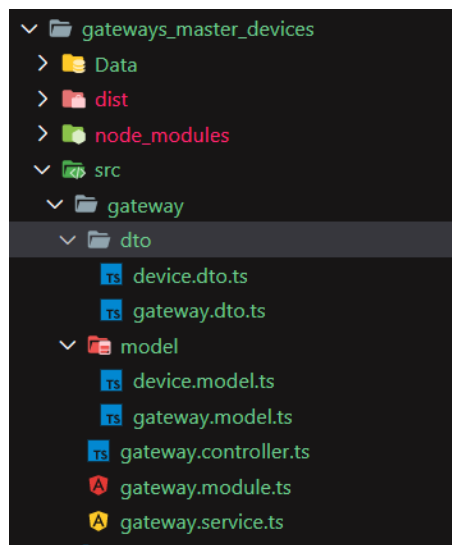


## DOCUMENTACIÓN DE REFERENCIA

El documento que se expone a continuación tiene como finalidad brindar una explicación general del mini proyecto de prueba realizado, así como una guía para la realización de pruebas.

El mini proyecto fue implementado en NestJS y está compuesto por un módulo “gateway”, el cual lo conforma una clase controladora donde estarán los endpoints, una clase servicio para la realización de las operaciones, una carpeta que contendrá los modelos al igual que otra para los DTOs.



*Ilustración 1: Estructura del módulo*

Los modelos que se manejan son Gateway y Device, en los ficheros gateway.model.ts y device.model.ts respectivamente. Estos representan las abstracciones de las estructuras de datos que se gestionarán en la base de datos, en este caso MongoDB.

La clase Gateway posee los atributos: serialNumber, name, ipv4Address y peripheralDevices (array de devices). En cuanto a la clase Device posee los atributos: uid que será de valor único (caso de intentar insertar un device con valor existente genere excepción), vendor, createDate y status (enum).

```

gateway.model.ts U X
gateways_master_devices > src > gateway > model > gateway.model.ts > Gateway
1  import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
2  import { Document, Types } from 'mongoose';
3  import { Device, DeviceSchema } from './device.model';
4
5  export type GatewayDocument = Gateway & Document;
6
7  @Schema()
8  export class Gateway {
9    @Prop({ required: true })
10    serialNumber: string;
11
12    @Prop({ required: true })
13    name: string;
14
15    @Prop({ required: true })
16    ipv4Address: string;
17
18    @Prop({ type: [{ type: Types.ObjectId, ref: 'Device' }] })
19    peripheralDevices: Types.ObjectId[];
20  }
21
22  export const GatewaySchema = SchemaFactory.createForClass(Gateway);
23

```

*Ilustración 2: Modelo Gateway*

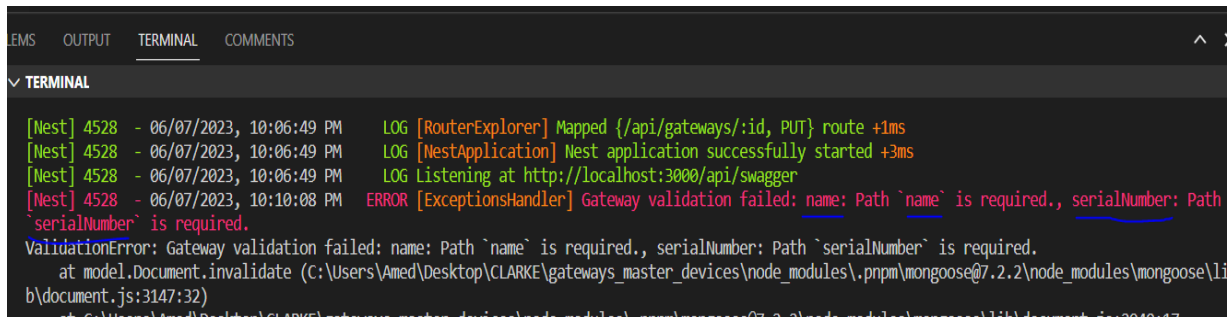
```

device.model.ts U X
gateways_master_devices > src > gateway > model > device.model.ts > Device > createdDate
1  import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
2  import { Document } from 'mongoose';
3
4  export type DeviceDocument = Device & Document;
5
6  @Schema()
7  export class Device {
8    @Prop({ required: true, unique: true })
9    uid: number;
10
11    @Prop({ required: true })
12    vendor: string;
13
14    @Prop({ required: true })
15    createdAt: Date;
16
17    @Prop({ enum: ['online', 'offline'], default: 'offline' })
18    status: string;
19  }
20
21  export const DeviceSchema = SchemaFactory.createForClass(Device);

