Proyecto API con ORM My SQL en nestjs

(TUDAI + <https://www.adictosaltrabajo.com/2018/03/05/introduccion-a-nestjs/>)

1. Para crear el proyecto con la CLI de Nest, habrá que ejecutar el siguiente comando de nest

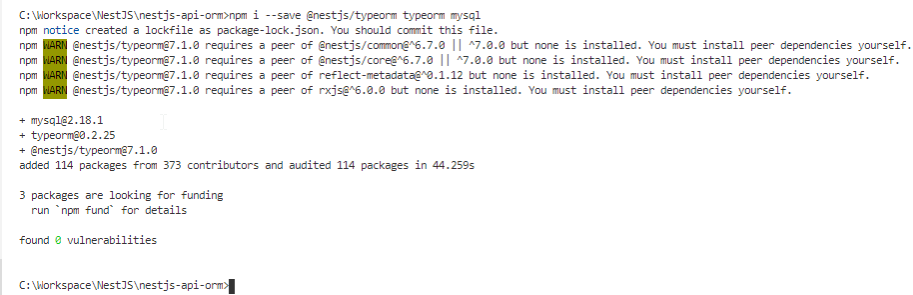
**nest new nombre-proyecto**

suponiendo que ya tenemos instalado @nestjs/cli globalmente (ya lo hice para otros proyectos).

1. Para acceder a la base de datos MySQL vamos a utilizar TypeORM por lo que necesitamos incluir las siguientes dependencias

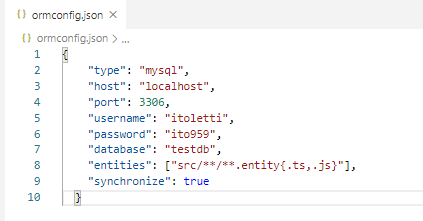
**npm i --save @nestjs/typeorm typeorm mysql**

lo referente al ORM y el cliente de MySQL para Nodejs



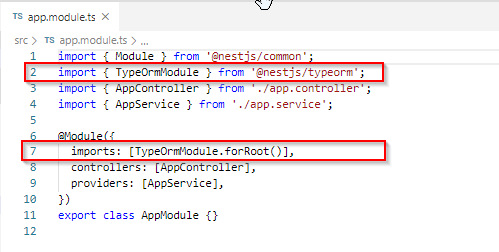
**Nota**: recordar, porque me paso en este caso, luego de instalar esta librerías anteriores, posteriormente a haber creado el proyecto nestjs. Que algunos archivos me indicaban un error como que no podían ubicar la librería @nestjs/common o la @nestjs/core, que en realidad las veía que estaban instaladas de manera global en la carpeta /node\_modules de la instalación de NodeJS. Esto lo solucione reinstalando todas las dependencias mediante el comando npm install.

1. Ahora creamos el archivo de configuración de TypeORM llamado «ormconfig.json» con la configuración de acceso a la base de datos y donde residen las entidades de la aplicación:

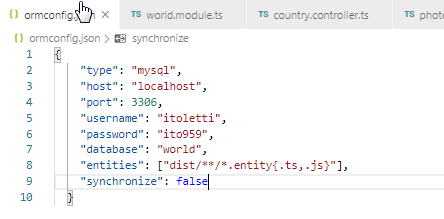


**Nota**: Recordar cuando me conecte a la MySQL hacerlo con el usuario itoletti y no con el usuario root, pues este ultimo va a dar error de conexión porque MySQL usa por default para los usuarios el modo de autenticacion **caching\_sha2\_password** y el cliente mysql que utiliza Nodejs no lo soporta, usa el modo de autenticación antiguo **mysql\_native\_password**, para tal motivo cree el usuario itoletti con los mismos privilegios que root pero con el modo de autenticación nativo.

1. Ahora establecemos esta configuración en la aplicación editando el archivo «src/app.module.ts» para añadir el módulo de TypeORM.



