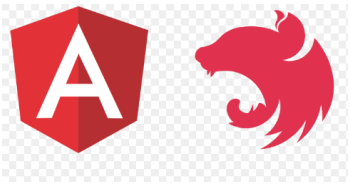
Estructura en Nest Js



¿Que es Nest Js?

NestJS es un framework para el desarrollador de aplicaciones Node.js en el lado del servidor. Se programa en TypeScript y proporciona una arquitectura en la aplicación que permite el desarrollo de aplicaciones más fáciles de mantener. Su arquitectura está bastante inspirada en Angular lo que facilita el trabajo al equipo de desarrollo al no tener que usar dos formas diferentes de trabajo en el backend y en el frontend.

Nuestro Proyecto de la tabla Curriculum se conforma de lo siguiente por el momento:

1.Dtos

2.Entytis

3. Services

4. Controller

5.Module

- ***Dtos :***

Dto son las siglas de Data Transfer Object y no es más que un objeto que se transfiere por la red entre dos sistemas, típicamente usados en aplicaciones cliente/servidor y en las aplicaciones web modernas.

- **Import {}**

- Lo primero que tenemos en nuestro dto son las importaciones de la clase “class-validator ” las cuales son las siguientes

```
import {
  IsNumber,
  IsString,
  MinLength,
  IsDate,
  Min,
  Max,
  IsOptional,
  MaxLength,
  IsPositive,
  IsDateString,
} from 'class-validator';
```

- Las cuales nos sirven para dar validaciones a nuestros datos de nuestras tablas en este caso de Curriculum

➢ Export class CreateCurriculumDto {}

- Tenemos nuestro CreateCurriculumDto el cual nos sirve para poner nuestros campos de nuestra tabla con validaciones acertadas por ejemplo para los tipo de dato : s**tring ,number Date , Bolean** con su respectivo mensaje de lo que se requiera en cada campo .

```
export class CreateCurriculumDto {
  @IsNumber()
  //@IsPositive()
  readonly careerId: number;//fk

  @IsNumber()
  //@IsPositive()
  readonly stateId: number;//fk

  @IsString()
  @MinLength(3,{message:"El numero de caracteres minimo es 3"})
  @MaxLength(80,{message:"El numero de caracteres maximo es 80"})
  readonly code: string; /// si

  //@IsDate()
  //@Type(() => Date)
  @IsDateString({message:"Dato tipo fecha"})
  readonly endedAt: Date;//si

  @IsString()
  @MinLength(2,{message:"El numero de caracteres minimo es 2"})
  @MaxLength(100,{message:"El numero de caracteres maximo es 100"})
  readonly description: string;//si



  @IsString()
  @MinLength(2,{message:"El numero de caracteres minimo es 2"})
  @MaxLength(255,{message:"El numero de caracteres maximo es 255"})
  readonly name: string;//si


  @IsNumber()
  @IsPositive()
  @Min(1,{message:"El numero minimo es 1 "})
  @Max(40,{message:"El numero maximo es 40"})
  readonly periodicAcademicNumber: number;//si
```

- ***Entity :***

- Una Entity es el cual define un tipo de recurso con el cual vamos a trabajar en una aplicación, que se asociará directamente con una tabla de una base de datos

➢ Creamos la Entity

- Procedemos a crear la Entity con un decorador = “@Entity” con su respectiva clase =” Curriculum Entity” el cual tiene otro decorador llamado =” @PrimaryGeneratedColum()”

```
@Entity("curricular")
export class CurriculumEntity{
    @PrimaryGeneratedColumn()
    id:number;
```

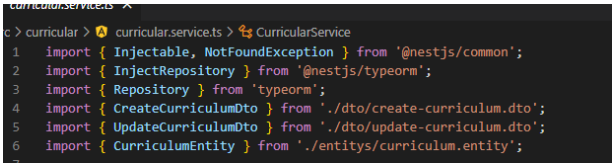
➢ Dentro de la Entity

- Dentro de la clase “Curriculum Entity{}”usamos el decorador =”@Column” para campo de nuestros campos de la tabla Curriculum el cual tendrá su respectivo tipo de dato , como se va a llamar en la Base de datos por ejemplo endedAt = “ended_at” con un comentario que se refiera a lo que necesita en cada campo e decir de que se trata el campo seleccionado .

