Tendencias de programación en la actualidad

ALEXANDER BRAVO

5to Desarrollo de Sotfware

Information Student

Tabla de contenido

Dto

CreateInformationStuden.dto.ts

Data Transfer Object("DTO")

Esto nos ayuda a validar solo los valores que van en el cuerpo de la solicitud. Estos deben respetar las validaciones si no se devuelve una respuesta con mensajes personalizables indicando que los valores de estos campos son incorrectos. Tenemos varias aserciones para diferentes tipos de valores incluidos;
las cuales en el siguiente ejemplo se pueden identificar tipos de validaciones que existen

```
@IsString({
  message: 'Se acepta solo string',
})
@IsEthereumAddress()
@IsAlphanumeric()
@IsOptional()
@Type()
@IsDefined()
@Max(100)
@MinLength(1)
@IsLocale()
@IsPositive()
@MaxLength(255, {
  message: 'Maximo 255 caracteres',
})
readonly ancestralLanguage: string;
```

@IsString() " Comprueba si un valor dado es una cadena real. **"**
@IsEthereumAddress() "Compruebe si la cadena es una dirección de Ethereum usando expresiones regulares básicas. No valida sumas de verificación de direcciones. Si el valor dado no es una cadena, entonces devuelve falso."
@IsAlphanumeric() "Comprueba si la cadena contiene solo letras y números. Si el valor dado no es una cadena, entonces devuelve falso."
@IsOptional() "Comprueba si falta el valor y, de ser así, ignora todos los validadores."
@Type() "Especifica un tipo de propiedad. La TypeFunction dada puede devolver un constructor. Se puede dar un discriminador en las opciones.Solo se puede aplicar a propiedades."
@IsDefined() "Comprueba si el valor está definido (!== indefinido, !== nulo)."
@Max(100)" Checks if the first number is less than or equal to the second. "
@MinLength(1) "Checks if the string's length is not less than given number. Note: this function takes into account surrogate pairs. If given value is not a string, then it returns false."
@IsLocale() "Compruebe si la cadena es una configuración regional. Si el valor dado no es una cadena, entonces devuelve falso."
@IsPositive() " Comprueba si el valor es un número positivo mayor que cero. "
@MaxLength () "Comprueba si la longitud de la cadena no es mayor que el número dado. Nota: esta función tiene en cuenta los pares sustitutos. Si el valor dado no es una cadena, entonces devuelve falso. "

como podemos ver los campos de validaciones son los que se explicaron

UpdateInformationStudent.dto.ts

UpdateInformationStudentDto

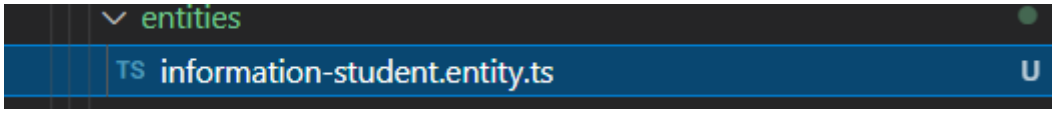
```
import { PartialType } from '@nestjs/swagger';
import { CreateInformationStudentDto } from './create-information-students.dto';

export class UpdateProductDto extends PartialType(CreateInformationStudentDto) {}
```

Al crear tipos de validación de entrada, también conocidos como **DTO**, suele ser útil crear y actualizar variantes del mismo tipo, en resumen, heredadas de la declaración anterior. **CreateInformationStudentDto** puede requerir todos los campos, mientras que **UpdateInformationStudentDto** puede hacer que todos los campos sean opcionales. Nest proporciona la función **PartialType()** a través de **@nestjs/swagger** para que sea fácil cometer el error de hacer lo mismo pero con campos opcionales. La función **PartialType()** devuelve un tipo de clase con todos los atributos de tipo de entrada

Entity

InformationStudent.entiti.ts

ENTITY



Estamos en ***InformationStudent.entity.ts*** donde se detalla la entidad, porque recuerden es una entidad en la base de datos, es una representación de la tabla en la base de datos, con esto claro la sintaxis de la entidad en "***NestJs***" es un clasificador InformationStudentEntity con propiedades como siempre a sido su declaración.
A continuación, se utiliza un decorador llamado @**Entity ()** para la clase. Por lo tanto, defina el nombre que usa la tabla en la base de datos como parámetro. Esto indica que la clase será una tabla en la base de datos.

```
@Entity('information_student')
```

