Índice

[**Integración de node-postgres con NestJS** 1](#_Toc133780311)

[**Conexión como inyectable y ejecutando un SELECT** 2](#_Toc133780312)

[**¿Qué es un ORM? Instalando y configurando TypeORM Module** 5](#_Toc133780313)

[**Para qué sirve un ORM** 5](#_Toc133780314)

[**Instalando TypeORM** 6](#_Toc133780315)

[**Creando tu primera entidad** 8](#_Toc133780316)

[**TypeORM: active record vs. repositories** 9](#_Toc133780317)

[**Implementación** 11](#_Toc133780318)

[**Sync Mode vs. Migraciones en TypeORM** 14](#_Toc133780319)

[**Recomendaciones** 14](#_Toc133780320)

[**Naming o Nombramiento de variables en BD** 15](#_Toc133780321)

[**Serializar** 18](#_Toc133780322)

[**Cómo solucionar una referencia circular entre módulos** 19](#_Toc133780323)

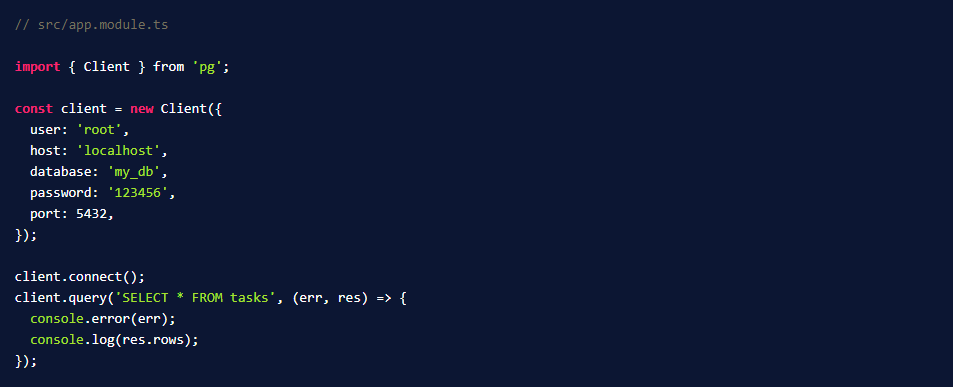
[**Solución 1: entidades en un Global Module** 21](#_Toc133780324)

[**Solución 2: Referencia directa** 22](#_Toc133780325)

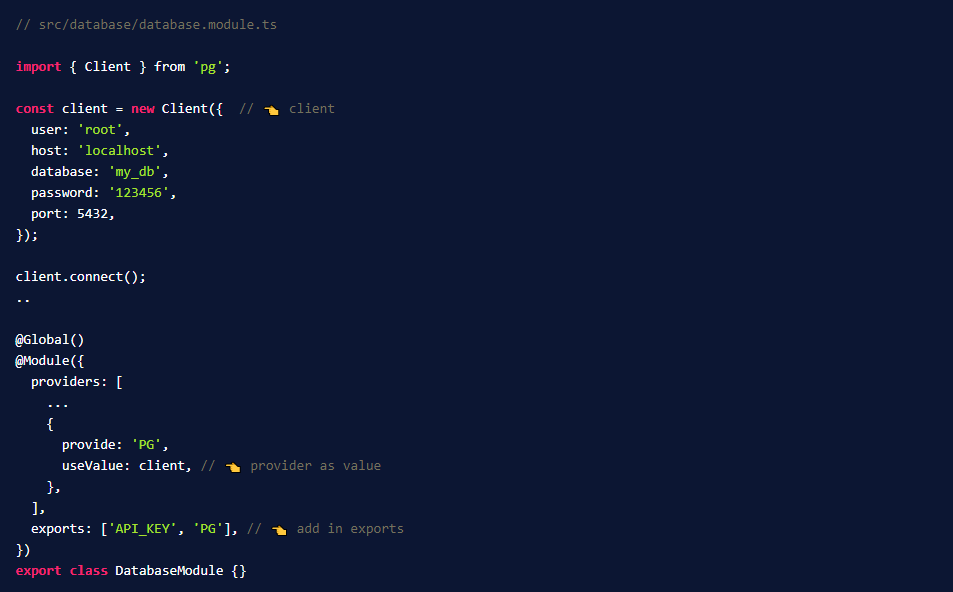
# **Integración de node-postgres con NestJS**

