesta y así crear los endpoints que necesites.

Existen diferentes tipos de endpoints que se identifican a través de los Verbos HTTP. Cada uno con un propósito determinado siguiendo el protocolo.

Obtención de datos con GET

En particular, el verbo GET suele utilizarse para endpoints que permiten la obtención de datos como un producto o una lista de productos.
Es frecuente la necesidad de que este tipo de endpoints también reciban información dinámica en las URL como el identificador de un producto.
Para capturar estos datos en NestJS, tienes que importar el
decorador Param desde @nestjs/common y emplearlo de la siguiente manera en tus endpoints.

```
import { Controller, Get, Param } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {

  constructor(private readonly appService: AppService) {}

  @Get('product/:idProduct')
  getProduct1(@Param() params: any): string {
```

```
    return `Producto id: ${params.idProduct}`;
  }

  @Get('product/:idProduct')
  getProduct2(@Param('idProduct') idProduct: string): string {
    return `Producto id: ${idProduct}`;
  }
}
```

Observa el decorador @Get() que posee el nombre del endpoint seguido de un :idProduct que identifica al parámetro dinámico. Luego puedes implementar el decorador @Param() para capturar todos los parámetros juntos en un objeto o @Param('idProduct') para capturar únicamente el parámetro con dicho nombre.
De esta forma, accede en un navegador a localhost:3000/product/12345 para capturar ese 12345 y posteriormente utilizarlo.

Contribución creada por: Kevin Fiorentino.

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

@Controller()
export class AppController {
  ...

  @Get('products/:productId')
  getProduct(@Param('productId') productId: string) {
    return `product ${productId}`;
  }

  @Get('categories/:id/products/:productId')
  getCategory(@Param('productId') productId: string, @Param('id') id: string) {
    return `product ${productId} and ${id}`;
  }
}
```

Hay varias maneras de enviar información a un endpoint del tipo GET, cada una con sus ventajas y desventajas. Profundicemos acerca de ellas.

Parámetros de ruta vs Parámetros query

Los parámetros de ruta son aquellos que forman parte del propio endpoint y suelen ser parámetros obligatorios.

```
@Get('product/:idProduct')
getProduct2(@Param('idProduct') idProduct: string): string {
  return `Producto id: ${idProduct}`;
}
```

En NestJS se capturan con el decorador @Param().

Por otro lado, están los parámetros de consulta o query en las URL como por ejemplo example.com/products?limit=10&offset=20 que se capturan con el decorador @Query() importado desde @nestjs/common.

```
@Get('products')
getProducts(@Query('limit') limit = 10, @Query('offset') offset = 0): string {
  return `Lista de productos limit=${limit} offset=${offset}`;
}
```

Su principal diferencia es que los parámetros de consulta suelen ser opcionales; el comportamiento del endpoint tiene que contemplar que estos datos pueden no existir con un valor por defecto.

Los parámetros de ruta se utilizan para IDs u otros identificadores obligatorios, mientras que los parámetros de consulta se utilizan para aplicar filtros opcionales a una consulta. Utilízalos apropiadamente en tus endpoints según tengas la necesidad.

Evitando el bloqueo de rutas

Un importante consejo a tener en cuenta para construir aplicaciones con NestJS es asegurar que un endpoint no esté bloqueando a otro.

Por ejemplo:

```
/* Ejemplo Bloqueo de Endpoints */
@Get('products/:idProduct')
endpoint1() {
  // ...
}
@Get('products/filter')
endpoint2() {
  // ...
}
```

El endpoint1 bloquea al **endpoint2, ya que este está esperando un parámetro :idProduct y si llamamos a localhost:3000/products/filter NestJS entenderá que la palabra filter es el ID que espera el primer endpoint ocasionando que no sea posible acceder al segundo endpoint.

Se soluciona de forma muy sencilla invirtiendo el orden de los mismos. Coloca los endpoints dinámicos en segundo lugar para que no ocasionen problemas.

```
/* Solución Bloqueo de Endpoints */
@Get('products/filter')
endpoint2() {
  // ...
}
@Get('products/:idProduct')
endpoint1() {
  // ...
```

```
}
```
Este es un inconveniente común que suele suceder en NestJS y es importante que lo conozcas para evitar dolores de cabeza.

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

@Controller()
export class AppController {
  ...

  @Get('products')
  getProducts(
    @Query('limit') limit = 100,
    @Query('offset') offset = 0,
    @Query('brand') brand: string,
  ) {
    return `products limit=> ${limit} offset=> ${offset} brand=> ${brand}`;
  }

  @Get('products/filter')
  getProductFilter() {
    return `yo soy un filter`;
  }
}
```

6. Separación de Responsabilidad

Single responsibility

NestJS le da mucha importancia a los Principios SOLID en el desarrollo de software para mantener las buenas prácticas de codificación. Una de ellas es la responsabilidad única.

