Índice

[**Encapsular lógica en módulos** 1](#_Toc131883127)

[**Cómo hacer la modularización de un proyecto en NestJS** 1](#_Toc131883128)

[**Interacción entre módulos** 2](#_Toc131883129)

[**Importaciones de servicios compartidos** 2](#_Toc131883130)

[**Entendiendo la inyección de dependencias** 3](#_Toc131883131)

[**Patrones de diseño en NestJS** 3](#_Toc131883132)

[ **Inyección de dependencias** 3](#_Toc131883133)

[ **Singleton** 4](#_Toc131883134)

[**Precauciones utilizando servicios** 4](#_Toc131883135)

[**STUPID vs SOLID** 5](#_Toc131883136)

[**useValue y useClass** 5](#_Toc131883137)

[**Cómo hacer la inyección con useClass** 5](#_Toc131883138)

[**Cómo hacer la inyección con useValue** 7](#_Toc131883139)

[**Cuadro de códigos para inyección de servicios** 8](#_Toc131883140)

[**useFactory** 9](#_Toc131883141)

[**Inyecciones Asíncronas** 10](#_Toc131883142)

[**Global Module** 11](#_Toc131883143)

[**Cómo usar el módulo global** 11](#_Toc131883144)

[**Cuadro de código para uso de global module** 12](#_Toc131883145)

[**Errores de dependencias circulares entre módulos** 14](#_Toc131883146)

[**Módulo de configuración** 16](#_Toc131883147)

[**Variables de entorno en NestJS** 16](#_Toc131883148)

[**Consejos sobre las variables de entorno** 17](#_Toc131883149)

[**Configuración por ambientes** 18](#_Toc131883150)

[**Configuración dinámica del entorno** 18](#_Toc131883151)

[**1.** **Archivo principal para manejo de ambientes** 18](#_Toc131883152)

[**2.** **Configuración por ambiente** 18](#_Toc131883153)

[**3.** **Importando variables de entorno** 19](#_Toc131883154)

[**4.** **Inicio de la aplicación** 20](#_Toc131883155)

[**5.** **Utilizando las variables de entorno** 20](#_Toc131883156)

[**Tipado en config** 21](#_Toc131883157)

[**Cómo hacer el tipado de variables de entorno** 21](#_Toc131883158)

[**1.** **Archivo de tipado de variables** 21](#_Toc131883159)

[**2.** **Importación del tipado de datos** 22](#_Toc131883160)

[**3.** **Tipado de variables de entorno** 22](#_Toc131883161)

[**Validación de esquemas en .envs con Joi** 24](#_Toc131883162)

[**Validando variables de entorno** 24](#_Toc131883163)

[**Integrando Swagger y PartialType con Open API** 26](#_Toc131883164)

[**Cómo hacer la documentación API Rest** 26](#_Toc131883165)

[**Tipado de la documentación** 28](#_Toc131883166)

[**Documentación** 30](#_Toc131883167)

[**Extendiendo la documentación** 31](#_Toc131883168)

[**Cómo hacer la documentación personalizada** 31](#_Toc131883169)

[**Descripción de las propiedades** 31](#_Toc131883170)

[**Descripción de los controladores** 32](#_Toc131883171)

[**Configuración de Heroku** 35](#_Toc131883172)

[**Configuración del proyecto al usar Heroku** 36](#_Toc131883173)

[**Configuración de puerto y CORS** 36](#_Toc131883174)

[**Configuración versión de NodeJS** 36](#_Toc131883175)

[**Configuración de Heroku** 37](#_Toc131883176)

[**Deploy de NestJS en Heroku** 37](#_Toc131883177)

[**Cómo hacer el despliegue en Heroku** 37](#_Toc131883178)

[**Creando proyecto en Heroku** 37](#_Toc131883179)

[**Variables de entorno en Heroku** 38](#_Toc131883180)

[**Cuadro de código para despliegue de Heroku** 38](#_Toc131883181)

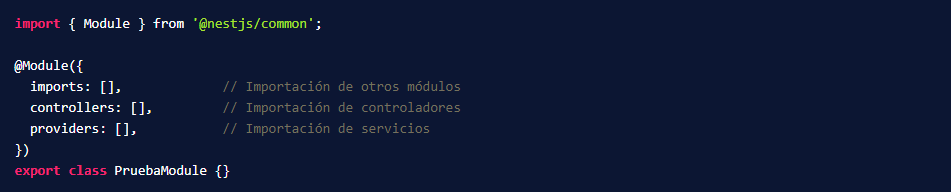
# **Encapsular lógica en módulos**

Las aplicaciones profesionales que se desarrollan con NestJS se realizan de forma modularizada para **dividir el código fuente de forma lógica** y que el proyecto sea más escalable y comprensible.

## **Cómo hacer la modularización de un proyecto en NestJS**

Para modularizar una aplicación, el **CLI** de NestJS trae consigo la posibilidad de autogenerar módulos con el comando **nest generate module <module-name>** o en su forma corta **nest g mo <module-name>**.

Los módulos son simples clases que utilizan el decorador **@Module()** para importar todo lo que construyan al mismo.



De esta manera, un módulo agrupará un conjunto de **controladores** y **servicios**, además de importar otros módulos.

A partir de aquí, tu aplicación podría tener un módulo para **usuarios**, otro para **productos**, otro para **comentarios**, etc. Crea tantos módulos como tu aplicación necesite.

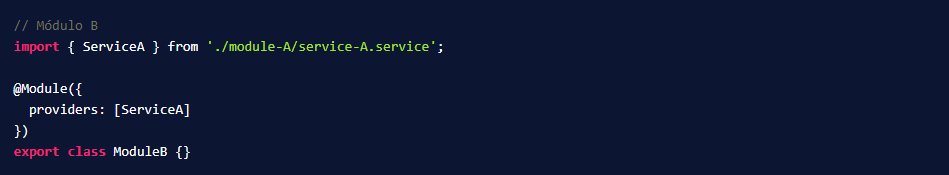
# **Interacción entre módulos**

Dentro de un módulo, puedes tener la necesidad de utilizar un **servicio** que pertenece a otro módulo. **Importar estos servicios en otros módulos requiere de un paso adicional**.

## **Importaciones de servicios compartidos**

Si tienes un **Módulo A** que posee un **Servicio A** y un segundo **Módulo B** requiere hacer uso de este, debes **exportar** el servicio para que otro módulo pueda utilizarlo.





