# Universidad de Cuenca

## Facultad de Ingeniería


# Nest.js

---

## Nest.js: Boletín

---


### Estudiantes

María Belén Sarmiento Quezada
Mateo Nicolás Once Campoverde
Jonathan Javier Ortega Molina


### Asignatura

Programación web

### Docente

Ing. Priscila Cedillo

### Carrera

Computación

### Fecha

Miércoles, 26 de junio de 2024

# 1. Nest.js

Nest.js es un framework de NodeJS para construir aplicaciones backend eficientes, escalables y de nivel empresarial. Su principal característica es que las aplicaciones desarrolladas con este framework son escalables y mantenibles.

Dado que Nest.js usa TypeScript, permite tener un código tipado que ayuda a la comprensión y mantención de mismo. Por otro lado, la incorporación de su potente CLI (Command-Line Interface) permite aumentar la productividad y facilidad en el desarrollo, permitiendo que sea muy fácil poner en marcha un proyecto a nivel de servidor

# 2. Nest.js CLI

Nest.js CLI es una herramienta de línea de comandos que simplifica bastante el proceso de desarrollo. Con la CLI se pueden generar nuevos proyectos, gestionar proyectos existentes y realizar tareas de desarrollo comunes como crear, probar o desplegar una aplicación de Nest.js.

# 3. Instalación de Nest.js CLI

Para poder instalar Nest.js CLI es necesario tener instalado Node, para eso se puede usar el siguiente enlace enlace. Luego, con el uso de **npm** se puede instalar Nest mediante el siguiente comando:

```
npm i -g @nestjs/cli
```

## 3.1. Comandos de Nest.js CLI

1. Nuevo proyecto

   ```
   nest new project-name
   ```

2. Generar componentes

   ```
   nest generate <comando>
   nest g <comando>
   ```

3. Componentes comunes

   ```
   # Crear una clase
   nest g cl <path/nombre>

   # Crear un controlador
   nest g co <path/nombre>

   # Crear un decorador
   nest g d <path/nombre>
   ```

```
# Crear un guard
nest g gu <path/nombre>

# Crear un interceptor
nest g in <path/nombre>

# Crear un módulo
nest g mo <path/nombre>

# Crear un pipe
nest g pi <path/nombre>

# Crear un servicio
nest g s <path/nombre>

# Crear un recurso completo
nest g resource <nombre>
```

# 4. Building bloks de Nest.js

## 4.1. Decoradores

Los decoradores son propios de JavaScript y TypeScript, sin embargo, son sumamente importantes en Nest.js, ya que expande funcionalidades de un método, propiedad o clase a la cual se adjuntan. Nest.js busca implementar el principio DRY (Don"t repeat yourself) con el uso de los decoradores.

### 4.1.1. Ejemplos

```
//Default Get
@Get()
//Obtener parametros  segmentos
@Param("id")
// Obtener el body de la peticion
@Body()
// Obtener los parametros de query
@Query()
// Obtener respose (Express/Fastify)
@Res()
```

Listing 1: Ejemplo de TypeScript

## 4.2. Pipes

Transforman la data recibida de los requests. De esta manera se asegura un tipo, valor o instancia de un objeto.

## 4.3. Controllers

Controlan un endpoint de una API, encargados de escuchar la solicitud y emitir una respuesta.

| | | |
|---|---|---|
| IsOptional | IsPositive | IsMongoId |
| IsArray | IsString | IsUUID |
| IsDecimal | IsDate | IsDateString |
| IsBoolean | IsEmail | IsUrl |

### 4.4. Services

Alojan la lógica de negocio, de manera que sea reutilizable mediante inyección de dependencias. En esta parte se realizan los CRUDS.

### 4.5. Módulos

Agrupan y desacoplan un conjunto de funcionalidad específica por dominio. Por ejemplo, en una aplicación puede existir un módulo de autenticación, que se encarga exclusivamente de todo lo relacionado con la autenticación.

### 4.6. DTO

Un DTO de Nest.js es una manera de validar la data que se recibe en las peticiones HTTP. Esto permite que no se ingrese información basura o errónea en la base de datos, ya que si la data no cumple lo que marca el dominio, no se puede ingresar esa información en la base de datos.

### 4.7. Resources

En NestJS, un **resource** es una representación modular de una entidad o conjunto de entidades en una aplicación. Los resources abarcan varios componentes que permiten la gestión de datos en una tabla de base de datos y la exposición de endpoints HTTP para la manipulación de estos datos. En estos están los controladores, servicios, dto"s y entidades.

