ESTRUCTURA EN NESTJS

Alvaro Curipoma

MODULE

En **nestJs** la programación es modular, lo que quiere decir que la aplicación divide el trabajo en partes para mejorar el orden y entendimiento del código.
Luego de haber realizado las debidas configuraciones. Lo primero en hacer es crear un módulo, pero nos vamos a ayudar de un comando que crea un módulo en su totalidad básica.
nest g res institutions--no-spec
Nos ubicamos dentro del directorio students el cual se creó, ubicamos el **institutions.module.ts** en el se describe;

- En el decorador **@Module()** indicamos mediante un objeto, los recursos que se van a usar en nuestra api.
- A la propiedad **imports** le asignamos la configuración de **typeorm** indicando cuales son las entidades presentes en este módulo.
- A la propiedad **providers** le asignamos el servicio ya que el proveerá las consultas indirectamente a la base de datos ya que se apoya de un patrón de repositorio para hacer las consultas **sql**.
- Luego, a la propiedad **controllers**, le indicamos los **controllers** que están presentes en este módulo.
- Por ultimo y mas importante la declaración de la clase **InstitutionsModule**, a la cual influye el decorador antes mencionado, convirtiendo la clase **InstitutionsModule** en un módulo de **nestJs**.

```
@Module({
  imports: [TypeOrmModule.forFeature([InstitutionEntity])],
  providers: [InstitutionsService],
  controllers: [InstitutionsController],
})
export class InstitutionsModule {}
```

ENTITY

Nos ubicamos en el institution.entity.ts donde se detalla una entidad, pues recordando que es una entidad en base de datos es una representación de la tabla en la base de datos, con eso claro la sintaxis de una entity en NestJs es una clase InstitutionEntity con propiedades como siempre a sido su declaración.
Después, sobre la clase se usa un decorador llamado @Entity(), ahí definimos como parámetro el nombre que va a tomar la tabla en la base de datos. Con ello le indicamos que nuestra clase será una tabla en la base de datos.
Luego, en cada campo comenzando por el id; usamos sobre la propiedad id un decorador llamado @PrimaryGeneratedColumn() indicando que este campo será la llave primaria la pk (primary key), será un número auto incrementable como sombre indica. Además, a este decorador también se le puede pasar como parámetro el nombre del campo en caso que queramos cambiar de nombre el campo cuando se cree la tabla.
Por último, tenemos campos la definición de campos comunes con el decorador @Column() podemos pasarle un montón de opciones después de pasar como primer parámetro el tipo de dato que tomara en la base de datos, después, dentro de un objeto como segundo parámetro, ya sea otro nombre para el campo, el tamaño de ese campo, un valor por defecto, si su valor puede ser nulo, si es único ese valor ósea similar a la pk, un comentario para ese campo y muchas más opciones.
Existen más decoradores para poder determinar que tipo de campo será en la base de datos, como terminar campos para cuando se actualizó, creó, eliminó ese registro y existen aún más decoradores.

Alvaro Curipoma

```typescript
@Entity('institutions')
export class InstitutionEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column('varchar', {
    name: 'acronym',
    length: 50,
    default: 'none',
    nullable: false,
    unique: false,
    comment: 'abreviatura del nombre del instituto',
  })
  acronym: string;

  @CreateDateColumn({
    name: 'create_at',
    type: 'timestamptz',
  })
  createAt: Date;

  @UpdateDateColumn({
    name: 'update_at',
    type: 'timestamptz',
  })
  updateAt: Date;

  @DeleteDateColumn({
    name: 'delete_at',
    type: 'timestamptz',
  })
  deleteAt: Date;
}
```

CreateInstitutionDto

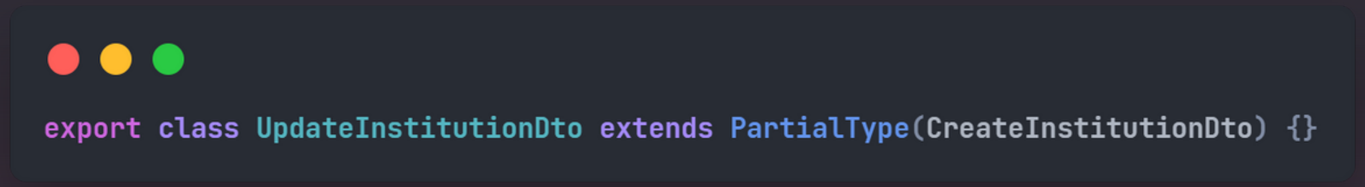
Nos ayuda a validar que los valores que ingresan por el cuerpo de la petición. Estos deben cumplir con las validaciones si no es devuelto una respuesta con mensajes que se puede personalizar diciendo que los valores de tales campos con incorrectos. Tenemos un sin número de validaciones para varios tipos de valores que ingresen;