Debes indicar en la propiedad **exports** del decorador **@Module()** que un módulo es exportable para que otro módulo pueda importarlo en sus **providers**.

De esta manera, evitas errores de compilación de tu aplicación que ocurren cuando importas servicios de otros módulos que no están siendo exportados correctamente.

# **Entendiendo la inyección de dependencias**

Es muy sencillo crear un servicio en NestJS, inyectarlo en un componente y utilizar su lógica. A pesar de esto, siempre es recomendable entender cómo lo está haciendo y qué sucede por detrás en tu aplicación.

## **Patrones de diseño en NestJS**

NestJS utiliza varios Patrones de Diseño para permitir que esto funcione. Te presentamos dos para tener en cuenta:

## **Inyección de dependencias**

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.

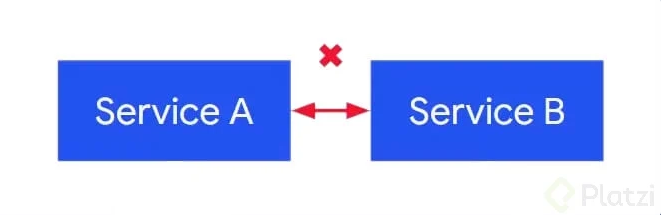
La inyección de dependencias llega para solucionar esto, resolver las dependencias de una clase por nosotros. Cuando instanciamos en el constructor el Servicio A, NestJS por detrás genera automáticamente la instancia del servicio B y C sin que nosotros nos tengamos que preocupar por estos.

## **Singleton**

La inyección de dependencias no es el único patrón de diseño que NestJS utiliza con sus servicios. También hace uso del patrón Singleton para crear una instancia única de cada servicio. Así es como, si tienes un servicio que se utiliza en N cantidad de componentes (u otros servicios) todos estos estarán utilizando la misma instancia del servicio, compartiendo el valor de sus variables y todo su estado.

## **Precauciones utilizando servicios**

Un **servicio** puede ser importado en muchos componentes u otros servicios a la vez. Puedes inyectar la cantidad de servicio que quieras en un componente (se recomienda hasta 3), siempre de una forma controlada y coherente.



**Solo debes tener cuidado con las dependencias circulares**. Cuando un servicio importa a otro y este al anterior. NestJS no sabrá cuál viene primero y tendrás un error al momento de compilar tu aplicación.

**NOTA**:

* Un **servicio** sólo pertenece a un **módulo**.
* Un **servicio A** puede ser inyectado en otro **servicio B**.
* **Nunca** el **servicio A** puede inyectar a un **servicio B** y un **servicio B** inyectar en un **servicio A**, esto ocasionaría una **dependencia circular**, lo cual crearía un error de referencia circular.

## **STUPID vs SOLID**

* **S.T.U.P.I.D**: Son una colección de principios que muestran cuales son las malas prácticas del desarrollo de software.
* **S.O.L.I.D**: son una colección de principios de diseño de software, introducidos por Robert Martin, estos principios nos ayudan a conseguir un código limpio, mantenible, escalable y tolerante a cambios.

[Aquí](https://codideep.com/blogpost/stupid-vs-solid) se detallan las diferencias de ambos patrones.

# **useValue y useClass**

NestJS posee **diferentes formas de inyectar servicios** en un módulo según la necesidad. Exploremos algunas de ellas, sus diferencias y cuándo utilizarlas.

## **Cómo hacer la inyección con useClass**

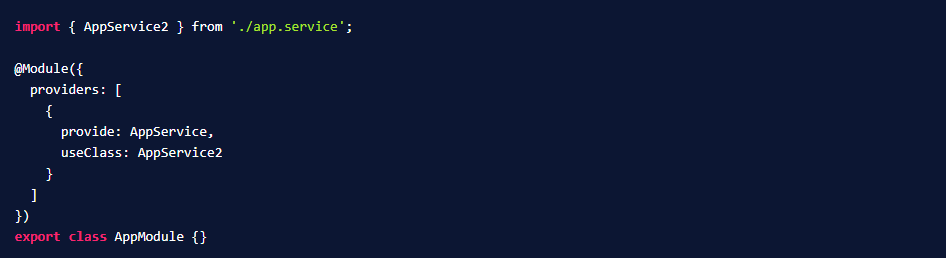
Cuando realizas un **import** de un servicio en un módulo:



Internamente, NestJS realiza lo siguiente:



Ambas sintaxis son equivalentes, **useClass** es el tipo de **inyección por defecto**. Básicamente, indica que un servicio debe utilizar **X** clase para funcionar. Si el día de mañana, por algún motivo en tu aplicación, el servicio **AppService** queda obsoleto y tienes que reemplazarlo por uno nuevo, puedes realizar lo siguiente:



De este modo, no tienes necesidad de cambiar el nombre **AppService** en todos los controladores donde se utiliza, este será reemplazado por la nueva versión del servicio.

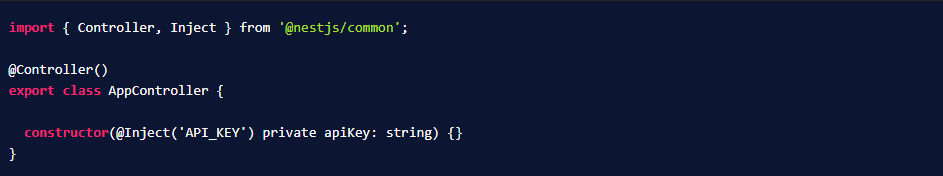
## **Cómo hacer la inyección con useValue**

Además de **clases**, puedes inyectar valores como un **string**, un **número**, etc. **useValue** suele utilizarse para **inyectar globalmente** en tu aplicación la llave secreta de una API o alguna otra variable de entorno que tu app necesita.

Para esto, simplemente inyecta el valor de una constante en el providers.