Qué es la responsabilidad única

La S de SOLID hace referencia a “Single Responsibility” y recomienda que cada pieza de software debe tener una única función. Por ejemplo, un controlador de productos no debería encargarse de categorías o de usuarios. Se debe crear un controlador para cada entidad que la aplicación necesite.

Lo mismo ocurre con los métodos. Un método para la obtención de datos solo debe realizar dicha acción y no estar actualizando o manipulando los datos de otra manera.

Responsabilidades en NestJS

En NestJS, una buena práctica es crear un directorio llamado controllers donde se agruparán todos los controladores que tu aplicación necesite. Ese ya es un buen paso para mantener el orden en tu proyecto.

Apóyate del CLI de NestJS para autogenerar código rápidamente con el comando nest generate controller <controller-name> o en su forma corta nest g co <controller-name>.

Es una buena forma de comenzar a seguir las buenas prácticas a la hora de escribir código y estructurar una aplicación.

Controllers y responsabilidades
nest g cor controllers/categories --flat
src/controllers/categories.controller.ts

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

@Controller('categories') // 👈 Route
export class CategoriesController {

  @Get(':id/products/:productId')
  getCategory(
    @Param('productId') productId: string,
    @Param('id') id: string
  ) {
    return `product ${productId} and ${id}`;
  }
}
```

nest g co controllers/products --flat
src/controllers/products.controller.ts

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

@Controller('products') // 👈 Route
export class ProductsController {
  @Get()
  getProducts(
    @Query('limit') limit = 100,
    @Query('offset') offset = 0,
    @Query('brand') brand: string,
  ) {
    return `products limit=> ${limit} offset=> ${offset} brand=> ${brand}`;
  }
```

```
  @Get('filter')
  getProductFilter() {
    return `yo soy un filter`;
  }

  @Get(':productId')
  getProduct(@Param('productId') productId: string) {
    return `product ${productId}`;
  }
}
```

src/app.module.ts

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductsController } from './controllers/products.controller';
import { CategoriesController } from './controllers/categories.controller';

@Module({
  imports: [],
  controllers: [
    AppController,
    ProductsController, // 👈 New controller
    CategoriesController  // 👈 New controller
  ],
  providers: [AppService],
})
export class AppModule {}
```

Contribución creada por: Kevin Fiorentino.

7. Post

   Así como el verbo HTTP GET se utiliza para la obtención de datos, el verbo HTTP Post se utiliza para la creación de los mismos previamente.

   Qué es el método Post
   Es el método para creación de datos. Para utilizarlo en Nest.js debemos importar el decorador.

   Crear registro con Post
   En tu proyecto NestJS, tienes que importar los decoradores Post y Body desde @nestjs/common. El primero para indicar que el endpoint es del tipo POST y el segundo para capturar los datos provenientes del front-end en el cuerpo del mensaje.

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

@Controller()
export class AppController {

 @Post('product')
 createProducto(@Body() body: any): any {
  return {
   name: body.name,
   price: body.price
  };
 }
}
```

Un buen endpoint del tipo POST tiene que devolver el registro completo recientemente insertado en la BBDD para que el front-end pueda actualizarse inmediatamente y no tener que realizar una consulta por el mismo.

Recuerda también que, al tratarse de un endpoint POST, no puedes realizar la solicitud desde el navegador al igual que con los endpoints GET. Para probar tu aplicación, tienes que utilizar una plataforma de APIs como Postman.

```
import { ..., Post, Body } from '@nestjs/common';

@Controller('products')
export class ProductsController {
 ...

 @Post() // 👈 New decorator
 create(@Body() payload: any) {
  return {
   message: 'accion de crear',
   payload,
  };
 }
}
```

Contribución creada por: Kevin Fiorentino.

8. Put Delete

El verbo HTTP GET se utiliza para la obtención de datos y el verbo POST para la creación de estos. También existe el verbo PUT y DELETE para la actualización y borrado de datos respectivamente.