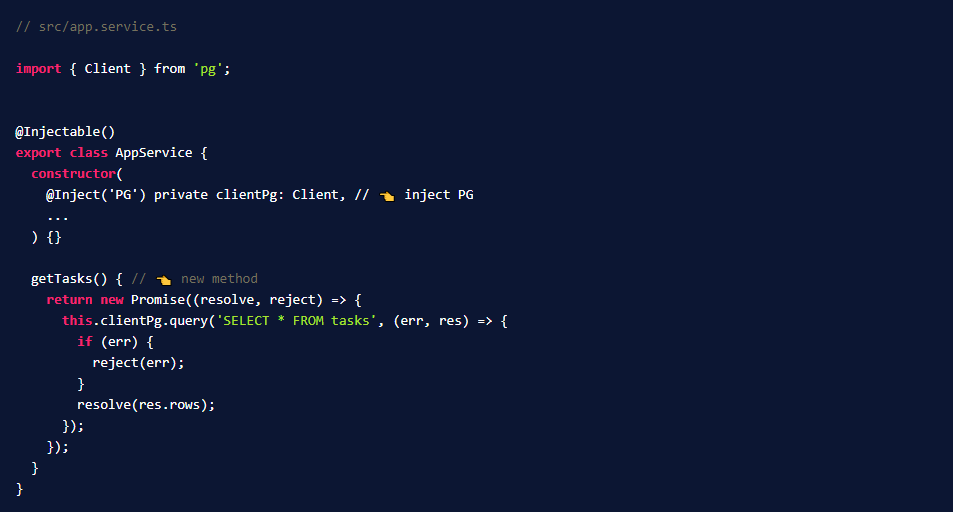
* Descargamos la librería oficial de postgres.
* Luego descargamos el tipado para esta librería, ya que de forma nativa no soporta tipados.





# **Conexión como inyectable y ejecutando un SELECT**







# **¿Qué es un ORM? Instalando y configurando TypeORM Module**

**ORM** (**O**bject **R**elational **M**apping) es una técnica de programación que nos ayuda a manipular y consultar la información almacenada dentro de una base de datos usando programación orientada a objetos. [Un ORM](https://platzi.com/clases/1920-eloquent-laravel/28515-que-es-un-orm-y-para-que-sirve-eloquent/) se encarga de la conexión y también de manejar todo con base en **modelos** o **entidades**.

Una principal característica de un ORM es que hace más transparente las conexiones a PostgreSQL y MySQL, además nos protege de algunas vulnerabilidades de SQL y facilita algunas validaciones a la información.

## **Para qué sirve un ORM**

Como esto es una abstracción, no vamos a tener que ejecutar código SQL directamente para hacer una búsqueda, una inserción o una actualización. La ORM va a dar métodos muy prácticos para utilizarlo dentro de nuestro código.

Un ORM nos ayuda en la extracción de código complejo SQL, sin embargo, nunca está de más que en verdad sepas cómo hacer tus propias consultas SQL en caso de que necesites una consulta muy potente o avanzada.