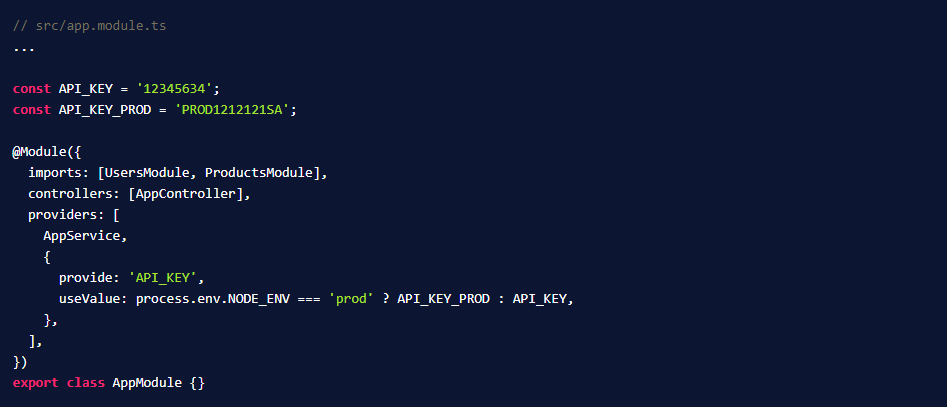


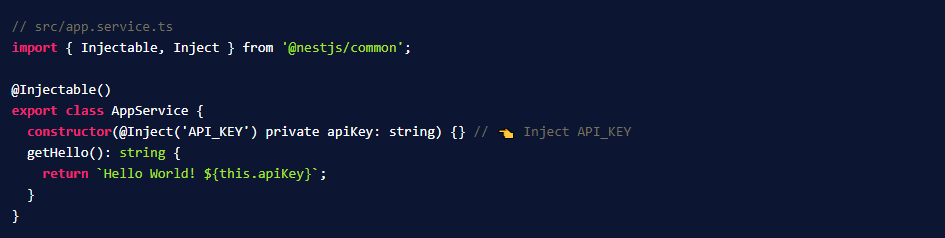
Importa este valor en los **controladores** u otros **servicios** donde se necesite de la siguiente manera:



Ahora tienes a disposición el valor de este dato en tu controlador para utilizarlo en lo que necesites.

## **Cuadro de códigos para inyección de servicios**







# **useFactory**

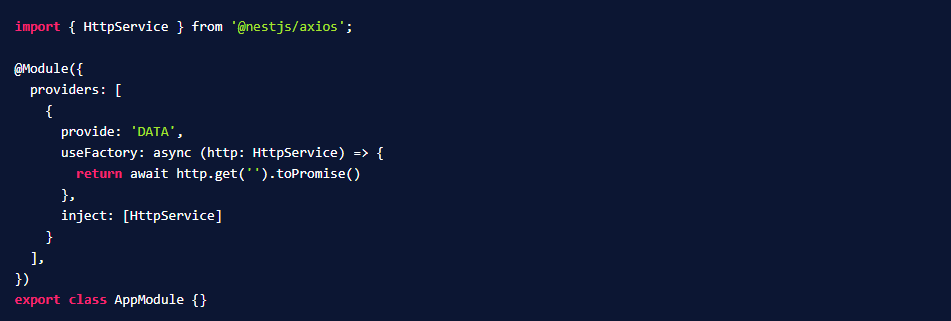
NestJS permite **inyecciones de servicios** o **datos** que necesiten de alguna petición HTTP o algún proceso **asíncrono**.

## **Inyecciones Asíncronas**

El tipo de inyección **useFactory** permite que realices un proceso asíncrono para inyectar un **servicio** o **datos** provenientes de una API.

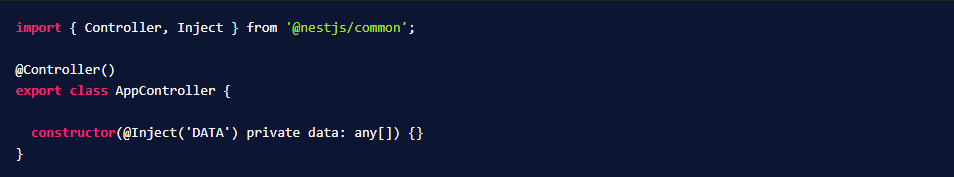
**NOTA**:

A partir de NestJS v8, el servicio **HttpService** importado desde **@nestjs/common** fue deprecado. Instala la dependencia **@nestjs/axios** e impórtalo desde ahí. No deberás realizar ningún otro cambio en tu código. También debes asegurarte de importar el módulo **HttpModule** desde la misma dependencia.



La propiedad **inject** permite que inyectes dentro de esta función asíncrona del **useFactory** otros **servicios** que este pueda necesitar. En el ejemplo anterior, se está haciendo una llamada a un **request** para obtener datos.

Importa estos datos en el controlador que lo necesite de la siguiente manera.



Así podrás hacer uso de estos datos que fueron cargados de forma asíncrona.

**NOTA**: Ten en cuenta que, al realizar una solicitud asíncrona, el controlador dependerá de la finalización de este proceso para estar disponible, pudiendo retrasar el inicio de tu aplicación. Esta funcionalidad suele utilizarse para conexiones de base de datos o procesos asíncronos similares.

# **Global Module**

Al desarrollar una aplicación con NestJS, existen necesidades de **importar módulos** **cruzados** o de **importar un mismo servicio en varios módulos**. Lo anterior, hace que la cantidad de **imports** en cada módulo crezca y se vuelva complicado de escalar.

## **Cómo usar el módulo global**

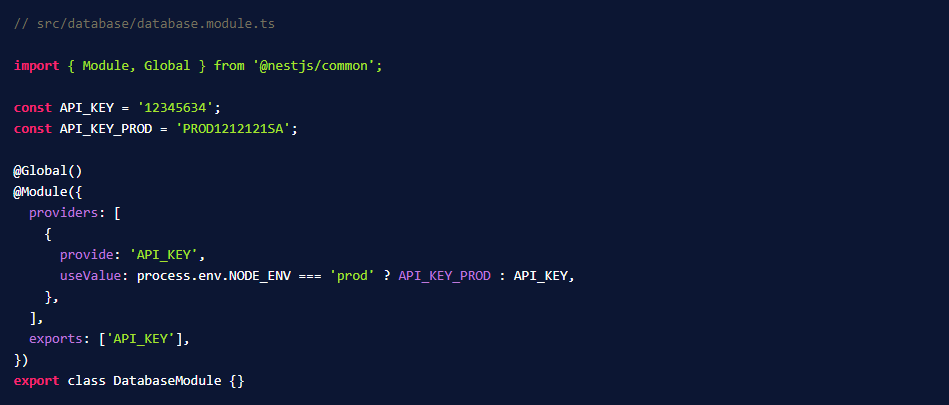
NestJS otorga la posibilidad de crear **módulos globales** que se importarán automáticamente en todos los otros módulos de la aplicación, sin necesidad de importarlos explícitamente.

