NOMBRE: Brayan Andrade
CURSO: 5to “A”

Documentación.

Tenemos al servicio que nos ayuda a manipular la base de datos de datos, y se comunica con el backend. Podemos encontrar en el código lo siguiente:

Las importaciones que vamos a usar.

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { Repository } from 'typeorm';
import { InjectRepository } from '@nestjs/typeorm';
import { CareerEntity } from './entities/career.entity';
import { CreateCareerDto } from './dto/create-career.dto';
import { UpdateCareerDto } from './dto/update-career.dto';
```

Y los injectables.

```
@Injectable()
export class CareersService {
  constructor(
    @InjectRepository(CareerEntity)
    private careerRepository: Repository<CareerEntity>,
  ) {}
```

Los metodos

```
async create(payload: CreateCareerDto) {
  const newCareer = this.careerRepository.create(payload);

  return await this.careerRepository.save(newCareer);
}

async delete(id: number) {
  return await this.careerRepository.softDelete(id);
}

async findAll() {
  return await this.careerRepository.find();
}
```

NOMBRE: Brayan Andrade
CURSO: 5to “A”

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

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

  return career;
}

async update(id: number, payload: UpdateCareerDto) {
  const career = await this.careerRepository.findOne({
    where: {
      id: id,
    },
  });

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

Encontramos los metodos que nos ayudaran a crear, actualizar, crear, y elminar los datos según su id en especifico o masivamente. En el cual lo realizamos asincronamente para que no nos devuelva un objeto vacio, al igual el await que funciona conjuntamente con el async, y nos dice que va esperar que nos llegue algol cualquier cosa de la base de datos o de donde estemos realizando la consulta. Tambien encontramos un contructor que con la ayuda de la inyeccion de dependencias, podemos comunicarlo con el controlador y la dto.

LOS CONTROLADORES.

Se comunica con el servidor y nos ayuda a responder las solicitudes del mismo.

Tenemos decoradores los cuales se inicializan con un arroba y nombre del decorador. El decorador trabaja con las clases, funciones y variables.

El nombre de las rutas lo estamos trabajando en plural.

NOMBRE: Brayan Andrade
CURSO: 5to “A”

```
@Controller("careers")
export class CareersController {
    constructor(private careersService: CareersService) {}

    @Get('')
    @HttpCode(HttpStatus.OK)
    findAll(@Query() params: any){
        const response = this.careersService.findAll();
        return response;
            //data: response,
            //message: 'index'
        //};
    }
```

El el sigueinte codigo tenemos un decorador que es un metodo el cual es el metodo GET

El decorador @Httpcode() me va a permitir modificar el estado de la respuesta.

El metodo **FINDALL()** nos ayuda a realizar una consulta de todos los elementos que tenemos en nuestra tabla

El metodo FindOne el cual nos sirve para buscar un objeto en especifico en este caso con su respectivo id.

```
  @Get(':id')
  @HttpCode(HttpStatus.OK)
  findOne(@Param('id', ParseIntPipe) id: number) {
    const response = this.careersService.findOne(id);

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

Tenemos el metodo **@Post** el cual nos sirve para crear un nuevo dato. Dentro de nuestra tabla.

NOMBRE: Brayan Andrade
CURSO: 5to “A”

```
 @Post('')
@HttpCode(HttpStatus.CREATED)
create(@Body() payload:CreateCareerDto) {
  const response = this.careersService.create(payload);
return response;

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

Tenemos el método @Put que es para actualizar un elemento en específico, según su id

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

Tenemos el método @Delete el cual nos sirve para eliminar mediante la id.

```
@Delete(':id')
@HttpCode(HttpStatus.CREATED)
delete(@Param('id', ParseIntPipe) id: number) {
  const response = this.careersService.delete(id);

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

El import nos sirve para importar las funciones que han sido exportadas desde un moduli externo.

NOMBRE: Brayan Andrade
CURSO: 5to “A”

En el modulo tendremos las exportaciones que haremos y allí estarán los controladores y los servicios.

```
@Module({
  imports:[TypeOrmModule.forFeature([CareerEntity])],
  controllers: [CareersController],
  providers: [CareersService]
})
export class CareerModule {}
```

ENTITIES

Las entidades nos sirven para mapear y poder conectar entre la base de datos.

```
@Entity('careers')
export class CareerEntity {
  @PrimaryGeneratedColumn()
  id: number;
```

Tenemos que importar un decorador el cual es el @column y el nombre de nuestra tabla a continuación tenemos los campos que están en nuestra tabla con sus respectivas validaciones.

```
@Column('varchar', {
    length: 10,
    comment: 'Acronimo de la carrera',
    name: 'acronym'
})
acronym: string;
```

Los DTO.

Me van a servir para validad la data que viene en el cuerpo de la petición.

NOMBRE: Brayan Andrade
CURSO: 5to “A”

```
from 'class-validator';

export class CreateCareerDto {

  @IsNumber()
  @IsPositive()
  readonly institutionId: number;

  @IsNumber()
  @IsPositive()
  readonly state: number;

  @IsNumber()
  @IsPositive()
  readonly type: number;

  @IsString()
  @MinLength(2, { message: 'El acronimo debe tener al menos 2 caracteres' })
  @MaxLength(10, { message: 'El acronimo no puede tener más de 10 caracteres' })
  readonly acronym: string;

  @IsString()
  @MinLength(1, { message: 'El codigo debe tener al menos 1 caracter' })
  @MaxLength(50, { message: 'El codigo no puede tener más de 50 caracteres' })
  readonly code: string;

  @IsString()
  @MinLength(1, { message: 'El codigo debe tener al menos 1 caracter' })
  @MaxLength(50, { message: 'El codigo no puede tener más de 50 caracteres' })
  readonly codeSniese: string;

  @IsString()
  @IsOptional()
  readonly logo: string;
```

Usamos decoradores para realizar validaciones y en caso de no cumplir con alguna de estas nos enviara un mensaje.

Aquí ejemplo de validaciones:

IsInt: esta validación nos sirve para reconcoer si es un entero o un decimal, solo acepta enteros.

Length: Valida el numero maximo de caracteres en un campo.

IsEmail: Valida que sea un correo electrónico**.**

IsDate: Valida que sea una fecha.

Min: Valida el número mínimo que podemos usar en un campo.

Max: Valida el número máximo que podemos usar en un campo.

IsFQDN: Valida que sea una dirección de internet.