Luego, en cada campo que comienza con **id;** usamos un decorador llamado @**PrimaryGeneratedColumn()** en el atributo id, lo que indica que este campo será la clave principal, PK, será un número de incremento automático como indica la sombra. Además, este decorador también puede tomar el nombre del campo como parámetro en caso de que queramos cambiar el nombre del campo al crear una tabla en la base de datos.

```
12  export class InformationStudentEntity {
13    @PrimaryGeneratedColumn()
14    id: number;
```

Finalmente, tenemos campos que definen campos comunes con el decorador @**Column()**, podemos pasar un montón de opciones después de pasar como primer parámetro el tipo de datos para obtener de la base de datos base y luego dentro de un objeto como segundo parámetro, si el nombre del campo es diferente, el tamaño de este campo, el valor predeterminado, si su valor puede ser nulo, si el valor Este valor de dato similar a **pk** es único, comentarios para este campo y muchas otras opciones.

```
@Column('varchar', {
  name: 'degree_obtained_superior',
  length: 255,
  comment: 'Titulo superior que obtuvo el estudiante',
})
```

Hay más decoradores para poder definir qué tipos de campos deben estar presentes en la base de datos, como los campos que siguen cuando este registro se **actualiza**, **crea**, **elimina** e incluso más decoradores.

```
  @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;
}
```

InformationStudent

informationStudent.controller.ts

Controller

```
18  @Controller('information-student')
19  export class InformationStudentsController {
20    constructor(private informationStudent: InformationStudentsService)
21
```

Un controlador es un controlador que expone puntos finales utilizados sobre direcciones HTTP. Encargado de responder a las solicitudes al servidor.
es básicamente una clase, añadiendo un decorador llamado **@Controller().**

```
18  @Controller('information-student')
```

Como primer parámetro, especifique el nombre del siguiente controlador:
Para usar el punto final del controlador, ingrese la URL **completa http: // localhost: 3000 /informationStudent**. Dentro del controlador, especifique algunos métodos que son los puntos finales para consumir y un constructor que inserta servicios que ayudan a establecer una conexión indirecta con la base de datos. El constructor tiene una inyección de dependencia privada de InformationStudentsService llamada por el método para usar el método de servicio.

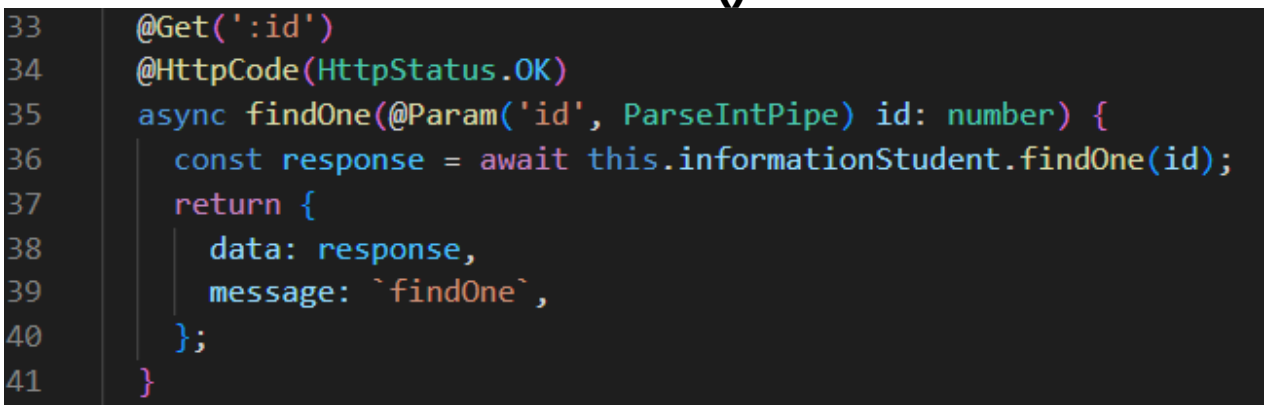
```
20    constructor(private informationStudent: InformationStudentsService)
```