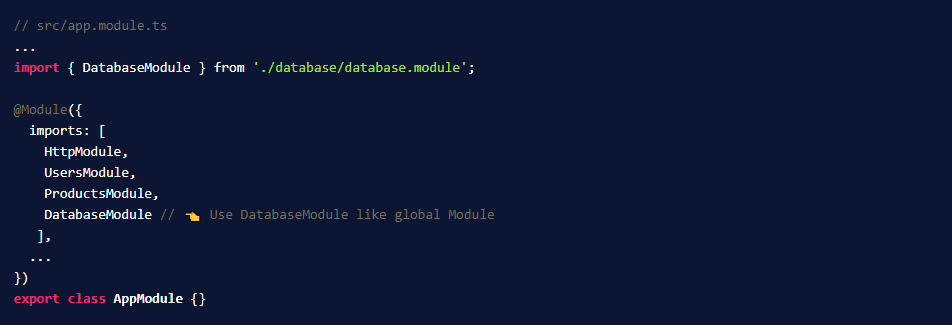


Todos los **servicios** que importes en este módulo, estarán disponibles para su utilización en cualquier otro módulo.

**NOTA**: Es importante no abusar de esta característica y no tener más de un módulo global para controlar las importaciones. Pueden ocurrir **errores de dependencias circulares** que suceden cuando el **Módulo A** importa al **Módulo B** y este a su vez importa al **Módulo A**. El decorador **@Global()** te ayudará a resolver estos problemas.

## **Cuadro de código para uso de global module**



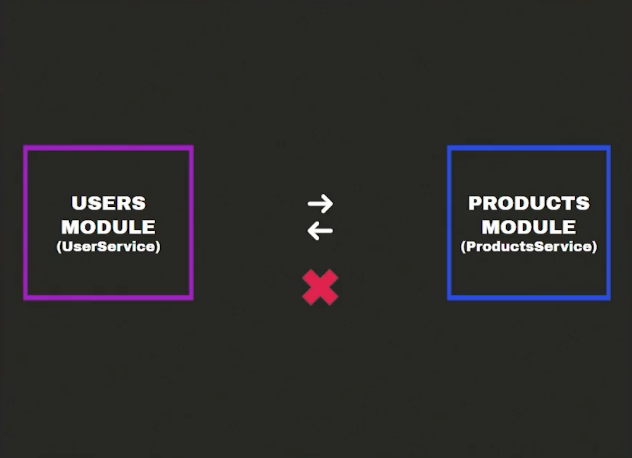




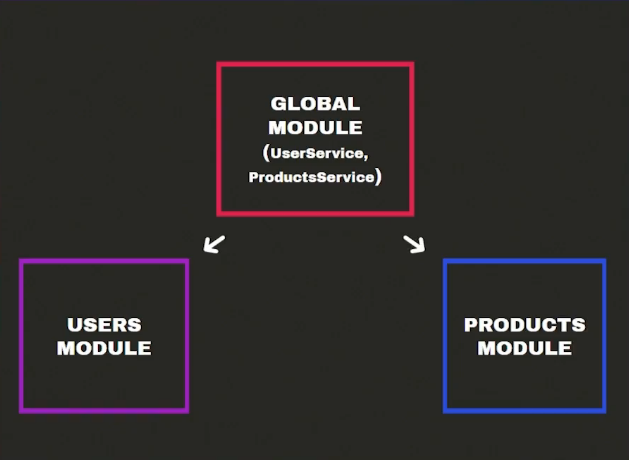
## **Errores de dependencias circulares entre módulos**

Los errores de dependencias circulares también se pueden dar entre módulos, por ej:

Sí tenemos **UserModule** y **ProductsModule**, en **ProductsModule** se necesita a **UserService** y dentro de **ProductsModule** necesitamos a **ProductsService**, esto nos causaría un error de dependencias circulares.



Para solucionar este problema, se puede tener un módulo Global donde estén los servicios que coinciden en varios módulos.



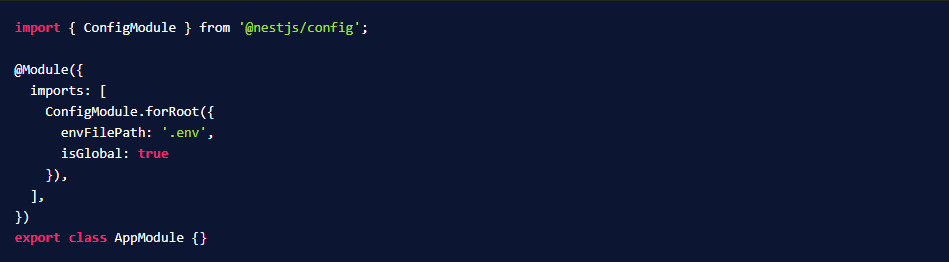
**NOTA**: Trata de evitar lo más posible esto, ya que cada modulo debería tener su propio dominio, pero si de verdad necesitas esto de utilizarlo de forma circular, sepáralo en un módulo a parte y que sea Global, de esta forma podrías utilizarlo en cualquier servicio, pero hay que tener mucha delicadeza con esto y elegir muy bien los servicios que van a ser globales.

# **Módulo de configuración**

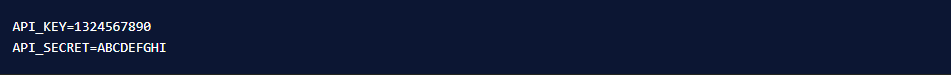
A medida que tu aplicación crezca, puedes llegar a necesitar decenas de **variables de entorno**. Variables que cambian de valor dependiendo si estás en un entorno de **desarrollo**, de **pruebas** o de **producción**.

## **Variables de entorno en NestJS**

El manejo de variables de entorno en NestJS se realiza de una forma muy sencilla. Instala la dependencia **@nestjs/config** e importa el módulo **ConfigModule** en el módulo principal de tu aplicación.



El archivo que almacena las variables de entorno suele llamarse **.env**. Créalo en la raíz de tu proyecto con las variables que necesitas.



De esta manera, las variables de entorno estarán disponibles en tu aplicación y utilizando el objeto global de NodeJS llamado **process** puedes acceder a estos valores de la siguiente manera:



## **Consejos sobre las variables de entorno**

Es muy importante **NO VERSIONAR** el archivo **.env** en el repositorio de tu proyecto. No guardes las claves secretas de tu aplicación en GIT.

