

## Documentación Dto, Entities, Modules, Services y Controllers

**Dto Data transfer object** es un objeto que contiene información que se transfiere entre procesos, como por ejemplo al enviar un objeto a cliente para que el servidor lo acepte y transfiera esa información

```
▼ dto
TS create-information-students.dto.ts
TS update-information-student.dto.ts
```

Class validator nos permite hacer validaciones mediante clases.

PartialType devuelve un tipo (clase) con todas las propiedades del tipo de entrada establecidas en opcional y minimiza el texto.

```
You, ayer | 1 author (You)
import { IsBoolean, IsNumber, IsString, MaxLength } from 'class-validator';
import { PartialType } from '@nestjs/mapped-types';
```

Las validaciones nos ayudan a comprobar que la información tenga el formato o tipo correcto.

```
You, ayer | 1 author (You)
4 export class CreateInformationStudentDto {
5   @IsString({
6     message: 'Se acepta solo string',
7   })
```

Exportamos la clase y estamos indicando con el decorador de nestjs @IsString que esta validación solo se acepta string y si no recibe ese tipo de dato recibirá un mensaje que indica que solo acepta datos tipo string o texto.

```
8   @MaxLength(255, {
9     message: 'Maximo 255 caracteres',
10  })
```

El decorador @MaxLength no indica que un parámetro donde solo se aceptan máximo 255 caracteres y envía el mensaje si no cumple ese parámetro

```
@IsNumber()
readonly codanisNumber: number;
```

Este decorador nos indica que solo acepta tipos de dato numéricos.

```
@IsEmail({ message: 'email debe ser un email' })
@IsOptional({ message: 'email es opcional' })
readonly email: string;
```

Estos decoradores nos hacen validar cuando el dato sea un mail y el decorador @IsOptional nos indica que este campo no es obligatorio llenarlo sino opcional y se leerá el mail en datos tipo texto.

```
@IsString({ message: 'El telefono debe ser texto' })  
@MinLength(5, { message: 'El telefono debe tener mínimo 5 caracteres' })  
@MaxLength(10, { message: 'El telefono debe tener máximo 10 caracteres' })  
readonly phone: string;
```

Como vimos anteriormente el decorador @IsString solo acepta datos tipo string, el decorador @MinLength nos indica que el número mínimo de caracteres a ingresar deben ser 5 y con el decorador @MaxLength se debe ingresar hasta máximo 10 caracteres

```
@IsString({ message: 'El website debe ser texto' })  
@IsUrl({ message: 'El website debe ser una url válida' })  
readonly web: string;
```

El decorador @IsString que nos dará el mensaje si no se cumple la validación diciendo que deber ser tipo string o texto, el decorador @IsUrl nos permite que valide un sitio web.

```
@IsNumber({}, {message: 'En el campo Carnet Codanis debe ingresar números'})  
@IsPositive({message: 'Se deben ingresar solo números positivos en el campo Carnet Codanis'})  
readonly codanisNumber: number;
```

El decorador @IsNumber valida solo a números y si no cumple el parámetro recibe el mensaje de que solo se deben ingresar números en ese campo, el decorador @IsPositive solo acepta números positivos y envía un mensaje de que se deben ingresar solo números positivos si no cumple con este parámetro y leerá el nombre del campo por ejemplo en este caso codanisNumber y el número que se ingresó en este campo.

## Entities

```
▼ entities  
  TS information-student.entity.ts
```

Las entidades de TypeORM son clases que debemos decorar con una anotación, en este caso @Entity, dentro de la clase, en el código de la entidad, se definen las propiedades de un recurso, con sus tipos de datos, a los que ahora vamos a añadir un nuevo decorador, que indicará que esa propiedad se corresponde con una columna que se va a crear en la tabla.

```
You, ayer | 1 author (You)
import { Column, CreateDateColumn, DeleteDateColumn, Entity, PrimaryColumn, PrimaryGeneratedColumn } from 'typeorm'
You, ayer | 1 author (You)
@Entity('information_student')
export class InformationStudentEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column('varchar', {
    name: 'ancestral_language',
    length: 255,
    comment: 'El idioma ancestral que el estudiante maneja',
  })
  ancestrallanguage: string;

  @Column('varchar', {
    name: 'company_name',
    length: 255,
    comment: 'El nombre de la compania donde el estudiante trabaja',
  })
  companyName: string;

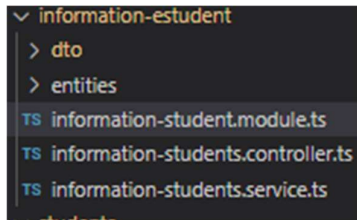
  @Column('varchar', {
    name: 'cell_phone',
    length: 10,
    comment: 'Numero de celular del estudiante',
  })
  cellPhone: string;

  @Column('integer', {
    name: 'codanis_number',
    comment: 'Numero que tiene el carnet del conais',
  })
  codanisNumber: number;
}
```

Ahora para definir columnas de una base de datos necesitamos decorar nuestras interfaces, por lo que ya no pueden ser interfaces sino clases.

