Documentación de Nest(NestJS)

Nombre: Odalis Rea

Servicios

El servicio se encarga de conectarse con la base de datos, utiliza el patrón singleton para conectar a la base de datos. Me permite crear, actualizar, editar, eliminar. Para poder interactuar con la base de datos necesitamos inyectar la dependencia del repository.

Los interceptores

Un interceptor es una clase anotada con el @Injectable()decorador, que implementa la NestInterceptor interfaz. Los interceptores tienen un conjunto de capacidades útiles que están inspiradas en la técnica de Programación Orientada a Aspectos (AOP). Hacen posible:

- Vincular lógica adicional antes/después de la ejecución del método.
- Transformar el resultado devuelto por una función.
- Transformar la excepción lanzada desde una función.
- Ampliar el comportamiento de la función básica.
- Anular completamente una función dependiendo de condiciones específicas.Por ejemplo, para fines de almacenamiento en caché.

```
@Injectable()
```

Aquí tenemos a nuestro decorador **@InjectRepository,** el cuál utilizaremos para inyectar nuestra entidad **SubjectEntity.**
También vamos a inyectar nuestro servicio **CataloguesService.**

```
constructor(
    @InjectRepository(SubjectEntity)
    private subjectRepository: Repository<SubjectEntity>,
    private catalogueService: CataloguesService,
  ) { }
```

El método create(payload) sirve para crear el objeto, utilizando async el método create(payload)devuelve un Promise, y la función del método create(payload) puede await realizar tareas asincrónicas. En una variable llamada payload guardamos el Dto **CreateSubjectDto.** Con una constante **newSubject** guardamos la llamada que hacemos con el this a nuestro repository **subjectRepository** y con el método **create(payload)** encontramos el id de **academicPeriod,** haciendo la llamada que hacemos con el this a nuestro inyector de dependencias **catalogueService.** Definimos una constante llamada response en la cual guarda la llamada que hacemos con el this a

nuestro repository **subjectRepository** y con el método **create(payload)** vamos a crear la nueva asignatura **newSubject**. Con **return** lo retornamos la variable response. Con save() guardamos la asignatura en la base de datos.

```
async create(payload: CreateSubjectDto) {
    const newSubject = this.subjectRepository.create(payload);
    newSubject.academicPeriod = await this.catalogueService.findOne(
      payload.academicPeriodId
    );
    const response = await this.subjectRepository.save(newSubject);
    return await this.subjectRepository.save(response);
  }
```

Acabamos de usar la clase **CreateSubjectDto** como si fuera un tipo, por lo que no debemos olvidarnos de importarla.

```
import { CreateSubjectDto } from './dto/create-subject.dto';
```

El método **findAll()** sirve para que retorne todos los datos de la base de datos, utilizando async el método **findAll()** devuelve un Promise, y la función del método **findAll()** puede await realizar tareas asincrónicas. Con this llamamos a nuestro repository **subjectRepository** con **find()** encontramos todos los datos y con **return** lo retornamos.

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

El método **findOne(id)** sirve para que retorne los datos de un id de la base de datos, utilizando async el método **findOne(id)** devuelve un Promise, y la función del método **findOne(id)** puede await realizar tareas asincrónicas. Creamos una const **subject** y guardamos la llamada a nuestro repository **subjectRepository**, con el método **findOne(id)** encontramos el id en la base de datos y con **return** lo retornamos. Cuando nuestra constante **subject** sea nulo retorna la asignatura no se encontró.

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

    if (subject === null) {
      throw new NotFoundException('La asignatura no se encontro');
    }
```

```
    return subject;
  }
```

El método **update(id, payload)** sirve para actualizar los datos del id de la base de datos, utilizando async el método **update(id, payload)** devuelve un Promise, y la función del método **update(id, payload)** puede await realizar tareas asincrónicas. En nuestros parámetros el id lo definimos numérico y el payload guardamos el Dto **UpdateSubjectDto.** Creamos una const **subject** y guardamos la llamada a nuestro repository **subjectRepository,** con el método **findOne(id)** encontramos el id en la base de datos y con **return** lo retornamos. Cuando nuestra constante **subject** sea nulo retorna la asignatura no se encontró. Utilizamos la función **merge()** para fusionar o unir los dos conjuntos de datos de muestra, en este caso vamos unir **subject y payload** y lo guardamos en nuestro repository.
Utilizando **return** retornamos nuestro repository, con save() guardamos a nuestra constante **subject**.

