Índice

[**¿Qué es NestJS?** 1](#_Toc130569417)

[**¿Por qué usar NestJS?** 1](#_Toc130569418)

[**Preparando el Entorno** 1](#_Toc130569419)

[**Primer proyecto NestJS** 2](#_Toc130569420)

[**Estructura de aplicaciones en NestJS** 3](#_Toc130569421)

[**Aplicaciones y carpetas de NestJS** 3](#_Toc130569422)

[**Introducción a controladores** 5](#_Toc130569423)

[**Qué son los controladores en NestJS** 5](#_Toc130569424)

[**Estructura de un controlador** 5](#_Toc130569425)

[**Verbos HTTP** 8](#_Toc130569426)

[**GET: Cómo recibir parámetros** 9](#_Toc130569427)

[**Obtención de datos con GET** 9](#_Toc130569428)

[**GET: Parámetros query** 10](#_Toc130569429)

[**Parámetros de ruta vs Parámetros query** 11](#_Toc130569430)

[**Parámetros de ruta** 11](#_Toc130569431)

[**Parámetros query** 11](#_Toc130569432)

[**Evitando el bloqueo de rutas** 11](#_Toc130569433)

[**Orden de endpoint** 12](#_Toc130569434)

[**Separación de responsabilidades** 14](#_Toc130569435)

[**Qué es la responsabilidad única** 14](#_Toc130569436)

[**Responsabilidades en NestJS** 14](#_Toc130569437)

[**Instalación de Postman o Insomnia** 15](#_Toc130569438)

[**Qué es el método POST** 15](#_Toc130569439)

[**Qué es el método Post** 15](#_Toc130569440)

[**Crear registro con Post** 16](#_Toc130569441)

[**Métodos PUT y DELETE para editar y eliminar** 17](#_Toc130569442)

[**Actualización de datos con PUT** 17](#_Toc130569443)

[**Eliminar datos con DELETE** 18](#_Toc130569444)

[**Códigos de estado o HTTP response status codes** 20](#_Toc130569445)

[**Cuáles son los códigos HTTP** 20](#_Toc130569446)

[**Cómo manejar los códigos de estado HTTP con NestJS** 20](#_Toc130569447)

[**Introducción a servicios: crea tu primer servicio** 22](#_Toc130569448)

[**Qué son los servicios en NestJS** 23](#_Toc130569449)

[**Primer servicio con NestJS** 23](#_Toc130569450)

[**Implementando servicios en tu controlador** 25](#_Toc130569451)

[**Inyección de dependencias** 25](#_Toc130569452)

[**Controladores y servicios** 25](#_Toc130569453)

[**Controllers** 27](#_Toc130569454)

[**Refactor update** 28](#_Toc130569455)

[**Manejo de errores con throw y NotFoundException** 28](#_Toc130569456)

[**Manejo de errores con NestJS** 28](#_Toc130569457)

[**Introducción a pipes: usa tu primer pipe** 30](#_Toc130569458)

[**Casos de uso de PIPES** 30](#_Toc130569459)

[**Implementando tu primer PIPE** 31](#_Toc130569460)

[**Crea tu propio pipe** 31](#_Toc130569461)

[**Cómo crear custom PIPES** 32](#_Toc130569462)

[1. **Crea tu primer Pipe** 32](#_Toc130569463)

[2. **Implementa la lógica del Pipe** 32](#_Toc130569464)

[3. **Importa y utiliza el Pipe** 33](#_Toc130569465)

[**Creando Data Transfers Objects** 33](#_Toc130569466)

[**Qué son objetos de transferencia de datos o data transfers objects** 34](#_Toc130569467)

[**Creando DTO** 34](#_Toc130569468)

[**Importando DTO** 34](#_Toc130569469)

[**Validando parámetros con class-validator y mapped-types** 37](#_Toc130569470)

[**Validación de datos con DTO** 37](#_Toc130569471)

[**Cómo reutilizar código de los DTO** 38](#_Toc130569472)

[**Cómo evitar parámetros incorrectos** 42](#_Toc130569473)

[**Cómo hacer la prohibición de datos** 42](#_Toc130569474)

# **¿Qué es NestJS?**

**NestJS** es un potente framework de NodeJS para la construcción de aplicaciones del lado del servidor.

## **¿Por qué usar NestJS?**

**NestJS** ha crecido mucho en los últimos años y se ha convertido en una de las principales opciones que los desarrolladores eligen para la programación de APIs con NodeJS.

Gracias a la utilización de **TypeScript** y la Programación Orientada a Objetos, es un framework ideal para iniciar en el desarrollo backend.

Si vienes de frontend y conoces Angular, te sentirás como en casa, ya que los desarrolladores de NestJS se inspiraron en Angular para desarrollar este maravilloso framework.

NestJS también te dará patrones para:

* Solid.
* Typescript.
* Orientado a objetos.
* Programación funcional.
* Programación reactiva.

# **Preparando el Entorno**

Asegúrate de tener instalado en tu ordenador tanto **NodeJS** como **NPM**. Se recomienda al menos NodeJS versión 10 u 12.

* **Consejo**: Siempre utilizar versiones pares de NodeJS ya que éstas tienen soporte a largo plazo.

Puedes verificar la versión de Node con el comando **node --version** y de NPM con el comando **npm --version**.

Una vez los hayas instalado, es hora de instalar el CLI de NestJS de forma global con el comando **npm install -g @nest/cli**.

**CLI**: **C**ommand **L**ine **I**nterface o Interfaz de Línea de Comandos por sus siglas en español.

Una vez instalado el CLI, verifica que el proceso es correcto con el comando **nest --version**.



## **Primer proyecto NestJS**

Para crear tu primer proyecto con este framework basta con utilizar el comando **nest new <project-name>**. El CLI te realizará una pregunta sobre qué gestor de dependencias quieres utilizar.

Para este ejemplo, escogeremos **NPM**. Luego de algunos segundos, tu primer proyecto estará listo.

Ahora, utiliza el comando **npm run start** para levantar el servidor de pruebas básico que trae consigo NestJS y finalmente, ingresa a **http://localhost:3000/** para visualizar tu primer “**Hola Mundo**” con esta tecnología.



# **Estructura de aplicaciones en NestJS**

**NestJS** posee desde el inicio de un proyecto varios directorios y archivos que se generan por defecto al crearlo. Veamos para que te sirve cada uno.

## **Aplicaciones y carpetas de NestJS**

De arriba hacia abajo:

1. **Node\_modules**

Todo proyecto de Javascript posee este directorio donde se almacenan las librerías y dependencias que se descarguen con NPM.

1. **SRC**

Directorio principal del proyecto donde encontramos:

* + **app.controller.spec.ts**: Archivo de pruebas unitarias del controlador con el mismo nombre.
  + **app.controller.ts**: Controlador que contiene endpoints a diferentes recursos.
  + **app.module.ts**: Módulo principal de toda la aplicación NestJS.
  + **app.service.ts**: Servicio consumido por los controladores para determinados propósitos.
  + **main.ts**: Archivo Core de la aplicación NestJS donde se realizan **configuraciones** e **imports** básicos para su funcionamiento.