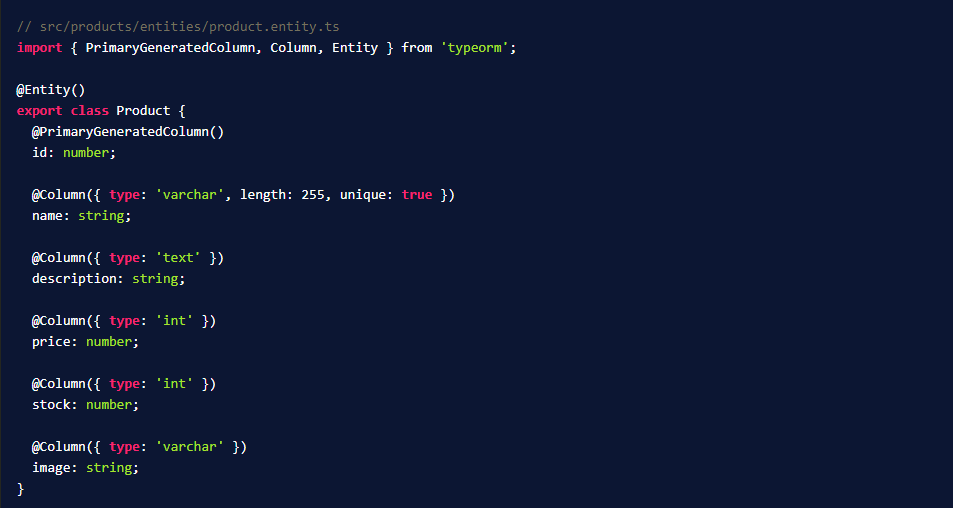
## **Instalando TypeORM**

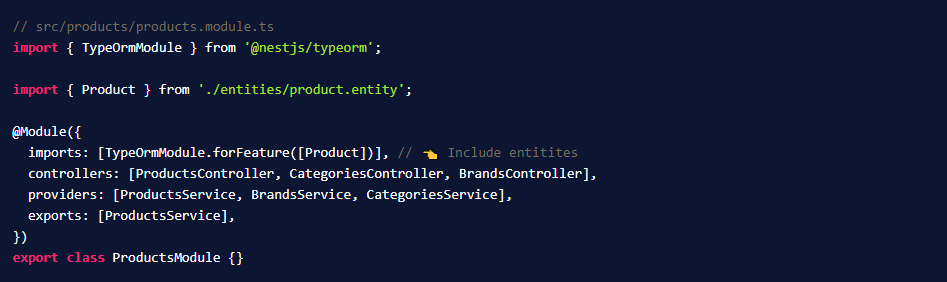
TypeORM está desarrollado con typescript, así que la integración con NestJS va a hacer buena.

Para hacer la integración, lo primero que debes hacer es instalar la dependencia



# **Creando tu primera entidad**



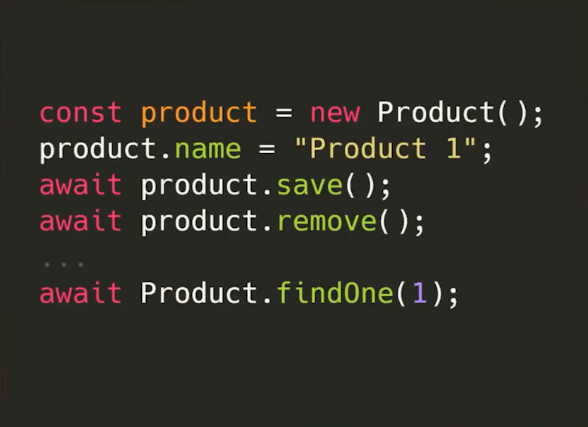


# **TypeORM: active record vs. repositories**

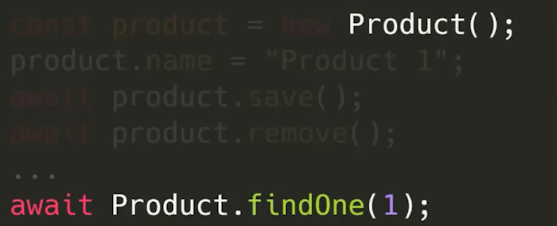
TypeORM maneja dos patrones para manipular la **creación**, **modificación**, **eliminación** y **búsquedas** en base a los modelos, estos son:

* **ActiveRecord**: Utiliza la misma instancia de la entidad. Este patrón tiene toda la responsabilidad del modelo.

