INSTITUTO SUPERIOR TECNOLÓGICO

DE TURISMO Y PATRIMONIO "YAVIRAC"

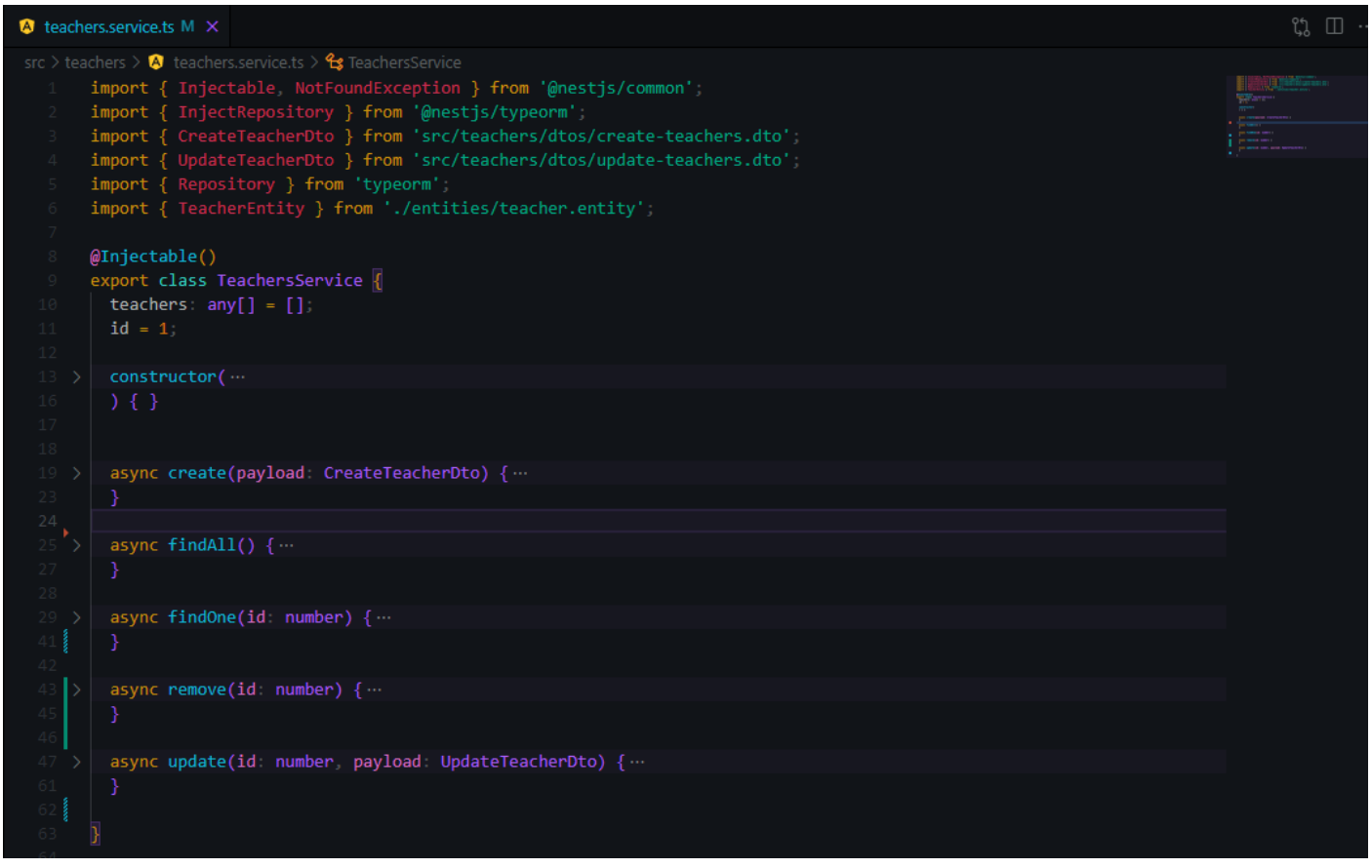
Dirección: García Moreno S435 y Ambato

Quito – Ecuador

Te DATOS GENERALES			
Carrera:	**Asignatura:**	**Apellidos:**	**Nombres:**
Tecnología Superior en Desarrollo de Software	Tendencias actuales de programación	Mejía Morocho	Zaida Stephania
Periodo Lectivo:	**Curso:**	**Fecha:**	**Docente:**
Mayo 2022 – Octubre 2022.	Quinto Vespertino "B"	26 de Junio del 2022.	Ing.MAURICIO TAMAYO

DOCUMENTACIÓN SOBRE SERVICIO, MODULE, DTO, CONTROLADORES Y ENTITY

- ## SERVICIO



Los servicios cuenta con un tipo especial llamado **provider** el cual nos permite englobar un conjunto diverso de servicios, repositorios, etc. Estos se los utilizan inyectándolos en los controladores de la aplicación y si es necesario se las inyecta en otras clases.