Las entidades se mapean en tablas, las cuales deben tener una clave primaria. Por ello, el campo identificador lo hemos decorado con `@PrimaryGeneratedColumn` y el resto de campos se pueden definir simplemente con `@Column` donde el ORM podrá simplemente decidir el tipo de datos de columna que se creará en la tabla a partir del tipo que hemos indicado mediante TypeScript.

Como vemos en el ejemplo en la entidad `InformationStudent` exportamos la clase de la entidad y con el decorador `@PrimaryGeneratedColumn` ponemos la clave primaria de nuestra tabla en este caso el `id` que es un dato numérico y en los demás decoradores `@Column` los demás campos de la tabla en la que estamos trabajando con sus validaciones de igual manera como en los `dto`.



## Modulo

Cuando se necesita tener diferentes rutas para consultas en backend creamos un módulo en que contiene su propio controller, servicio y módulo que debe ser importado en el módulo principal.

Los módulos no son más que clases de programación orientada a objetos. Para conseguir que las clases se comporten como un módulo usamos el decorador `@Module()`, que nos permite asociar metadatos a una clase, en el módulo principal ya pudimos ver el decorador `@Module()` en su código. Básicamente a la hora de definirlo se declaran determinados elementos de tipo array con los componentes que manejaba ese módulo.

En un módulo podremos definir los siguientes elementos:

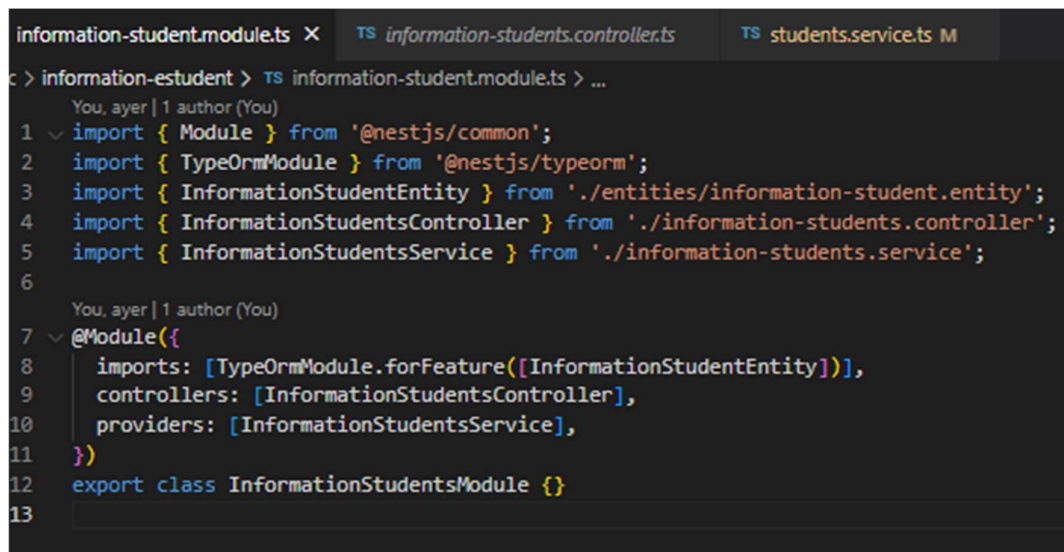
**Controladores:** Son las clases controlador que un módulo defina.

**Providers:** Son los servicios, factorías y otros elementos inyectables que son definidos por este módulo y que por tanto serán instanciados por el propio Nest para poder entregarlos allá donde se necesiten. Como dijimos, estos providers son instanciados una única vez y a lo largo del módulo se compartirá esa única instancia.

**Exports:** Es una lista de providers que podrán ser usados desde otros módulos. No todos los providers declarados en un módulo deben de ser conocidos fuera de él. Por supuesto, los providers seguirán siendo "singleton" aunque se usen desde otros módulos externos donde se puedan compartir.