@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.
@MaxLength(255): 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.
@MinLength(1): 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.
@IsPositive(): Comprueba si el valor es un número positivo mayor que cero.
@Max(1000): 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.
@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
@IsBase64(): Comprueba si una cadena está codificada en base64. Si el valor dado no es una cadena, entonces devuelve falso.
@IsBase32(): Compruebe si una cadena está codificada en base32. Si el valor dado no es una cadena, entonces devuelve falso.
@IsBoolean(): Comprueba si un valor es booleano.
@IsEnum({}): Comprueba si un valor dado es una enumeración
@IsByteLength(10): Comprueba si la longitud de la cadena (en bytes) se encuentra dentro de un rango. Si el valor dado no es una cadena, entonces devuelve falso.
@IsHexadecimal(): Comprueba si la cadena es un número hexadecimal. Si el valor dado no es una cadena, entonces devuelve falso.

```ts
export class CreateInstitutionDto
{ @IsOptional()
  @IsString()
  @IsDateString()
  @MaxLength(255)
  @MinLength(1)
  @IsNumber()
  @IsPositive()
  @Max(1000)
  @Min(0)
  @IsEmail()
  @IsUrl()
  @IsArray()
  @IsBase64()
  @IsBase32()
  @IsBoolean()
  @IsEnum({})
  @IsByteLength(10)
  @IsHexadecimal()
  readonly name: string;
}
```

UpdateInstitutionDto

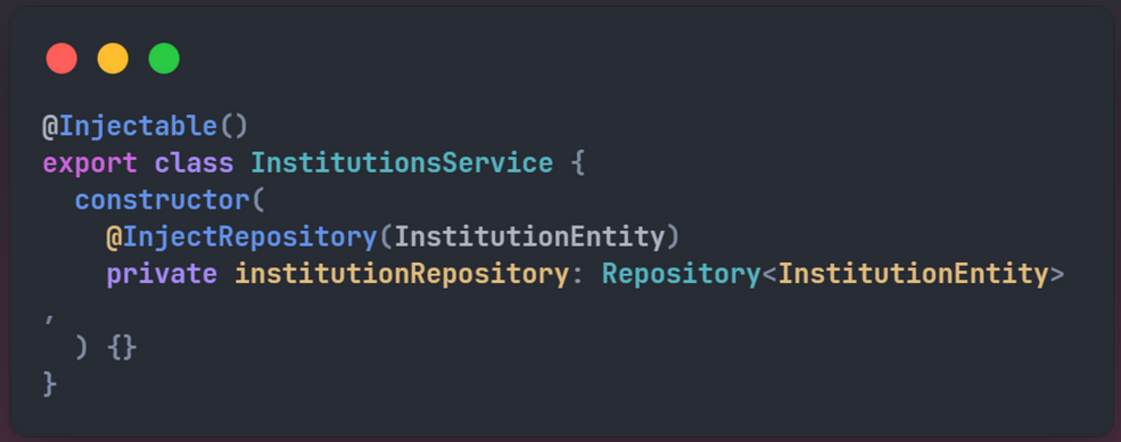
Al crear tipos de validaciones de entrada, también llamados DTO, suele ser útil crear y actualizar variaciones del mismo tipo, en pocas palabras heredar la antigua declaración. Por ejemplo, CreateInstitutionDto puede requerir todos los campos, mientras que UpdateInstitutionDto puede hacer que todos los campos sean opcionales.
Nest proporciona la función PartialType() mediante @nestjs/swagger para facilitar esta tarea y no cometer errores al hacer la misma tarea pero con campos opcionales.
La función PartialType()devuelve un tipo de clase con todas las propiedades del tipo de entrada establecidas en opcional.