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

Para poder crear los servicios existe un comando especial el cual es: **nest generate service teachers** este nos ayudara a crear solo el archivo de services este comando es de la forma larga pero también existe uno que es más corto y cumple la misma función el cual es: **nest g s teachers**. Existe un comando que nos ayuda a crear todos los archivos necesarios que utilizares como el controlador, services, module, etc el cual es: **nest g s teachers --no-spec** este nos permite crear todos los archivos exceptuando eltest unitariosque no se utiliza.

Este archivo cuenta con un decorador llamado **@Inyectable** el cual es especialmente importante dentro del services, ya que gracias a este decorador estamos creando una clase que será capaz de inyectarse en los controladores (todos los servicios deben contar con este decorador para poder usar la inyección de dependencias).

```
@Injectable()
export class TeachersService {
```

Los services se conectan a los controladores mediante inyección de dependencia para que este bien inyectado tenemos que seguir dos pasosimportantes:

- **Importar el servicio (service) que queremos usar**
- **Inyectarlo en el constructorde la clase**

Teniendo realizados los pasos anteriores el código quedaría así:

```
import { ApiTags, ApiOperation } from '@nestjs/swagger';
import { CreateTeacherDto } from 'src/teachers/dtos/create-teachers.dto';
import { UpdateTeacherDto } from 'src/teachers/dtos/update-teachers.dto';
import { TeachersService } from 'src/teachers/teachers.service';

@Controller('teachers')
export class TeachersController {
  constructor(private teachersService: TeachersService) { }
```

Los servicios están pensados para comunicarse con la base de dato, es decir sirve como un intermediario entre los dos la base de datos y el framework, en otras palabras es el encargado de gestionar los **datos**de la aplicación, de modo que realiza las operaciones para obtener esos datos, actualizarlos, crearlos y eliminarlos; cada una de estas acciones tienen sus propios comandos los cuales son:

Para Crear (create):

```
async create(payload: CreateTeacherDto) {
  const newTeacher = this.teacherRepository.create(payload);

  return this.teacherRepository.save(newTeacher);
}
```

Este usa directamente un método **create** el cual nos ayuda a crear un nuevo array con los datos que nosotros hayamos solicitado anteriormente como: nombre, apellido, etc. Todos estos datos que se solicitan se encuentran guardados en el **teacherRepository,** el **payload** nos ayuda a aceptar los datos que se envían ya que estos vienen en formato **JSON** en el **body** o cuerpo del mensaje, para que seguidamente se guarden en la base de datos **(returnthis.teacherRepository.save(newTeacher)**y se devuelva **(return)** al cliente los datos ingresados y guardados.

Para Eliminar (remove):

```
43    async remove(id: number) {
44      return this.teacherRepository.softDelete(id);
45    }
```

Este usa directamente un método **remove** el cual necesita un **id** del docente que deseamos eliminar este nos retorna un array vacío en forma de que la petición se realizo correctamente, el **softDelete** nos permite hacer una eliminación lógica no física esto quiere decir que el dato eliminado no estará disponible para el servicio y solo será visible en la base de datos.

```
"generatedMaps": [],
"raw": [],
"affected": 1
```

Para Consultar todos los datos (findAll):

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

Este utiliza directamente el método **findAll** el cual realiza la consulta a todos los datos que estén dentro de la tabla, este no devuelve la data este solo consulta lo que ya está guardado anterior mente en la base de datos.