## 5. Métodos HTTP comunes

```
Import {
Get, Post, Put, Path, Delete
} from "@nestjs/common";
```

### 5.1. Argumentos en métodos HTTP

```
# Default Get
@Get()
# Con segmento dinámico
@Get(":id")
# Especificando una ruta
@Get("cats/breed")
@Get(["cats","breed"])
# Paths dinámicos
@Get(":product/:size")
```

# 6. Ejemplo práctico

Se realizará un CRUD completo de las tablas Persona y Teléfono, que tienen una relación 1:n como se observa en el siguiente diagrama:

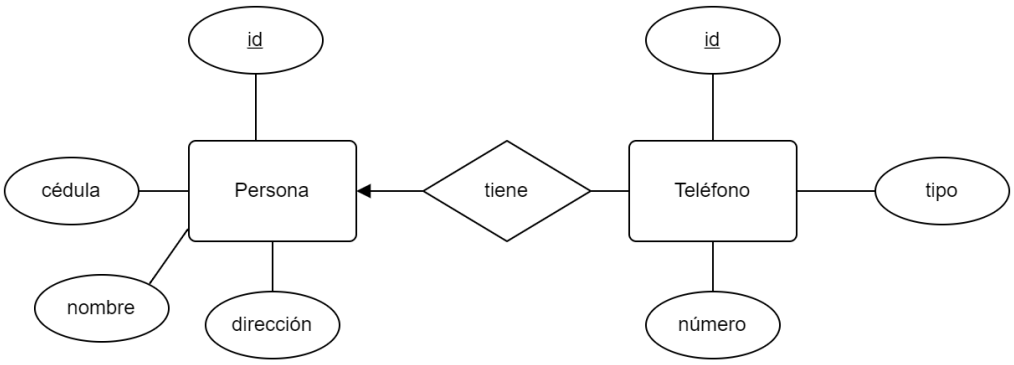


Figura 1: Diagrama entidad-relación

Revisar la guía de instalación para crear el proyecto e instalar las dependencias necesarias. Una vez creado el proyecto tendrá una estructura como la que se muestra a continuación:

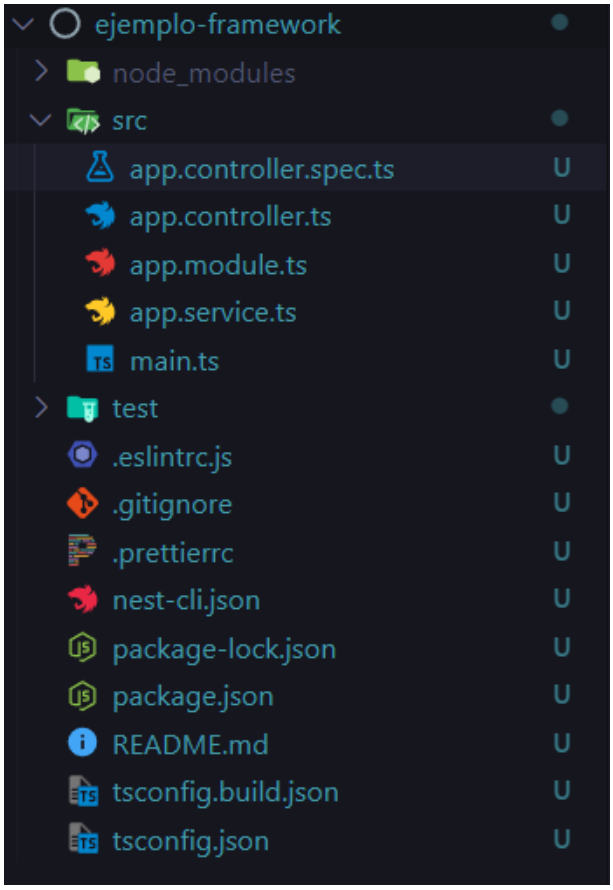


Figura 2: Estructura inicial del proyecto

## 1. Ejecutar la aplicación

Para evitar interrupciones, se recomienda no usar la terminal integrada de VSC, sino abrir una consola aparte. Se debe navegar a la carpeta donde se encuentra el proyecto con

```
cd ubicacion_archivo
```

Verificar que el proyecto esté corriendo con el comando:

```
npm run start:dev
```