Alvaro Curipoma

SERVICE

El servicio InstitutionsService es el encargado de conectarse con la base de datos e interactuar con ella para conseguir almacenar nuestros registros. Se debe tener en cuenta que una instancia de InstitutionsService es una conexión con la base de datos, entonces no sería conveniente instanciar cada vez que la usemos el InstitutionsService, entonces para eso es indispensable usar @Injectable() y así usar una instancia que se crea globalmente para no saturar la aplicación con demasiadas conexiones innecesarias. Su sintaxis es como el de una clase en sí. Denota un constructor donde realizamos inyección de una dependencia privada con el nombre de institutionRepository sobre este actúa un decorador llamado @InjectRepository() donde como parámetro debemos enviar nuestro Entity es decir, InstitutionEntity para saber a qué tabla hace referencia este repositorio y así realizar operaciones, como tipo de dato es indicar que es un repositorio del tipo de nuestra entidad.



```
@Injectable()
export class InstitutionsService {
  constructor(
    @InjectRepository(InstitutionEntity)
    private institutionRepository: Repository<InstitutionEntity>
,
  ) {}
}
```

Alvaro Curipoma

Service.create()

El método .create() es el que recibe como parámetro payload de tipo CreateInstitutionDto, en eso guardamos en una constante la creación de nuestro registro que se prepara para ser guardada cuando se resuelva la promesa. En el return guardamos el registro en la base de datos, esperamos a que se resuelva la promesa y devolvemos la creación del objeto de tipo InstitutionEntity.

```
@Injectable()
export class InstitutionsService {
  constructor(
    @InjectRepository(InstitutionEntity)
    private institutionRepository: Repository<InstitutionEntity>,
  ) {}

  async create(payload: CreateInstitutionDto): Promise<InstitutionEntity> {
    const newInstitution = await this.institutionRepository.create(payload);
    return await this.institutionRepository.save(newInstitution);
  }
}
```

Service.findAll()

El método .findAll() es el encargado de devolver una lista de completa de todos los registros de tipo InstitutionEntity. En el return esperamos a que se resuelva la promesa con el await de consumir el método .find().

```
@Injectable()
export class InstitutionsService {
  constructor(
    @InjectRepository(InstitutionEntity)
    private institutionRepository: Repository<InstitutionEntity>,
  ) {}

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

Service.findOne()

El método .findOne() es el encargado de devolver una objeto de tipo InstitutionEntity siempre y cuando el id sea igual al enviado en los parámetros, en caso contrario si no encuentra un objeto con ese id pues nos regresa null, ya que mediante throw new NotFoundException('error') atrapa el error y nosotros podemos tratarla. Si luego re resolver la promesa del método del repositorio consigue el objeto siendo diferente de nulo, entonces podemos retornar el objeto de tipo InstitutionEntity.

```
@Injectable()
export class InstitutionsService {
  constructor(
    @InjectRepository(InstitutionEntity)
    private institutionRepository: Repository<InstitutionEntity>,
  ) {}

  async findOne(id: number): Promise<InstitutionEntity> {
    const institution = await this.institutionRepository.findOne({
      where: { id },
    });
    if (institution === null)
      throw new NotFoundException('not found institution');
    return institution;
  }
```

Alvaro Curipoma

Service.update()

El método .update() es el encargado de devolver un objeto actualizado. Recibe como parámetro un id y un payload que son los nuevos datos a los que se va a actualizar el objeto. Luego de resolver la promesa con await, si no encuentra el objeto devuelve null y mediante throw new NotFoundException('error') tratamos ese error. Si encuentra el objeto llegado al return y devolvemos el objeto sin problemas.

```
@Injectable()
export class InstitutionsService {
  constructor(
    @InjectRepository(InstitutionEntity)
    private institutionRepository: Repository<InstitutionEntity>,
  ) {}

  async update(
    id: number,
    payload: UpdateInstitutionDto,
  ): Promise<InstitutionEntity> {
    const institution = await this.institutionRepository.findOne({
      where: { id },
    });

    if (institution === null)
      throw new NotFoundException('not found institution');
    await this.institutionRepository.merge(institution, payload);
    return await this.institutionRepository.save(institution);
  }
}
```

Alvaro Curipoma

Service.remove()

El método .remove() es el encargado de eliminar un objeto de la base de datos según el id que recibe en los parámetros. Luego de resolver la promesa con await, si encuentra el objeto lo elimina entonces devuelve la eliminación en un objeto de tipo DeleteResult.