Para Consultar un dato en específico (findOne):

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

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

  return teacher;
}
```

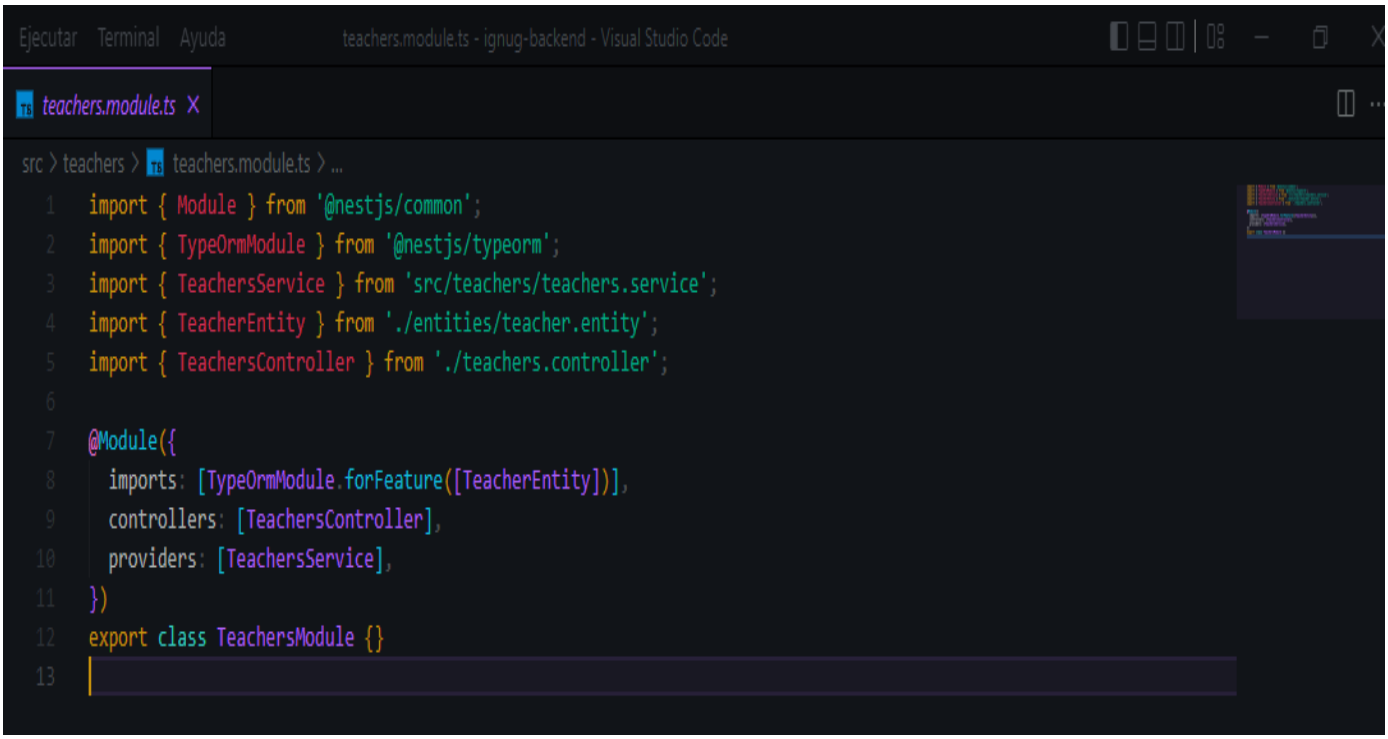
Este usa directamente un método **findOne** el cual nos devuelve un array con la información del docente que se le solicito al momento de mandar el **id**; en caso de que este **id** no exista dará paso a la sentencia **if** la cual dice que si el valor es nulo nos saldrá un mensaje **(NotFoundException)** el cual dice "El teacher no se encontró", si el valor existe nos dará un return con la información existente.

Para Actualizar (upDate):

```
async update(id: number, payload: UpdateTeacherDto) {
  const teacher = await this.teacherRepository.findOne({
    where: {
      id: id,
    },
  });

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

  this.teacherRepository.merge(teacher, payload);