Para asegurar esto, agrega el archivo **.env** a la configuración del archivo **.gitignore** para que no sea reconocido por Git y este no lo guarde en el repositorio.

Lo que puedes hacer es crear un archivo llamado **.env.example** que contendrá un modelo de las variables de entorno que tu aplicación necesita, pero no sus valores.



De este modo, cuidas tu aplicación y guardas un archivo para que cualquier desarrollador que tome el proyecto, sepa qué variables necesita configurar para el funcionamiento de la misma.

# **Configuración por ambientes**

Una aplicación profesional suele tener más de un ambiente. Ambiente **local**, ambiente de **desarrollo**, ambiente de **pruebas**, **producción**, entre otros, dependiendo la necesidad del equipo y de la organización. Veamos cómo puedes administrar **N** cantidad de ambientes en NestJS.

## **Configuración dinámica del entorno**

Configuremos la aplicación para intercambiar fácilmente entre diversos ambientes, cada uno con su propia configuración.

### **Archivo principal para manejo de ambientes**

Crea un archivo llamado **enviroments.ts** (o el nombre que prefieras) que contendrá un objeto con una propiedad por cada ambiente que tenga tu aplicación.



### **Configuración por ambiente**

Crea un archivo **.env** por cada ambiente que necesites. **Recuerda que todos los archivos finalizados en “.env” no deben guardarse en GIT**.

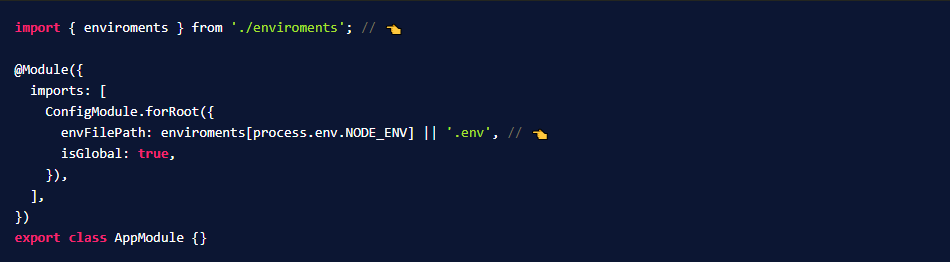




### **Importando variables de entorno**

Importa en el módulo principal de tu aplicación el archivo principal para manejo de ambientes y, a través de una única variable de entorno llamada NODE\_ENV, elegirás qué configuración usar.

**NODE\_ENV es una variable de entorno propia de NodeJS y del framework Express que se encuentra preseteada en tu aplicación.**

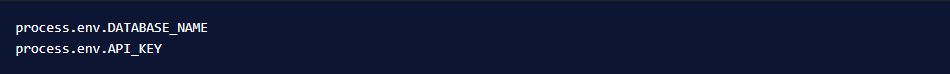
****

### **Inicio de la aplicación**

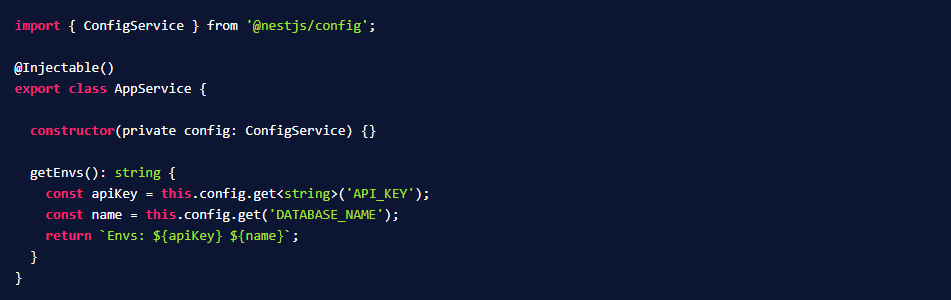
Finalmente, para iniciar tu aplicación basta con el comando **NODE\_ENV=test** **npm run start:dev** o **NODE\_ENV=prod npm run start:dev** para configurar la variable de entorno principal **NODE\_ENV** y escoger qué configuración utilizar.

### **Utilizando las variables de entorno**

Puedes utilizar las variables de entorno en tu aplicación de dos maneras. Con el objeto global de NodeJS llamado **process**:



O puedes usar estas variables a través del servicio **ConfigService** proveniente de **@nestjs/config**.



De este modo, configura de la mejor manera que necesites para tu aplicación el manejo de múltiples ambientes, cada uno con su propia configuración.

# **Tipado en config**

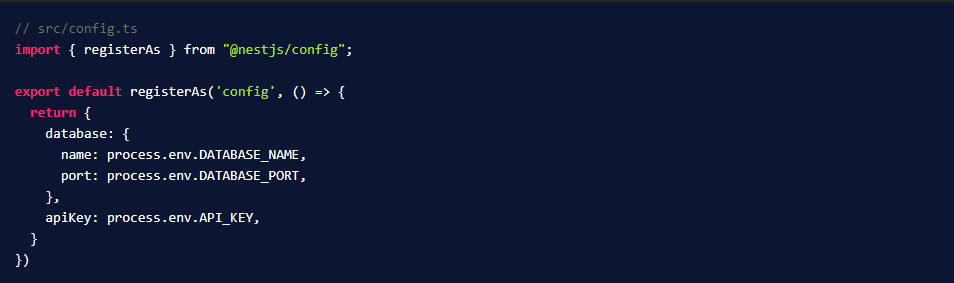
A medida que tu aplicación acumule más y más variables de entorno, puede volverse inmanejable y es propenso a tener errores el no recordar sus nombres o escribirlos mal. A continuación, verás como tipar variables.

## **Cómo hacer el tipado de variables de entorno**

Seguriza tu lista de variables de entorno de manera que evites errores que son difíciles de visualizar. Veamos cómo puedes tipar tus variables.