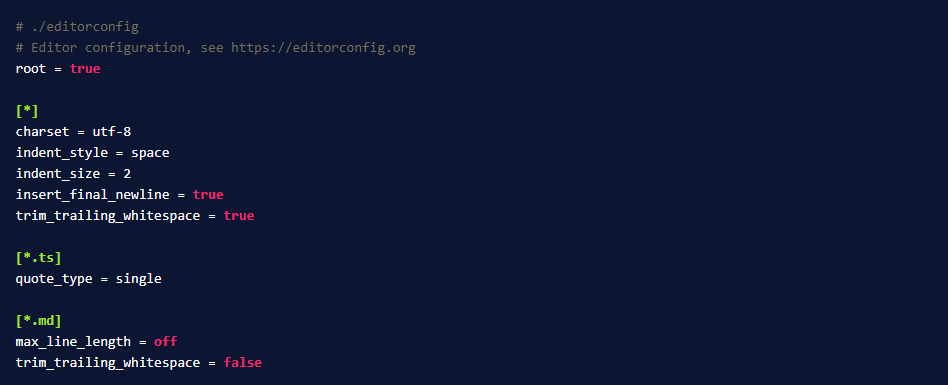
1. **Test**

Directorio de pruebas unitarias y de integración. NestJS utiliza por defecto **Jest** para escribir las pruebas.

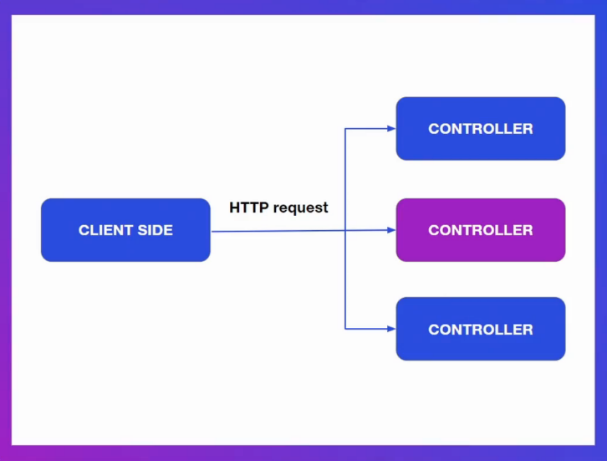
* **.editorconfig**: Este archivo no viene por defecto, pero se recomienda crearlo e instalar el plugin con el mismo nombre en el editor. Permite autoformatear los archivos, espacios, indentación, etc.
* **.eslintrc.js**: Permite la configuración de un analizador de código para detectar tempranamente errores y resolverlos. Requiere instalación de un plugin en el editor (ESLint).
* **.gitignore**: Indicarle a GIT qué archivos/directorios ignorar.
* **.prettierrc**: Archivo de configuración para el autoformateo de código. Requiere instalación de un plugin en el editor (Prettier).
* **nest-cli.json**: Archivo con configuraciones de NestJS. Algunos plugins del framework requieren de configuraciones adicionales en este archivo.
* **package-lock.json**: Describe las dependencias exactas que se generaron en la instalación del proyecto.
* **package.json**: Archivo para el manejo de dependencias, scripts y metadatos relevantes para el proyecto.
* **README.md**: Archivo markdown para la documentación del proyecto.
* **tsconfig.build.json**: Archivo principal para la configuración de TypeScript.
* **tsconfig.json**: Extensión con más configuraciones de TypeScript.

Poco a poco irás conociendo la funcionalidad de cada archivo. ¡No te preocupes y continuemos!

**Archivo ./editorconfig**



# **Introducción a controladores**

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

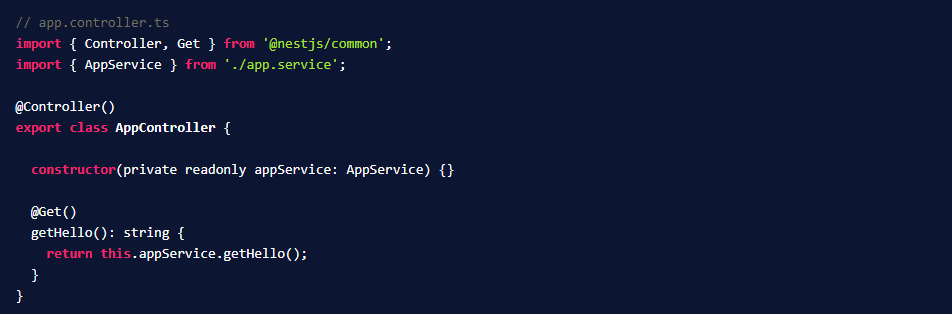
## **Qué son los controladores en NestJS**

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

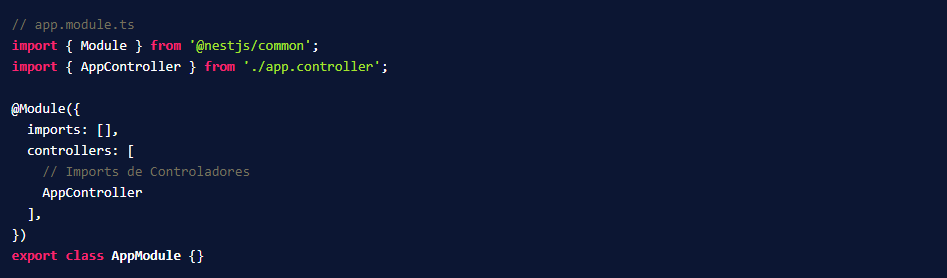
## **Estructura de un controlador**

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.



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

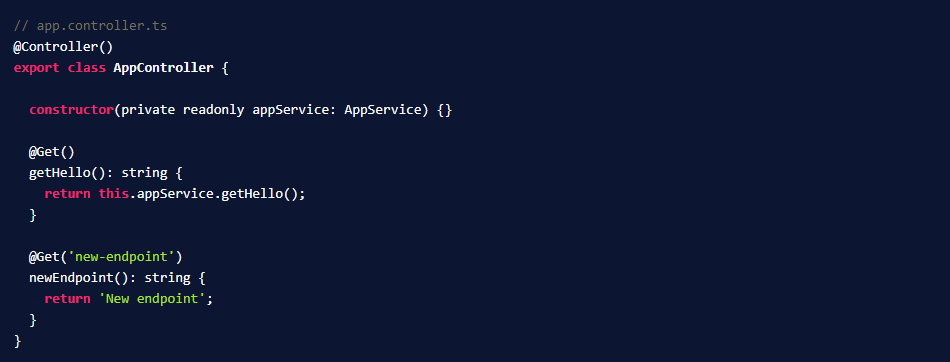


El controlador importa un **servicio** que son los responsables de la **lógica** y **obtención de datos** desde una BBDD que el controlador requiere.