  return this.teacherRepository.save(teacher);
}
```

Este usa directamente un método **upDate** el cual nos solicita un **id** y su **payload** (datos solicitados) con estos dos el método sabe a qué docente queremos realizarle una actualización de la información, el **teacherRepository.findOne (nos ayuda a traer la información que ya tenía anteriormente el Id);** en caso de que el **id** no exista nos saltara directo a la sentencia **if** la cual dice que si el valor es nulo nos saldrá un mensaje **(NotFoundException)** el cual dice "El docente no se encontró", en caso de que este si exista nos actualizara los datos que le solicitamos y nos devolverá un array con los nuevos datos los cuales se guardaron de ante mano en la base de datos.

- **MODULE**



Los módulos son clases que funcionan como contenedores de otras clases o artefactos, como son los **controladores, servicios y otros componentes;** estos agrupan los elementos de modo que una aplicación puede tener varios módulos con clases altamente relacionadas entre ellas.

@Module es un decorador del archivo module.ts, este toma un solo objeto cuyas propiedades describen el módulo. El módulo **encapsula** proveedores por defecto, esto significa que es imposible inyectar proveedores que no sean directamente parte del módulo actual ni exportados de los módulos importados. Por lo tanto, puede considerar los proveedores exportados de un módulo como la interfaz pública o API del módulo.

Dentro del@Module podremos definir los siguientes elementos:

`providers`	los proveedores que serán instanciados por el inyector de Nest y que pueden compartirse al menos en este módulo
`controllers`	el conjunto de controladores definidos en este módulo que deben ser instanciados
`imports`	la lista de módulos importados que exportan los proveedores que se requieren en este módulo
`exports`	el subconjunto de `providers` eso lo proporciona este módulo y debería estar disponible en otros módulos que importan este módulo. Puede usar el proveedor en sí o solo su token (`provide` valor)

Este archivo module.ts se puede crear atraves del comando **nest generate module teachers**(es el comando de forma larga) pero también de forma corta el cual es: **nest g mo teachers**(esta es su manera corta), también este archivo se crea junto con los demás con el comando **nest g s teachers --no-spec**

- **CONTROLADORES**

```
15  import { CreateTeacherDto } from 'src/teachers/dtos/create-teachers.dto';
16  import { UpdateTeacherDto } from 'src/teachers/dtos/update-teachers.dto';
17  import { TeachersService } from 'src/teachers/teachers.service';
18
19  @Controller('teachers')
20  export class TeachersController {
21    constructor(private teachersService: TeachersService) { }
22
```

Los controladores nos sirven para poder dar soporte o responder las solicitudes realizadas al servidor en otras palabras los controladores hace tres acciones principales las cuales son:

1. **Publica las rutas o recursos que tiene el backend**
2. **Recibe la petición**
3. **Responde la petición realizada al servidor**

El uso de un prefijo de ruta en un **@Controller()** decorador nos permite agrupar fácilmente un conjunto de rutas relacionadas y minimizar el código repetitivo, asi traer los archivos services.

Estos archivos controller.ts se crean con el comando **nest generate controller teachers** (esta es su forma larga) y su forma más corta **nest g co teachers,** también este se puede crear en un solo conjunto con los demás con el comando **nest g s teachers --no-spec.**

```
23    @Get('')
24    @HttpCode(HttpStatus.OK)
25    index(@Query() params: any) {
26      const response = this.teachersService.findAll();
27
28      return response;
```

El decorador **@Get** de la solicitud HTTP, crea un controlador para una acción en especifico el cual se va a conectar al service en la parte de **findAll** para que este consulte todos los datos que se encuentren registrados en la base de datos; cuando toda la consulta está bien generada el **@HttpCode** nos bota un código status **200 OK** diciendo que la acción esta correcta.

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

El decorador **@Delete** de la solicitud HTTP, crea un controlador para una acción en especifico el cual se va a conectar al service en la parte de **remove** para que este elimine los dalos de la base; esto se realiza mandando el **id** pero este tiene tubo llamado el **ParseIntPipe** el cual hace que el campo solicitado acepte solo números (**id);** cuando toda la consulta está bien generada el **@HttpCode** nos bota un código status **201** diciendo que la acción esta correcta; cuando está mal lanza un **404 Not Found**.

```
@Put(':id')
@HttpCode(HttpStatus.CREATED)
update(
  @Param('id', ParseIntPipe) id: number,
  @Body() payload: UpdateTeacherDto,
) {
  const response = this.teachersService.update(id, payload);
  return response;
```

El decorador **@Put** de la solicitud HTTP, crea un controlador para una acción en especifico el cual se va a conectar al service en la parte de **upDate** para que este coja el **id** y actualice los datos según le pidamos y seguidamente los guarde en la base de datos; cuando toda la acción está bien generada el **@HttpCode** nos bota un código status **201 Created** diciendo que la acción se realizo correctamente; cuando está mal lanza un **404 Not Found**.

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

El decorador **@Post** de la solicitud HTTP, crea un controlador para una acción en especifico el cual se va a conectar al service en la parte de **create** para que este coja el **id** y actualice los datos según le pidamos y seguidamente los guarde en la base de datos; cuando toda la acción está bien generada el **@HttpCode** nos bota un código status **201 Created** diciendo que la acción se realizo correctamente; cuando está mal lanza un **400 Bad Request**.

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

El decorador **@Get** de solicitud HTTP, crea un controlador para una acción en especifico el cual se va a conectar al service en la parte de **findOne** para que este coja el **id y** muestre la información que estaba en la base de datos según le pidamos; cuando toda la acción está bien generada el **@HttpCode** nos bota un código status **200 OK** diciendo que la acción se realizo correctamente; cuando está mal lanza un **404 Not Found**.

- **ENTITY**