## 2. Configurar el archivo main.ts

Por cuestiones de convención, versionado, claridad y organización, se recomienda usar el prefijo api en la aplicación. Por otro lado para que los decoradores aplicados a los DTOs, propios de Nest, se apliquen en tiempo de ejecución, se debe configurar los pipes de forma global. Para configurar esto, se modifica el archivo main.ts de la siguiente forma:

```
import { NestFactory } from "@nestjs/core";
import { AppModule } from "./app.module";

import { ValidationPipe } from "@nestjs/common";
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors();
  app.setGlobalPrefix("api");
  await app.listen(3000);
  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true,
      forbidNonWhitelisted: true
    })
  )

}
bootstrap();
```

## 3. Conectar con la base

Se va a trabajar usando TypeORM porque permite trabajar con diferentes motores de bases de datos como MySQL, MariaDB, PostgreSQL, SQLite, entre otros. Se ha escogido SQLite porque su configuración es sencilla y este ejemplo es pequeño. SQLite creará un archivo .db que gracias a la extensión del VSC nos permitirá ver como se van creando las tablas. Para configurarlo, se modifica el archivo app.module.ts. como se muestra a continuación:

```
import { Module } from "@nestjs/common";
import { AppController } from "./app.controller";
import { AppService } from "./app.service";
import { TypeOrmModule } from "@nestjs/typeorm";

@Module({
  imports: [

    TypeOrmModule.forRoot({
    type: "sqlite",
    database: "ejemplo.db",
```

```
    entities: [__dirname + "/**/*.entity{.ts,.js}"],
    synchronize: true,
  }),],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

## 4. Crear los resources

Los resources contienen el DTO, la entidad, el controlador, el servicio y el módulo. Para respetar la arquitectura de Nest crear una carpeta dentro de src llamada resources. Después desde la terminal de VSC hacer:

```
cd src/resources
```

Para crear los recursos usar los comandos

```
nest g res personas --no-spec
nest g res telefonos --no-spec
```

Se debe escoger REST API

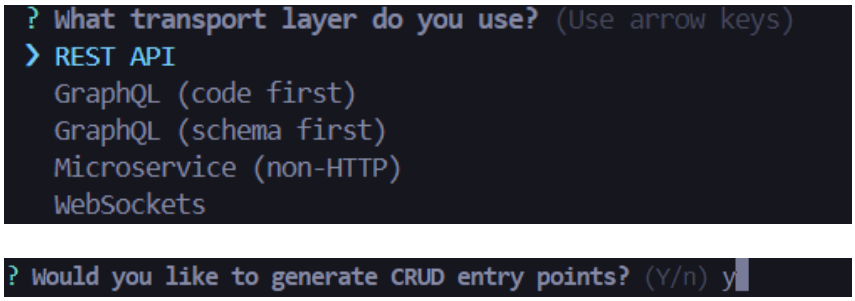


Figura 3: Opciones al crear los resources

## 5. Importar los módulos creados

Importar los módulos de Personas y Teléfonos en el app.module.ts

```
@Module({
  imports: [
    PersonasModule,
    TelefonosModule,

    TypeOrmModule.forRoot({
      type: "sqlite",
      database: "ejemplo.db",
      entities: [__dirname + "/**/*.entity{.ts,.js}"],
      synchronize: true,
    }),],
  controllers: [AppController],
  providers: [AppService],
})
```

## 6. Definir los DTOs

En el archivo create-persona.dto.ts, agregar el siguiente código:

```
import { IsNumberString, IsString } from "@nestjs/class-validator";
import { Length } from "class-validator";

export class CreatePersonaDto {

    @IsNumberString()
    @Length(10)
    cedula: string;

    @IsString()
    nombre: string;

    @IsString()
    direccion: string;

}
```

En el archivo create-telefono.dto.ts, agregar el siguiente código:

```
import { IsNumberString, IsString } from "@nestjs/class-validator";

export class CreateTelefonoDto {

    @IsString()
    tipo: string;

    @IsNumberString()
    numero: string;

    @IsString()
    id_persona: string;
}
```

## 7. Definir las entidades

En archivo persona.entity.ts, agregar el siguiente código:

```
import { Telefono } from "src/resources/telefonos/entities/telefono.entity";
import { Column, Entity, PrimaryGeneratedColumn, OneToMany, JoinColumn } from
"typeorm";

@Entity()
export class Persona {

    @PrimaryGeneratedColumn("uuid")
    id: string;
```

```
    @Column({
        type: "text",
        unique: true
    })
    cedula: string;