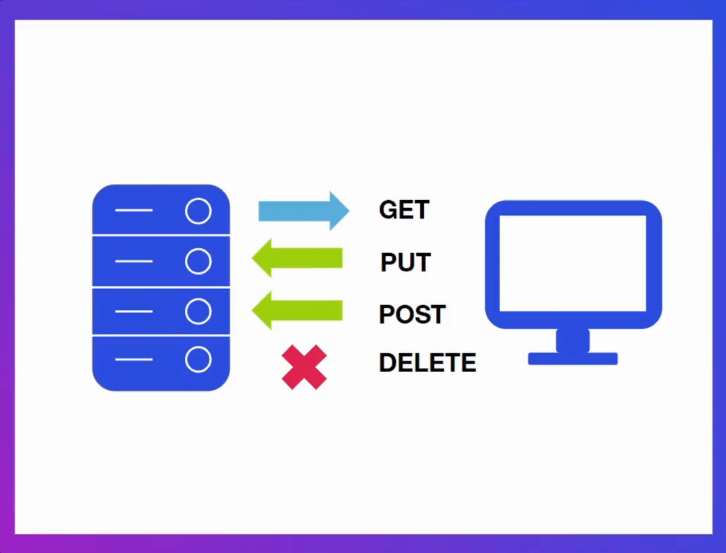
Puedes correr el servidor de NestJS con el comando **npm run** **start:dev** e ingresar a la ruta **localhost:3000/** para visualizar el contenido que el controlador envía.

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.



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

## **Verbos HTTP**



# **GET: Cómo recibir parámetros**

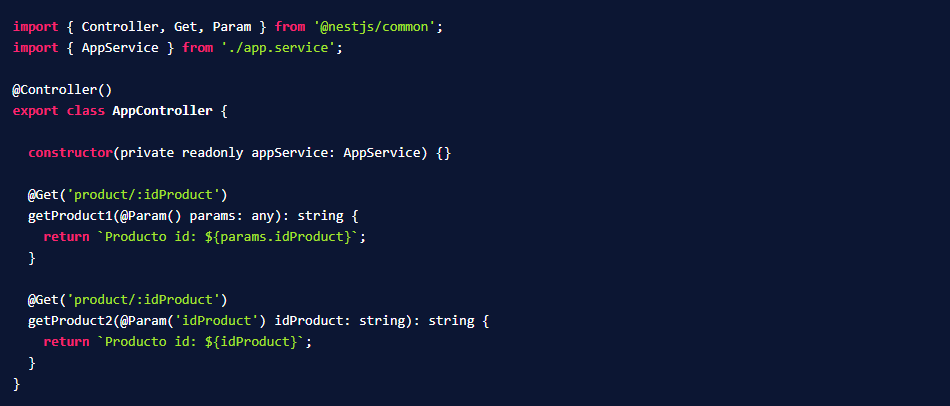
Existen diferentes tipos de endpoints que se identifican a través de los [Verbos HTTP](https://developer.mozilla.org/es/docs/Web/HTTP/Methods). Cada uno con un propósito determinado siguiendo el protocolo.

## **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.

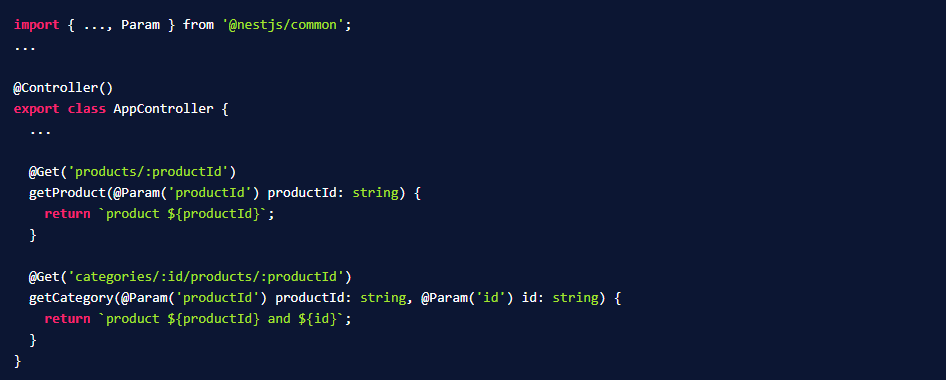
Es frecuente la necesidad de que este tipo de endpoints también reciban información dinámica en las URL como el identificador de un producto.

Para capturar estos datos en NestJS, tienes que importar el decorador Param desde **@nestjs/common** y emplearlo de la siguiente manera en tus endpoints.



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.



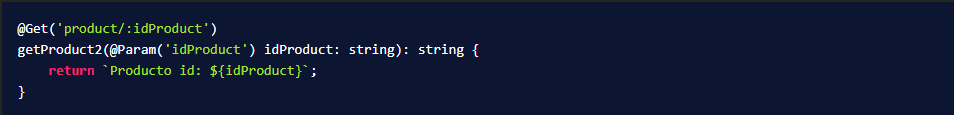
# **GET: Parámetros query**

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.

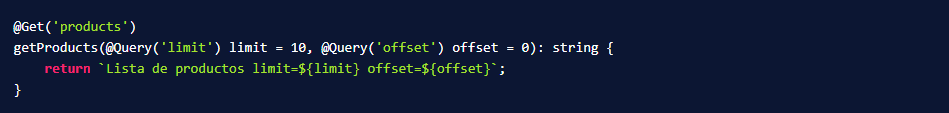
## **Parámetros de ruta**



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

## **Parámetros query**

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**.



**Su principal diferencia es que los parámetros de consulta o query 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 o query **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:



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.

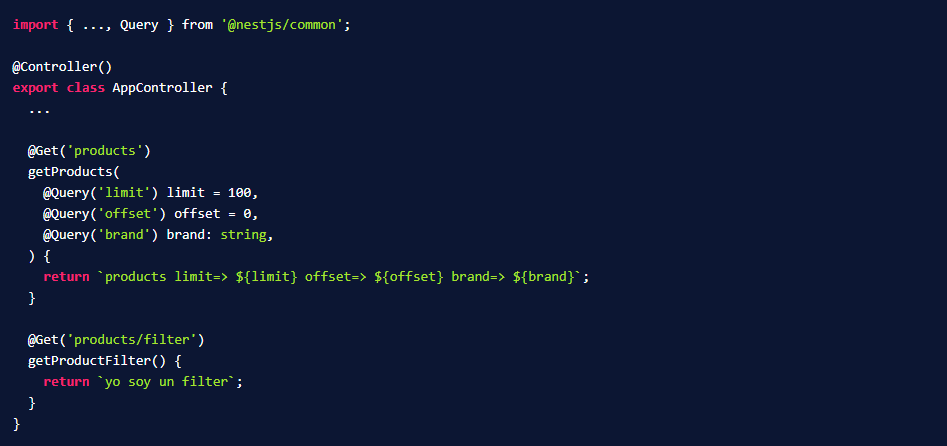
Se soluciona de forma muy sencilla, invirtiendo el orden de los mismos. **Coloca los endpoints dinámicos en segundo lugar** para que no ocasionen problemas.

