**Estructura de carpetas del api**

En la carpeta **src** están los módulos iniciales para poder trabajar con ella e irán los futuros módulos del proyecto los cuales de pondrán su nombre en minuscula.

Explicación de los módulos ubicados en **src**.

|  |  |
| --- | --- |
| Nombre | Descripción |
| database | Módulo para la conexión con la base de datos. |
| dpa | Módulo para la división política administrativa del país. |
| security | Módulo para la seguridad de la aplicación. |
| shared | Módulo que tiene todas las clases que son comunes para utilizar en los otros módulos. |
| nomenclator | Módulo para trabajar con los nomencladores que se creen. |

A continuación, se explica la estructura de carpetas dentro del módulo security que es el más amplio de todos hasta ahora.

|  |  |
| --- | --- |
| Nombre | Descripción |
| controller | En esta carpeta se ponen todos los controladores que utilizamos en este módulo |
| decorator | En esta carpeta se ponen todos los decoradores que utilizamos en este módulo |
| dto | En esta carpeta se ponen todos los DTOs que utilizamos en este módulo |
| entity | En esta carpeta se ponen todas las entidades que utilizamos en este módulo |
| enum | En esta carpeta se ponen todos los enumeradores que utilizamos en este módulo |
| guard | En esta carpeta se ponen todos los guardianes que utilizamos en este módulo |
| interface | En esta carpeta se ponen todas las interfaces que utilizamos en este módulo |
| mapper | En esta carpeta se ponen todos los mapeadores que utilizamos en este módulo |
| repository | En esta carpeta se ponen todos los repositorios que utilizamos en este módulo |
| service | En esta carpeta se ponen todos los servicios que utilizamos en este módulo |
| strategy | En esta carpeta se ponen todas las estrategias que utilizamos en este módulo |

A medida que nos hagan falta trabajar con otras bondades que nos brinda nestjs haremos una carpeta en singular con su nombre y los ficheros que contengan dentro le pondremos de sufijo ese nombre. Estos ficheros estarán en minúscula y dentro le llamaremos a la clase en UpperCamelCase ejemplo para nombre del fichero dentro de la carpeta entity (role.entity.ts) y dentro de ella.

**export class** RolEntity **extends** GenericNomencaldorEntity

*{…}*

**Metodología para trabajar con los genéricos del api o también puede ser una forma de trabajo nuestra.**

1. Crear las entidades a trabajar.
2. Crear los dto para la entidad que serían los que se corresponden con el crud básico: son 6 DTOs. Ejemplos (listado.dto, create-role.dto, read-role.dto, update-role.dto, update-multiple-role.dto, response.dto).
3. Crear el mapper que será el intermediario entre los DTO y la entidad.
4. Se crea el repository que será el que contiene las consultas a la base de datos ya sean especificas o que hereden del GenericRepository.
5. Se crea el service que contiene las llamadas a las funciones hecha en el repository o que hereden del GenericService.
6. Se crea el controller para documentar el swagger ya sea que hereden del GenericController o no.

Explicación detallada:

Lo primero es crear nuestras entidades por buenas prácticas de nestjs le agregaremos el sufijo **Entity**. La clase **GenericEntity** contiene los atributos genéricos para trabajar con el softDelete y para lleva el control de cuando se crea y modifica los datos. Es obligado que si no heredamos de ella al menos tener el campo activo de tipo boolean en nuestra entidad si queremos trabajar con los genéricos.

En el ejemplo que se muestra **RolEntity** extiende de **GenericNomencladorEntity** que ha su vez esta está extendiendo de **GenericEntity**.

@Entity*(*'rol', *{* schema: 'mod\_auth' *})***export class** RolEntity **extends** GenericNomencaldorEntity *{}*

Por último, crearemos el constructor de la clase que servirá para trabajar con el mapper. Este ejemplo es de la entity **GenericNomencladorEntity** pero si fueran a trabajar haciendo otra entidad se debe de crear el constructor.