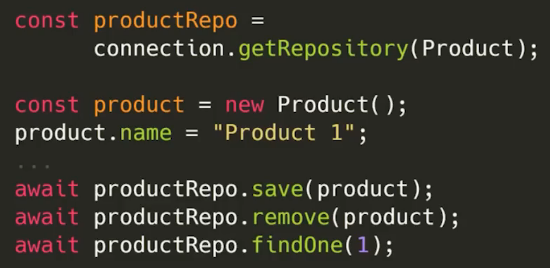
Ya que se cuenta con la referencia del objeto, se puede hacer nameObject.**save()** (el cual también puede ser utilizado para actualizar) o nameObject.**remove()**



Sin embargo, para realizar una búsqueda se utiliza el **nombre de clase**, ya que no se cuenta con la referencia al objeto.



* **Repositories**: Este patrón divide las responsabilidades del modelo, es decir, la responsabilidad del modelo va a ser de sus atributos, pero el repositories es el que se encarga de las operaciones.



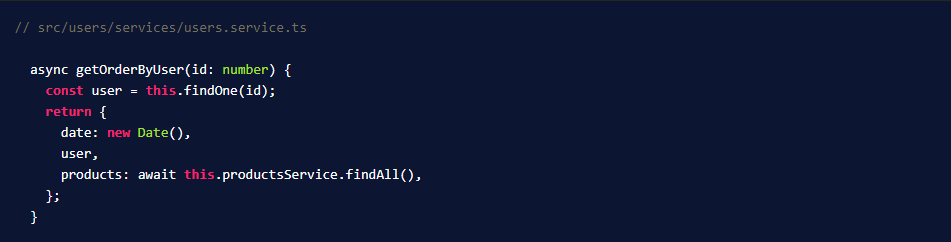
En el ejemplo anterior, primero se crea un repositorio, luego el producto, pero para realizar búsquedas, crear, actualizar y eliminar, lo haríamos a través del **repositorio**.

Ambos patrones se basan en tener un modelo o una entidad.

**NOTA**: El patrón recomendado para TypeORM y NestJs es el patrón de **Repositories**.

## **Implementación**







# **Sync Mode vs. Migraciones en TypeORM**

Las **migraciones** es una forma de controlar los cambios hacia la base de dato. Hay dos tipos de hacer esto:

* **Mode sync**: El flag **synchronize: true** esto hace que los cambios que hagamos sobre nuestro modelo de clases impacten directamente en las tablas de nuestra base de datos.

## **Recomendaciones**

Solo utilizar este flag sobre el entorno de **desarrollo** y **testing**. En producción **es una práctica riesgosa**, **porque cualquier cambio en el código, se pueda corromper la base de datos**. Para eso se crearon las migraciones.

* **Migrations**: Son un proceso que se utiliza para realizar cambios en la estructura de una base de datos de manera controlada y documentada. Se puede hacer cosas como:
  + Agregar, eliminar tablas.
  + Agregar, modificar columnas de una tabla.
  + Etc...

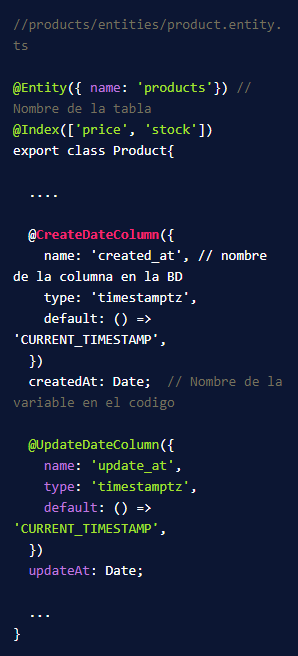
Al utilizar migraciones, se puede tener un control más preciso sobre los cambios que se están realizando en la base de datos y revertir cambios si es necesario.