Actualización de datos con PUT

El verbo PUT se usa para la actualización de un registro en la BBDD. Suele recibir un Body con los datos a actualizar, pero también es importante que reciba el ID del registro para buscar al mismo.

```
import { Controller, Put, Param, Body } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {

  @Put('product/:idProduct')
  updateProducto(@Param('idProduct') idProduct: string, @Body() body: any): any {
    return {
      idProduct: idProduct,
      name: body.newName,
      price: body.newPrice
    };
  }
}
```

El ID suele recibirse por parámetros de URL para que sea obligatorio, mientras que reservamos el cuerpo del mensaje para los datos actualizados. Finalmente, retornamos el registro completo luego de ser actualizado.

Eliminar datos con DELETE

Eliminar un registro es sencillo. Basta con decorar el endpoint con DELETE. Suele recibir el ID del registro a borrar únicamente.

```
import { Controller, Delete, Param } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {

  @Delete('product')
  deleteProducto(@Param('idProduct') idProduct: string): any {
    return {
      idProduct: idProduct,
      delete: true,
      count: 1
    };
```

```
  }
}
```
Una buena práctica para este tipo de endpoints es retornar un booleano que indique si el registro fue eliminado o no. Además de incluir un count que indique cuántos registros fueron eliminados.

src/controllers/products.controller.ts

```
import {..., Put, Delete } from '@nestjs/common';

@Controller('products')
export class ProductsController {
  ...
  @Put(':id')
  update(@Param('id') id: number, @Body() payload: any) {
    return {
      id,
      payload,
    };
  }

  @Delete(':id')
  delete(@Param('id') id: number) {
    return id;
  }
}
```
Contribución creada por: Kevin Fiorentino.

9. Código de estados http

El protocolo HTTP tiene estandarizado una lista de códigos de estado que indican los tipos de respuesta que las API deben enviar dependiendo la situación. Como profesional en el desarrollo de software, debes conocerlos y diferenciarlos.

Cuáles son los códigos HTTP
Hay cinco familias de códigos de estado HTTP que tienes que utilizar apropiadamente para que tus APIs informen correctamente la situación de la solicitud.

Estados informativos (100–199)
Estados de éxito (200–299)
Estados de redirección (300–399)
Estados de error del cliente (400–499)
Estados de error del servidor (500–599)
Cómo manejar los códigos de estado HTTP con NestJS

En NestJS, puedes manejar los códigos de estado HTTP importando el decorador HttpCode y el enumerado HttpStatus desde @nestjs/common.

El primero te servirá para indicar cuál será el código de estado HTTP que un endpoint tiene que devolver; el segundo para ayudarte por si no recuerdas qué código pertenece a cada tipo de respuesta.

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

@Get('product/:idProduct')
@HttpCode(HttpStatus.OK)
getProduct2(@Param('idProduct') idProduct: string): string {
   return `Producto id: ${idProduct}`;
}

@Post('product')
@HttpCode(HttpStatus.CREATED)
createProducto(@Body() body: any): any {
   return {
     name: body.name,
     price: body.price
   };
}
```

El enumerado HttpStatus.OK indica código de estado 200 que es el que suele devolver por defecto todos los endpoints cuando la operación sale exitosamente. Los endpoints POST suelen devolver HttpStatus.CREATED o código 201 para indicar la creación exitosa del registro.

src/controllers/products.controller.ts

```
import { ..., HttpStatus, HttpCode, Res } from '@nestjs/common';
import { Response } from 'express';

@Controller('products')
export class ProductsController {
  ...
  @Get(':productId')
  @HttpCode(HttpStatus.ACCEPTED) // 👈 Using decorator
  getOne(
    @Res() response: Response,
    @Param('productId') productId: string
  ) {
    response.status(200).send({...}); // 👈 Using express directly
  }
}
```

Contribución creada por: Kevin Fiorentino.

10. Creación de Servicios

Los servicios en NestJS son los que suelen tener la lógica del negocio y la conexión con la base de datos.

Qué son los servicios en NestJS

Los servicios son una pieza esencial de las aplicaciones realizadas con el framework NestJS. Están pensados para proporcionar una capa de acceso a los datos que necesitan las aplicaciones para funcionar.

Un servicio tiene la responsabilidad de gestionar el trabajo con los datos de la aplicación, de modo que realiza las operaciones para obtener esos datos, modificarlos, etc.

Primer servicio con NestJS

Para crear un servicio puedes utilizar el comando nest generate service <service-name> o en su forma corta nest g s <service-name>.