```
import {
  PrimaryGeneratedColumn,
  Column,
  Entity,
  CreateDateColumn,
  UpdateDateColumn,
  DeleteDateColumn
} from 'typeorm';

@Entity('teachers')
export class TeacherEntity {

  @PrimaryGeneratedColumn()
  id: number;

  /* FK ...

  @Column('varchar', {
    name: 'academic_unit',
    length: 255,
    comment: 'Nombre de la unidad academica',
  })
  academicUnit: string;

  @Column('float', {
    name: 'administrative_hours',
    unsigned: true,
    comment: 'Horas dedicadas a la administracion al mes',
  })
  administrativeHours: number;
```

Las **entity o entidades** nos ayudan a obtener una relación con la base de datos, en este archivo es necesario tener un **"typeorm"** la cal es obligatoria para todos los archivos entity ya que este nos va ayudar a asociarnos con la base de datos, puesto que este nos ayuda a mapear directamente el objeto a registrar en la BD.

Dentro de este se definen las propiedades de un recurso con sus tipos de datos junto con un decorador el cual indicara que esa propiedad corresponde a una columna que se va a crear en la tabla; como en base de datos las tablas se mapean con una clave primaria

(PK)por ende se debe identificar el campo primario con un decorador **@PrimaryGeneratedColumn** el cual en la imagen es el **id** este decorador solo se puede usar una vez dentro de una archivo entity ya que en ninguna tabla puede llegar a existir más de una **PK**.

Los campos restantes se pueden usar el decorador **@Column(),** este de decorador nos permite definir que tipo de dato es poniéndolo dentro de sus paréntesis entre comillas simple si un valor en 'varchar' en el caso de este podemos definir la longitud del campo**({length:20})**, 'numeric', 'boolean', etc. Estos deben ser nombres que la BD pueda reconocer en su sistema dentro de estos paréntesis también podemos poner el**name:**en el cual le indicamos con que nombre queremos q se cree en la BD (name: 'total_publications',); también podemos poner un **comment** el cual nos permite da una descripción del campo creado su uso (comment: 'Total de las publicaciones realizadas sata el momento',). Estos son los dos mínimos que deben tener los campos de los entitys.

Existe decoradores o columnas especiales como:

- **@CreateDateColumn:**es una columna especial que se establece automáticamente en la fecha de inserción de la entidad. No necesita configurar esta columna, se configurará automáticamente.
- **@UpdateDateColumn:**es una columna especial que se establece automáticamente en la hora de actualización de la entidad cada vez que llama saveal administrador de la entidad o al repositorio. No necesita configurar esta columna, se configurará automáticamente.
- **@DeleteDateColumn:**es una columna especial que se establece automáticamente en el tiempo de eliminación de la entidad cada vez que llama a la eliminación temporal del administrador o repositorio de la entidad. No necesita configurar esta columna, se configurará automáticamente. Si se establece @DeleteDateColumn , el alcance predeterminado será "no eliminado" en esta columna se llena cuando hacemos una eliminación lógica con el softDelete.

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

@UpdateDateColumn({
  name: 'updated_at',
  type: 'timestamptz',
  default: () => 'CURRENT_TIMESTAMP',
})
updatedAt: Date;

@DeleteDateColumn({
  name: 'deleted_at',
  type: 'timestamptz',
  nullable: true,
})
deletedAt: Date;
```

- **DTO(Data Transfer Object)**

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


export class CreateTeacherDto {

  /* FK...