**Nota**: la función **forRoot()** busca el archivo de configuración ormconfig.json en el raíz del proyecto.



Dos cosas tuve que tener en cuenta a la hora de probar el proyecto

* 1. Que syncronize este a false, si las tablas en la BD ya están creadas, porque sino intenta crearlas y da error. Indicaban setearlo a true
  2. En entities usar dist/ en vez de src/ como lo indican. (la notacion

1. Vamos a utilizar la BD world de MySQL y de ella las tablas City y Country. La interaccion con la BD de MySQL la vamos a hacer a través de una variable del tipo **Repositoy**, esta va a formar parte del objeto servicio, va a ser un miembro del objeto **servicio** (**cityRepository** para el caso de las ciudades y **countryRepository** para el caso de los países Se definirá en el constructor de cada clase **CityService** y **CountryService**, luego en estas clases se implementaran los métodos get*entity*(), get*entities*(), add*entity*(), delete*entity*(), update*entity*() que utilizaran los métodos find(), findone(), save(), merge(), delete() del atributo ***entity*Repository** de la clase que corresponda. Estas funciones a su vez responderán a las llamadas a de los métodos GET, POST, DELTE, y UPDATE, que implementaran los diferentes endpoints en los **controladores** correspondientes **CityController** y **CountryController**.

Entonces, en principio, vamos a pensar en un modulo **World** con entidades **city** y **country**, sus **DTO** (data transfer objects), sus **servicios** y sus **controladores** que se referenciaran desde el modulo secundario world.module.ts

\src

\world

\enities

city.entity.ts

country.entity.ts

\dto

city.dto.ts

country.dto.ts

\services

city.service.ts

country.service.ts

\controllers

city.controller.ts

country.controller.ts

word.module.ts

app.service.ts

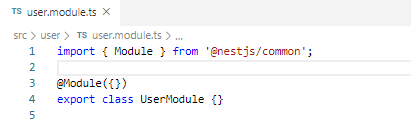
app.controller.ts

app.module.ts

Nest nos permite generar un modulo con el comando

nest generate module world (<https://christiantola.me/blog/create-api-rest-nestjs.html>)

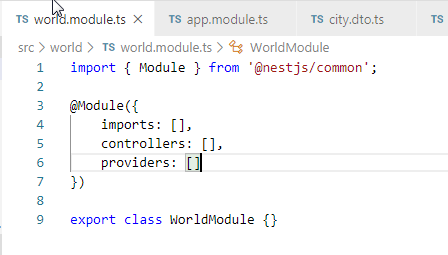
lo que genera una carpeta \world y el archivo world.module.ts



Por el momento dejamos el archivo world.module.ts y arrancamos definiendo

* las **endidades**, es decir, creando un archivo por cada tabla con la clase correspondiente a cada tabla, utilizando los decoradores del modelo TypeORM (que se debe importar al archivo para que los reconozca)
* los **DTO,** un archivo por cada objeto, con una clase definiendo los atributos de cada objeto correspondiente a cada tabla.

1. Después debiéramos de proseguir con la creación del archivo world.module.ts, sin información aun de los providers, los controles que aun no hemos escrito pero se hara en un futuro.



la documentación indica que dentro de imports: [] se debe agregar información para configurar el repositorio de entidades a través del método **forFeature()**, para esto será necesario hacer el import de TypeOrmModule desde @nestjs/typeorm, además de importar las entidades City y Country para agregarlas en el forFeature()



1. Es momento de escribir los controladores y los servicios. De la misma manera que con el modulo Nest cuenta con comando s para crear controladores y servicios en blanco

nest generate controller world (<https://christiantola.me/blog/create-api-rest-nestjs.html>)

nest generate service world

luego hay que ir completandolos.