## **Orden de endpoint**

1. Estáticos.
2. Dinámicos.



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



# **Separación de responsabilidades**

**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.

# **Instalación de Postman o Insomnia**

Para poder probar nuestra API de forma más productiva vamos a tener que usar algún cliente de APIs que nos permita hacer las pruebas de funcionamiento de lo que estás construyendo, los dos clientes más famosos son Insomnia y Postman y vas a necesitar de alguno de ellos para ir probando cada característica que vamos a ir construyendo en nuestro servicio.

[**Insomnia**](https://insomnia.rest/download)

Como cliente para probar APIs tiene grandes características, destaca principalmente por una interfaz limpia y sencilla, pero a la vez potente, en donde puede configurar **ambientes**, **exportar** e **importar**, **gran soporte con GraphQL**, etc.

[**Postman**](https://www.postman.com/downloads/)

Es uno de los más usados y legendariamente nos ha acompañado por mucho tiempo, además cuenta con características similares a Insomnia como: **exportar** e **importar**, **ambientes**, **entornos**, y provee una API para hacer testing muy potente.

# **Qué es el método POST**

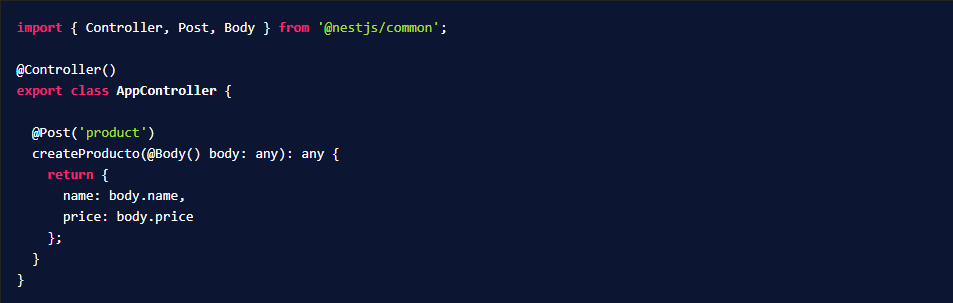
Así como el verbo HTTP **GET** se utiliza para la **obtención de datos**, el verbo HTTP **Post** se utiliza para la **creación de los mismos** previamente.

## **Qué es el método Post**

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

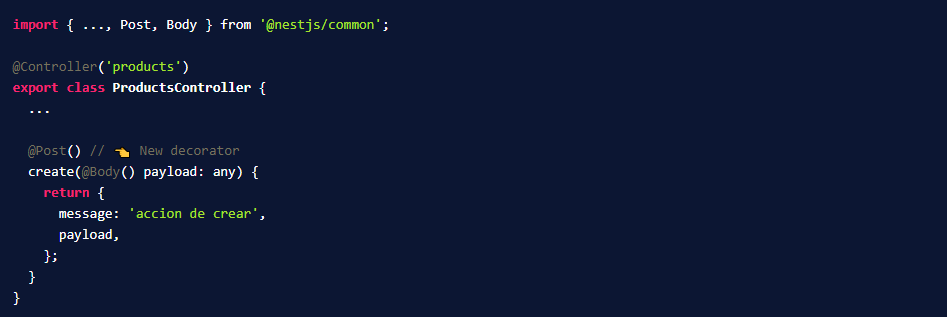
## **Crear registro con Post**

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 frontend en el cuerpo del mensaje**.



**Un buen endpoint del tipo POST tiene que devolver el registro completo recientemente insertado en la BBDD para que el frontend 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**.

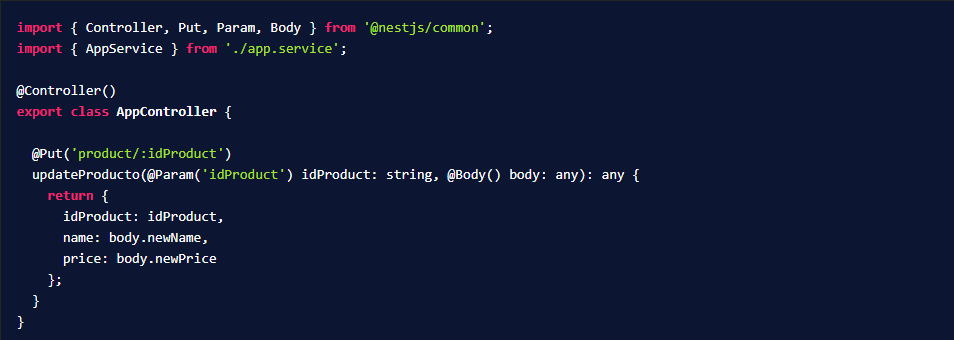


# **Métodos PUT y DELETE para editar y eliminar**

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.



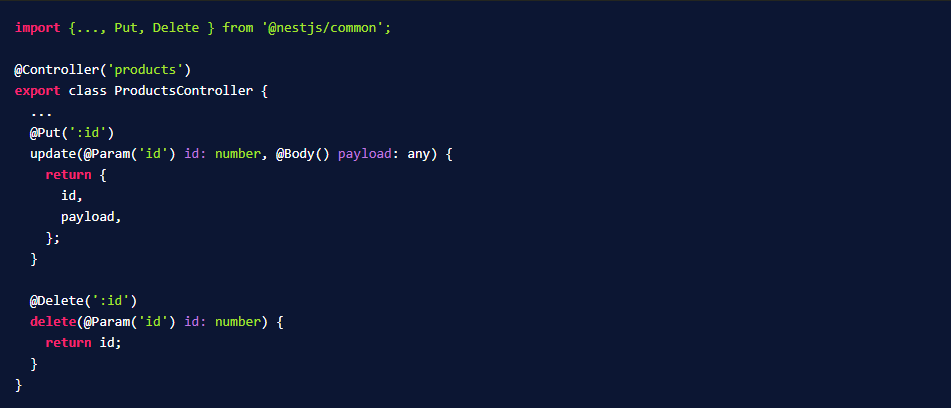
El **ID** suele recibirse por parámetros de URL para que sea obligatorio, mientras que reservamos el cuerpo del mensaje para los datos actualizados. Finalmente, retornamos el registro completo luego de ser actualizado.

