

“Instituto Tecnológico Superior Benito Juárez”

Nombre: Ronald Apolo

Curso: 5 Vespertino “B”

Fecha: 26/06/2022

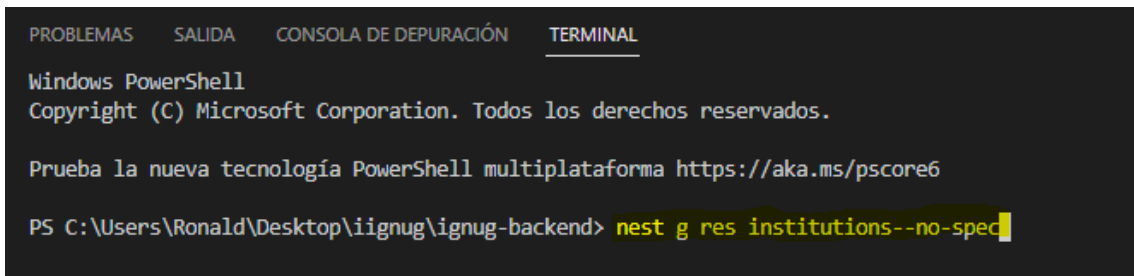
Module

Un módulo es una clase anotada con un `@Module()` decorador.

El `@Module()` decorador proporciona metadatos que Nest utiliza para organizar la estructura de la aplicación.

En Nest Js el modulo nos ayuda a dividir nuestro trabajo en diferentes partes y asi poder mejor el entendimiento con el orden del código

Para crear un nuevo modulo ingresamos en la consola del visual el siguiente comando:

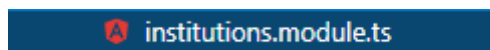


```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\Users\Ronald\Desktop\iignug\ignug-backend> nest g res institutions--no-spec
```

Al ingresar el comando se nos creara un archivo **modulo.ts**



Mediante un objeto indicamos al decorador `@Module()`, y asi indicaremos los recursos que utilizaremos en nuestra API.

Le asignamos a la configuración de typeorm la propiedad `imports`, y a continuación vamos a indicar las entidades que tenemos presentes en el modulo

Establecemos el servicio en la propiedad `providers` porque proporcionará consultas indirectas a la base de datos porque depende del modelo de repositorio para ejecutar consultas sql.

A continuación, en las propiedades del controlador, definimos los controladores que se incluyen en el modulo

Declaramos la clase `InstitutionsModule`, que se ve afectada por el decorador `@Module()`, y asi la convertimos en un modulo.

```

10 @Module({
11   imports: [TypeOrmModule.forFeature([InstitutionEntity, CatalogueEntity])],
12   providers: [InstitutionsService, CataloguesService],
13   controllers: [InstitutionsController, CataloguesController],
14 })
15 export class InstitutionsModule {}
16

```

Entity

El ORM TypeORM funciona con un concepto llamado **entidad**, o en inglés "Entity", 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.

Como otros elementos de NestJS, las entidades de TypeORM son clases que debemos decorar con una anotación, en este caso `@Entity`:

```

14 @Entity('institutions')
15 export class InstitutionEntity {
16   @PrimaryGeneratedColumn()
17   id: number;
18

```

Dentro de la clase, en el código de la entidad, se definen las propiedades de un recurso, con sus tipos de datos, añadimos un nuevo decorador, que indicará que esa propiedad se corresponde con una columna que se va a crear en la tabla.

```

14 @Entity('institutions')
15 export class InstitutionEntity {
16   @PrimaryGeneratedColumn()
17   id: number;
18
19   @OneToOne(() => CatalogueEntity)
20   @JoinColumn({ name: 'address_id' })
21   address: CatalogueEntity;
22
23   @ManyToOne(() => CatalogueEntity, { nullable: true })
24   @JoinColumn({ name: 'state_id' })
25   state: CatalogueEntity;

```

Las entidades se mapean en tablas, las cuales deben tener una clave primaria. Por ello, el campo identificador lo hemos decorado con `@PrimaryGeneratedColumn`.

El resto de campos se pueden definir simplemente con `@Column` y el propio 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.

```
14 @Entity('institutions')
15 export class InstitutionEntity {
16     @PrimaryGeneratedColumn()
17     id: number;
18
19     @OneToOne(() => CatalogueEntity)
20     @JoinColumn({ name: 'address_id' })
21     address: CatalogueEntity;
22
23     @ManyToOne(() => CatalogueEntity, { nullable: true })
24     @JoinColumn({ name: 'state_id' })
25     state: CatalogueEntity;
26
27     @Column('varchar', {
28         name: 'acronym',
29         length: 50,
30         default: 'none',
31         nullable: false,
32         unique: false,
33         comment: 'abreviatura del nombre del instituto',
34     })
35     acronym: string;
36
37     @Column('varchar', {
38         name: 'cellphone',
39         nullable: true,
40         length: 50,
41         comment: 'teléfono móvil directo de contacto con el instituto',
42     })
43     cellphone: string;
```

CreateInstitutionDto

Nos ayuda a validar solo los valores que se ingresan en la solicitud.