**constructor***(*nombre: **string**, descripcion: **string***) {* **super***()*;  
 **this**.nombre = nombre;  
 **this**.descripcion = descripcion;  
*}*

Por ultimo nos pedirá implementar el método **toString()** para utilizarlo en los mapper. Este ejemplo es de la entity **GenericNomencladorEntity** pero si fueran a trabajar haciendo otra entidad se debe de crear.

**public** toString*()*: **string** *{* **return this**.nombre;  
*}*

Para pasar al segundo paso que son los DTO. Se trabajan en el genérico con 6 DTO que sirven entre otras cosas para documentar los servicios en el controller y además actúan como plantillas para listar, mostrar, crear, actualizar, actualizar múltiples elementos. Por buenas prácticas se les puso así los nombres para poder identificarlos mejor, pero le pueden poner el nombre que quieran. Solo 2 de los 6 son reutilizables así que solo sería llamarlos en cada respuesta. Aclarar que se debe de crear los constructores para el DTO de **crear** y de **read** que serán utilizado en el mapper.

En el DTO de **read** tenemos que crear un atributo llamado **dtoToString** de tipo string pues este servirá para trabajar en el listado en el visual.

Se les pone un ejemplo del DTO de crear rol.

**import***{*IsNumber, IsString*}* **from** "class-validator";  
**import** *{*ApiProperty*}* **from** "@nestjs/swagger";**export class** ReadRolDto*{*

@IsString*({*message: 'El dtoToString debe de ser un string'*})* dtoToString: **string**;

@IsNumber*()* @ApiProperty*({* description: 'id del rol.', example: 1 *})* id: **number**;  
@IsString*()* @ApiProperty*({* description: 'Nombre del rol.', example: 'Administrador' *})* nombre : **string**;  
@IsString*()* @ApiProperty*({* description: 'Descripción del rol.', example: 'Tiene permiso total del api' *})* description: **string**;  
  
 **constructor***(*id: **number**, nombre: **string**, description: **string,** dtoToString: **string***) {* **this**.id = id;  
 **this**.nombre = nombre;  
 **this**.description = description;

**this**.dtoToString = dtoToString;  
 *}  
}*

El tercer paso son los mapper para pasar los datos de las entidades hacia el DTO y viceversa.

Como los mapper son proveedores de datos y estos son utilizados en los genéricos se utilizan las funciones llamadas **dtoToEntity**, **entityToDto** y **dtoToUpdateEntity.** En el método que cambia de la entidad hacia el dto utilizaremos el método toString() creado en la entidad. Ejemplo.

**import** *{*Injectable*}* **from** '@nestjs/common';  
**import** *{*CreateRolDto, ReadRolDto, UpdateRolDto*}* **from** '../dto'  
**import** *{*RolEntity*}* **from** "../entity";  
  
@Injectable*()***export class** RolMapper *{* dtoToEntity*(*createRolDto: CreateRolDto*)*: RolEntity *{* **return new** RolEntity*(* createRolDto.nombre,  
 createRolDto.descripcion  
 *)*;  
 *}* dtoToUpdateEntity*(*updateRolDto: UpdateRolDto, updateRolEntity: RolEntity*)*: RolEntity *{* updateRolEntity.nombre = updateRolDto.nombre;  
 updateRolEntity.descripcion = updateRolDto.descripcion;  
 **return** updateRolEntity;  
 *}* entityToDto*(*roleEntity: RolEntity*)*: ReadRolDto *{*

**const** dtoToString: **string** = rolEntity.toString*()*;**return new** ReadRolDto*(* rolEntity.id,  
 rolEntity.nombre,  
 rolEntity.descripcion,

dtoToString  
 *)*;  
 *}  
}*

Si ya todo esto esta implementado se puede pasar al paso cuatro que sería crear el repository acá va un ejemplo de cómo sería si utilizamos los genéricos por supuesto se pueden hacer en este repository las otras consultas e incluso sobrescribirlas.