**En TypeORM es solo un archivo con queries tipo SQL para actualizar los cambios en el esquema o aplicar cambios en una base de datos existente**.

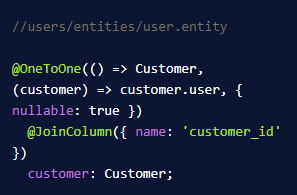
# **Naming o Nombramiento de variables en BD**

Algunas de las buenas prácticas en bases de datos son

* El naming es una buena práctica.
* Nombre de una variable tiene que ser representativa.
* Tiene que ser escrita en **camelCase** para JavaScript.
* En BD una buena práctica es no tener caracteres especiales como la mayúscula.
* Debido que si se hace consultas directas a la BD habría que escapar caracteres especiales.
* En BD debe ser separadas con \_(**snake\_case**) el nombre de un campo con palabras compuestas, ej, **product\_id**.
* Nombre de la tabla debe ser en **plural**, por ej, products.

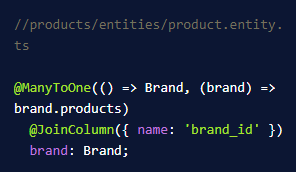


En los casos de relaciones **OneToOne** se hace la modificación en la tabla que tenga el decorador **@JoinColumn**.



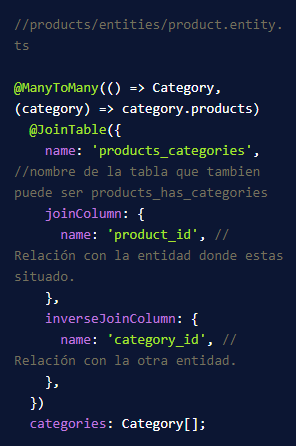
En una relación **ManyToOne** se hace la modificación en quien lleve este decorador.

Se agrega el decorador **@JoinColumn**



En las tablas N:N que es manejada por TypeORM se modifica en la entidad que tenga el decorador **@JoinTable**

* Recuerda que en este caso se crea una **tabla ternaria**.



# **Serializar**

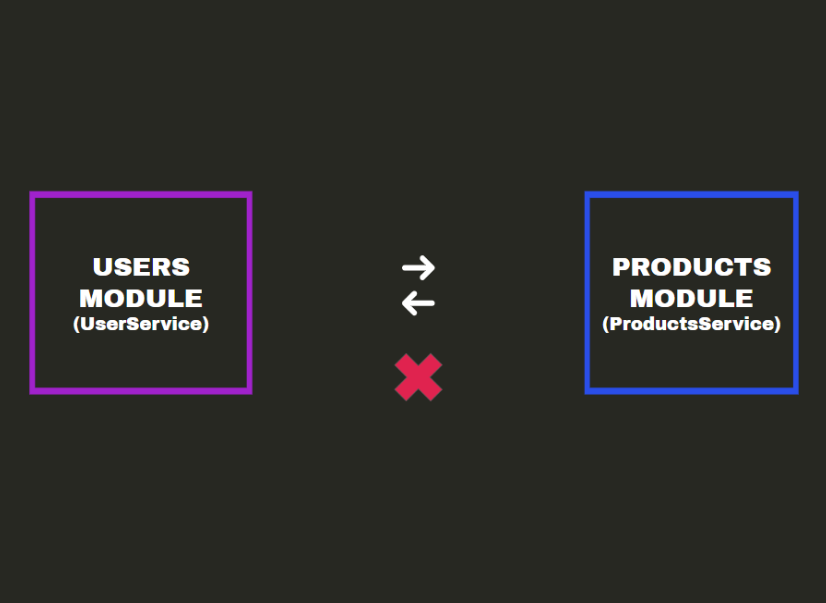
Serializar **es transformar la información antes** de que el controlador la retorne a un servicio, es decir, a un **response** como tal.