```
// app.service.ts
import { Injectable } from '@nestjs/common';

@Injectable()
export class AppService {

  getHello(): string {
    return 'Hello World!';
  }
}
```

Los servicios utilizan el decorador @Injectable() y deben ser importados en los providers del módulo al que pertenecen o tu aplicación no lo reconocerá y tendrás errores al levantar el servidor.

```
// app.module.ts
import { Module } from '@nestjs/common';
import { AppService } from './app.service';

@Module({
  imports: [],
  providers: [
    // Imports de Servicios
    AppService
  ],
```

```
})
export class AppModule {}
```

Crea un método en el servicio para cada propósito que necesites. Uno para obtener un producto, otro para obtener un listado de productos. Uno para crear producto, para actualizar, eliminar, etc.

Servicios en NestJS

```
// src/entities/product.entity.ts

export class Product {
  id: number;
  name: string;
  description: string;
  price: number;
  stock: number;
  image: string;
}
nest g s services/products --flat

// src/services/products.service.ts
import { Injectable } from '@nestjs/common';

import { Product } from './../entities/product.entity';

@Injectable()
export class ProductsService {
  private counterId = 1;
  private products: Product[] = [
    {
      id: 1,
      name: 'Product 1',
      description: 'bla bla',
      price: 122,
      image: '',
      stock: 12,
    },
  ];

  findAll() {
    return this.products;
  }

  findOne(id: number) {
    return this.products.find((item) => item.id === id);
```

```
  }

  create(payload: any) {
    this.counterId = this.counterId + 1;
    const newProduct = {
      id: this.counterId,
      ...payload,
    };
    this.products.push(newProduct);
    return newProduct;
  }
}
// src/app.module.ts
import { Module } from '@nestjs/common';
...
import { ProductsService } from './services/products.service';

@Module({
  imports: [],
  controllers: [...],
  providers: [AppService, ProductsService], // 👈 New Service
})
export class AppModule {}
```

Contribución creada con los aportes de: Kevin Fiorentino y Christian Moreno.

11. Errores

Desarrollar una API correctamente también implica manejar los errores que sus endpoints pueden tener de manera clara para el front-end.

Manejo de errores con NestJS

NestJS implementa de forma muy sencilla la posibilidad de responder con errores al cliente que realiza las consultas. Esto lo hace con una serie de clases que implementan los códigos HTTP correctos dependiendo el tipo de error que necesites.

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

@Get('product/:idProduct')
@HttpCode(HttpStatus.OK)
async getProduct(@Param('idProduct') idProduct: string): string {
  const product = await this.appService.getProducto(idProduct);
```

```
  if (!product) {
     throw new NotFoundException(`Producto con ID #${idProduct} no encontrado.`);
  }
  return product;
}
```

Importando NotFoundException puedes arrojar un error con la palabra reservada throw indicando que un registro no fue encontrado. Esta excepción cambiará el estado HTTP 200 que envía el decorador @HttpCode(HttpStatus.OK) por un 404 que es el correspondiente para la ocasión.

También puedes lanzar errores cuando el usuario no tiene permisos para acceder a un recurso.

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

@Get('product/:idProduct')
@HttpCode(HttpStatus.OK)
async getProduct(@Param('idProduct') idProduct: string): string {
  // ...
  throw new ForbiddenException(`Acceso prohibido a este recurso.`);
}
```

O incluso lanzar errores de la familia del 5XX cuando ocurre un error inesperado en el servidor.

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

@Get('product/:idProduct')
@HttpCode(HttpStatus.OK)
async getProduct(@Param('idProduct') idProduct: string): string {
  // ...
  throw new InternalServerErrorException(`Ha ocurrido un error inesperado.`);
}
```

Explora todas las clases con estados HTTP que NestJS posee para desarrollar tus endpoints de manera profesional y manejar correctamente los errores.

SRC services

```
// src/services/products.service.ts

import { ..., NotFoundException } from '@nestjs/common';

@Injectable()
export class ProductsService {
  ...

  findOne(id: number) {
    const product = this.products.find((item) => item.id === id);
```

```
    if (!product) {
      throw new NotFoundException(`Product #${id} not found`);
    }
    return product;
  }

  update(id: number, payload: any) {
    const product = this.findOne(id);
    const index = this.products.findIndex((item) => item.id === id);
    this.products[index] = {
      ...product,
      ...payload,
    };
    return this.products[index];
  }

  remove(id: number) {
    const index = this.products.findIndex((item) => item.id === id);
    if (index === -1) {
      throw new NotFoundException(`Product #${id} not found`);
    }
    this.products.splice(index, 1);
    return true;
  }
}
// src/controllers/products.controller.ts

  @Delete(':id')
  delete(@Param('id') id: string) {
    return this.productsService.remove(+id);
  }
```

Contribución creada por: Kevin Fiorentino.