```
@Injectable()
export class InstitutionsService {
  constructor(
    @InjectRepository(InstitutionEntity)
    private institutionRepository: Repository<InstitutionEntity>,
  ) {}

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

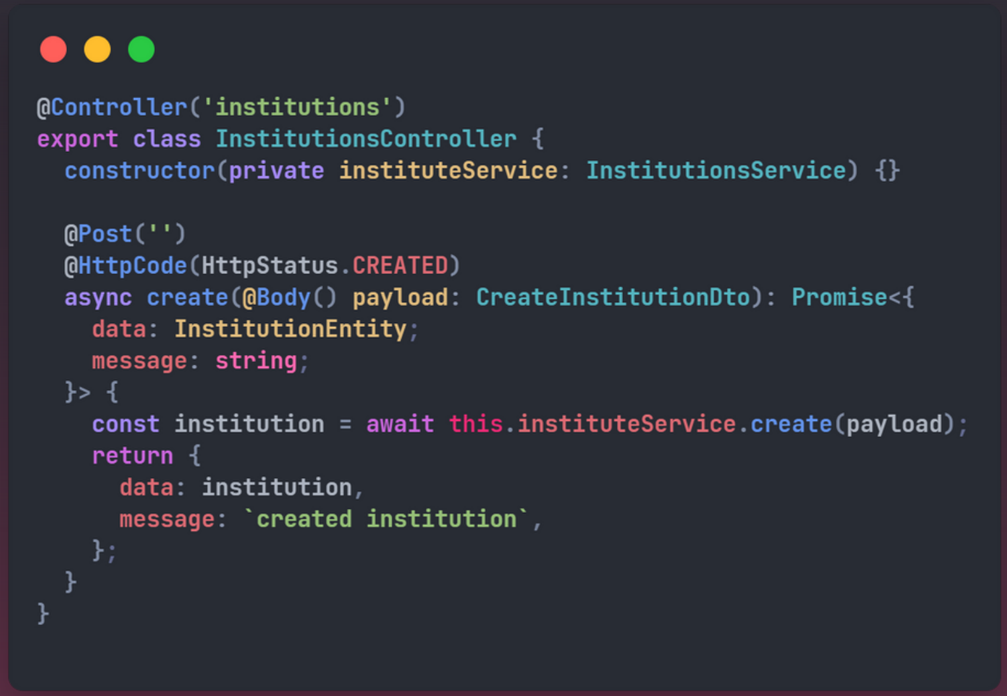
CONTROLLER

El controller es quien expone los endpoints para ser consumidos mediante direcciones http. Es el responsable de responder a las peticiones realizadas al servidor.
Su sintaxis es en lo básico una clase, añadimos a eso un decorador llamado @Controller() como primer parámetro le indicamos el nombre del controller que va ayudar a completar la url completa, http://localhost:3000/institutions , para poder consumir los endpoints del controller.
Dentro del controller (class) le indicamos algunos métodos que serán los endpoints a consumir, y un constructor donde inyectaremos un servicio, el cual nos ayudará a conectarnos indirectamente con la base de datos.
El constructor posee una inyección de dependencia privada de InstitutionsService, el cual será llamado en los métodos para consumir los métodos del servicio.

Alvaro Curipoma

Controller.create()

En el endpoint .create() usamos un decorador @Post() para indicar que ese endpoint es uno atendiendo una petición POST, podemos insertar un string para identificar este endpoint de mejor manera si quisiéramos. Además, añadimos otro decorador @HttpCode() es el encargado de devolver 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. Luego en una constante guardamos una la devolución al consumir el método .create() del servicio inyectado, donde le pasamos como parámetro el payload, luego de resolver la promesa con el await el return de manera descriptiva devuelve un objeto donde la data es el objeto creado, y un mensaje de que elobjeto fue creado.



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

  @Post('')
  @HttpCode(HttpStatus.CREATED)
  async create(@Body() payload: CreateInstitutionDto): Promise<{
    data: InstitutionEntity;
    message: string;
  }> {
    const institution = await this.instituteService.create(payload);
    return {
      data: institution,
      message: `created institution`,
    };
  }
}
```

Alvaro Curipoma

Controller.findAll()

En el endpoint .findAll() usamos un decorador @Get() para indicar que ese endpoint es uno atendiendo una petición GET, podemos insertar un string para identificar este endpoint de mejor manera si quisiéramos. Además, añadimos otro decorador @HttpCode() es el encargado 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 devuelve un objeto donde la data será una lista de todos los registros de esa tabla en la base de datos y un menaje de todos los listados.