.FindAll()

```
22    @Get('')
23    @HttpCode(HttpStatus.OK)
24    async findAll(@Query() params: any) {
25      const response = await this.informationStudent.findAll();
26
27      return {
28        data: response,
29        message: `findAll`,
30      };
31    }
```

Para el .findAll() usaremos el decorador **@Get()** para indicar que este punto final es un punto final que admite solicitudes **GET**, podemos insertar una cadena para definir mejor este punto final si queremos. También agregamos otro decorador **@HttpCode()** que es responsable de devolver el estado http correcto, 200 OK en este caso, ya que solo recibe datos. En los parámetros usamos el decorador **@Query()** que nos ayuda a obtener los parámetros de la url. Luego, en una constante, usamos el método **.findAll()** del servicio inyectado. Después de resolver este método **returnpromise with expectation, return** devuelve un objeto donde los datos serán una lista de todos los registros de esa tabla en la base de datos y un mensaje de todas las listas.

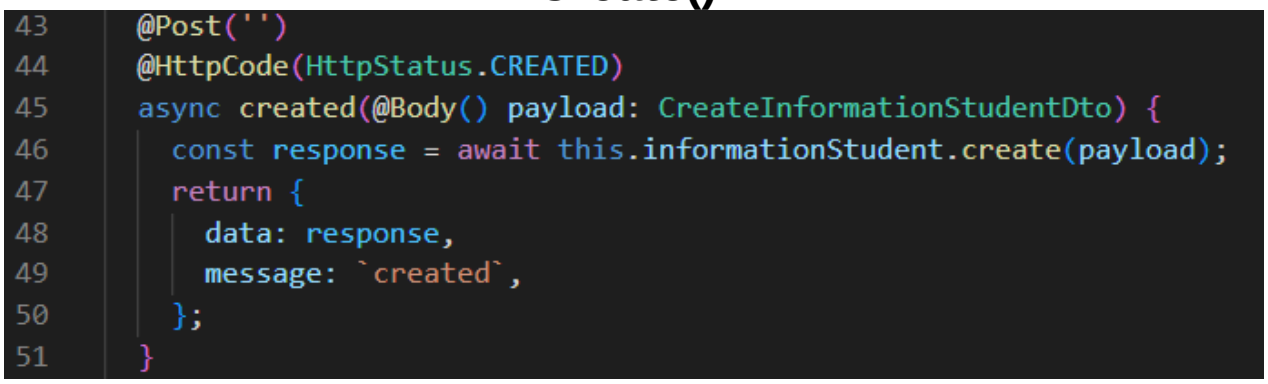
.findOne()



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

usamos el decorador @**Get()** para indicar que este punto final es el que admite la solicitud **GET**, podemos insertar una cadena para identificar mejor este punto final, luego con **id(":id' ")**, indicamos que registraremos lo que continúa de la URL después de insertar el nombre en el decorador @Controller(). Además, agregamos otro decorador @**HttpCode()** En los parámetros usamos el decorador @**Param()** que nos ayuda a obtener los parámetros de la URL, por lo que obtenemos la identificación en este parámetro. Luego, en una constante, usamos el método **.findOne()** del servicio inyectado, donde pasamos la identificación de inicialización en los parámetros como parámetros

Create().



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

El **.create()** usamos **el** decorador @**Post()** para indicar que este punto final está atendiendo una **solicitud POST**, podemos insertar una cadena para identificar mejor este punto final si queremos**.** Además, agregamos otro decorador @**HttpCode()** responsable de devolver el estado http correcto, 201 que indica que el objeto se creó con éxito**.** Los primeros parámetros usamos el decorador @**Body()**, este decorador se encarga del cuerpo de la solicitud, luego lo almacenamos en un parámetro llamado payload, el tipo de datos es un DTO que nos ayuda a validar los campos enviados en la solicitud. Luego**,** en una **constante,** guardamos el retorno usando el método **.create()** del servicio inyectado, donde pasamos **el payload** como parámetro**,** después de resolver la promesa con un tiempo de espera**,** devuelve la descripción de un objeto donde los datos son el objeto creado y notifica que el objeto ha sido creado.