12. Pipe

Crear tus propias validaciones de datos será muy importante para segurizar tu aplicación y evitar errores inesperados.

Cómo crear custom PIPES
Crea tu propio PIPE para implementar lógica custom de validación de datos.

Crea tu primer Pipe

Con el CLI de NestJS autogenera un nuevo pipe con el comando nest generate pipe <pipe-name> o en su forma corta nest g p <pipe-name>.
Por defecto, verás un código como el siguiente.

```
import { ArgumentMetadata, Injectable, PipeTransform } from '@nestjs/common';

@Injectable()
export class ParseIntPipe implements PipeTransform {

  transform(value: any, metadata: ArgumentMetadata) {
    return value;
  }
}
```

3. Implementa la lógica del Pipe
Implementa aquí tu propia lógica de transformación y validación de datos. Ten en cuenta que si los datos no son válidos, puedes arrojar excepciones para informarle al front-end que los datos son erróneos.

```
import { ArgumentMetadata, Injectable, PipeTransform, BadRequestException } from
'@nestjs/common';

@Injectable()
export class ParseIntPipe implements PipeTransform {

  transform(value: string, metadata: ArgumentMetadata) {
    const finalValue = parseInt(value, 10);
    if (isNaN(finalValue)) {
      throw new BadRequestException(`${value} no es un número.`);
    }
    return finalValue;
  }
}
```

3. Importa y utiliza el Pipe
Finalmente, implementa tu custom PIPE en el controlador.

```
import { ParseIntPipe } from './pipes/parse-int.pipe';

@Get('product/:idProduct')
getProduct(@Param('idProduct', ParseIntPipe) idProduct: string): string {
  // ...
}
```

Puedes desarrollar la lógica para validar y transformar los datos que más se adecue a tus necesidades. Es fundamental no permitir el ingreso de datos erróneos a tus controladores. Por eso, los pipes son una capa previa a los controladores para realizar esta validación.

Generar un pipe con nest g pi common/parse-int

```
// src/common/parse-int.pipe.ts

import {
  ArgumentMetadata,
  Injectable,
  PipeTransform,
  BadRequestException,
} from '@nestjs/common';

@Injectable()
export class ParseIntPipe implements PipeTransform {
  transform(value: string, metadata: ArgumentMetadata) {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      throw new BadRequestException(`${value} is not an number`);
    }
    return val;
  }
}
```

Contribución creada por: Kevin Fiorentino.

13. Propio Pipe

Crear tus propias validaciones de datos será muy importante para segurizar tu aplicación y evitar errores inesperados.

Cómo crear custom PIPES

Crea tu propio PIPE para implementar lógica custom de validación de datos.

1. Crea tu primer Pipe
Con el CLI de NestJS autogenera un nuevo pipe con el comando nest generate pipe <pipe-name> o en su forma corta nest g p <pipe-name>.
Por defecto, verás un código como el siguiente.

```
import { ArgumentMetadata, Injectable, PipeTransform } from '@nestjs/common';

@Injectable()
```

```
export class ParseIntPipe implements PipeTransform {

  transform(value: any, metadata: ArgumentMetadata) {
    return value;
  }
}
```

3. Implementa la lógica del Pipe

Implementa aquí tu propia lógica de transformación y validación de datos. Ten en cuenta que si los datos no son válidos, puedes arrojar excepciones para informarle al front-end que los datos son erróneos.

```
import { ArgumentMetadata, Injectable, PipeTransform, BadRequestException } from
'@nestjs/common';

@Injectable()
export class ParseIntPipe implements PipeTransform {

  transform(value: string, metadata: ArgumentMetadata) {
    const finalValue = parseInt(value, 10);
    if (isNaN(finalValue)) {
      throw new BadRequestException(`${value} no es un número.`);
    }
    return finalValue;
  }
}
```

3. Importa y utiliza el Pipe

Finalmente, implementa tu custom PIPE en el controlador.

```
import { ParseIntPipe } from './pipes/parse-int.pipe';

@Get('product/:idProduct')
getProduct(@Param('idProduct', ParseIntPipe) idProduct: string): string {
    // ...
}
```

Puedes desarrollar la lógica para validar y transformar los datos que más se adecue a tus necesidades. Es fundamental no permitir el ingreso de datos erróneos a tus controladores. Por eso, los pipes son una capa previa a los controladores para realizar esta validación.