## **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.



**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.



# **Códigos de estado o HTTP response status codes**

El protocolo **HTTP** tiene estandarizado una [lista de códigos](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status) 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)

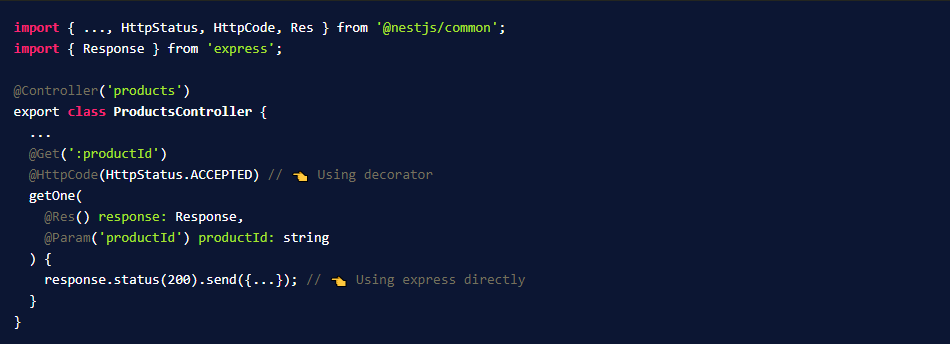
## **Cómo manejar los códigos de estado HTTP con NestJS**

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 que código pertenece a cada tipo de respuesta.

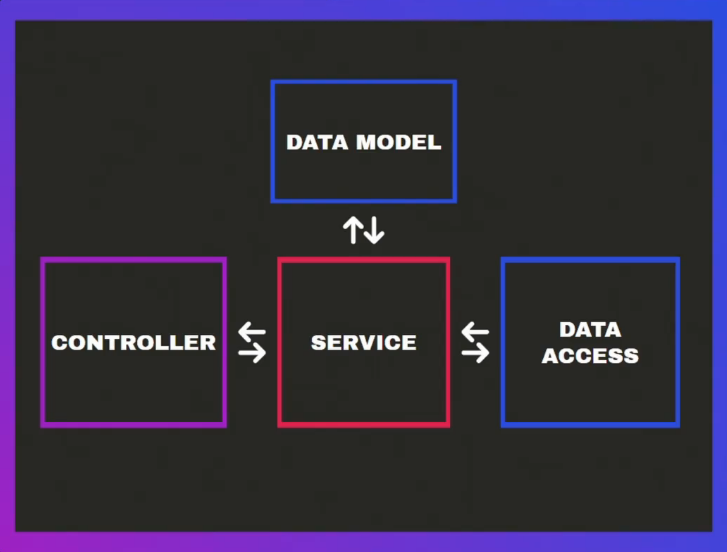


El enumerado **HttpStatus.OK** indica código de estado **200** que es el que suele devolver por defecto todos los endpoints cuando la operación sale exitosamente. Los endpoints **POST** suelen devolver **HttpStatus.CREATED** o código **201** para indicar la creación exitosa del registro.



# **Introducción a servicios: crea tu primer servicio**

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.

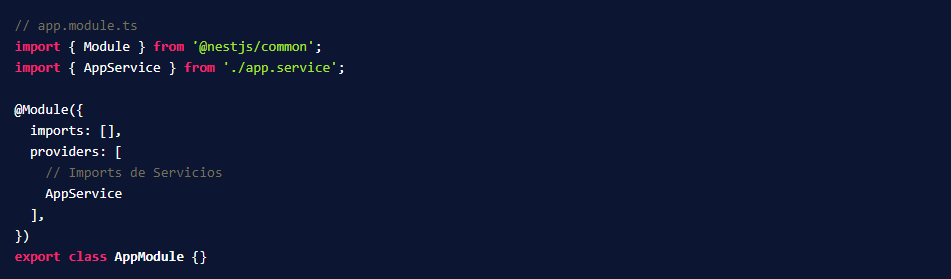
Un servicio tiene la responsabilidad de gestionar el trabajo con los datos de la aplicación, de modo que realiza las operaciones para obtener esos datos, modificarlos, etc.

## **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>**.



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.



# **Implementando servicios en tu controlador**

Los **servicios** son el otro 50% de los controladores. Podría decirse que un controlador **siempre hará uso de uno o más servicio**s para implementar lógica de negocio. Veamos cómo se relacionan.

## **Inyección de dependencias**

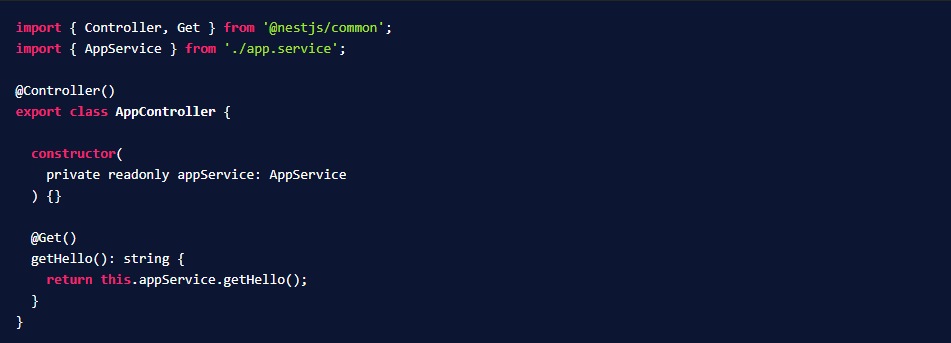
Antes de hablar de la relación entre **servicios** y **controladores**, hay que hablar del patrón de diseño que NestJS utiliza internamente.

Imagínate que tienes un **Servicio A** que utiliza el **Servicio B** y este a su vez utiliza el **Servicio C**. Si tuvieses que instanciar el **Servicio A**, primero deberías instanciar el **C** para poder instanciar el **B** y luego sí hacerlo con el **A**. Se vuelve confuso y poco escalable si en algún momento también tienes que instanciar el **Servicio D** o **E**.

La inyección de dependencias llega para solucionar esto, resolver las dependencias de una clase por nosotros. Cuando instanciamos en el constructor el **Servicio A**, NestJS internamente crea automáticamente la instancia del **servicio B** y **C** sin que nosotros nos tengamos que preocupar por estos.

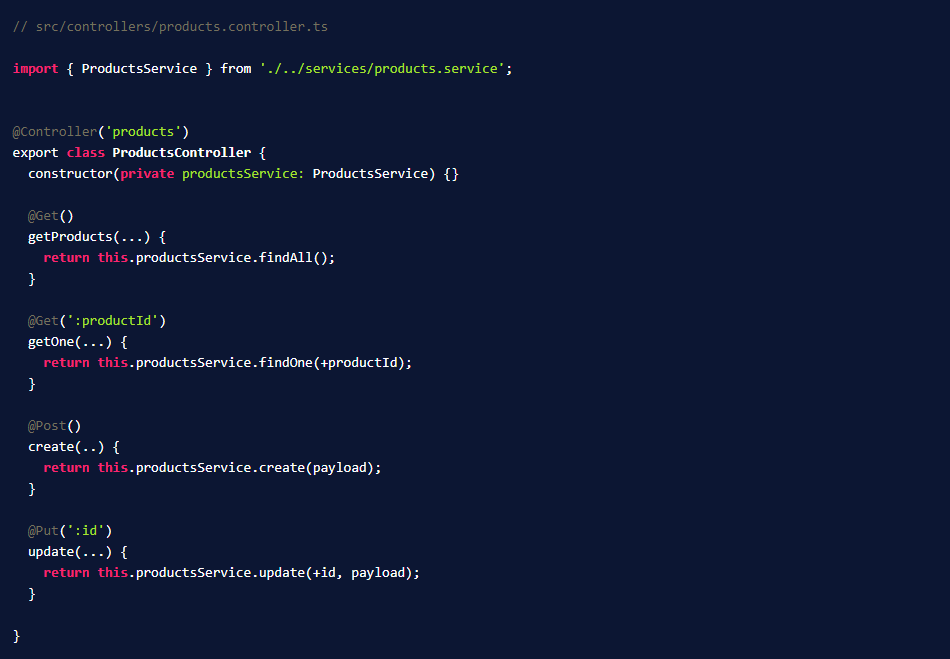
## **Controladores y servicios**

Los controladores inyectan los servicios desde el constructor. De esta manera, cada endpoint puede hacer uso de la lógica del servicio.