```
  async update(id: number, payload: UpdateSubjectDto) {
    const subject = await this.subjectRepository.findOne({
      where: {
        id: id,
      },
    });

    if (subject === null) {
      throw new NotFoundException('La asignatura no se encontro');
    }

    this.subjectRepository.merge(subject, payload);
    return this.subjectRepository.save(subject);
  }
```

Acabamos de usar la clase **UpdateSubjectDto** como si fuera un tipo, por lo que no debemos olvidarnos de importarla.

El método **remove(id)** sirve para eliminar los datos utilizando el id de la base de datos, utilizando async el método **remove(id)** devuelve un Promise, y la función del método **remove(id)** puede await realizar tareas asincrónicas. Con this llamamos a nuestro repository **subjectRepository.** Utilizamos **softDelete()** para que muestre los registros borrados o los no borrados, con el id que está definido como number.

```
async remove(id: number) {
    return await this.subjectRepository.softDelete(id);
  }
```

Módulos

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. El módulo raíz es el punto de partida que utiliza Nest para crear la estructura de datos interna que utiliza Nest para resolver las relaciones y dependencias entre módulos y proveedores. El @Module()decorador toma un solo objeto cuyas propiedades describen el módulo:

```
@Module({
    imports: [TypeOrmModule.forFeature([SubjectEntity])],
  controllers: [SubjectsController],
  providers: [SubjectsService],
  exports: [TypeOrmModule, SubjectsService],
})
```

providers
Los proveedores que serán instanciados por el inyector y que pueden compartirse al menos en este módulo como el **SubjectsService**

controllers
El conjunto de controladores definidos en este módulo que deben ser instanciados como **SubjectsController**
imports
La lista de módulos importados que exportan los proveedores que se requieren en este módulo son **TypeOrmModule y la entidad SubjectEntity**
exports
El subconjunto de providers eso lo proporciona este módulo y debería estar disponible en otros módulos que importan este módulo como el **TypeOrmModule, SubjectsService**

Controladores

Los controladores son responsables de manejar las solicitudes entrantes y devolver las respuestas al cliente, es decir el controlador se encarga de publicar las rutas o recursos, recibir la petición del cliente y responder la petición del cliente.
El propósito de un controlador es recibir solicitudes específicas para la aplicación, utilizando el mecanismo de enrutamiento, controla qué controlador recibe qué solicitudes.

Para crear un controlador básico
-Usamos clases y decoradores.

Lo primero es el import del decorador que hace posible que podamos convertir una clase en un controlador. Lo importamos desde **'@nestjs/common'.**

```
import {
  Body,
  Controller,
  Delete,
  Get,
  HttpCode,
  HttpStatus,
  Param,
  ParseIntPipe,
  Post,
  Put,
  Query,
} from '@nestjs/common';
```

Los decoradores

Los decoradores asocian las clases con los metadatos requeridos, es decir que agregan funcionalidades a la clase.

Ejemplo

Usaremos el **@Controller()decorador.-** se requiere para definir un controlador básico,especificaremos un prefijo de ruta de ruta opcional de subjects.

```
@Controller('subjects')
```

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.

El export class sirve para exportar todo el contenido o lo que retorna la clase de **SubjectsController** del controlador.

```
export class SubjectsController {
```

Inyección de dependencia con un proveedor

La idea principal de un proveedor es que se puede inyectar como una dependencia; esto significa que los objetos pueden crear varias relaciones entre sí, y la función de "conectar" instancias de objetos puede delegarse en gran medida al sistema de tiempo de ejecución de Nest.

En el siguiente ejemplo, Nest resolverá el problema **subjectsService** y devolviendo una instancia de **SubjectsService** o, en el caso normal de un singleton, devolviendo la instancia existente si ya se

solicitó en otro lugar. Esta dependencia se resuelve y pasa al constructor de su controlador o se asigna a la propiedad indicada:

```
constructor(private subjectsService: SubjectsService) { }
```

El @Get()es un decorador de la petición get() de los métodos de solicitud HTTP. El decorador **@Get()** está asociado al método **findAll()** para que retorne todos los datos del **SubjectsService.** Utilizamos el decorador **@Query()**, este decorador sirve para enviar parámetros en la url con datos clave y valor. En la variable params le ponemos any para que no tenga ningún valor por defecto. El decorador **@HttpCode** sirve para definir qué tipo de status o código HTTP me devuelve. En este caso escogimos el status 200 con el OK. Definimos una constante llamada response en la cual guarda la llamada que hacemos con el this a nuestro inyector **subjectsService** y con el método **findAll()** vamos a retornar todos los datos. Con **return** lo retornamos la variable response.

No te olvides de importar la declaración del decorador @Get(),@Query(),@HttpCode()

```
  @Get('')
 @HttpCode(HttpStatus.OK)
findAll(@Query() params: any) {
    const response = this.subjectsService.findAll();
    return response;
  }
```

Utilizamos el decorador **@Get(':id')** incluimos ‘:id’ dentro del decorador para definir la ruta, con los : lo ponemos para que vaya algo en la ruta. El decorador **@HttpCode** sirve para definir qué tipo de status o código HTTP me devuelve. En este caso escogimos el status 200 con el OK. Con el decorador **@Param()** sirve para recuperar el id. **ParseIntPipe** sirve para transformar un dato a su valor integer. Declaramos el tipo de dato con la variable id: number. Definimos una constante llamada response en la cual guarda la llamada que hacemos con el this a nuestro inyector **subjectsService** y con el método **findOne()** vamos a retornar un id con su data. Con **return** lo retornamos la variable response.

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

El @Post()es un decorador de la petición post() de los métodos de solicitud HTTP. No te olvides de importar la declaración del decorador @Post().
También queremos proporcionar un **método de solicitud HTTP** que cree nuevos registros. Para esto, vamos a utilizar el decorador **@Post().** En el controlador vamos a usar el DTO para que, cuando nos piden hacer una inserción, podamos decirle que el dato que se recibirá en el body de la request, será del tipo del DTO que acabamos de crear.El decorador **@HttpCode** sirve para definir qué tipo de status o código HTTP me devuelve. En este caso escogimos el status 201 con el **CREATED**. En el decorador **@Body()** viaja la data en la cabecera porque para enviar un objeto necesito el **@Body().** Utilizamos la variable payload donde guarda el dto **CreateSubjectDto,** en la **const response** guardamos la llamada con this a nuestro inyector del servicio llamado **subjectsService** y la llamada a nuestro método **create(payload),** vamos a crear una nueva asignatura. Con **return** lo retornamos la variable response.
Entonces, nuestro método @Post, tendrá el siguiente código.

```
  @Post('')
  @HttpCode(HttpStatus.CREATED)
 create(@Body() payload: CreateSubjectDto) {
    const response = this.subjectsService.create(payload);
    return response;
  }
```

Acabamos de usar la clase **CreateSubjectDto** como si fuera un tipo, por lo que no debemos olvidarnos de importarla.

```
import { CreateSubjectDto } from './dto/create-subject.dto';
```

El @Put()es un decorador de la petición put() de los métodos de solicitud HTTP.
Utilizamos el decorador **@Put(':id')** incluimos ‘:id’ dentro del decorador para definir la ruta, con los : lo ponemos para que vaya algo en la ruta. Con el decorador **@Param()** sirve para recuperar el id. **ParseIntPipe** sirve para transformar un dato a su valor integer. Declaramos el tipo de dato con la variable id: number. El decorador **@HttpCode** sirve para definir qué tipo de status o código HTTP me devuelve. En este caso escogimos el status 201 con el **CREATED**.
En el decorador **@Body()** viaja la data en la cabecera porque para enviar un objeto necesito el **@Body().**
Utilizamos la variable payload donde guarda el dto **UpdateSubjectDto** y luego le llamamos con this a nuestro inyector **subjectsService,** a nuestro método **update(id, payload)** y lo guardamos en la **const response,** vamos a actualizar la asignatura. Con **return** lo retornamos la variable response.