Generar un pipe con nest g pi common/parse-int

```
// src/common/parse-int.pipe.ts

import {
  ArgumentMetadata,
  Injectable,
```

```
  PipeTransform,
  BadRequestException,
} from '@nestjs/common';

@Injectable()
export class ParseIntPipe implements PipeTransform {
  transform(value: string, metadata: ArgumentMetadata) {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      throw new BadRequestException(`${value} is not an number`);
    }
    return val;
  }
}
```

Contribución creada por: Kevin Fiorentino.

Creando Data Transfers Objects

NestJS utiliza el concepto de Objetos de Transferencia de Datos, o simplemente abreviado como DTO, para el tipado de datos y su segurización.

Qué son objetos de transferencia de datos o data transfers objects
Los DTO no son más que clases customizadas que tu mismo puedes crear para indicar la estructura que tendrán los objetos de entrada en una solicitud.

1. Creando DTO
Crea un nuevo archivo que por lo general lleva como extensión .dto.ts para indicar que se trata de un DTO.

```
// products.dto.ts
export class CreateProductDTO {
  readonly name: string;
  readonly description: string;
  readonly price: number;
  readonly image: string;
}
```

La palabra reservada readonly es propia de TypeScript y nos asegura que dichos datos no sean modificados.

Crea tantos atributos como tu clase CreateProductDTO necesite para dar de alta un nuevo producto.

2. Importando DTO

Importa la clase en tu controlador para tipar el Body del endpoint POST para la creación de un producto.

```
import { CreateProductDTO } from 'products.dto.ts';

@Post('product')
createProducto(@Body() body: CreateProductDTO): any {
   // ...
}
```

De esta forma, ya conoces la estructura de datos que tendrá el parámetro body previo a la creación de un producto.

SRC: DTOS

```
// src/dtos/products.dtos.ts
export class CreateProductDto {
  readonly name: string;
  readonly description: string;
  readonly price: number;
  readonly stock: number;
  readonly image: string;
}

export class UpdateProductDto {
  readonly name?: string;
  readonly description?: string;
  readonly price?: number;
  readonly stock?: number;
  readonly image?: string;
}
// src/controllers/products.controller.ts
export class ProductsController {
  @Post()
  create(@Body() payload: CreateProductDto) { // 👈 Dto
    ...
  }

  @Put(':id')
  update(
    @Param('id') id: string,
    @Body() payload: UpdateProductDto  // 👈 Dto
   ) {
   ...
  }
```

```
}
// src/services/products.service.ts
export class ProductsService {
  create(payload: CreateProductDto) { // 👈 Dto
    ...
  }

  update(id: number, payload: UpdateProductDto) { // 👈 Dto
    ...
  }

}
```

Contribución creada por: Kevin Fiorentino.

14. Validando parámetros con class-validator y mapped-types

Los DTO no solo sirven para tipar y determinar la estructura de los datos de entrada de un endpoint, también pueden contribuir en la validación de los datos y en la entrega de mensajes al front-end en caso de error en los mismos.

Validación de datos con DTO

Utiliza el comando npm i class-validator class-transformer para instalar dos dependencias que nos ayudarán en la validación de los datos.
Estas librerías traen un set de decoradores para las propiedades de los DTO y así validar los tipos de datos de entrada.

```
import { IsNotEmpty, IsString, IsNumber, IsUrl } from 'class-validator';

export class CreateProductDTO {
  @IsNotEmpty()
  @IsString()
  readonly name: string;

  @IsNotEmpty()
  @IsString()
  readonly description: string;

  @IsNotEmpty()
  @IsNumber()
  readonly price: number;
```

```
  @IsNotEmpty()
  @IsUrl()
  readonly image: string;
}
```

Estas validaciones contribuyen en la experiencia de desarrollo devolviendo mensajes al front-end sobre qué datos están faltando o cuáles no son correctos. Por ejemplo, si en el Body de la petición enviamos.

```
{
  "name": "Nombre producto",
  "price": 100,
  "image": "imagen"
}
```

El servidor nos devolverá el siguiente mensaje:

```
{
  "statusCode": 400,
  "message": [
    "description should not be empty",
    "description must be a string",
    "image must be an URL address"
  ],
  "error": "Bad Request"
}
```

Indicando que la solicitud espera de forma obligatoria un campo description del tipo string y un campo image con una URL.