En los controladores se inyectara en su constructor una instancia del servicio, se definirán las rutas (endpoints) con los diferentes métodos http (GET, POST, PUT, DELETE) que terminaran siendo llamadas a los métodos del servicio. Por el otro lado, en los servicios nos vamos a encargar de recuperar los datos, inyectando a través del constructor una instancia del repositorio que entre otros métodos tiene find(), el save() que finalmente serán los responsables de recuperar y grabar los datos. Una vez definidos controladores y servicios se deberán agregar en los arreglos controllers[] y provides[] quedando el world.module.ts finalmente

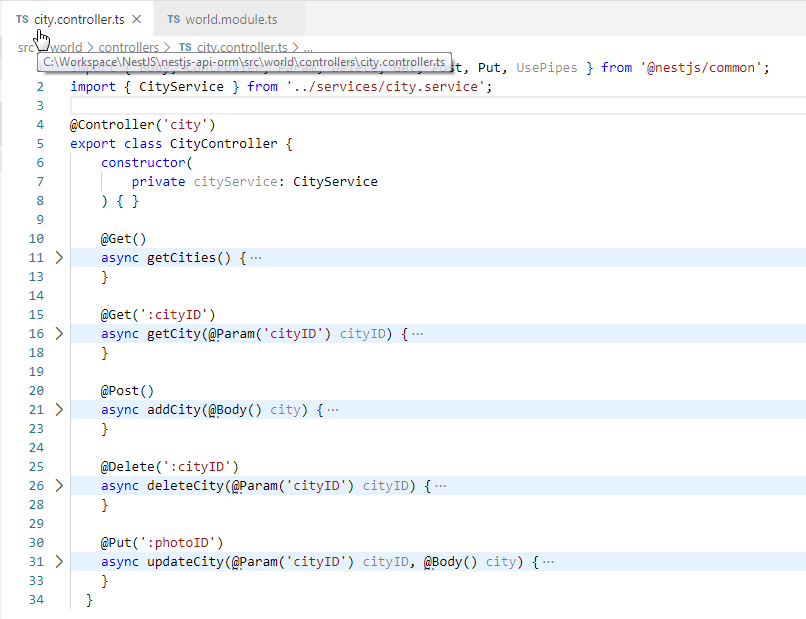


Arranaquemos por el controlador de City

En el archivo city.controller.ts definiremos la clase **CityController**, donde vamos a inyectar el servicio creado de recuperación **cityService** y a través de anotaciones (decoradores) vamos a exponer los datos a través del endpoint (‘city’)

Necesitaremos importar todos los tipos de decoradores a utilizar (Controller(), Param(), Body(), Get(), Post(), Put(), Delete()) desde ‘@nest/common’.

Definimos el punto de entrada (end point) @Controller(‘city’), definimos entonces la clase CitryController. el constructor para la clase **CitryController,** por lo menos contendrá en un principio un atributo del tipo CityService al que definiremos como cityService (es el que contiene los métodos definidos para cada una de las acciones GET, POST, PUT y DELETE).



arrancamos con los GET que serían los más sencillos para escribir en el Servicio. Estos retornaran el resultado del llamado a los métodos getCities() y getCity(id) de cityService. El primero retornara todas las ciudades mientras que el segundo recibirá como parámetro un id de ciudad y retornará la ciudad correspondiente.