**Imports:** es una lista de otros módulos cuyos providers podemos usar desde este módulo.

En nuestro módulo de `information-students` vamos a ir colocando todos los archivos que antes pertenecían al módulo principal, para hacer que el código este bien organizado.



```
information-student.module.ts X TS information-students.controller.ts TS students.service.ts M
c > information-estudent > TS information-student.module.ts > ...
  You, ayer | 1 author (You)
1  import { Module } from '@nestjs/common';
2  import { TypeOrmModule } from '@nestjs/typeorm';
3  import { InformationStudentEntity } from '../entities/information-student.entity';
4  import { InformationStudentsController } from '../information-students.controller';
5  import { InformationStudentsService } from '../information-students.service';
6
  You, ayer | 1 author (You)
7  @Module({
8    imports: [TypeOrmModule.forFeature([InformationStudentEntity])],
9    controllers: [InformationStudentsController],
10   providers: [InformationStudentsService],
11  })
12  export class InformationStudentsModule {}
13
```

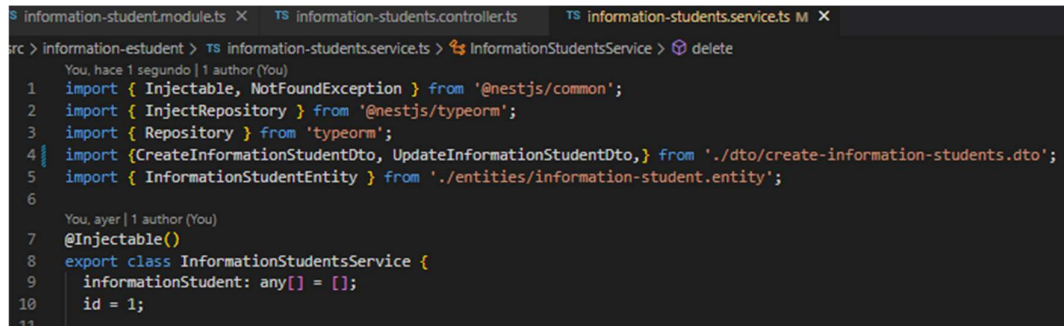
## Servicio

Un servicio tiene la responsabilidad de gestionar el trabajo con los datos de la aplicación, de modo que realiza las operaciones para obtener esos datos, modificarlos, etc.

Los servicios son clases, de programación orientada a objetos, como otros componentes de las aplicaciones. Son clases clasificadas como "provider", un tipo especial de artefactos dentro del framework. Por eso es interesante que expliquemos qué es un provider antes de ponernos a ver código de servicios.

Los providers son un concepto que se usa en el framework Nest, para englobar a un conjunto diverso de clases o artefactos, entre los que se encuentran los servicios, repositorios, factorías, etc.

Los providers están preparados para ser inyectados en los controladores de la aplicación, y otras clases si fuera necesario, a través del sistema de inyección de dependencias



```
1 import { Injectable, NotFoundException } from '@nestjs/common';
2 import { InjectRepository } from '@nestjs/typeorm';
3 import { Repository } from 'typeorm';
4 import { CreateInformationStudentDto, UpdateInformationStudentDto } from '../dto/create-information-students.dto';
5 import { InformationStudentEntity } from '../entities/information-student.entity';
6
7 @Injectable()
8 export class InformationStudentsService {
9   informationStudent: any[] = [];
10   id = 1;
11 }
```

Los decoradores son expresiones que devuelve una función y puede tomar un objetivo, nombre y descriptor de propiedad como argumentos. Los decoradores pueden definirse para una clase o una propiedad.

El decorador `@Injectable()`, nos permite llamar a la clase mediante inyección de dependencias.

Para hacer una inyección de dependencia los servicios se envían a los controladores, para conseguirlo en el controlador tenemos que importar el servicio que queremos usar e inyectarlo en el constructor de la clase, el constructor creará una propiedad dentro de la clase que tendrá el servicio inyectado. Hacemos el import de la clase `InformationStudentService`

```
constructor(  
  @InjectRepository(InformationStudentEntity)  
  private informationStudentRepository: Repository<InformationStudentEntity>,  
) {}  
  
async create(payload: CreateInformationStudentDto) {  
  const newInformationsStudent =  
    this.informationStudentRepository.create(payload);  
  
  return await this.informationStudentRepository.save(newInformationsStudent);  
}  
  
async delete(id: number) {  
  return await this.informationStudentRepository.softDelete(id);  
}  
  
async findAll() {  
  return await this.informationStudentRepository.find();  
}
```

You, ayer • actualizado

Creamos un constructor y agregamos el decorador `@InjectRepository` para inyectar al repositorio en este caso de `InformationStudentEntity`

Dentro del constructor realizamos la inyección en los parámetros

La palabra `private` permite que el objeto inyectado se asocie al controlador por medio de una propiedad privada.