**import** *{*Injectable, NotFoundException*}* **from** "@nestjs/common";  
**import** *{*GenericRepository*}* **from** "../../shared/repository/generic.repository";  
**import** *{*IRepository*}* **from** "../../shared/interface";  
**import** *{*InjectRepository*}* **from** "@nestjs/typeorm";  
**import** *{*Repository*}* **from** "typeorm";  
**import** *{*RolEntity*}* **from** "../entity";  
  
@Injectable*()***export class** RolRepository **extends** GenericRepository*<*RolEntity*>* **implements** IRepository*<*RolEntity*>{* **constructor***(* @InjectRepository*(*RolEntity*)* **private** rolRepository: Repository*<*RolEntity*>){* **super***(*rolRepository*)*;  
 *}  
}*

El quinto paso es crear el service en él se llaman al mapper y al repository para devolver los resultados al controller. Tambien si estamos trabajando con nuestros servicios y queramos que exista una traza tendríamos que llamarla desde aca. Si estamos utilizando también los genéricos nos pedirá el **TrazaService** y le pasamos un boolean para que sepa si queremos que utilice las trazas en los métodos ya implementados del crud. Ejemplo.

**import** *{* Injectable*}* **from** '@nestjs/common';  
**import** *{* RolEntity*}* **from** '../entity';  
**import** *{*RolRepository*}* **from** "../repository";  
**import** *{*RolMapper*}* **from** "../mapper";  
**import** *{*TrazaService*}* **from** "./traza.service";  
**import** *{*GenericService*}* **from** "../../shared/service/generic.service";  
**import** *{*IService*}* **from** "../../shared/interface";  
  
@Injectable*()***export class** RolService **extends** GenericService*<*RolEntity*>* **implements** IService*<*RolEntity*>{* **constructor***(* **protected** rolRepository: RolRepository,  
 **protected** rolMapper: RolMapper,  
 **protected** trazaService: TrazaService,  
 *){* **super***(*rolRepository,roleMapper,trazaService,**true***)*;  
 *}  
}*

Ya teniendo los servicios implementados pasamos al último paso que es el controller donde crearemos las documentaciones de swagger y las llamadas a los servicios. Se muestra un ejemplo del RoleController utilizando los genéricos, también se pueden guiar de este ejemplo para documentar cualquier otro controller.

**import** *{*Body, Controller, Get, Param, ParseIntPipe, Patch, Post, Query, UseGuards*}* **from** '@nestjs/common';  
**import** *{*CreateRoleDto, ReadRoleDto, UpdateMultipleRoleDto, UpdateRoleDto*}* **from** '../dto';  
**import** *{*RoleService*}* **from** '../service';  
**import** *{*GetUser, Roles*}* **from** "../decorator";  
**import** *{*RolType*}* **from** "../enum/roltype.enum";  
**import** *{*AuthGuard*}* **from** "@nestjs/passport";  
**import** *{*RolGuard*}* **from** "../guards/rol.guard";  
**import** *{*RolEntity, UserEntity*}* **from** "../entity";  
**import** *{*ConfigService*}* **from** "@atlasjs/config";  
**import** *{* ApiBearerAuth, ApiBody,  
 ApiNotFoundResponse,  
 ApiOperation,  
 ApiResponse,  
 ApiTags  
*}* **from** "@nestjs/swagger";  
**import** *{*GenericController*}* **from** "../../shared/controller";  
**import** *{*IController*}* **from** "../../shared/interface";  
**import** *{*ListadoDto, ResponseDto*}* **from** "../../shared/dto";  
  