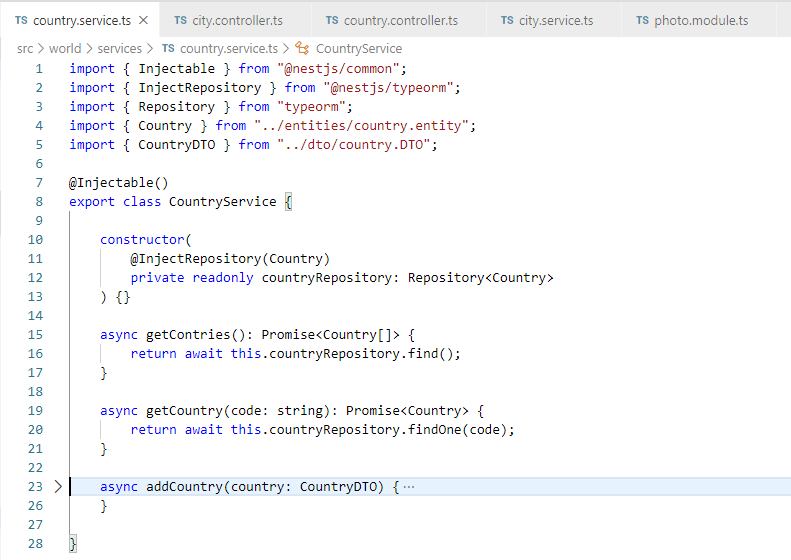
Escribimos entonces estos dos métodos en el servicio city.Service.ts. Vamos a definir allí la clase **CityService** necesitaremos importar primero algunas como el decorador Inyectable del @nestjs/common, el decorador InyectRepository de @nestjs/typeorm, la entidad City de citi.entity.ts, el DTO cityDTO de CityDTO.ts, el objeto Repository del modulo typeorm.

Definimos la clase CityService, y un constructor para la misma con una atributo del tipo Repository cityRepository (de esta manera decimos que se inyecta el repositorio). Y después empezamos a escribir cada una de las funciones que se llaman desde el Controller. Arrancamos con getCities() y getCity() que son las más sencillas. Allí recién se utilizan los métodos de typeORM find(), findOne() para lidiar contra la BD



De manera similar arrancaremos con el Servicio para Country (country.service.ts – CountryService) y el Controlador (country.controller.ts – Country.Controller)

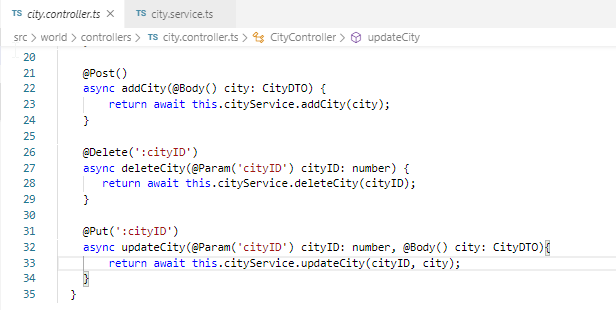
Entonces el servicio arrancaría más o menos así



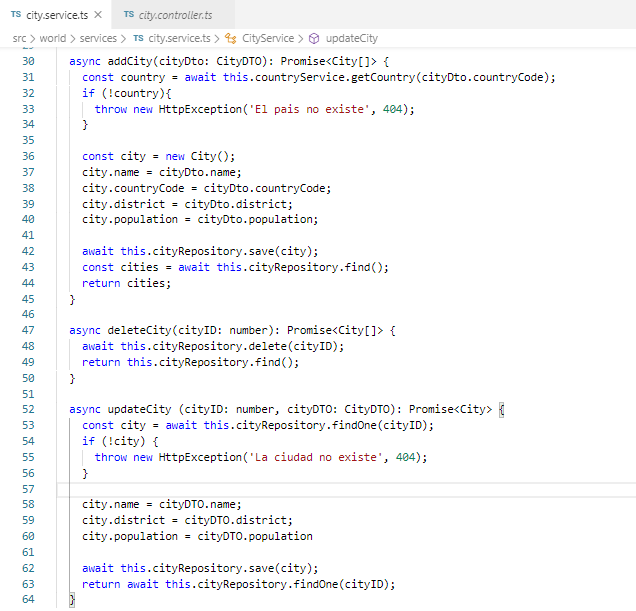
y el controlador así



Agregamos ahora para City tanto en el controlador como en el servicio las funciones para los métodos POST, DELETE y PUT