```
@Put(':id')
```

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

El @Delete()es un decorador de la petición delete() de los métodos de solicitud HTTP. Utilizamos el decorador **@Delete**(':id') incluimos ‘:id’ dentro del decorador para definir la ruta, con los : lo ponemos para que vaya algo en la ruta. El decorador **@HttpCode** sirve para definir qué tipo de status o código HTTP me devuelve. En este caso escogimos el status 201 con el **CREATED**. Con el decorador **@Param()** sirve para recuperar el id. **ParseIntPipe** sirve para transformar un dato a su valor integer. Declaramos el tipo de dato con la variable id: number. Definimos una constante llamada **response** en la cual guarda la llamada que hacemos con el this a nuestro inyector subjectsService y con el método **remove**(id) vamos a borrar un solo id. Con return lo retornamos la variable response.

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

Dtos

El Dto sirve para validar la data del cuerpo de la petición es decir para validar lo que el cliente me envíe.
DTO son las siglas de Data Transfer Object y no es más que un objeto que se transfiere por la red entre dos sistemas, típicamente usados en aplicaciones cliente/servidor y en las aplicaciones web modernas.
Un DTO no es más que una clase o interfaz que define los datos que debemos recibir para trabajar con una entidad.
En Nest, con la intención de crear aplicaciones robustas, es útil tipar los datos que se van a enviar y recibir desde el frontend al backend, especificando qué propiedades tendrán los objetos DTO y de qué tipos.
Lo primero que haremos es instalar unos paquetes necesarios para que el componente ValidationPipe nos facilite las validaciones en el framework. Dichos paquetes los instalamos de la siguiente manera:

```
PS C:\Users\bebecita\ignug-backend> yarn add class-validator class-transformer
```

Ahora vamos a especificar los tipos de datos de nuestro DTO.
Los tipos de datos son: **number, string, etc.**
Para definir una propiedad utilizamos **readonly**
Utilizamos decoradores como: **@IsNumber(), @IsPositive(), @IsOptional(), @IsString(), @Min(), @MinLength(), @MaxLength(), @Max(),** para validar el tipo de dato de nuestra propiedad.
Utilizamos el atributo **message** en cada decorador para especificar el error que le va a devolver al cliente.
@IsNumber(),para validar que el campo sea de tipo número.
@IsPositive(), para validar que el campo sea de tipo número positivo.
@IsOptional(), para validar que el campo sea opcional.
@IsString(), para validar que el campo sea de tipo string.
@Min(), para validar que el campo tenga un número mínimo.
@MinLength(), para validar que el campo tenga una longitud mínima de caracteres.
@MaxLength(), para validar que el campo tenga una longitud máxima de caracteres.
@Max(), para validar que el campo tenga un número máximo.
@IsNotEmpty(), booleano . Devuelve verdadero si la colección no está vacía .

```
export class CreateSubjectDto {
    @IsNumber({}, { message: 'El campo academicPeriodId debe ser un número' })
    @IsPositive({ message: 'El campo academicPeriodId debe ser un entero positivo' })
    readonly academicPeriodId: number;

    @IsNumber({}, { message: 'El campo curriculumId debe ser un número' })
    @IsPositive({ message: 'El campo curriculumId debe ser un entero positivo' })
    readonly curriculumId: number;

    @IsNumber({}, { message: 'El campo stateId debe ser un número' })
    @IsPositive({ message: 'El campo stateId debe ser un entero positivo' })
    readonly stateId: number;

    @IsNumber({}, { message: 'El campo typeId debe ser un número' })
    @IsPositive({ message: 'El campo typeId debe ser un entero positivo' })
    readonly typeId: number;
```

```
    @IsNumber({}, { message: 'Debe ser un número' })
    @Min(0, { message: 'El número mínimo es 0' })
    readonly autonomousHour: number;

    @IsNumber({}, { message: 'Debe ser un número' })
    @Min(0, { message: 'El número mínimo es 0' })
    @IsOptional()
    readonly credit: number;