Update().

```
53   @Put(':id')
54   @HttpCode(HttpStatus.CREATED)
55   async update(
56     @Param('id', ParseIntPipe) id: number,
57     @Body() payload: UpdateInformationStudentDto,
58   ) {
59     const response = await this.informationStudent.update(id, payload);
60     return {
61       data: response,
62       message: `updated ${id}`,
63     };
64   }
65
```

usamos el decorador **@Put()** para indicar que este punto final es el que admite la solicitud **PUT**, podemos insertar una cadena para identificar mejor este punto final si queremos, luego agregamos una identificación **(":id")** indicamos que registrará la continuación de la url después del nombre insertado en el decorador **@Controller()**. Además, agregamos otro decorador **@HttpCode()** responsable de devolver el estado **http correcto, 201** que indica que el objeto se actualizó correctamente. En los parámetros, primero usamos el decorador **@Param()** que nos ayuda a obtener los parámetros de la URL, por lo que obtenemos la identificación en este parámetro. Luego el decorador **@Body()**, este decorador se encarga del cuerpo de la solicitud, entonces lo almacenamos en un parámetro llamado payload, el tipo de dato es **DTO** el cual nos ayuda a validar los campos enviados en las solicitudes. Luego en una constante registramos un retorno cuando usamos el método **.update()** del servicio inyectado, donde pasamos el payload y el id como parámetros, luego de resolver la promesa con expect, devuelve una descripción que devuelve un objeto cuyos datos son el objeto a ser actualizado y notifica que el objeto ha sido actualizado.

Remove().

```
66   @Delete(':id')
67   @HttpCode(HttpStatus.CREATED)
68   async remove(@Param('id', ParseIntPipe) id: number) {
69     const response = await this.informationStudent.remove(id);
70     return {
71       data: response,
72       message: `deleted`,
73     };
74   }
75 }
```

En el punto final .remove() usamos el decorador **@Delete()** para indicar que este punto final es el que admite solicitudes **DELETE,** podemos insertar una cadena para definir mejor este punto final, luego con id similar **(": id")** indicar que registraremos la siguiente parte de la url después de insertar el nombre en el decorador **@Controller().** Además, agregamos otro decorador **@HttpCode()** responsable de devolver el estado http correcto**, 200 OK** en este caso para indicar

que la solicitud fue exitosa. En los parámetros usamos el decorador **@Param()** que nos ayuda a obtener los parámetros de la **URL**, por lo que obtenemos la identificación en este parámetro. Luego, en una constante, usamos el método **.remove()** del servicio inyectado, donde pasamos la identificación como parámetro. Después de resolver esta promesa de devolución del método con expectativa, el resultado de devolución devuelve un objeto donde los datos serán un registro para esa tabla en la base de datos y un mensaje de que el registro se ha eliminado.

informationStudent.service.ts

SERVICE

```
10  @Injectable()
11  export class InformationStudentsService {
12    informationStudent: any[] = [];
13    id = 1;
14
15    constructor(
16      @InjectRepository(InformationStudentEntity)
17      private informationStudentRepository: Repository<InformationStudentEntity>,
18    ) {}
```

El servicio **InformationStudentService** es responsable de conectarse e interactuar con la base de datos para almacenar nuestros registros. Cabe señalar que una instancia de **InformationStudentService** es una conexión a la base de datos, por lo que no será práctico instanciar **InformationStudentService** cada vez que lo usemos, por lo que debemos usar **@Injectable()** y, por lo tanto, usar una instancia generada globalmente para que para no saturar la aplicación con demasiadas conexiones inútiles. Su sintaxis es la misma que la de una clase en sí. Especifique un constructor que incluyamos una dependencia privada llamada **InformationStudentRepository**, esta acción actuará un decorador llamado **@InjectRepository()** donde tenemos que enviar la Entidad, es decir, **InformationStudentEntity** para saber a qué tabla se refiere este repositorio y así realizar operaciones porque el tipo de datos es para indicar que se trata de una tienda de nuestra entidad .