    @Column("text")
    nombre: string;

    @Column("text")
    direccion: string;

    @OneToMany(
        () => Telefono,
        (telefono) => telefono.persona
    )
    telefonos: Telefono[];
}
```

## 8. Inyección de dependencias y variables de clase

La inyección de dependencias se realiza en los archivos de extensión .service.ts, en el constructor de la clase. Se inyectan los repositorios que se vayan a utilizar. Además se define una variable de clase llamada logger para un mejor manejo de los errores. Insertar el siguiente código dentro de la clase PersonasService que se encuentra en el archivo personas.service.ts:

```
private logger = new Logger("PersonasService");
  constructor(
    @InjectRepository(Persona)
    private readonly personaRepository: Repository<Persona>
  ) { }
```

Insertar el siguiente código dentro de la clase TelefonosService que se encuentra en el archivo telefonos.service.ts:

```
private logger=new Logger("TelefonosService");

  constructor(
    @InjectRepository(Telefono)
    private readonly telefonoRepository: Repository<Telefono>,
    @InjectRepository(Persona)
    private readonly personaRepository: Repository<Persona>
  ){}
```

## 9. Crear web service POST

Agregar un decorador para el método POST en los controladores de tal manera que en el personas.controller.ts quede:

```
  @Post()
  @UsePipes( ValidationPipe )
```

```
  create(@Body() createPersonaDto: CreatePersonaDto) {
    return this.personasService.create(createPersonaDto);
  }
```

Hacer lo mismo en el archivo telefonos.controller.ts

```
  @Post()
  @UsePipes( ValidationPipe )
  create(@Body() createTelefonoDto: CreateTelefonoDto) {
    return this.telefonosService.create(createTelefonoDto);
  }
```

Modificar los módulos agregando la importación de las entidades. En el archivo personas.module.ts, agregar el siguiente código:

```
import { Module } from "@nestjs/common";
import { PersonasService } from "./personas.service";
import { PersonasController } from "./personas.controller";
import { TypeOrmModule } from "@nestjs/typeorm";
import { Persona } from "./entities/persona.entity";
import { Telefono } from "../telefonos/entities/telefono.entity";

@Module({
  imports:[
    TypeOrmModule.forFeature([Persona])
  ],
  controllers: [PersonasController],
  providers: [PersonasService],
})
export class PersonasModule {}
```

En el archivo telefonos.module.ts, agregar el siguiente código:

```
  import { Module } from "@nestjs/common";
  import { TelefonosService } from "./telefonos.service";
  import { TelefonosController } from "./telefonos.controller";
  import { TypeOrmModule } from "@nestjs/typeorm";
  import { Telefono } from "./entities/telefono.entity";
  import { Persona } from "../personas/entities/persona.entity";

  @Module({
    imports: [
      TypeOrmModule.forFeature([Telefono, Persona])
    ],
    controllers: [TelefonosController],
    providers: [TelefonosService],
  })
  export class TelefonosModule {}
```

Modificar la función create de los servicios. En el archivo personas.service.ts:

```
 async create(createPersonaDto: CreatePersonaDto) {
  try {
```

```
      const persona = this.personaRepository.create(createPersonaDto);
      const personaDB = await this.personaRepository.save(persona);
      return {
        statusCode: 200,
        response: {
          title: "Correcto",
          message: "Persona creada exitosamente",
        },
        personaDB
      }
    } catch (error) {
      this.logger.error(error);
    }
  }
```

De igual manera en el archivo telefonos.service.ts:

```
async create(createTelefonoDto: CreateTelefonoDto) {
    const { id_persona, ...rest } = createTelefonoDto;
    try {

      const registro_Persona = await this.personaRepository.findOne({
        where: { id: id_persona }
      });

      if (!registro_Persona) {
        return {
          statusCode: 404,
          response: {
            title: "No encontrado",
            message: "Persona con id no encontrada"
          }
        }
      }
      const telefono = this.telefonoRepository.create({
        ...rest,
        persona: registro_Persona
      });

      const telefonoDB = await this.telefonoRepository.save(telefono);
      return {
        statusCode: 200,
        response: {
          title: "Correcto",
          message: "Telefono creado exitosamente",
        },
        telefonoDB
      }
    } catch (error) {
```