Muy importante, para que funcione la inyección indicamos el nombre de la clase del objeto que estamos inyectando declarando el tipo del servicio que deseamos inyectar, (con la primera en mayúscula).

En el método `create` llamamos al dto por medio de un `payload` y en la constante `newInformationStudent` se crea el nuevo estudiante creado en el repositorio, y nos devolverá ese dato y se guardara en la base.

Método `delete (id: number)` nos permite borrar un elemento del array, es decir que se eliminara el elemento con el id recibido por parámetro.

El método `FindAll` nos permite encontrar todos los datos guardados en el repositorio de `informationStudentRepository`.



```
You, ayer * actualizado
async findOne(id: number) {
  const informationStudent = await this.informationStudentRepository.findOne({
    where: {
      id: id,
    },
  });

  if (informationStudent === null) {
    throw new NotFoundException(
      'La informacion del estudiante no se encontro'
    );
  }

  return informationStudent;
}

async update(id: number, payload: UpdateInformationStudentDto) {
  const informationStudent = await this.informationStudentRepository.findOne({
    where: {
      id: id,
    },
  });

  if (informationStudent === null) {
    throw new NotFoundException(
      'La informacion del estudiante no se encontro',
    );
  }

  this.informationStudentRepository.merge(informationStudent, payload);

  return this.informationStudentRepository.save(informationStudent);
}
```

Método `findOne(id: number)` nos permite encontrar un elemento del array, es decir que se mostrara el elemento con el id recibido por parámetro.

Método `update(id: number, payload: any)` este método permite actualizar un campo cuyo id se recibe por parámetro, con respecto a un contenido, que es el mismo payload enviado desde el controlador.

### Controlador

Son una de las piezas principales de las aplicaciones. Básicamente nos sirven para dar soporte o responder las solicitudes realizadas al servidor.

Lo primero en un controlador es el import del decorador que hace posible que podamos convertir una clase en un controlador. Lo importamos desde '@nestjs/common', así como también importamos los dto y el servicio.

```
TS information-student.module.ts TS information-students.controller.ts X TS information-students.service.ts
src > information-estudent > TS information-students.controller.ts > ...
  You, ayer | 1 author (You)
1  import {
2    Body,
3    Controller,
4    Delete,
5    Get,
6    HttpStatusCode,
7    HttpStatus,
8    Param,
9    ParseIntPipe,
10   Post,      You, ayer * actualizado
11   Put,
12   Query,
13 } from '@nestjs/common';
14 import { CreateInformationStudentDto } from '../dto/create-information-students.dto';
15 import { UpdateInformationStudentDto } from '../dto/create-information-students.dto';
16 import { InformationStudentsService } from '../information-students.service';
17
```

El decorador `@Controller`, es muy importante para que se le otorgue la condición de controlador a la clase que se está implementando en este caso la de `InformationStudentsController` dentro de esta un constructor con la palabra reservada `private` que permite que el objeto se asocie al controlador por medio de una propiedad privada.

Dentro de el controlador se realizan las peticiones al primera petición es `@Get` y le indicamos la ruta dentro de este decorador para que nos devuelva la respuesta por medio del servidor, en este caso un `findOne` que necesita del `id` para que nos devuelva el valor de ese `id` y en la siguiente petición de igual manera indicamos la ruta pero queremos que nos encuentre todos los datos guardados por medio del `findAll`.

En el decorador `@Post` de igual manera indicamos la ruta dentro del paréntesis del decorador Si enviamos `body` `params` debemos utilizar el decorador `Body` para poder recuperarlos de esta manera, reiniciamos el servidor y observamos que el método `post` ya se encuentra mapeado y nos mietra el elemento creado.



```
You, ayer | 1 author (You)
@Controller('information-student')
export class InformationStudentsController {
  constructor(private informationStudent: InformationStudentsService) {}

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

    return {
      data: response,
      message: `findAll`,
    };
  }

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

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

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

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

Decorador @Put se utiliza para gestionar rutas que son invocadas en el servidor con el método PUT aquí realizamos una actualización de un recurso dado, así que usamos parámetros en la URL y datos que nos llegarán en el cuerpo de la solicitud (body).

Decorador @Delete dentro del decorador indicamos la ruta y necesitamos el recurso (id) que se debe eliminar.