```

*Ilustración 3: Modelo Device*

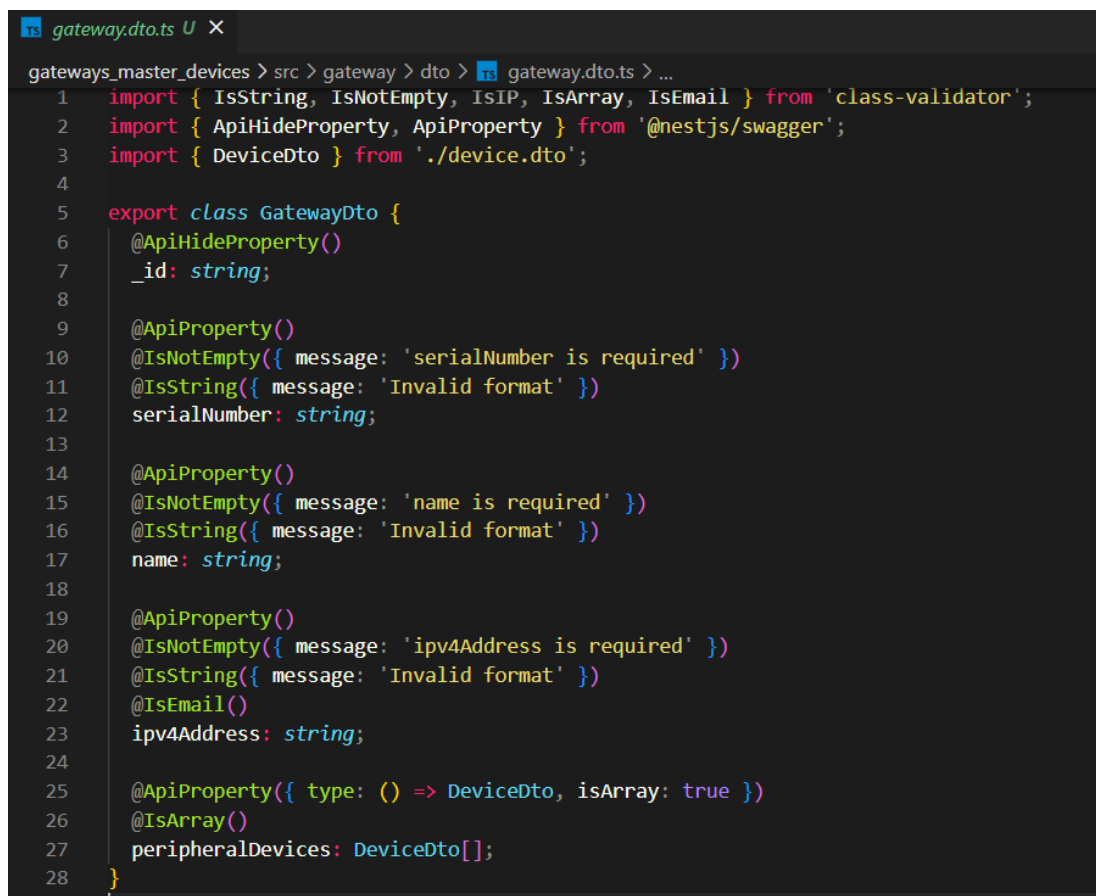
Para el manejo de la información que viajará entre el backend y el cualquier frontend se implementaron las clases DTOs pertinentes, GatewayDto y DeviceDto. Cada una de estas clases presentan validaciones básicas que en caso de no cumplirlas el dato que se reciba generará excepciones. Por ejemplo:



```
LEMS  OUTPUT  TERMINAL  COMMENTS
▼ TERMINAL
[Nest] 4528 - 06/07/2023, 10:06:49 PM    LOG [RouterExplorer] Mapped {/api/gateways/:id, PUT} route +1ms
[Nest] 4528 - 06/07/2023, 10:06:49 PM    LOG [NestApplication] Nest application successfully started +3ms
[Nest] 4528 - 06/07/2023, 10:06:49 PM    LOG Listening at http://localhost:3000/api/swagger
[Nest] 4528 - 06/07/2023, 10:10:08 PM    ERROR [ExceptionsHandler] Gateway validation failed: name: Path `name` is required., serialNumber: Path `serialNumber` is required.
ValidationException: Gateway validation failed: name: Path `name` is required., serialNumber: Path `serialNumber` is required.
    at model.Document.invalidate (C:\Users\Amed\Desktop\CLARKE\gateways_master_devices\node_modules\.pnpm\mongoose@7.2.2\node_modules\mongoose\lib\document.js:3147:32)
    at C:\Users\Amed\Desktop\CLARKE\gateways_master_devices\node_modules\.pnpm\mongoose@7.2.2\node_modules\mongoose\lib\document.js:2940:17
```