  @IsString({ message: 'Debe ser un texto' })
  @MaxLength(255, { message: 'Debe tener como Maximo 255 caracteres' })
  readonly academicUnit: string;//pendiente de revisar

  @IsNumber({}, { message: 'El campo tiene que ser número.' })
  @Min(0, { message: 'El número de caracteres mínimo es 0.' })
  readonly administrativeHours: number;

  @IsNumber({}, { message: 'El campo tiene que ser número.' })
  @Min(0, { message: 'El número de caracteres mínimo es 0.' })
  readonly classHours: number;

  @IsNumber({}, { message: 'El campo tiene que ser número.' })
  @Min(0, { message: 'El número de caracteres mínimo es 0.' })
```

Los DTO son objetos que se transfieren por la red de dos sistemas, estos son mas usados en aplicaciones **cliente/servidor**; este transfiere los objetos **(JSON)** desde el cliente al servidor o viceversa para realizar la operativa necesaria la que la aplicación no de errores. En otras palabras, los DTO son los datos que recibe la aplicación en formato JSON que vienen desde el cliente al backend (no es necesario especificar el id).

Este archivo se crea con un comando largo como: **nestgenerateclassteacher.dto** y su forma más corta como: **nest g clteacher.dto**, o su tercera forma con el comando general.

Estos DTO tienen sus propias validaciones como los entity en este caso vamos a ver 10 validaciones diferentes como:

- **@ValidateNested()**

Este decorador nos permite validar objetos animados, debemos tener en cuenta que debe ser una instancia de una clase de lo contrario no sabrá que clase es el objeto de la validación

```
import { ValidateNested } from 'class-validator';

export class Post {
  @ValidateNested()
  user: User;
}
```

- **@ValidatePromise()**

Este decorador funciona para cuando tenemos una propiedad con **promise,** para este tipo de propiedad funciona también un segundo decorador el cual es **@ValidateNested**

```
import { ValidateNested, ValidatePromise } from 'class-validator';

export class Post {
  @ValidateNested()
  @ValidatePromise()
  user: Promise<User>;
}
```

- **@ValidateIf**

Este es un decorador de validación condicional se puede usar para ignorar las validaciones de una propiedad cuando la función de la condición proporcionada devuelva falso.

```
import { ValidateIf, IsNotEmpty } from 'class-validator';

export class Post {
  otherProperty: string;

  @ValidateIf(o => o.otherProperty === 'value')
  @IsNotEmpty()
  example: string;
}
```

- **@Min()**

Este decorador nos permite definir cual será el mínimo de letras o números que puede tener una propiedad en caso de no cumplir con el mínimo se lanzara un mensaje

diciendo que debe tener mínimo de caracteres, ese mensaje se puede personalizar con un message.

```
@IsNumber()
@Min(0)
readonly administrativeHours: number;
```

- **@IsNumber()**

Este decorador nos permite definir una propiedad que solo sea tipo numérico y no permita ingresar letras en el caso de ingresar letras nos dará un mensaje de error diciendo que solo puede ser numérico, este mensaje se puede personalizar con message.

```
@IsNumber()
@Min(0)
readonly administrativeHours: number;
```

- **@MaxLength()**

Este decorador nos permite definir un máximo de letras que puede tener una entidad, en caso de sobrepasar ese máximo puesto nos dará un mensaje de error diciendo que sobrepasamos el máximo, este mensaje puede ser personalizado con un message.

```
@MaxLength(255)
readonly degreeHigherEducation: string;
```

- **@IsString()**

Este decorador nos permite definir que nuestra propiedad sean solo letras en caso de ingresar números nos lanzara un mensaje de error diciendo que la propiedad tiene que tener letras o debe ser string, este mensaje de error se puede personalizar con un message.

```
@IsString({ message: 'Debe ser un texto' })
@MaxLength(255)
readonly degreeHigherEducation: string;
```

- **@IsDateString**

Este decorador nos permite validar una propiedad con fecha y este nos permite mandar entre comillas dobles para q se guarde en la BD, en caso de que no sea de tipo fecha se lanzara un mensaje diciendo que el dato debe se de tipo date, este mensaje se puede personalizar con un message.

```
@IsDateString({ message: 'El campo es tipo fecha' })
readonly homeVacation: Date;
```

- **@IsOptional**

Comprueba si el valor dado está vacío (=== nulo, === indefinido) y, de ser así, ignora todos los validadores de la propiedad, a esta validación se le puede personalizar el mensaje con un message.

```
@IsOptional({ message: 'El campo es opcional.' })
@IsDateString({ message: 'El campo es tipo fecha' })
readonly homeVacation: Date;
```

- **@IsBoolean()**

Este decorador comprueba si el valor es booleano (false o true), de no ser asi nos lanza una mensaje de error diciendo que el valor tiene que ser boolean, este mensaje se puede personalizar con un message.

```
@IsBoolean()
readonly technology: string;
```