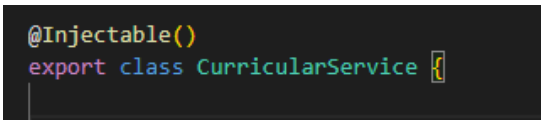
```
@Column("varchar", {length:400, name:"code",comment:"Codigo del Curriculum"})
 code:string;

@Column("varchar", {length:400,name:"description",comment:"Descripcion del Curriculum"})
description:string;

@Column("date",{name:"ended_at" ,comment:"Fecha de Finalizacion"})
 endedAt:Date;


@Column("varchar", {length:400,name:"name" ,comment:"Nombre del curriculum"})
name:string;

@Column("int",{name:"periodic_academic_number",comment:"Numero Academico del curriculum"})
periodicAcademicNumber:number;


@Column("varchar", {length:100,name:"resolution_number",comment:"Numero de Resolucion del curriculum"})
resolutionNumber:string;

@Column("date",{name:"started_at",comment:"Fecha De Incio"})
startedAt:Date;

@Column("int",{name:"weeks_number",comment:"Numero de Semanas del Curriculum"})
weeksNumber:number;


@CreateDateColumn({
    name:"created_at",
    type:"timestamptz",
    default: () => "CURRENT_TIMESTAMP",
})
createdAT:Date;
```

- ***Services :***
  - Los servicios son una pieza esencial de las aplicaciones realizadas con el framework NestJS, el proporciona una capa de acceso a los datos, mediante los servicios podemos liberar de código a los controladores y conseguir desacoplar éstos de las tecnologías de almacenamiento de datos que estemos utilizando.

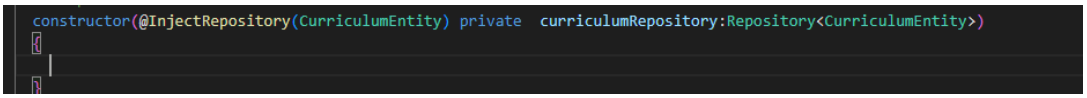
- **Creamos El Servicio/ Service**
  - Primer Paso siempre traer las importaciones que requerimos.



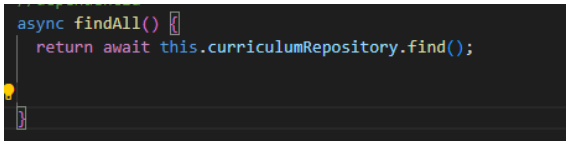
  - Creamos un decorador ="Injectable() " dentro de este decorador exportamos una clase en este caso llamado "CurriculumService{}"



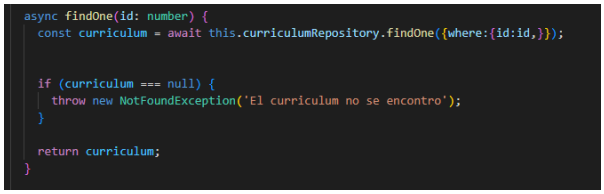
  - Dentro del CurriculumService{} creamos un constructor en cual tiene un decorador =" @InjectRepository" el cual de la Entity de Curriculum haga un repositorio llamado "curriculumRepository"



  - **findAll:** es una función asíncrona que nos va a devolver un valor con el Postman en este caso va a devolver el curriculumRepository con el.find()

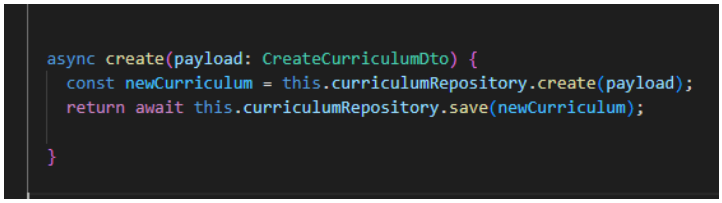


- **findOne:** Es una función asíncrona que nos va a devolver un valor con el Postman, pero con un valor en este caso con un id eso quiere decir que el curriculumRepository nos dará nuestros datos en función al id que estemos buscando y si en caso no nos encuentra con el id requerido pues nos saltará a una sentencia if el cual nos dirá que con ese id en el repositorio no ha sido encontrado.



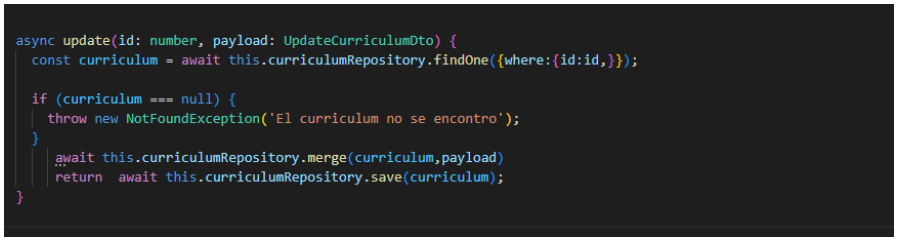
```
async findOne(id: number) {
  const curriculum = await this.curriculumRepository.findOne({where:{id:id,}});


  if (curriculum === null) {
    throw new NotFoundException('El curriculum no se encontro');
  }

  return curriculum;
}
```

- **créate:** Es una función asíncrona que nos va a devolver un valor creado con nuestros datos una vez le digamos al Postman es decir un Curriculum completo creado aleatoriamente



```
async create(payload: CreateCurriculumDto) {
  const newCurriculum = this.curriculumRepository.create(payload);
  return await this.curriculumRepository.save(newCurriculum);

}
```