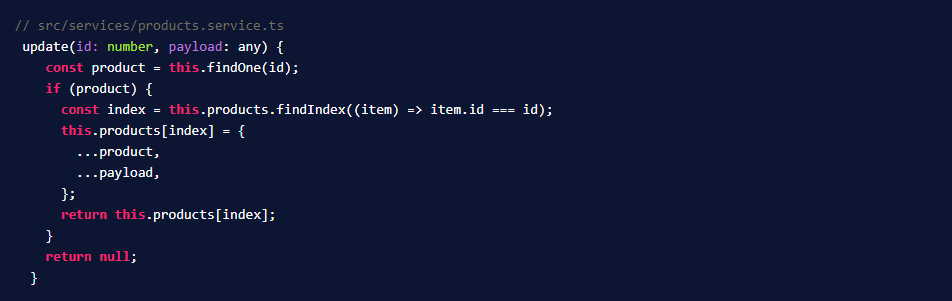


Importa los servicios que necesites, pero hazlo de una manera controlada para mantener la escalabilidad de tu proyecto. Si necesitas importar 20 servicios en un mismo controlador, tal vez tengas que mejorar la estructura del proyecto.

## **Controllers**



## **Refactor update**

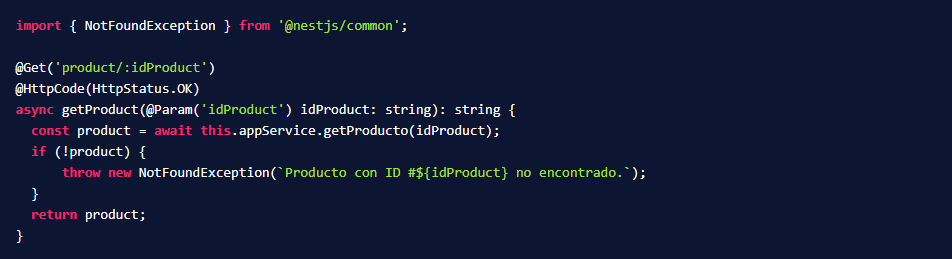


# **Manejo de errores con throw y NotFoundException**

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

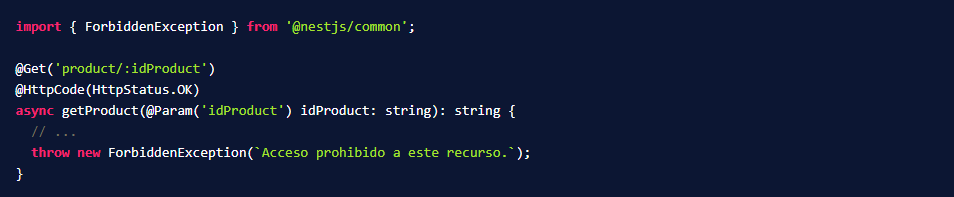
## **Manejo de errores con NestJS**

**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.

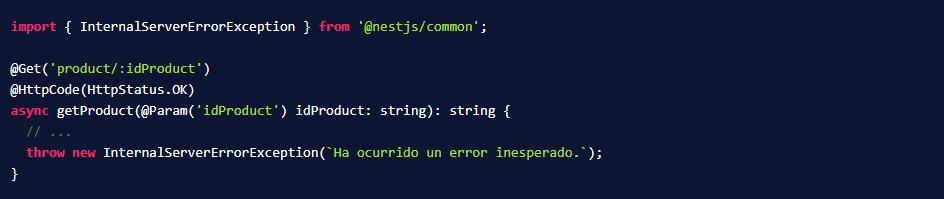


Importando **NotFoundException** puedes arrojar un error con la palabra reservada **throw** indicando que un registro no fue encontrado. Esta excepción cambiará el estado **HTTP 200** que envía el decorador **@HttpCode(HttpStatus.OK)** por un **404** que es el correspondiente para la ocasión.

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



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



Explora todas las [clases con estados HTTP](https://docs.nestjs.com/exception-filters#built-in-http-exceptions) que NestJS posee para desarrollar tus endpoints de manera profesional y manejar correctamente los errores.

# **Introducción a pipes: usa tu primer pipe**

NestJS utiliza el concepto de **PIPES** para la **validación** y **transformación** de los datos antes del ingreso de estos a un controlador.

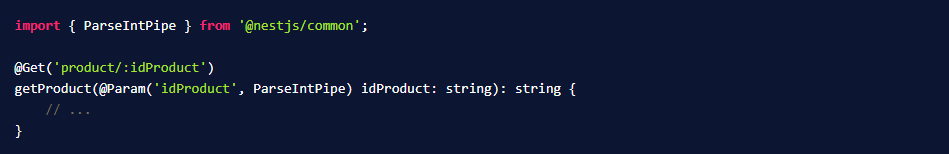
## **Casos de uso de PIPES**

Los **pipes** tienen dos casos de uso típicos:

* **Transformación**: Transforma los datos de entrada a la forma deseada (por ejemplo, de cadena a entero).
* **Validación**: Evalúa los datos de entrada y, si son válidos, simplemente los pasa sin cambios, de lo contrario, lanza una excepción cuando los datos son incorrectos.

## **Implementando tu primer PIPE**

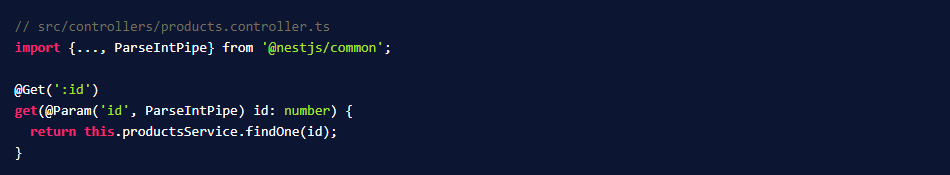
NestJS ya trae consigo una serie de **pipes** que puedes utilizar para la manipulación de datos. Impórtalos desde **@nestjs/common** y úsalos de la siguiente manera.