Para ello se crea un interceptor global en NestJs. Esto es utilizado por lo general para ocultar campos que no son de importancia o para no mostrar información sensible, como por ej, el **password**.

# **Cómo solucionar una referencia circular entre módulos**

**NestJS** Realmente es un framework potente que nos permite tener una arquitectura sólida y escalable. Pero al conectarnos con bases de datos podemos llegar a enfrentarnos a una referencia circular, problema que debemos evitar y/o aprender a solucionar.

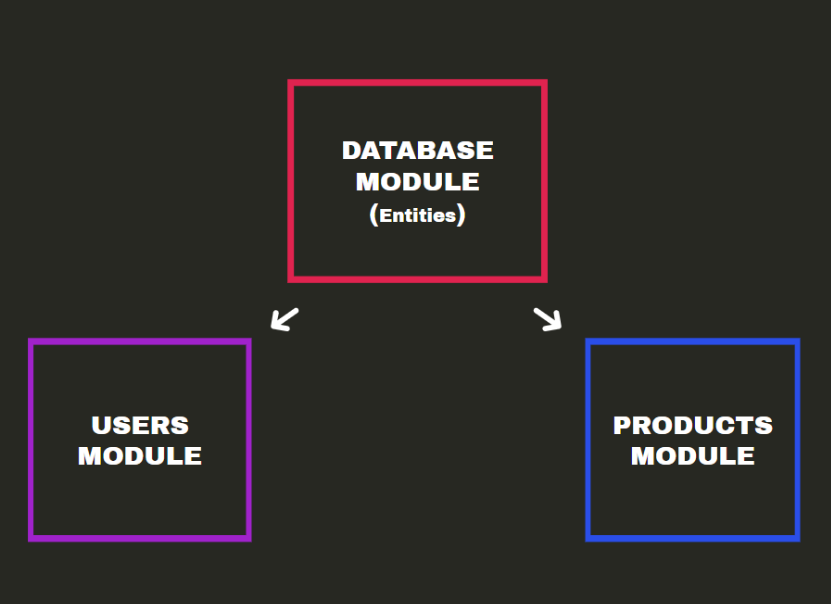
En nuestro proyecto tenemos a **UsersModule** que importa a **ProductsModule** con el fin usar al **ProductsService**. Pero ¿qué pasa si **\_ProductsModule\_** necesita hacer una consulta hacia users? Recuerda que dentro de **UsersModule** está el UserService y la entidad de Users. Si importamos a UserModule dentro de ProductsModule, tendríamos un problema de referencia circular entre módulos.



Existen dos maneras de solucionarlo

## **Solución 1: entidades en un Global Module**

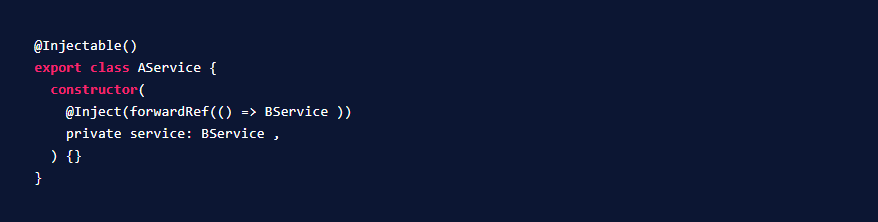
Una de las soluciones es poner todas las entidades de tu proyecto en el **DatabaseModule** de manera global haciendo que cada uno de los demás módulos pueda usar estas entidades sin tener problemas de referencia circular.



De esta manera cada módulo puede usar el **Repository Pattern** y hacer la consulta que necesite de una entidad.

## **Solución 2: Referencia directa**

Una de las formas que tiene NestJS para resolver la referencia circular es tener una referencia directa, por ejemplo, si **AService** y **BService** dependen el uno del otro, ambos lados de la relación pueden usar **@Inject()** y la utilidad **forwardRef()** para resolver la dependencia circular, ejemplo:



De la misma manera en el otro servicio.