- **update:** Es una función asíncrona que nos va a devolver un valor actualizado es decir que alguna de nuestros datos de algún Curriculum q tengamos realizado podamos cambiar un valor en algún tipo de dato requerido y si en caso no tengamos ese Curriculum creado pues nos devolverá una excepción que el Curriculum no se ha encontrado



```
async update(id: number, payload: UpdateCurriculumDto) {
  const curriculum = await this.curriculumRepository.findOne({where:{id:id,}});

  if (curriculum === null) {
    throw new NotFoundException('El curriculum no se encontro');
  }
    await this.curriculumRepository.merge(curriculum,payload)
    return  await this.curriculumRepository.save(curriculum);
}
```

- **update:** Es una función asíncrona que nos va a devolver un Curriculum eliminado es decir eliminara de la base de datos este id con los datos que tenga en el.

```
async delete(id: number) {
  return await  this.curriculumRepository.softDelete(id);

  }
}
```

- ***Controller :***
  - Los controladores son una de las piezas principales de las aplicaciones. Básicamente nos sirven para dar soporte o responder las solicitudes realizadas al servidor.

➢ Creamos El Controlador/ Controller

- Primer Paso importamos todos los métodos que vayamos a usar de nest/common al Postman:

```
import {
  Body,
  Controller,
  Delete,
  Get,
  HttpCode,
  HttpStatus,
  Param,
  ParseIntPipe,
  Post,
  Put,
  Query,
} from '@nestjs/common';
```

- Importamos los servicios y los dto

```
import { CreateCurriculumDto } from './dto/create-curriculum.dto';
import { UpdateCurriculumDto } from './dto/update-curriculum.dto';
import { CurricularService } from './curricular.service';
```

- Creamos un decorador = " Controller" el cual nos permita hacer una clase llamada = CurricularController el cual tenga dentro un constructor que llame al CurriculumService

```
export class CurricularController {
  constructor(private curricularService: CurricularService) {}
```

- Llamamos al findAll() mediante el servicio de Curriculum para que nos devuelva valores en el Postman en este caso todos los id realizados

```
@Get('')
@HttpCode(HttpStatus.OK)
async findAll (@Query() params: any) {
  const response = await this.curricularService.findAll();

  return response;
  //return {
    //data: response,
    //message: `index`,
  //};
}
```

- Llamamos al findOne() para que nos devuelva un valor en el Postman que requerimos buscar con un Id

```
@Get(':id')
@HttpCode(HttpStatus.OK)
async findOne(@Param('id', ParseIntPipe) id: number) {
  const response =  await this.curricularService.findOne(id);
  return response;
  //return {
    //data: response,
    //message: `show`,
  //};
}
```

- Llamamos al create() para que nos crea un valor nuevo en el Postman con nuestros tipos de datos es decir uno nuevo

```
@Post('')
@HttpCode(HttpStatus.CREATED)
async  create (@Body() payload: CreateCurriculumDto) {
  const response = this.curricularService.create(payload);
  return response;
  //return {
    //data: response,
   // message: `created`,
  //};
}
```

- Llamamos al update() para que nos actualiza un valor en el Postman del dato o la tabla que requerimos cambiar con el id

```
@Put(':id')
@HttpCode(HttpStatus.CREATED)
update(
  @Param('id', ParseIntPipe) id: number,
  @Body() payload: UpdateCurriculumDto,
) {
  const response = this.curricularService.update(id, payload);
  return response;
  //return {
    //data: response,
    //message: `updated ${id}`,
  //};
}
```

- Llamamos al delete() para que nos elimine un valor en el Postman en este caso con el id queremos eliminar una tabla lo borra de la base de datos

```
@Delete(':id')
@HttpCode(HttpStatus.CREATED)
async delete (@Param('id', ParseIntPipe) id: number) {
  const response = this.curricularService.delete(id);
  return response;
  //return {
    //data: response,
    //message: `deleted`,
  //};
}
}
```

- ***Module :***
  - Los módulos son clases que funcionan como contenedores de otras clases o artefactos, como son los controladores, servicios y otros componentes desarrollados con Nest.
  - Los módulos sirven para agrupar elementos, de modo que una aplicación podrá tener varios módulos con clases altamente relacionadas entre sí.

➢ Creamos El Module

- Primer Paso importamos todos lo que vayamos a usar en el modulo

```
app.module.ts > ...
import { HttpModule, Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ConfigModule } from '@nestjs/config';
import { enviroments } from './enviroment';
import config from './config';
import { DatabasesModule } from './database/database.module';
import { CurricularModule } from './curricular/curricular.module';
```

- Usamos un decorador = “Module” el cual dentro importamos todo que usemos en nuestra app en este caso la Database Module y el Curricular Module también los controladores y el App Service y exportamos de la clase App Module

```
@Module({
  //decorador
  imports: [
    ConfigModule.forRoot({
      envFilePath: enviroments[process.env.NODE_ENV] || '.env',
      isGlobal: true,
      load: [config],
      validationSchema: Joi.object({
        DB_NAME: Joi.string().required(),
        DB_PORT: Joi.number().required(),
      }),
    }),
    HttpModule,
    DatabasesModule,
    CurricularModule,
    CurricularModule,

  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {} //afecta aqui
```