El pipe **ParseIntPipe**, agrégalo como segundo parámetro del decorador **Param** para transformar el parámetro **idProduct** y asegurar que este sea un número entero.

De no serlo, arrojará un error y al mismo tiempo estás protegiendo tu aplicación de datos erróneos o maliciosos.

Explora [todos los PIPES](https://docs.nestjs.com/pipes#built-in-pipes) que NestJS ya tiene preparados para ti.



# **Crea tu propio pipe**

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

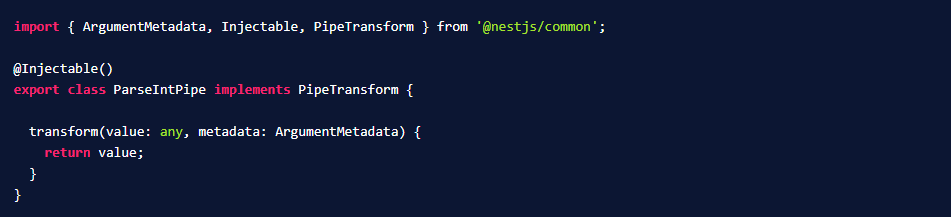
## **Cómo crear custom PIPES**

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

## **Crea tu primer Pipe**

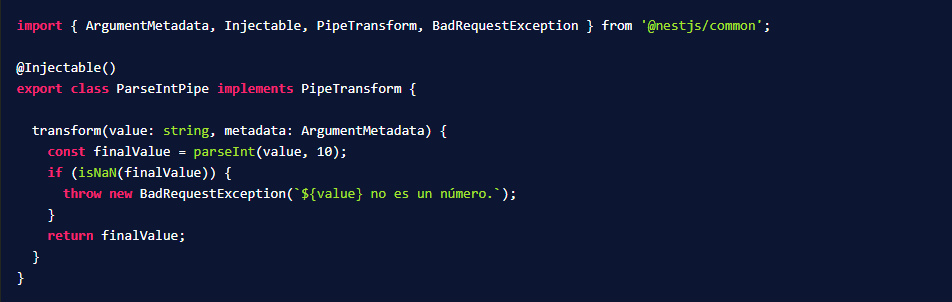
Con el **CLI** de NestJS autogenera un nuevo **pipe** con el comando **nest generate pipe <pipe-name>** o en su forma corta **nest g p <pipe-name>**.

Por defecto, verás un código como el siguiente.



## **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 frontend que los datos son erróneos.



## **Importa y utiliza el Pipe**

Finalmente, implementa tu custom **PIPE** en el controlador.

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.

# **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 tú mismo puedes crear para indicar la estructura que tendrán los objetos de entrada en una solicitud. **Esto nos ayuda a no cometer errores cuando estemos desarrollando, cuando haya muchos desarrolladores o equipos, no cometer errores fáciles, es decir, errores que son de tipado.**

## **Creando DTO**

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

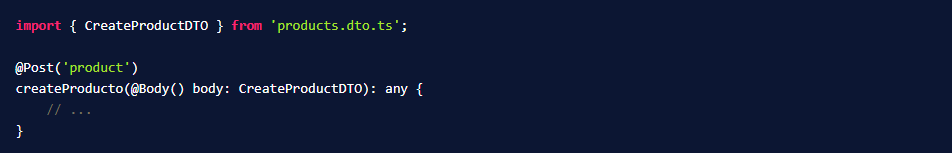


La palabra reservada **readonly** es propia de TypeScript y nos asegura que dichos datos **no sean modificados**.

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

## **Importando DTO**

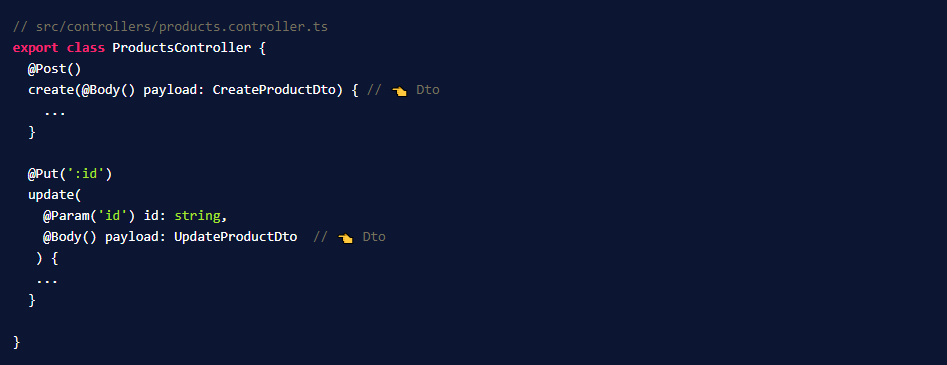
Importa la clase en tu controlador para tipar el **Body** del endpoint **POST** para la creación de un producto.

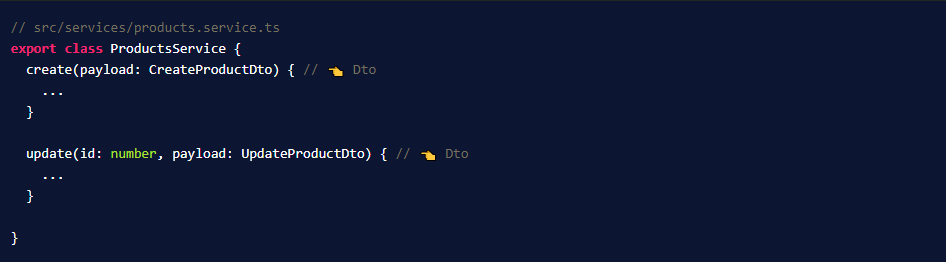


De esta forma, ya conoces la estructura de datos que tendrá el parámetro **body** previo a la creación de un producto.

**SRC: DTOS**

****

****



**NOTA**: Es una buena práctica tener separado los **DTO** y las **entidades**, esto es porque hay ocasiones que se necesitan atributos agregados en la transferencia de la información que no necesariamente van a estar insertada en la base de datos.

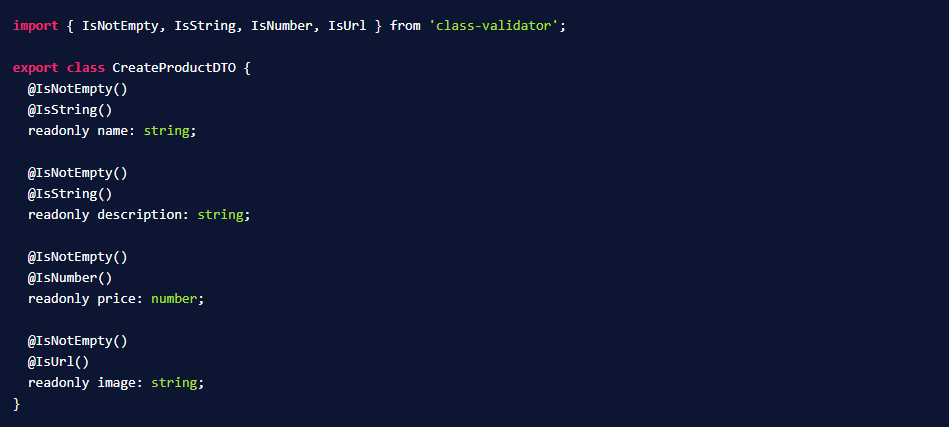
# **Validando parámetros con class-validator y mapped-types**

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 frontend 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.