```typescript
@Controller('institutions')
export class InstitutionsController {
  constructor(private instituteService: InstitutionsService) {}

  @Get('')
  @HttpCode(HttpStatus.OK)
  async findAll(@Query() params: any): Promise<{
    data: InstitutionEntity[];
    message: string;
  }> {
    const institutions = await this.instituteService.findAll();
    return {
      data: institutions,
      message: `all institutions`,
    };
  }
}
```

Controller.findOne()

En el endpoint .findOne() usamos un decorador @Get() para indicar que ese endpoint es uno atendiendo una petición GET, podemos insertar un string para identificar este endpoint de mejor manera, entonces con como id (“:id”) indicamos que vamos a guardar lo que continua de la url después del nombre insertado en el decorador @Controller(). Además, añadimos otro decorador @HttpCode() que es el encargado de devolver el estado http correcto, 200 OK en este caso porque solo está obteniendo datos. En los parámetros usamos el decorador @Param() el que nos ayuda a obtener los parámetros de la url, entonces estamos obteniendo el id en ese parámetro. Luego en una constante consumimos el método .findOne() del servicio inyectado, donde le pasamos como parámetro el id instanciado en los parámetros. Después de resolver la promesa que devuelve ese método con el await el return devuelve un objeto donde la data será un registro de esa tabla en la base de datos y un menaje.

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

  @Get(':id')
  @HttpCode(HttpStatus.OK)
  async findOne(@Param('id', ParseIntPipe) id: number): Promise<{
    data: InstitutionEntity;
    message: string;
  }> {
    const institution = await this.instituteService.findOne(id);
    return {
      data: institution,
      message: `show institution ${id}`,
    };
  }
}
```

Alvaro Curipoma

Controller.update()

En el endpoint .update() usamos un decorador @Put() para indicar que ese endpoint es uno atendiendo una petición PUT, podemos insertar un string para identificar este endpoint de mejor manera si quisiéramos, entonces le añadimos id (“:id”) indicamos que vamos a guardar lo que continua de la url después del nombre insertado en el decorador @Controller(). Además, añadimos otro decorador @HttpCode() es el encargado de devolver el estado http correcto, 201 indicando que se actualizó el objeto correctamente. En los parámetros usamos primero al decorador @Param() el que nos ayuda a obtener los parámetros de la url, entonces estamos obteniendo el id en ese parámetro. Luego el 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. Luego en una constante guardamos una la devolución al consumir el método .update() del servicio inyectado, donde le pasamos como parámetro el payload y el id, luego de resolver la promesa con el await el return de manera descriptiva devuelve un objeto donde la data es el objeto actualizado, y un mensaje de que el objeto fue actualizado.

```typescript
@Controller('institutions')
export class InstitutionsController {
  constructor(private instituteService: InstitutionsService) {}

  @Put(':id')
  @HttpCode(HttpStatus.CREATED)
  async update(
    @Param('id', ParseIntPipe) id: number,
    @Body() payload: UpdateInstitutionDto,
  ): Promise<{
    data: InstitutionEntity;
    message: string;
  }> {
    const institution = await this.instituteService.update(id, payload);
    return {
      data: institution,
      message: `updated institution ${id}`,
    };
  }
}
```

Alvaro Curipoma

Controller.remove()

En el endpoint .remove() usamos un decorador @Delete() para indicar que ese endpoint es uno atendiendo una petición DELETE, podemos insertar un string para identificar este endpoint de mejor manera, entonces con como id (“:id”) indicamos que vamos a guardar lo que continua de la url después del nombre insertado en el decorador @Controller(). Además, añadimos otro decorador @HttpCode() que es el encargado de devolver el estado http correcto, 200 OK en este caso para indicar que tuvo éxito la petición. En los parámetros usamos el decorador @Param() el que nos ayuda a obtener los parámetros de la url, entonces estamos obteniendo el id en ese parámetro. Luego en una constante consumimos el método .remove() del servicio inyectado, donde le pasamos como parámetro el id. Después de resolver la promesa que devuelve ese método con el await el return devuelve un objeto donde la data será un registro de esa tabla en la base de datos y un menaje indicando que fue eliminado ese registro.

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

  @Delete(':id')
  @HttpCode(HttpStatus.OK)
  async remove(@Param('id', ParseIntPipe) id: number): Promise<{
    data: DeleteResult;
    message: string;
  }> {
    const institution = await this.instituteService.remove(id);
    return {
      data: institution,
      message: `deleted institution ${id}`,
    };
  }
}
```

Alvaro Curipoma