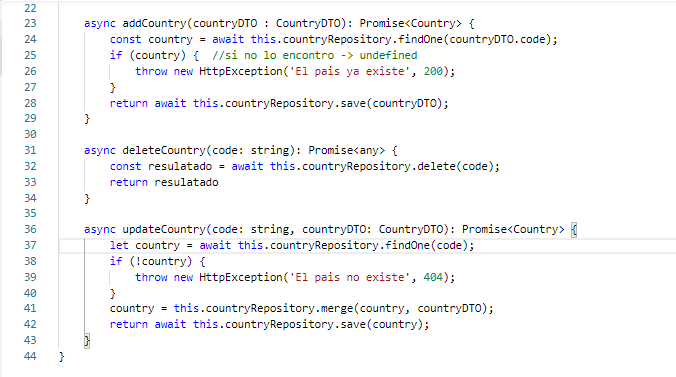
* Para el POST (addCity()) buscamos primero si existe el país con el codeCountry que posee la ciudad. Si no existe (retorna undefined) lanza una excepción indicando que el país no existe y allí termina sin insertar la ciudad. En otro caso, crea una entidad ciudad y la completa con la información que viene en el data transfer object cityDTO que viaja en el @Body del request y finalmente hace un save() en el repositorio cityRepository con esa entidad. Finalemente retorna el resultado de hacer un find() de las ciudades, podría haber realizado un findeone() de la ciudad para indicar, con el resultado, que la insertó. Lo que no se está controlando es que la ciudad ya exista, si quisiésemos hacerlo habría que realizar una búsqueda previa (con findone()) por el nombre dentro de las ciudades.
* Para el DELETE (deleteCity()) simplemente efectúa un delete() sobre el repositorio cityRepository, utilizando el argumento del cityID que se pasa en el request a través del @Param. El delete() retorna un json que muestra entre otras cosas si afecto a algún registro o no, pero en este caso se opto por retornar el resultado de hacer un find() de las ciudades, que mostrará que la misma ya no está. Pero podría haber elegido retornar (any) del resultado de la ejecución del delete().
* Para el PUT (updateCity()), se opto par hacer una búsqueda primero en el cityRepository con el cityID que se pasa en el request a través del @Param. Si la ciudad no existe se lanza una excepción indicando que la ciudad no existe. En otro caso, sobre la entidad city se asignan los valores del data transfer object cityDTO que viaja en el request en el @Body y finalmente se usa el método save() del repositorio cityRepository para persistir el cambio. Se podría haber utilizado el método de repositorio merge(entidad, dto) que realiza la actualización de la entidad conlos datos del dto y luego del merge() hacer el save() de la entidad. Debiera de obtenerse el mismo resultado. Finalmente en este caso retorno el resultado de un findone() del cityID sobre el respositorio, lo que debería mostrar la ciudad con el cambio efectuado.



Para Country vamos a proceder de la misma manera con los métodos POST, DELETE y PUT en el controlador como en el service donde escribiremos las funciones que los resolverán addCountry(), deleteCountry() y updateCountry(), respectivamente.



* Para el POST (addCountry()) buscamos primero si existe el país con el codeCountry que viaje en el request en el @Param. Si no existe (retorna undefined) lanza una excepción indicando que el país no existe y allí termina sin insertar el pais. En otro caso, realiza directamente un save() sobre el repositorio countryRepository con la información que viene en el data transfer object cityDTO que viaja en el @Body del request. El retorno el resultado del mismo save(), mostrará el json con los datos de la entidad que se insertó.



Para hacer una prueba de los controladores obviamente antes de haber desarrollado los servicios podríamos escribir sendas clases CityServiceFake y CountryServiceFake que leen los datos de de sendos archivos (city.data.ts y country.data.ts) que exportan variables que poseen un subconjunto de las ciudades y países. Las funciones getCities() y getCountries() retornan el contenido de estos archivos en vez de hacerlo desde la BD. Obviamente hay que hacer unos acomodos en los servicios inyectados en los constructores de los controladores. Pero a veces es mas rápido hacer una demostración con pocos datos sin tener en cuenta el tiempo de involucrar a la BD.