*Ilustración 4: Validaciones en Dto*

A continuación, las clases DTOs:



```
gateway.dto.ts U x
gateways_master_devices > src > gateway > dto > gateway.dto.ts > ...
1  import { IsString, IsNotEmpty, IsIP, IsArray, IsEmail } from 'class-validator';
2  import { ApiHideProperty, ApiProperty } from '@nestjs/swagger';
3  import { DeviceDto } from '../device.dto';
4
5  export class GatewayDto {
6    @ApiHideProperty()
7    _id: string;
8
9    @ApiProperty()
10   @IsNotEmpty({ message: 'serialNumber is required' })
11   @IsString({ message: 'Invalid format' })
12   serialNumber: string;
13
14   @ApiProperty()
15   @IsNotEmpty({ message: 'name is required' })
16   @IsString({ message: 'Invalid format' })
17   name: string;
18
19   @ApiProperty()
20   @IsNotEmpty({ message: 'ipv4Address is required' })
21   @IsString({ message: 'Invalid format' })
22   @IsEmail()
23   ipv4Address: string;
24
25   @ApiProperty({ type: () => DeviceDto, isArray: true })
26   @IsArray()
27   peripheralDevices: DeviceDto[];
28 }
29
```

*Ilustración 5: GatewayDto*

```

device.dto.ts U x
gateways_master_devices > src > gateway > dto > device.dto.ts > DeviceDto
1  import { IsNumber, IsString, IsNotEmpty, IsEnum, IsDate, isEmpty } from 'class-validator';
2  import { ApiProperty } from '@nestjs/swagger';
3
4  export class DeviceDto {
5      @ApiProperty()
6      @IsNotEmpty({ message: 'uid is required' })
7      @IsNumber()
8      uid: number;
9
10     @ApiProperty()
11     @IsNotEmpty({ message: 'vendor is required' })
12     @IsString({ message: 'Invalid format' })
13     vendor: string;
14
15     @ApiProperty()
16     @IsNotEmpty({ message: 'createdDate is required' })
17     @IsDate()
18     createdDate: Date;
19
20     @ApiProperty({ enum: ['online', 'offline'], default: 'offline' })
21     @IsNotEmpty({ message: 'status is required' })
22     @IsEnum(['online', 'offline'])
23     status: string;
24 }

```

Ilustración 6: DeviceDto

Como se mencionó anteriormente, se emplea como base de datos MongoDB, por lo tanto, se debe tener instalado previamente el servidor y en caso de tener sistema operativo Windows entonces se recomienda también la interfaz MongoDB Compass. Cuando se realiza la primera transacción a la base de datos esta será generada en conjunto con el registro que se haya insertado.