Cómo reutilizar código de los DTO

A medida que tu aplicación crezca, tendrás que crear muchos DTO, para la creación de un producto, edición, filtros, etc. Una buena práctica es la reutilización de las clases DTO que ya tengas implementado para no repetir propiedades.

Instala la dependencia **npm i @nestjs/mapped-types** que nos ayudará con la reutilización de código de la siguiente manera.

```
import { IsNotEmpty, IsString, IsNumber, IsUrl } from 'class-validator';
import { PartialType, OmitType } from '@nestjs/mapped-types';

export class CreateProductDTO {

  @IsNotEmpty()
  @IsString()
  readonly name: string;
```

```
  @IsNotEmpty()
  @IsString()
  readonly description: string;

  @IsNotEmpty()
  @IsNumber()
  readonly price: number;

  @IsNotEmpty()
  @IsUrl()
  readonly image: string;
}

export class UpdateProductDto extends PartialType(
  OmitType(CreateProductDTO, ['name']),
) {}
```

Importa PartialType y OmitType desde @nestjs/mapped-types.

PartialType permite extender una clase de otra y que todos sus campos sean opcionales. Así, el DTO UpdateProductDto no tiene como obligatorio sus campos y es posible editar todos o solo uno.

Por otro lado, OmitType, permite la omisión de campos haciendo que cierta cantidad de ellos no formen parte del DTO en el caso de que dichos campos no deban ser editados.

Instalar npm i class-validator class-transformer @nestjs/mapped-types

```
// src/dtos/products.dtos.ts
import {
  IsString,
  IsNumber,
  IsUrl,
  IsNotEmpty,
  IsPositive,
} from 'class-validator';
import { PartialType } from '@nestjs/mapped-types';

export class CreateProductDto {
  @IsString()
  @IsNotEmpty()
  readonly name: string;

  @IsString()
  @IsNotEmpty()
  readonly description: string;
```

```
  @IsNumber()
  @IsNotEmpty()
  @IsPositive()
  readonly price: number;

  @IsNumber()
  @IsNotEmpty()
  @IsPositive()
  readonly stock: number;

  @IsUrl()
  @IsNotEmpty()
  readonly image: string;
}

export class UpdateProductDto extends PartialType(CreateProductDto) {}
// src/main.ts
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  ...
  app.useGlobalPipes(new ValidationPipe());
  ...
}
bootstrap();
```

Contribución creada por: Kevin Fiorentino.

15. Evitar Parametros incorrectos.

Los DTO ayudan con el tipado y la validación de datos, además de indicar la obligatoriedad de los mismos para que los registros se creen completos. Es importante también evitar que haya datos que no deben estar en las solicitudes, ya que podrían ser ataques maliciosos.

Cómo hacer la prohibición de datos

Busca el archivo main.ts que contiene el bootstrap de tu aplicación, es decir, el punto inicial de la misma. Agrega aquí la siguiente configuración.

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

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

```
  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true,                // Ignorar datos que no esten en los DTO
      forbidNonWhitelisted: true,     // Lanzar error si existen datos prohibidos
      disableErrorMessages: true,     // Desabilitar mensajes de error (producción)
    })
  );
  await app.listen(process.env.PORT || 3000);
}
bootstrap();
```

Importa ValidationPipe desde @nestjs/common y configura en true las propiedades whitelist para ignorar datos que no estén en el DTO. Usa forbidNonWhitelisted para lanzar errores si existen datos prohibidos y disableErrorMessages que es recomendable activarlo solo en producción para no enviar mensajes de error y no dar información al front-end.

De esta simple manera, tus endpoints gracias a los DTO son súper profesionales, seguros y contribuyen a una buena experiencia de desarrollo.

```
// src/main.ts

app.useGlobalPipes(
  new ValidationPipe({
    whitelist: true,
    forbidNonWhitelisted: true,
  }),
);
```

Contribución creada por: Kevin Fiorentino.