    @IsString({ message: 'Debe ser un string' })
    @IsNotEmpty({ message: 'Devuelve verdadero si el campo no está
vacío' })
    @MinLength(4, { message: 'El número de caracteres mínimo es 4' })
    @MaxLength(255, { message: 'Maximo 255 caracteres' })
    readonly name: string;

    @IsNumber({}, { message: 'Debe ser un número' })
    @Min(0, { message: 'El número mínimo es 0' })
    readonly practicalHour: number;

    @IsNumber({}, { message: 'Debe ser un número' })
    @Min(0, { message: 'El número mínimo es 0' })
    @Max(1, { message: 'El número máximo es 1' })
    readonly scale: number;

    @IsNumber({}, { message: 'Debe ser un número' })

    @Min(0, { message: 'El número mínimo es 0' })
    readonly teacherHour: number;
}
```

- Como ves, se trata de una clase normal, en la que solamente estamos especificando sus propiedades y los tipos.
- En el DTO de creación de un elemento no es necesario especificar el id, puesto que se generará en el momento de crearlo.

Entidades

La entidad sirve para conectar backend con la base de datos.
Podemos generar las llaves primarias con el decorador **`@PrimaryGeneratedColumn()`**
Utilizamos tipos de datos como number, date, string, etc.

```
  @PrimaryGeneratedColumn()
```

```
  id: number;
```

Podemos generar la fecha de creación con el decorador `@CreateDateColumn()`
Utilizamos variables para definir el campo, el name sirve para definir el nombre en el que iría el campo en la base de datos y type define el tipo de dato, utilizamos una función con default para saber la fecha actual.

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

Podemos generar la fecha de actualización con el decorador `@UpdateDateColumn()`
Utilizamos variables para definir el campo, el name sirve para definir el nombre en el que iría el campo en la base de datos y type define el tipo de dato, utilizamos una función con default para saber la fecha actual.

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

Podemos generar la fecha de eliminación con el decorador `@DeleteDateColumn()`
Utilizamos variables para definir el campo, el name sirve para definir el nombre en el que iría el campo en la base de datos y type define el tipo de dato, utilizamos nullable para definir si es que el campo puede ser nulo, con true el campo es nulo, con false el campo no es nulo.

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

Podemos generar un campo con el decorador `@Column()` dentro de este decorador vamos a definir el tipo de dato que en este caso es un entero `'int'`.
Utilizamos tipos de datos para definir cada campo, el name sirve para definir el nombre en el que iría el campo en la base de datos, default define el tipo de dato que viene por defecto y comment sirve para poner un comentario de acuerdo al campo.
Al final escribimos el campo en forma de Case Sensitive y definimos el tipo de dato en este caso es number.

```
@Column('int', {
    name: 'autonomous_hour',
    default: 0,
    comment: 'Hora autónoma de la asignatura',
  })
  autonomousHour: number;
```

Podemos generar un campo con el decorador `@Column()`dentro de este decorador vamos a definir el tipo de dato que en este caso es una cadena `'varchar'`.
Utilizamos tipos de datos para definir cada campo, el name sirve para definir el nombre en el que iría el campo en la base de datos, length define la longitud de los caracteres que se van a ingresar en el campo y comment sirve para poner un comentario de acuerdo al campo.
Al final escribimos el campo en forma de Case Sensitive y definimos el tipo de dato en este caso es string.

```
 @Column('varchar', {
    name: 'code',
    length: 50,
    comment: 'Código de la asignatura',
  })
  code: string;
```

Podemos generar un campo con el decorador `@Column()`dentro de este decorador vamos a definir el tipo de dato que en este caso es un flotante `'float'`.
Utilizamos tipos de datos para definir cada campo, el name sirve para definir el nombre en el que iría el campo en la base de datos, utilizamos nullable para definir si es que el campo puede ser nulo, con true el campo es nulo, con false el campo no es nulo, default define el tipo de dato que viene por defecto y comment sirve para poner un comentario de acuerdo al campo.
Al final escribimos el campo en forma de Case Sensitive y definimos el tipo de dato en este caso es number.

```
@Column('float', {
```

```
    name: 'credit',
    nullable: true,
    default: 0,
    comment: 'Creditos de la asignatura',
  })
  credit: number;
```