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



El servidor nos devolverá el siguiente mensaje:

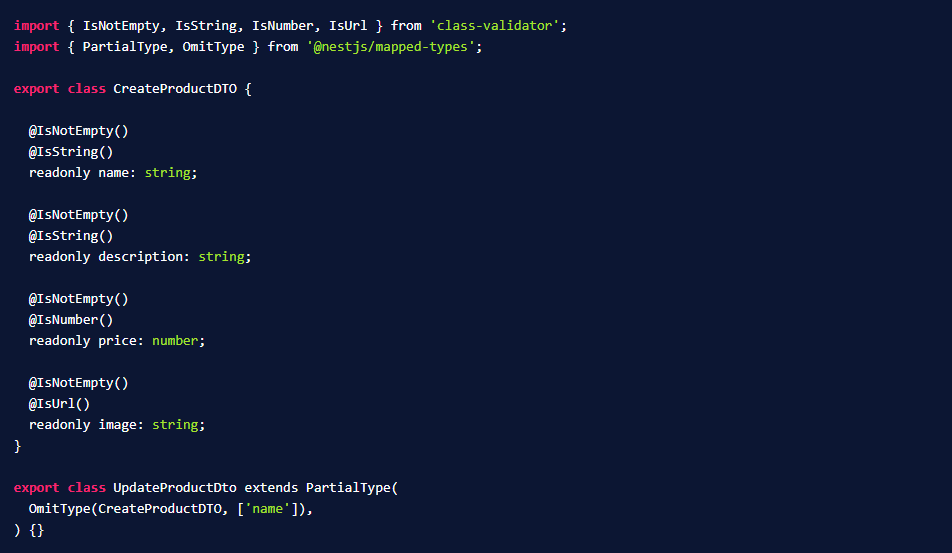


Indicando que la solicitud espera de forma obligatoria un campo description del tipo **string** y un campo **image** con una URL.

## **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 **@nestjs/mapped-types** que nos ayudará con la reutilización de código de la siguiente manera.

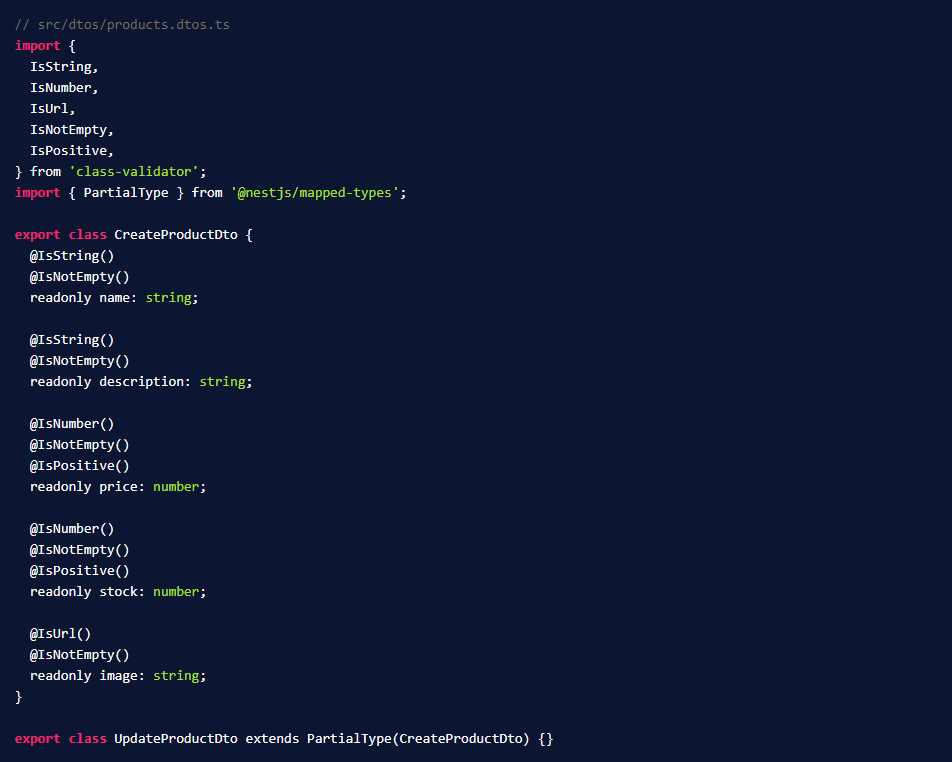


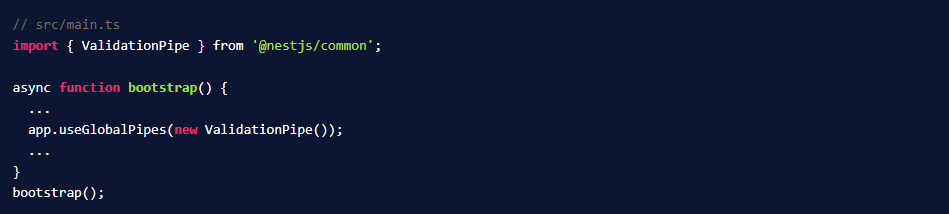
Importa **PartialType** y **OmitType** desde **@nestjs/mapped-types**.

**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**



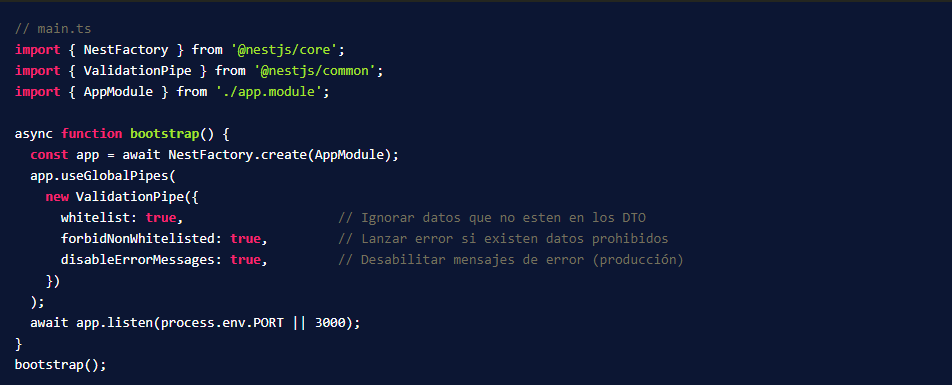


# **Cómo evitar parámetros incorrectos**

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.



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 frontend.

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