### **Archivo de tipado de variables**

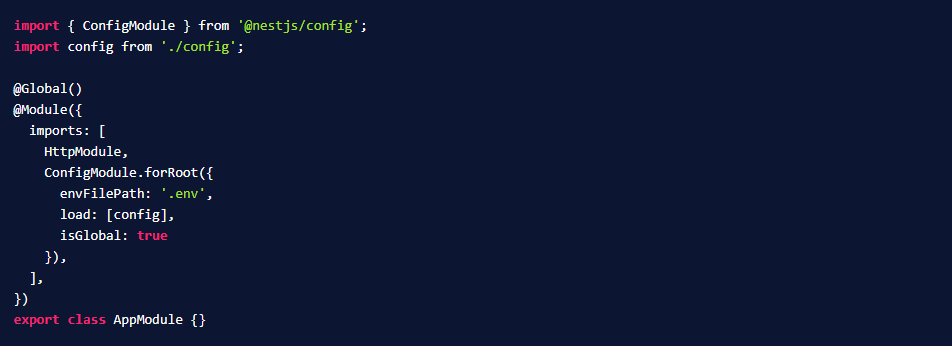
Crea un archivo al que denominaremos **config.ts** que contendrá el tipado de tu aplicación con ayuda de la dependencia **@nestjs/config**.



Importa **registerAs** desde **@nestjs/config** que servirá para crear el tipado de datos. Crea un objeto con la estructura de datos que necesita tu aplicación. Este objeto contiene los valores de las variables de entorno tomados con el objeto global de NodeJS, **process**.

### **Importación del tipado de datos**

Importa el nuevo archivo de configuración en el módulo de tu proyecto de la siguiente manera para que este sea reconocido.



### **Tipado de variables de entorno**

Es momento de utilizar este objeto que genera una interfaz entre **nuestra aplicación** y **las variables de entorno** para no confundir el nombre de cada variable.



Observa la configuración necesaria para **inyectar** y tipar tus variables de entorno. Ahora ya no tendrás que preocuparte por

posibles errores al invocar a una de estas variables y evitar dolores de cabeza debugueando estos errores.

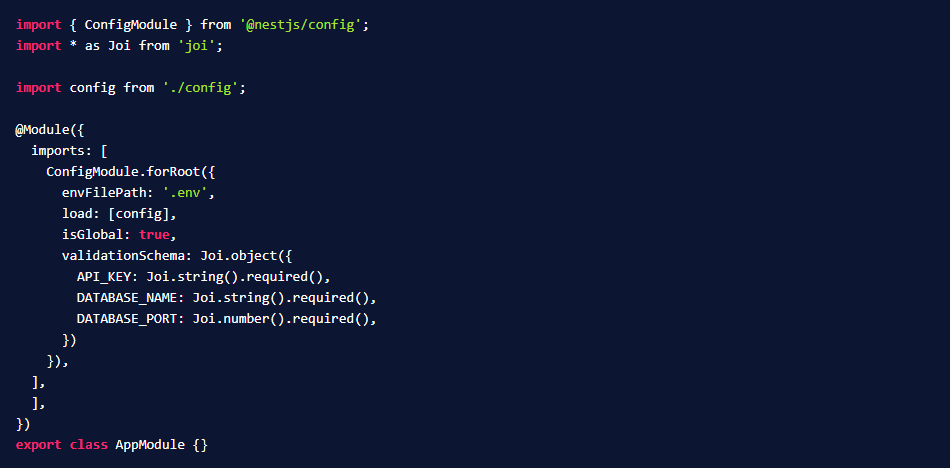
# **Validación de esquemas en .envs con Joi**

Las variables de entorno son sensibles, pueden ser requeridas o no, pueden ser un **string** o un **number**. Validemos tus variables de entorno para evitar errores u omisiones de las mismas.

## **Validando variables de entorno**

Instala la dependencia **Joi** con el comando **npm install joi --save**. La misma nos dará las herramientas para realizar validaciones de nuestras variables de entorno.

Importa **Joi** en el módulo de tu aplicación y a través de la propiedad **validationSchema** del objeto que recibe el **ConfigModule** crea el tipado y las validaciones de tus variables de entorno.



Lo que hace [**Joi**](https://www.npmjs.com/package/joi) es asegurar que, en el archivo **.env**, existan las variables de entorno indicadas dependiendo si son obligatorias o no, además de validar el tipo para no ingresar un **string** donde debería ir un **number**.

En equipos de trabajo profesional, quienes suelen desplegar las aplicaciones en los entornos es el equipo **DevOps** y ellos no necesariamente saben qué variables de entorno necesita tu aplicación y de qué tipo son. Gracias a esta configuración, tu app emitirá mensajes de error claros por consola cuando alguna variable no sea correcta.

# **Integrando Swagger y PartialType con Open API**

Una API profesional debe estar documentada. Cuando hablamos de documentación, nos suena a una tarea tediosa que nadie quiere realizar. Afortunadamente, NestJS permite automatizar fácilmente la creación de la misma.

## **Cómo hacer la documentación API Rest**

[**Swagger**](https://swagger.io/) es un reconocido set de herramientas para la documentación de API Rest. Instala las dependencias necesarias con el comando **npm install --save @nestjs/swagger swagger-ui-express** y configura el archivo **main.ts** con el siguiente código.



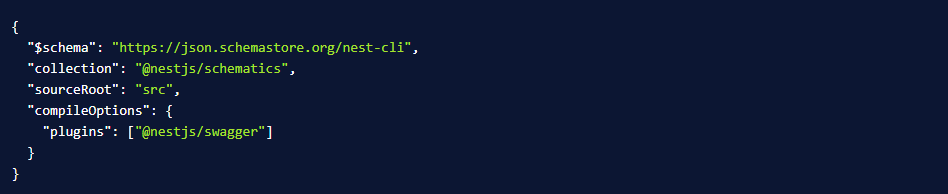
Setea el **título**, **descripción** y **versión** de tu documentación. Lo más importante es la URL para acceder a la misma.

Levanta el servidor con **npm run start:dev** y accede a **localhost:3000/docs** para visualizar la documentación autogenerada que mapea automáticamente todos los endpoints de tu aplicación.