@ApiTags*(*'Roles'*)*@Controller*(*'roles'*)*@Roles*(*RolType.*ADMINISTRADOR)*@UseGuards*(*AuthGuard*(*'jwt'*)*, RolGuard*)*@ApiBearerAuth*()***export class** RolController **extends** GenericController*<*RolEntity*>* **implements** IController*<*RolEntity*> {* **constructor***(* **protected** rolService: RolService,  
 **protected** configService: ConfigService  
 *) {* **super***(*rolService, configService, 'rol'*)*;  
 *}* @Get*()* @ApiOperation*({*summary: 'Obtener el listado de elementos del conjunto'*})* @ApiResponse*({* status: 200,  
 description: 'Listado de elementos del conjunto',  
 type: ListadoDto,  
 *})* @ApiNotFoundResponse*({* status: 404,  
 description: 'Elementos del conjunto no encontrados.',  
 *})* @ApiResponse*({*status: 401, description: 'Sin autorizacion.'*})* @ApiResponse*({*status: 500, description: 'Error interno del servicor.'*})* **async** findAll*(* @Query*(*'page'*)* page: **number** = 1,  
 @Query*(*'limit'*)* limit: **number** = 10*)*: Promise*<***any***> {* **const** data = **await super**.findAll*(*page, limit*)*;  
 **const** header: **string***[]* = *[*'id', 'Nombre', 'Descripción'*]*;  
 **const** listado: ListadoDto = **new** ListadoDto*(*header, data*)*;  
 **return** listado;  
 *}* @Get*(*':id'*)* @ApiOperation*({*summary: 'Obtener un elemento del conjunto'*})* @ApiResponse*({* status: 200,  
 description: 'Muestra la información de un elemento del conjunto',  
 type: ReadRolDto,  
 *})* @ApiNotFoundResponse*({* status: 404,  
 description: 'Elemento del conjunto no encontrado.',  
 *})* @ApiResponse*({*status: 401, description: 'Sin autorizacion.'*})* @ApiResponse*({*status: 500, description: 'Error interno del servicor.'*})* **async** findById*(*@Param*(*'id', ParseIntPipe*)* id: **number***)*: Promise*<*ReadRolDto*> {* **return await super**.findById*(*id*)*;  
 *}* @Get*(*'/elementos/multiples'*)* @ApiOperation*({*summary: 'Obtener multiples elementos del conjunto'*})* @ApiBody*({* description: 'Estructura para mostrar los multiples elementos del conjunto.',  
 type: *[*Number*]*,  
 *})* @ApiResponse*({* status: 200,  
 description: 'Muestra la información de multiples elementos del conjunto',  
 type: *[*ReadRolDto*]*,  
 *})* @ApiNotFoundResponse*({* status: 404,  
 description: 'Elementos del conjunto no encontrados.',  
 *})* @ApiResponse*({*status: 401, description: 'Sin autorizacion.'*})* @ApiResponse*({*status: 500, description: 'Error interno del servicor.'*})* **async** findByIds*(*@Body*()* ids: **number***[])*: Promise*<*ReadRolDto*[]> {* **return await super**.findByIds*(*ids*)*;  
 *}* @Post*()* @ApiOperation*({*summary: 'Crear un elemento del conjunto.'*})* @ApiBody*({* description: 'Estructura para crear el elemento del conjunto.',  
 type: CreateRolDto,  
 *})* @ApiResponse*({*status: 201, description: 'Crea un elemento del conjunto.', type: ResponseDto*})* @ApiResponse*({*status: 401, description: 'Sin autorizacion.'*})* @ApiResponse*({*status: 500, description: 'Error interno del servicor.'*})* **async** create*(*@GetUser*()* user: UserEntity, @Body*()* createRolDto: CreateRolDto*)*: Promise*<*ResponseDto*> {* **return await super**.create*(*user, createRolDto*)*;  
 *}* @Post*(*'/multiple'*)* @ApiOperation*({*summary: 'Crear un grupo de elementos del conjunto.'*})* @ApiBody*({* description: 'Estructura para crear el grupo de elementos del conjunto.',  
 type: *[*CreateRolDto*]*,  
 *})* @ApiResponse*({*status: 201, description: 'Crea un grupo de elementos del conjunto.', type: ResponseDto*})* @ApiResponse*({*status: 401, description: 'Sin autorizacion.'*})* @ApiResponse*({*status: 500, description: 'Error interno del servicor.'*})* **async** createMultiple*(*@GetUser*()* user: UserEntity, @Body*()* createRolDto: CreateRolDto*[])*: Promise*<*ResponseDto*> {* **return await super**.createMultiple*(*user, createRolDto*)*;  
 *}* @Patch*(*':id'*)* @ApiOperation*({*summary: 'Actualizar un elemento del conjunto.'*})* @ApiBody*({* description: 'Estructura para modificar el elemento del conjunto.',  
 type: UpdateRolDto,  
 *})* @ApiResponse*({*status: 201, description: 'El elemento se ha actualizado.', type: ResponseDto*})* @ApiResponse*({*status: 401, description: 'Sin autorizacion.'*})* @ApiResponse*({*status: 500, description: 'Error interno del servicor.'*})* **async** update*(*@GetUser*()* user: UserEntity, @Param*(*'id', ParseIntPipe*)* id: **number**, @Body*()* updateRolDto: UpdateRolDto*)*: Promise*<*ResponseDto*> {* **return await super**.update*(*user, id, updateRolDto*)*;  
 *}* @Patch*(*'/elementos/multiples'*)* @ApiOperation*({*summary: 'Actualizar un grupo de elementos del conjunto.'*})* @ApiBody*({* description: 'Estructura para modificar el grupo de elementos del conjunto.',  
 type: *[*UpdateMultipleRolDto*]*,  
 *})* @ApiResponse*({*status: 201, description: 'El grupo de elementos se han actualizado.', type: ResponseDto*})* @ApiResponse*({*status: 401, description: 'Sin autorizacion.'*})* @ApiResponse*({*status: 500, description: 'Error interno del servicor.'*})* **async** updateMultiple*(*@GetUser*()* user: UserEntity, @Body*()* updateMultipleRolDto: UpdateMultipleRolDto*[])*: Promise*<*ResponseDto*> {* **return await super**.updateMultiple*(*user, updateMultipleRolDto*)*;  
 *}  
}*