En esta parte se deben respetar estas afirmaciones, si la respuesta no se devuelve con mensajes personalizados que indiquen los valores de estos campos en ellos es incorrecto.

@MaxLength(100): Comprueba si la longitud de la cadena no es mayor que el número dado. Si el valor dado no es una cadena, entonces devuelve falso.

@Max(200): Comprueba si el primer número es menor o igual que el segundo.

@Min(0): Comprueba si el primer número es mayor o igual que el segundo.

@IsEmail(): Comprueba si la cadena es un correo electrónico. Si el valor dado no es una cadena, entonces devuelve falso.

@MinLength(5): Comprueba si la longitud de la cadena no es menor que el número dado. Si el valor dado no es una cadena, entonces devuelve falso.

@IsNumber(): Comprueba si el valor es un número.

@IsOptional(): Hace que este campo sea opcional, comprueba si falta el valor y, de ser así, ignora todos los validadores.

@IsString(): Comprueba si el valor dado es una cadena real.

@IsDateString(): Comprueba si es una fecha escrita como string.

@IsPositive(): Comprueba si el valor es un número positivo mayor que cero.

@IsUrl(): Comprueba si la cadena es una url. Si el valor dado no es una cadena, entonces devuelve falso.

@isArray(): Comprueba si un valor dado es una matriz

```
13 export class CreateInstitutionDto {
14     @IsNumber({}, { message: 'addressId debe ser un número' })
15     @IsPositive({ message: 'addressId debe ser un entero positivo' })
16     readonly addressId: number;
17
18     @IsNumber({}, { message: 'stateId debe ser un número' })
19     @IsPositive({ message: 'stateId debe ser un entero positivo' })
20     readonly stateId: number;
21
22     @IsString({ message: 'Acronym debe ser texto' })
23     @MinLength(2, { message: 'Acronym debe tener mínimo 2 caracteres' })
24     @MaxLength(50, { message: 'Acronym debe tener máximo 2 caracteres' })
25     readonly acronym: string;
26
27     @IsString({ message: 'Cellphone debe ser texto' })
28     @MinLength(5, { message: 'Cellphone debe tener mínimo 5 caractere' })
29     @MaxLength(20, { message: 'Cellphone debe tener máximo 20 caracteres' })
30     @IsOptional({ message: 'Cellphone es opcional' })
31     readonly cellphone: string;
32
33     @IsString({ message: 'Code debe ser texto' })
34     @MinLength(1, { message: 'Code debe tener mínimo 1 caracter' })
35     @MaxLength(50, { message: 'Code debe tener máximo 50 caracteres' })
36     readonly code: string;
37
38     @IsString({ message: 'codeSniese debe ser texto' })
39     @MinLength(1, { message: 'codeSniese debe tener mínimo 1 caracter' })
40     @MaxLength(50, { message: 'codeSniese debe tener máximo 50 caracteres' })
41     readonly codeSniese: string;
42 }
```

Service

Los servicios son una pieza esencial de las aplicaciones realizadas con el framework NestJS. Están pensados para proporcionar una capa de acceso a los datos que necesitan las aplicaciones para funcionar.

Para construir un servicio podemos usar el CLI de Nest. Para crear la clase de un servicio lanzamos el siguiente comando:


```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

Windows PowerShell
Copyright (C) Microsoft Corporation. Todos los derechos reservados.

Prueba la nueva tecnología PowerShell multiplataforma https://aka.ms/pscore6

PS C:\Users\Ronald\Desktop\iignug\ignug-backend> nest generate service InstitutionsService
```

Una vez construido nuestro servicio con el comando anterior podemos apreciar que se creó el siguiente archivo:

 institutions.service.ts

Ahora vamos a prestar atención al código base de un servicio, donde es especialmente importante el decorador `@Injectable`.

Este es el código base que nos aparece en el archivo **institutions.service.ts**:

```
9  @Injectable()
10 export class InstitutionsService {
11   constructor(
12     @InjectRepository(InstitutionEntity)
13     private institutionRepository: Repository<InstitutionEntity>,
14     private cataloguesService: CataloguesService,
15   ) {}
```

El servicio de momento está vacío de funcionalidad, pero gracias al decorador `@Injectable` estamos creando una clase que será capaz de inyectarse en los controladores.

Todo servicio debe tener ese decorador antes de la declaración de la clase que lo implementa, para poder usar la inyección de dependencias que Nest nos proporciona.