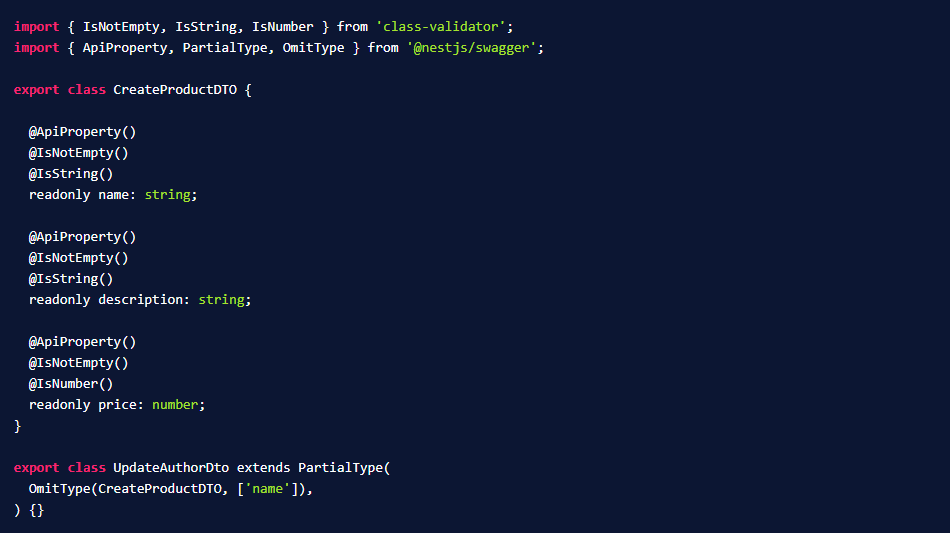
## **Tipado de la documentación**

La documentación autogenerada por **Swagger** es bastante simple, puedes volverla más completa tipando los datos de **entrada** y **salida** de cada endpoint gracias a los **DTO**.

Busca el archivo **nest-cli.json** en la raíz de tu proyecto y agrega el siguiente plugin:

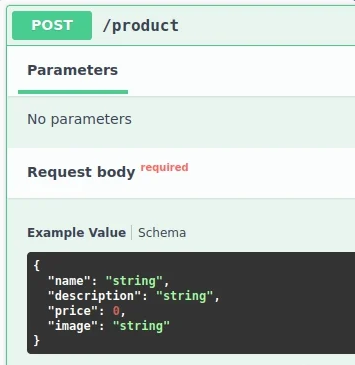


A continuación, prepara tus DTO de la siguiente manera:



Lo más relevante aquí es importar **PartialType** y **OmitType** desde **@nestjs/swagger** en lugar de importarlo desde **@nestjs/mapped-types**. Coloca también el decorador **@ApiProperty()** en cada una de las propiedades que el DTO necesita.

Finalmente obtendremos un resultado similar a este:



De esta manera, observarás en la documentación que indica el tipo de dato que requiere cada uno de tus endpoints.

## **Documentación**

* [OpenAPI](https://spec.openapis.org/oas/v3.1.0), buenas prácticas de escritura de documentación para API.
* [Configuración de OpenAPI en NestJS](https://docs.nestjs.com/openapi/introduction)
* [Configuración del CLI de NestJS](https://docs.nestjs.com/openapi/cli-plugin#using-the-cli-plugin)
* [Mapped types](https://docs.nestjs.com/openapi/mapped-types)

**NOTA**: Es importante que la ruta o nombres de carpetas no contengan espacio ni acentos u otros caracteres extraños, ya que, de a verlos, puede ocurrir excepciones de tipo **Module Not Found**, es decir, swagger puede no encontrar los módulos al compilar el proyecto.

# **Extendiendo la documentación**

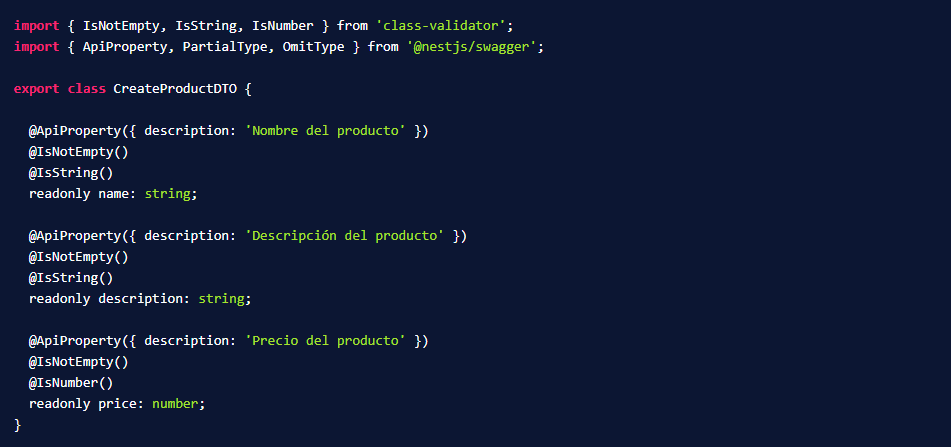
La documentación automática que genera **NestJS** y **Swagger** es muy fácil de implementar y otorga una buena base. La documentación de tu aplicación puede ser aún más completa y detallada, si así lo quieres con algo de trabajo de tu parte.

## **Cómo hacer la documentación personalizada**

Veamos varios decoradores que te servirán para ampliar la documentación de tu API.

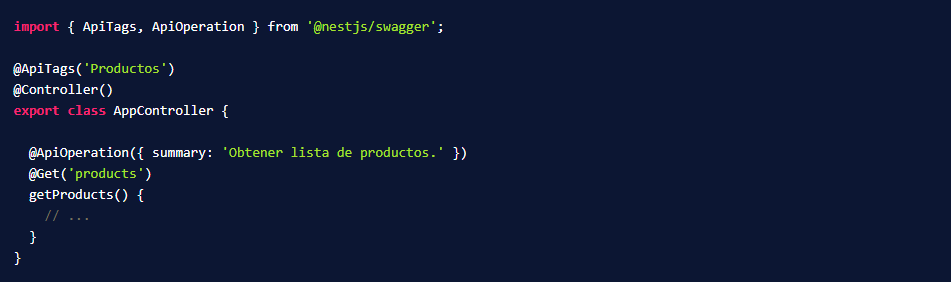
## **Descripción de las propiedades**

En tus **DTOs**, puedes dar detalle sobre qué se espera recibir en cada propiedad de tus endpoints gracias al decorador **@ApiProperty()**



## **Descripción de los controladores**

Puedes agrupar los endpoints en la documentación por controlador con el decorador **@ApiTags()** y describir, endpoint por endpoint, la funcionalidad de cada uno con el decorador **@ApiOperation()**.



Se puede extender la documentación de [swagger con muchos más decoradores](https://docs.nestjs.com/openapi/introduction#introduction).