16. Retro Controladores

Ya tienes todas las bases y hemos avanzado en el proyecto. Ahora llego la hora del **RETO** y este será hacer los demás controladores, así es debes crear los controladores, DTOs y servicios para:

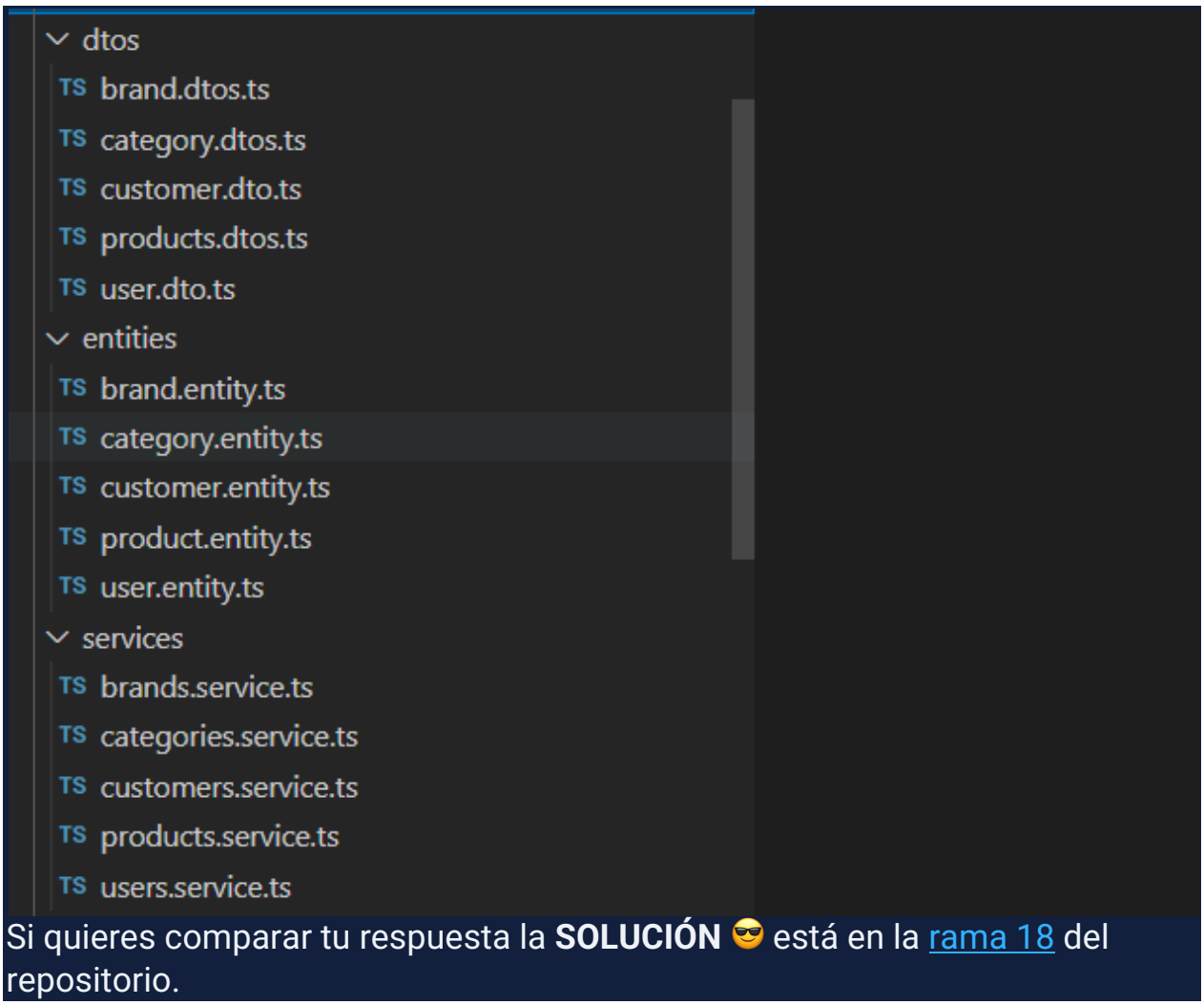
- Products
- Categories
- Brands
- Users
- Customers

Sin embargo en esta lectura vamos a ver como quedaría, y finalmente deberías haber terminado con una AppModule parecido a este:

```
...

@Module({
  imports: [],
  controllers: [
    AppController,
    ProductsController, // 👈
    CategoriesController, // 👈
    BrandsController,  // 👈
    CustomerController,  // 👈
    UsersController,  // 👈
  ],
  providers: [
    AppService,
    ProductsService,  // 👈
    CategoriesService,  // 👈
    BrandsService,  // 👈
    CustomersService,  // 👈
    UsersService,  // 👈
  ],
})
export class AppModule {}
```

Debes haber terminado con los controladores para Products, Categories, Brands, Users y Customers con sus servicios, entidades y DTOs respectivos, una vista general sería así:



Si quieres comparar tu respuesta la **SOLUCIÓN** 😎 está en la rama 18 del repositorio.

17. ccc