```
      this.logger.error(error);
    }
  }
```

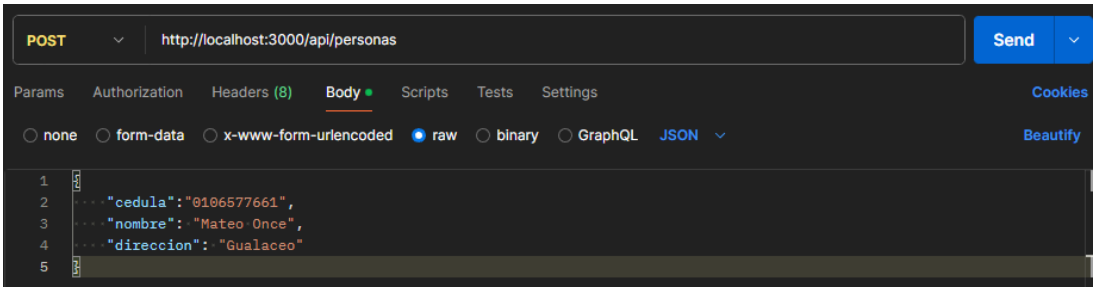


Figura 4: POST de personas

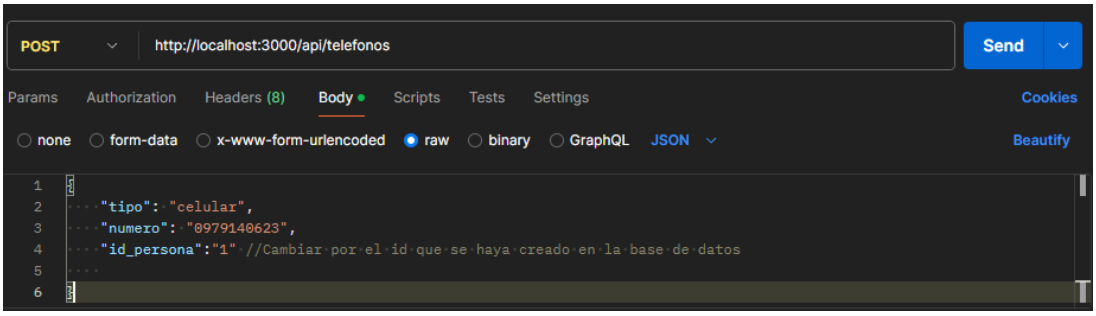


Figura 5: POST de teléfonos

## 10. Crear web service GET

En este web service se manejará la paginación que sirve para recuperar los datos en partes llamadas páginas. Para eso creamos dentro de la carpeta src una carpeta common, dentro una carpeta dto y dentro un archivo pagination.dto.ts que tiene el siguiente contenido

```
export class PaginationDto{

    @IsNumber()
    @IsPositive()
    itemsPage:number=10;

    @IsNumber()
    @IsPositive()
    page:number=1;
}
```

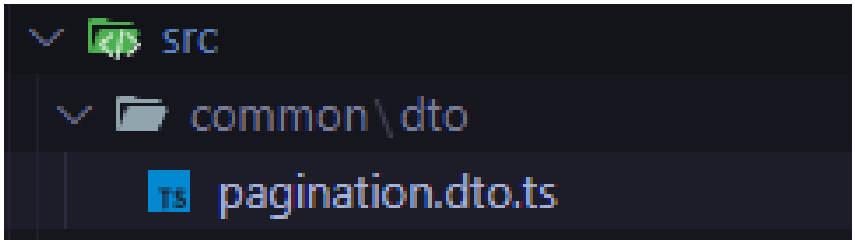


Figura 6: Estructura de archivos

Modificar los controladores, en el archivo personas.controller.ts:

```
  @Get()
  findAll(@Query() paginationDto: PaginationDto) {
    return this.personasService.findAll(paginationDto);
  }
```

En el archivo telefonos.controller.ts

```
@Get()
  findAll(@Query() paginationDto: PaginationDto) {
    return this.telefonosService.findAll(paginationDto);
}
```

Modificar los métodos findAll() de los services. Para el archivo personas.service.ts:

```
    async findAll(paginationDto?: PaginationDto) {

    const { itemsPage = 10, page = 1 } = paginationDto;
    try {
      const personas = await this.personaRepository.find({
        take: itemsPage,
        skip: (page - 1) * itemsPage,
        relations: ["telefonos"]
      });


      return {
        statusCode: 200,
        response: {
          title: "Correcto",
          message: "Personas recuperadas exitosamente",
        },
        pagination: {
          page,
          itemsPage
        },
        personas