para indicar tenemos diferentes tipos de Services

Service.create()

```
20    async create(payload: CreateInformationStudentDto) {
21      const newInformationsStudent =
22        this.informationStudentRepository.create(payload);
23
24      return await this.informationStudentRepository.save(newInformationsStudent);
25    }
26
```

El **.create()** es un método que toma un parámetro de carga útil de tipo **CreateInformationStudentDto**, donde guardamos la creación de nuestro registro en una constante lista para ser guardada cuando se resuelva la promesa. A su vez, guardamos el registro

en la base de datos, esperamos a que se resuelva la promesa y devolvemos un objeto de tipo **InformationStudentEntity.**

Service.findAll()

```
31    async findAll() {
32      return await this.informationStudentRepository.find();
33    }
```

El **.findAll()** es un metodo responsable de devolver una lista completa de todos los registros de tipo **InformationStudentEntity**. A cambio, esperamos la resolución de la promesa con la expectativa de usar el método **.find().**

Service.findOne()

```
35  async findOne(id: number) {
36    const informationStudent = await this.informationStudentRepository.findOne({
37      where: {
38        id: id,
39      },
40    });
41
42    if (informationStudent === null) {
43      throw new NotFoundException(
44        'La informacion del estudiante no se encontro'
45      );
46    }
```

El método **.findOne()** se encarga de devolver un objeto de tipo **InformationStudentEntity** siempre y cuando el ID sea igual al enviado en el parámetro. De lo contrario, si no se encuentra el objeto con esa ID, una nueva NotFoundException ("error") detectará el error y podremos manejarlo. Luego, al resolver la promesa del método de repositorio, si el objeto no es nulo, puede devolver un objeto de tipo InformationStudentEntity.

Service.remove()

```
27   async remove(id: number) {
28     return await this.informationStudentRepository.softDelete(id);
29   }
30
```

El **.remove()** elimina un objeto de la base de datos según el ID recibido por el parámetro. Después de resolver una promesa con await, si encuentra un objeto para eliminar, devuelve una operación de eliminación en un objeto de tipo DeleteResult.

Service.update()

```
50
51   async update(id: number, payload: UpdateInformationStudentDto) {
52     const informationStudent = await this.informationStudentRepository.findOne({
53       where: {
54         id: id,
55       },
56     });
57
58     if (informationStudent === null) {
59       throw new NotFoundException(
60         'La informacion del estudiante no se encontro',
61       );
62     }
```

El **.update()** se encarga de devolver el objeto actualizado. Recibe el ID y el payload como parámetros. Estos son los nuevos datos con los que se actualizará el objeto. Después de resolver una promesa con await, si no se encuentra el objeto, devuelve un valor nulo y maneja el error lanzando una nueva excepción NotFoundException ("error"). Si el artículo llega en el momento de la devolución, se devolverá sin ningún problema.

informationStudent.module.ts

MODULE

En nestJs, la programación es modular, lo que significa que la aplicación divide el trabajo en partes para mejorar el orden y la comprensión del código.
Después de hacer los ajustes apropiados. Lo primero que debe hacer es crear una unidad, pero usaremos una directiva para crear una unidad en su forma básica completa.

Comando

"nest g res informacion-student --no-spec"

Nos posicionamos en el directorio de information-student creado, y localizamos el information-student.module.ts en el que se describe;

```
TS information-student.module.ts
```

1. El decorador **@Module()** Nos referimos por un objeto a los recursos que serán utilizados en nuestra API

```
@Module({
```

2. La propiedad **providers** Lo configuramos como el servicio porque proporcionará consultas indirectas a la base de datos, ya que depende del patrón del repositorio para ejecutar consultas en la base de datos.

```
    providers: [InformationStudentsService],
```

3. La propiedad **imports** establecemos una configuración de typeorm que le dice a las entidades en este módulo.

```
8       imports: [TypeOrmModule.forFeature([InformationStudentEntity])],
```

4. La propiedad **controllers**, le indicamos los controllers que están presentes en este módulo.

```
9       controllers: [InformationStudentsController],
```

5. para el ultimo punto se debe declarar la clase InformationStudentModule, que se vio afectada por el decorador anterior, convierte la clase InstitutionsModule en un dato nestJs

```
export class InformationStudentsModule {}
```