Service.create()

El Metodo -Create recibe como parámetro payload de tipo CreateInstitutionDto, registramos la creación de nuestro registro en una constante, este está listo para guardarse cuando se cumple la promesa, A su vez, guardamos el registro en la base de datos y esperamos hasta que se resuelva la promesa y devolvemos el objeto de tipo InstitutionEntity.

```
9  @Injectable()
10 export class InstitutionsService {
11     constructor(
12         @InjectRepository(InstitutionEntity)
13         private institutionRepository: Repository<InstitutionEntity>,
14         private cataloguesService: CataloguesService,
15     ) {}
16
17     async create(payload: CreateInstitutionDto): Promise<InstitutionEntity> {
18         const newInstitution = this.institutionRepository.create(payload);
19         newInstitution.address = await this.cataloguesService.findOne(
20             payload.addressId,
21         );
22         newInstitution.state = await this.cataloguesService.findOne(
23             payload.stateId,
24         );
25         return await this.institutionRepository.save(newInstitution);
26     }
27 }
```

Service.findAll()

El método .findAll() devuelve una lista de completa de todos los registros de tipo InstitutionEntity. Esperamos a que se resuelva la promesa en el return con el await de consumir el método .find().

```
async findAll(): Promise<InstitutionEntity[]> {
    return await this.institutionRepository.find();
}
```

Service.findOne()

El método .findOne() devuelve una objeto de tipo InstitutionEntity siempre y cuando el id sea igual al enviado en los parámetros, de lo contrario si no encuentra un objeto con ese id pues nos regresa null.

```

32     async findOne(id: number): Promise<InstitutionEntity> {
33         const institution = await this.institutionRepository.findOne({
34             where: { id },
35         });
36         if (institution === null)
37             throw new NotFoundException('not found institution');
38         return institution;
39     }
40

```

Service.update()

Es el responsable de devolver un objeto actualizado. Toma como parámetros, el id y el payload los cuales serán los nuevos datos en los cuales se les va a actualizar el objeto.

Si no se encuentra el objeto nos devolverá un null y para poder tratar ese error usamos el `throw new NotFoundException('error')`.

```

41     async update(
42         id: number,
43         payload: UpdateInstitutionDto,
44     ): Promise<InstitutionEntity> {
45         const institution = await this.institutionRepository.findOne({
46             where: { id },
47         });
48
49         if (institution === null)
50             throw new NotFoundException('not found institution');
51         await this.institutionRepository.merge(institution, payload);
52         return await this.institutionRepository.save(institution);
53     }
54
55     async remove(id: number): Promise<DeleteResult> {
56         return await this.institutionRepository.softDelete(id);
57     }
58 }
59

```

Service.remove()

Este método nos ayuda a eliminar un objeto de nuestra base de datos según los parámetros que reciba, si encuentra el objeto lo elimina entonces devuelve la eliminación en un objeto de tipo `DeleteResult`.

```

55     async remove(id: number): Promise<DeleteResult> {
56         return await this.institutionRepository.softDelete(id);
57     }
58 }

```

CONTROLLER

Los controladores son responsables de manejar las solicitudes entrantes y devolver las respuestas al cliente.

El propósito de un controlador es recibir solicitudes específicas para la aplicación. El mecanismo de enrutamiento controla qué controlador recibe qué solicitudes. Con frecuencia, cada controlador tiene más de una ruta y diferentes rutas pueden realizar diferentes acciones.

```
21 @Controller('institutions')
22 export class InstitutionsController {
23     constructor(private instituteService: InstitutionsService) {}
24 }
```

Controller.create()

Utilizamos el decorador `@Post()` para poder indicar que esta atendiendo una petición POST, Añadimos otro decorador `@HttpCode()` que devuelve el estado http correcto,

201 indicando que se creó el objeto correctamente. En los parámetros usamos primero al decorador `@Body()`, este decorador es el encargado de poseer el cuerpo de la petición, entonces aquello lo guardamos en un parámetro llamado `payload` el tipo de datos es un DTO el cual nos ayuda a validar los campos enviados en las peticiones.

```
21 @Controller('institutions')
22 export class InstitutionsController {
23     constructor(private instituteService: InstitutionsService) {}
24
25     @Post('')
26     @HttpCode(HttpStatus.CREATED)
27     async create(@Body() payload: CreateInstitutionDto): Promise<{
28         data: InstitutionEntity;
29         message: string;
30     }> {
31         const institution = await this.instituteService.create(payload);
32         return {
33             data: institution,
34             message: `created institution`,
35         };
36     }
37 }
```


Controller.findAll()

Utilizamos un decorador `@Get()` para indicar que ese endpoint esta atendiendo una petición GET, si queremos identificar estes endpoint insertamos un string. Añadimos otro decorador `@HttpCode()`, Se encarga de devolver el estado http correcto,

200 OK en este caso porque solo está obteniendo datos. En los parámetros usamos el decorador `@Query()` el que nos ayuda a obtener los parámetros de la url.

Luego en una constante consumimos el método `.findAll()` del servicio inyectado. Después de resolver la promesa que devuelve ese método con el `await` el `return` nos devuelve un objeto donde la data será una lista de todos los registros de esa tabla en la base de datos.

```
37
38  @Get('')
39  @HttpCode(HttpStatus.OK)
40  async findAll(@Query() params: any): Promise<{
41    data: InstitutionEntity[];
42    message: string;
43  }> {
44    const institutions = await this.instituteService.findAll();
45    return {
46      data: institutions,
47      message: `all institutions`,
48    };
49  }
```