      }
    } catch (error) {
      this.logger.error(error);
      return {
        statusCode: 500,
        response: {
          title: "Error",
          message: "Error del servidor",
          error: error
        },
      }
    }
  }
```

Para el archivo telefonos.service.ts:

```
async findAll(paginationDto?: PaginationDto) {

    const { itemsPage = 10, page = 1 } = paginationDto;
    try {
      const telefonos = await this.telefonoRepository.find({
        take: itemsPage,
        skip: (page - 1) * itemsPage,
        relations: ["persona"]
      });


      return {
        statusCode: 200,
        response: {
          title: "Correcto",
          message: "Telefonos recuperados exitosamente",
        },
        pagination: {
          page,
          itemsPage
        },
        telefonos

      }
    } catch (error) {
      this.logger.error(error);
      return {
        statusCode: 500,
        response: {
          title: "Error",
          message: "Error del servidor",
          error: error
        },
      }
    }
  }
```

A continuación se observa el funcionamiento de los GETs.

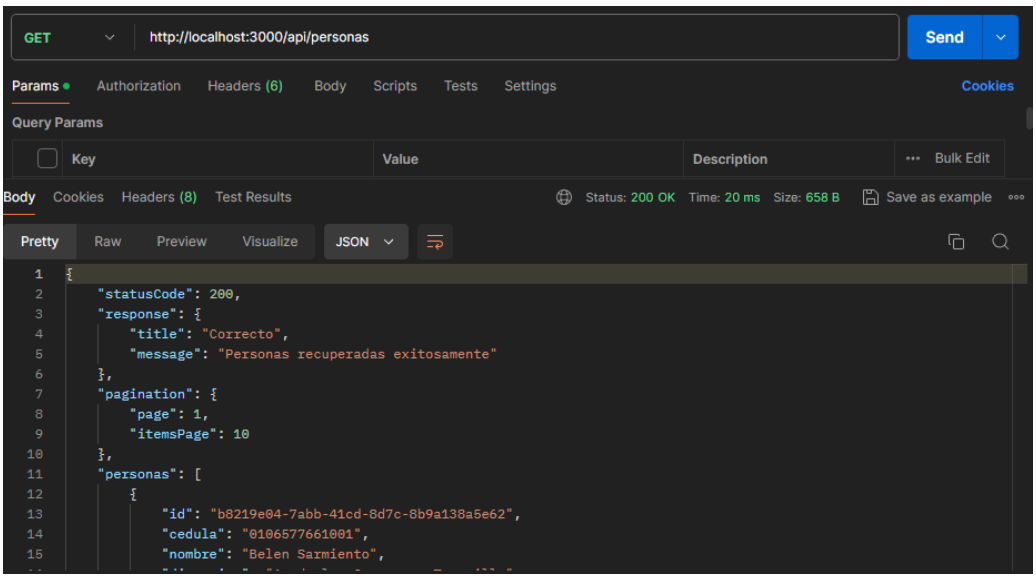


Figura 7: GET personas

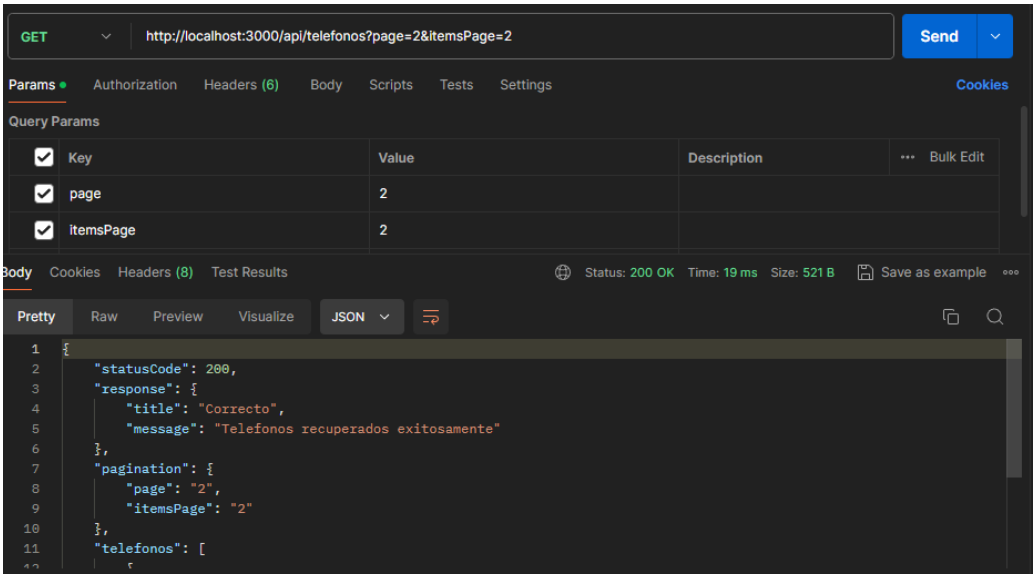


Figura 8: GET telefonos

## 11. Crear web service PATCH o PUT

En Nest se crea como Patch por defecto. Es posible cambiar a Put, ambos sirven para actualizar. Primero se realiza una modificación en los controladores. Para el archivo personas.controller.ts:

```
@Patch(":id")
@UsePipes( ValidationPipe )
update(@Param("id") id: string, @Body() updatePersonaDto: UpdatePersonaDto) {
  return this.personasService.update(id, updatePersonaDto);
}
```

Para el archivo telefonos.controller.ts:

```
@Patch(":id")
```

```
  @UsePipes( ValidationPipe )
  update(@Param("id") id: string, @Body() updateTelefonoDto: UpdateTelefonoDto) {
    return this.telefonosService.update(id, updateTelefonoDto);
  }
```

Modificar las funciones update de los servicios. En el archivo personas.service.ts:

```
    async update(id: string, body: Partial<Persona>) {
    try {
      const registroPersona = await this.personaRepository.findOne({where:{id}});
      if (!registroPersona) {
        return {
          statusCode: 404,
          response: {
            title: "Error",
            message: `Persona con id ${id} no encontrada`,

          },
        }
      }
      const updated = this.personaRepository.merge(registroPersona, body);
      await this.personaRepository.save(updated);

      return {
        statusCode: 200,
        response: {
          title: "Correcto",
          message: "Persona actualizada correctamente"
        },
        persona: updated
      }
    } catch (error) {
      this.logger.error(error);
      return {
        statusCode: 500,
        response: {
          title: "Error",
          message: "Error del servidor",
          error: error
        },
      }
    }

  }
```

Para el archivo telefonos.service.ts:

```
async update(id: string, body: Partial<Telefono>) {
    try {
    const registroTelefono = await this.telefonoRepository.findOne({where:{id}});
      if (!registroTelefono) {
        return {
```

```
        statusCode: 404,
        response: {
          title: "Error",
          message: `Telefono con id ${id} no encontrado`,

        },
      }
    }
    const updated = this.telefonoRepository.merge(registroTelefono, body);
    await this.telefonoRepository.save(updated);

    return {
      statusCode: 200,
      response: {
        title: "Correcto",
        message: "Telefono actualizado correctamente"
      },
      persona: updated
    }
  } catch (error) {
    this.logger.error(error);
    return {
      statusCode: 500,
      response: {
        title: "Error",
        message: "Error del servidor",
        error: error
      },
    }
  }