Acá es donde se aprecia mejor la utilización de los DTO y se logra dividir en capas nuestra estructura. No es conveniente en el controller devolver entidades por un tema de seguridad.

En esta tabla se puede ver las capas las cuales están muy relacionadas entre sí.

|  |  |  |
| --- | --- | --- |
| Capas | Clases | Descripción |
| Controller | DTOs | Utilizados para mostrar al cliente y ayudar a documentar en el swagger. |
| Servicios | Son los que traen los resultados y utilizan los respectivos DTOs para dárselo al controller. |
| Servicio | Mappers | Pasan los datos de las entities que devuelve el repository al respectivo DTO que devolverá el servicio. |
| Servicios | Llamadas a servicios de terceros para luego procesar la información en el mapper. |
| Repositories | Llamadas a las funciones que hacen las consultas en la base de datos y devuelven los resultados para procesarlos en el mapper. |
| Mapper | DTO | Estructura deseada para el cliente. |
| Entity | Datos que son obtenidos por los repositories de la base de datos y que es llamado desde el servicio. |
| Repository | Entity | Son utilizadas para captar y devolver información al servicio. |
| Repositories | Llamadas a las funciones que hacen otros repositorios a la base de datos para resolver la información que nos piden en la lógica de programación. |

Para trabajar con el módulo de nomencladores.

Este módulo se creo para gestionar todas las entidades que sean nomencladores se explicara como se debe de crear las cosas para que funcione. Además, contiene todos los enpoints para trabajar y uno que le dirá al programador frontend como se llaman los nomencladores que se encuentran implementados para así no tener error a la hora de llamarlos.

Primero se crean las entidades dentro de la carpeta **entity** las mismas extienden de **GenericNomencladorEntity.**

Luego tienen que actualizar el **NomencladorTypeEnum** en él se ponen los nombres de los nomencladores. Ejemplo

**export enum** NomencladorTypeEnum *{*

  TIPOALERTA = 'tipoAlerta',

*}*

Esto servirá para poder llamar a cada nomenclador cuando se utilicen en los otros módulos. También el nombre que pongan tiene que ser con el que empiezan los repository de cada entidad.

Después que tenemos actualizado el enum pasamos a crear los repository dentro de **GenericNomencladorRepository** este se encargara de las consultas para cada nomenclador. Se muestra un ejemplo para crear los repository. En amarillo está el nombre que tiene en el **NomencladorTypeEnum** para poder llamarlo en los servicios correctamente.

**export class** GenericNomencladorRepository *{* **constructor***(*

@InjectRepository(TipoAlertaEntity)  
 protected tipoAlertaRepository: Repository<TipoAlertaEntity>,  
 *) {}*

*}*