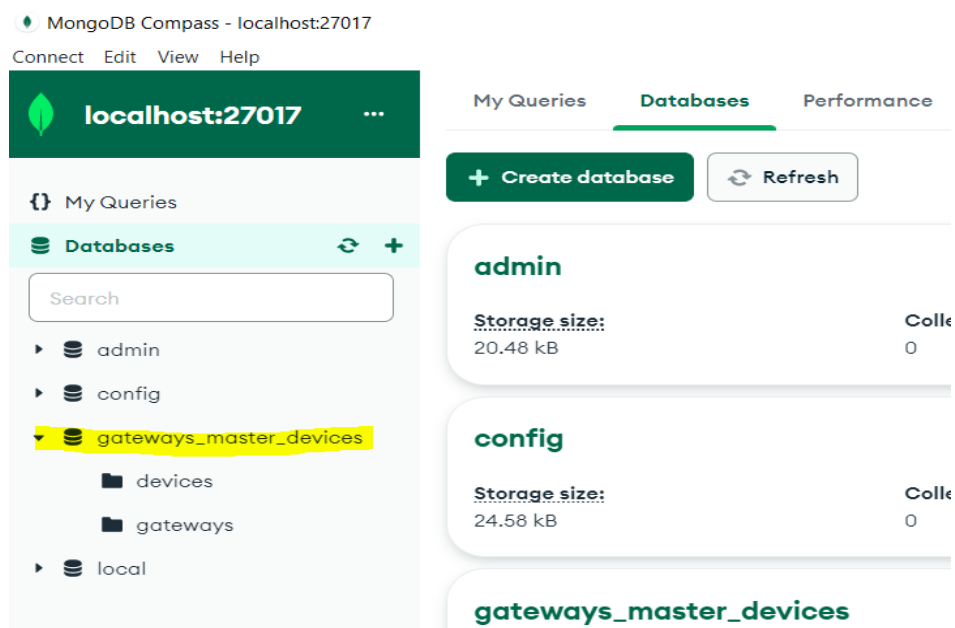


Ilustración 7: MongoDB Compass

Para probar la aplicación se debe ejecutar `npm start` o `pnpm start`, según sea el caso que se tenga instalado y dirigirse a la url señalada en los logs.

```
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [NestFactory] Starting Nest application...
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [InstanceLoader] AppModule dependencies initialized +93ms
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [InstanceLoader] MongooseModule dependencies initialized +1ms
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [InstanceLoader] MongooseCoreModule dependencies initialized +55ms
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [InstanceLoader] MongooseModule dependencies initialized +42ms
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [InstanceLoader] GatewayModule dependencies initialized +4ms
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [RoutesResolver] GatewayController {/api/gateways}: +72ms
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [RouterExplorer] Mapped {/api/gateways, POST} route +7ms
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [RouterExplorer] Mapped {/api/gateways, GET} route +1ms
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [RouterExplorer] Mapped {/api/gateways/:id, GET} route +1ms
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [RouterExplorer] Mapped {/api/gateways/:id/devices, POST} route +1ms
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [RouterExplorer] Mapped {/api/gateways/:id/devices/:deviceId, DELETE} route +9ms
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [RouterExplorer] Mapped {/api/gateways/:id, DELETE} route +4ms
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG [NestApplication] Nest application successfully started +12ms
[Nest] 28320 - 06/07/2023, 11:17:19 PM LOG Listening at http://localhost:3000/api/swagger
```

Ilustración 8: Ejecución del API

El API está conformada por EndPoints básicos. Para insertar, buscar, eliminar y listar gateways, así como para eliminar e insertar un device de un gateway dado:

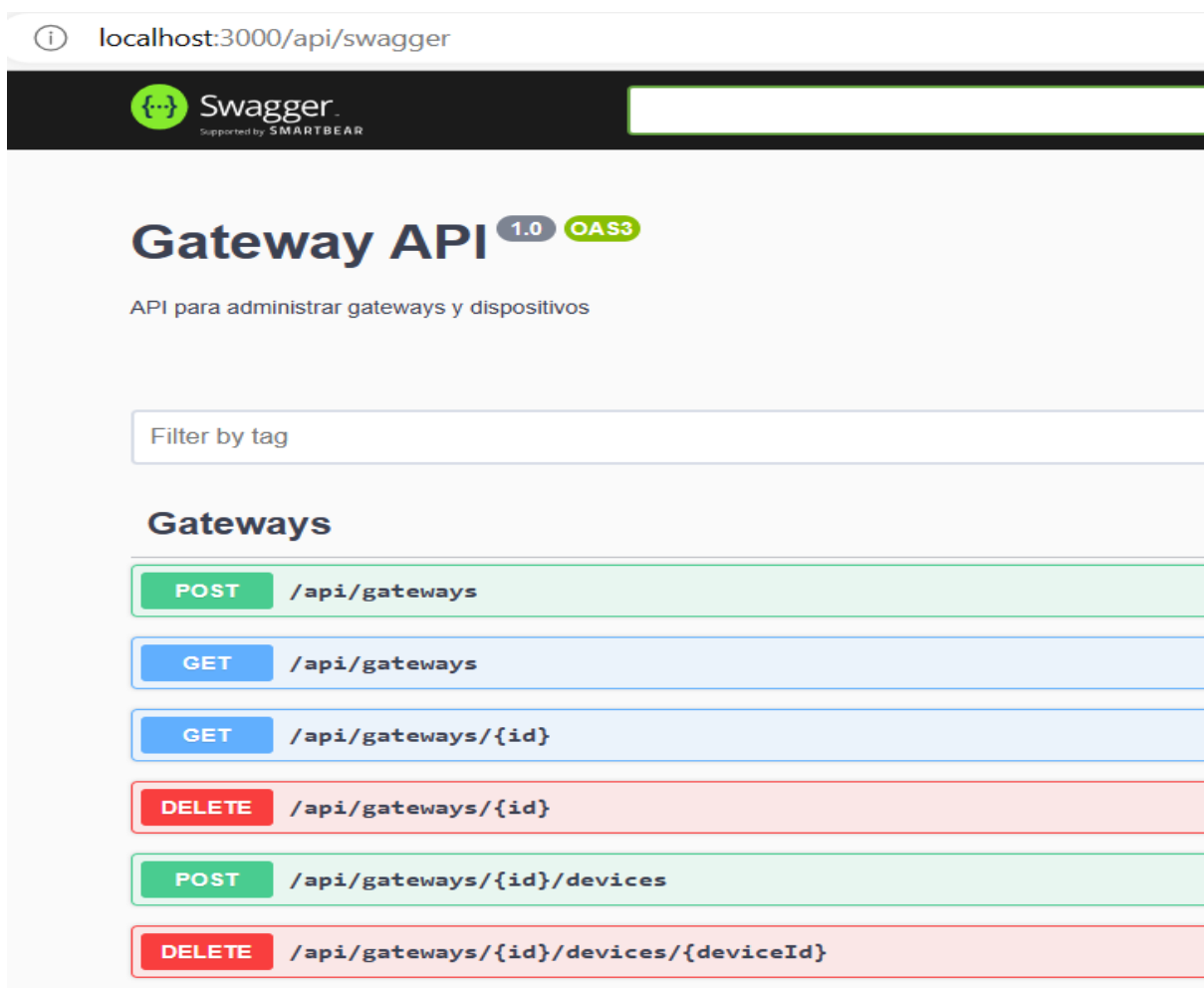
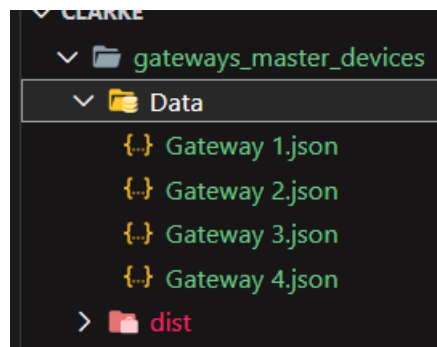
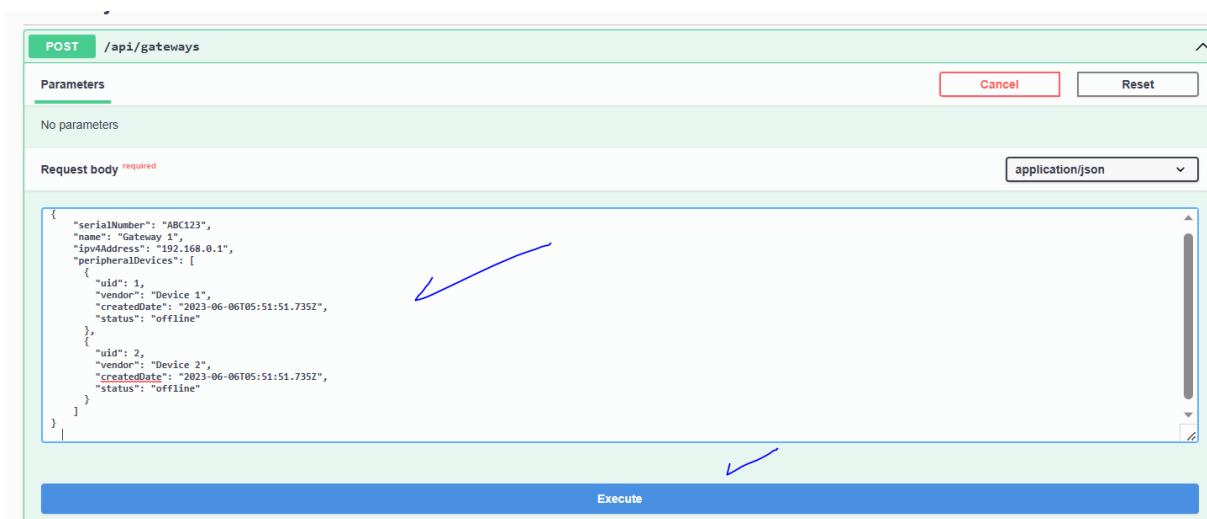