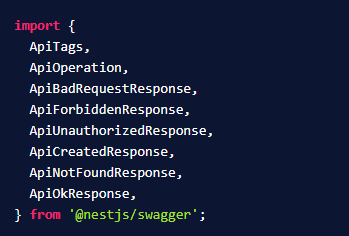
Para los controladores se utiliza:

* **ApiTags**: Para agrupar los métodos de los controladores.
* **ApiOperation**: Para documentar que es lo que hace el método.

Por cada método del controlador, se puede documentar las respuestas:

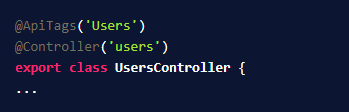
* **ApiResponse**: Se le define el **http code status** y la descripcion **description**.
  + Se puede extender con decoradores particulares para cada response que viene con su **status**:
    - **ApiOkResponse**: http code status 200.
    - **ApiBadRequestResponse**: http code status 400.
    - **ApiForbiddenResponse**: http code 403.
    - **ApiUnauthorizedResponse**: http code 401
    - **ApiCreatedResponse**: http code 201
    - **ApiNotFoundResponse**: http code 404

Importación

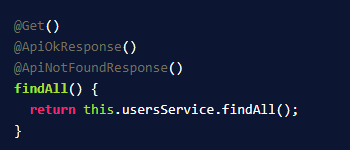


Y se implementan los decoradores:

* Para agrupar



* Para cada método, en este caso de un get, aparecía en swagger el **200** y **404**.

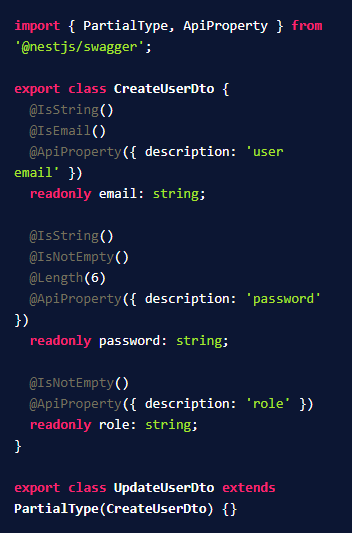


* Para POST



Para los DTOs:

* Se importa **PartialType** y **ApiProperty**



# **Configuración de Heroku**

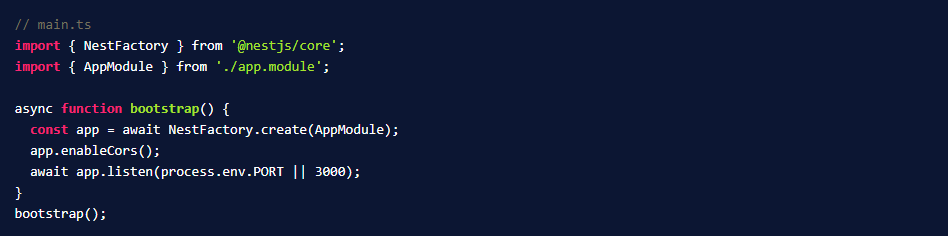
Llega el momento del despliegue de tu aplicación en un entorno productivo. Utilizaremos [Heroku](https://www.heroku.com/), uno de los proveedores de servidores Cloud más utilizado y fácil de utilizar de la industria.

## **Configuración del proyecto al usar Heroku**

Preparar tu aplicación para Heroku requiere de algunas configuraciones sencillas.

## **Configuración de puerto y CORS**

**Heroku**, por defecto, usa una variable de entorno llamada **PORT** para levantar la aplicación en un puerto aleatorio. Asegúrate de configurar esta variable dinámica en el **bootstrap** de tu **app**, además de activar **CORS** para no tener problemas con el mismo. Agrega las configuraciones en el archivo **main.ts**.



## **Configuración versión de NodeJS**

Edita el archivo **package.json** para especificar la versión de NodeJS que tu aplicación necesita con la siguiente configuración:



## **Configuración de Heroku**

Heroku requiere de un pequeño archivo adicional en la raíz de tu proyecto llamado **Procfile** que contiene el comando que da inicio a tu proyecto:



Además, [Heroku posee su propio CLI](https://devcenter.heroku.com/articles/heroku-cli) que nos ayudará en el despliegue de cualquier aplicación. Instálalo dependiendo tu sistema operativo para estar listo para el despliegue de tu app.

# **Deploy de NestJS en Heroku**

Teniendo tu aplicación configurada correctamente. Realiza el despliegue en Heroku instalando su **CLI** en primer lugar.

## **Cómo hacer el despliegue en Heroku**

Luego de instalar el **CLI**, realiza un **heroku login** para autenticarte. Si aún no posees una cuenta en Heroku, es el momento de [crearte una de forma gratuita](https://functions.salesforce.com/signups/).

## **Creando proyecto en Heroku**

Una vez situado en tu proyecto, utiliza el comando **heroku create -a <nombre\_proyecto>** para crear un nuevo proyecto remoto en tu cuenta de Heroku.

Heroku, internamente, posee su propio servidor de **GIT**. Si realizas un **git remote -v**, observarás que este ha agregado a tu proyecto nuevos servidores remotos. El despliegue se hará usando los propios de Heroku.

Con el simple comando **git push heroku master** la aplicación demorará unos pocos minutos en desplegarse. Podrás observar el progreso en la consola.

La aplicación quedará desplegada en una URL proporcionada por Heroku similar a **https://<nombre\_proyecto>.herokuapp.com/**, a la cual puedes acceder para observar si tu aplicación fue desplegada con éxito.

## **Variables de entorno en Heroku**

Si tu aplicación utiliza variables de entorno debes configurar estas. De manera muy sencilla, el siguiente comando te permite configurar cada una de tus variables de entorno: **heroku config:set APP\_KEY=12345**, mientras que el comando **heroku config** te permitirá ver una lista de las variables que ya están configuradas.

Recuerda que las variables de entorno son sensibles y debes cuidar quién tiene acceso a ellas.

¡**Felicidades**!

Has desplegado tu aplicación en un entorno productivo. Ahora el mundo puede acceder a tu app.

## **Cuadro de código para despliegue de Heroku**



También puedes usar el comando **heroku config:set APP\_KEY=12345** para configurar variables de ambiente desde el CLI y esto hace que la app se reinicie sin la necesidad de enviar un push.

**Notas**:

* Cuidado con los tipos 🙂
* No dejes comments en producción 📢