Ilustración 9: Imagen del API en Swagger

Para probar la creación de gateways nos dirigimos al Post `/api/gateways` e introducimos la data, teniendo en cuenta todas las consideraciones iniciales de validación y requerimientos de modelo antes mencionado. Para facilitar el proceso de prueba se deja en la raíz del proyecto una carpeta Data con 4 ficheros json que se pueden copiar y pegar directamente en el cuerpo de la solicitud sustituyendo la estructura que se sugiere por defecto por el contenido de cada json.



*Ilustración 10: Json para insertar*



*Ilustración 11: Inserción de Gateway*

Todos los objetos en los json serán creados excepto el cuarto que generará una excepción porque tiene 10 devices y ese era un requerimiento del mini proyecto. Si se desea realizar esto a mano recordar las recomendaciones iniciales. Los gateways serán creados con sus devices asociados.

Cuando se desea mostrar en pantalla una información es necesario listar y obtener por id y precisamente esto lo permiten los EndPoint `GET /api/gateway` y `GET /api/gateway/{id}`. Este último requiere el id del gateway a mostrar como parámetro.

GET

/api/gateways

Parameters

No parameters

Execute

Responses

Curl

```
curl -X 'GET' \
'http://localhost:3000/api/gateways' \
-H 'accept: */*'

```

Request URL

```
http://localhost:3000/api/gateways

```

Server response

Code	Details
200	<div>Response body</div> <pre>[   {     "id": "64814be02b4858d4d392e564",     "serialNumber": "ABC123",     "name": "Gateway 1",     "ipv4Address": "192.168.0.1",     "peripheralDevices": [       {         "uid": 1,         "vendor": "Device 1",         "createdDate": "2023-06-06T05:51:51.735Z",         "status": "offline"       },       {         "uid": 2,         "vendor": "Device 2",         "createdDate": "2023-06-06T05:51:51.735Z",         "status": "offline"       }     ]   } ]</pre>

Ilustración 12: Listar gateways

**GET** /api/gateways/{id}

**Parameters**

Name	Description
<b>id</b> * required string (path)	<input type="text" value="64814be02b4858d4d392e564"/>

**Execute**

**Responses**

**Curl**

```
curl -X 'GET' \
  'http://localhost:3000/api/gateways/64814be02b4858d4d392e564' \
  -H 'accept: */*'
```

**Request URL**

```
http://localhost:3000/api/gateways/64814be02b4858d4d392e564
```

**Server response**

Code	Details
200	<p><b>Response body</b></p> <pre>{   "_id": "64814be02b4858d4d392e564",   "serialNumber": "ABC123",   "name": "Gateway 1",   "ipv4Address": "192.168.0.1",   "peripheralDevices": [     {       "uid": 1,       "vendor": "Device 1",       "createdAt": "2023-06-06T05:51:51.735Z",       "status": "offline"     },     {       "uid": 2,       "vendor": "Device 2",</pre>

*Ilustración 13: Obtener gateway por Id*

Similar al Obtener por Id funciona el DELETE /api/gateway/{id}. Mediante el elimina el gateway correspondiente y sus devices asociados. En este mini proyecto las operaciones se realizan en función de que los devices dependan de sus gateways pero no es algo obligatorio pues es una base de datos no relacional y si no se le implementa la lógica correspondiente los devices quedarían por su cuenta sin depender de gateway alguno. Cuando se elimine o se agregue un gateway es posible verificar el estado actual de los gateways seleccionando el Endpoint de listar.

Otro endpoint de interés es el POST para adicionar un device a un gateway, el cual se le pasa la data del device a adicionar en el cuerpo de la solicitud y requiere como parámetro el id del gateway destino. Y por último, el endpoint DELETE para eliminar un device de un Gateway pasando como primer parámetro el id del gateway y como segundo parámetro el id del device a eliminar.



*Espero que esta información le sea útil para que pueda guiarse, probar y entender el mini proyecto sin morir en el intento. Muchas Gracias.*