}
```

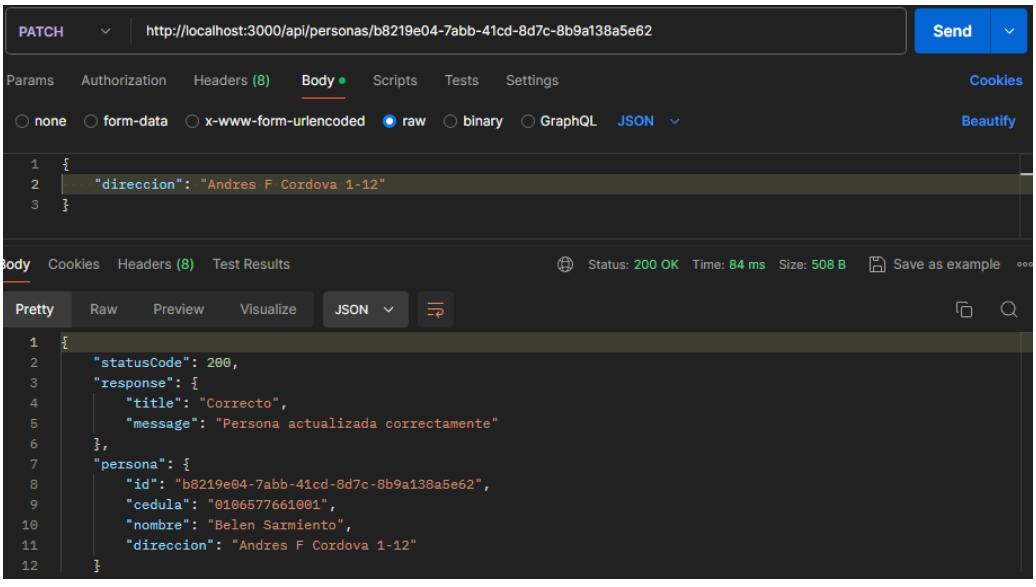


Figura 9: PATCH personas

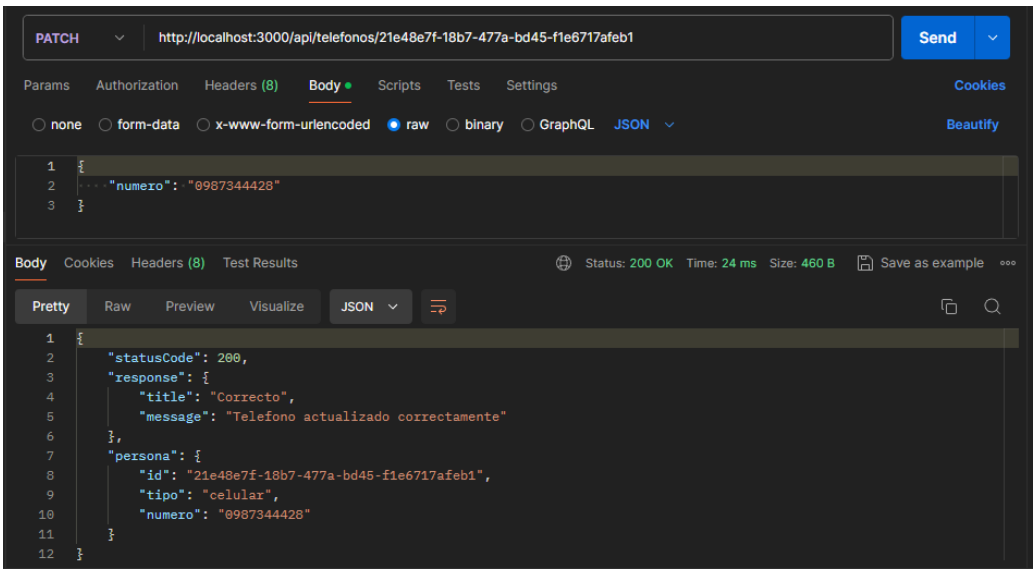


Figura 10: PATCH telefonos

## 11. Crear web service DELETE

Con este web service se maneja la eliminación de datos mediante el método DELETE. Realizar un cambio en el personas.service.ts:

```
@Delete(":id")
remove(@Param("id") id: string) {
  return this.personasService.remove(id);
}
```

En el archivo telefonos.service.ts:

```
@Delete(":id")
remove(@Param("id") id: string) {
  return this.telefonosService.remove(id);
}
```

Modifcar las funciones delete de los servicios. Para el archivo personas.service.ts:

```
async remove(id: string) {
try{
  const registroPersona = await this.personaRepository.findOne({where:{id}});
  if (!registroPersona) {
    return {
      statusCode: 404,
      response: {
        title: "Error",
        message: `Persona con id ${id} no encontrada`,

      },
    }
  }
  const  deletedPersona = await this.personaRepository.delete(registroPersona);
  return{
    statusCode: 200,
```

```
      response: {
        title: "Correcto",
        message: "Persona eliminada exitosamente"
      },
    }

  }catch(error){
    this.logger.error(error);
    return {
      statusCode: 500,
      response: {
        title: "Error",
        message: "Error del servidor",
        error: error
      },
    }
  }

}
```

Para el archivo telefonos.service.ts:

```
  async remove(id: string) {
  try{
    const registroTelefono = await this.telefonoRepository.findOne({where:{id}});
    if (!registroTelefono) {
      return {
        statusCode: 404,
        response: {
          title: "Error",
          message: `Teléfono con id ${id} no encontrado`,

        },
      }
    }
  const  deletedTelefono = await this.telefonoRepository.delete(registroTelefono);
    return{
      statusCode: 200,
      response: {
        title: "Correcto",
        message: "Teléfono eliminado exitosamente"
      },

    }

  }catch(error){
    this.logger.error(error);
    return {
      statusCode: 500,
      response: {
        title: "Error",
```

```
        message: "Error del servidor",
        error: error
      },
    }
  }

}
```

Listing 2: telefonos.service.ts

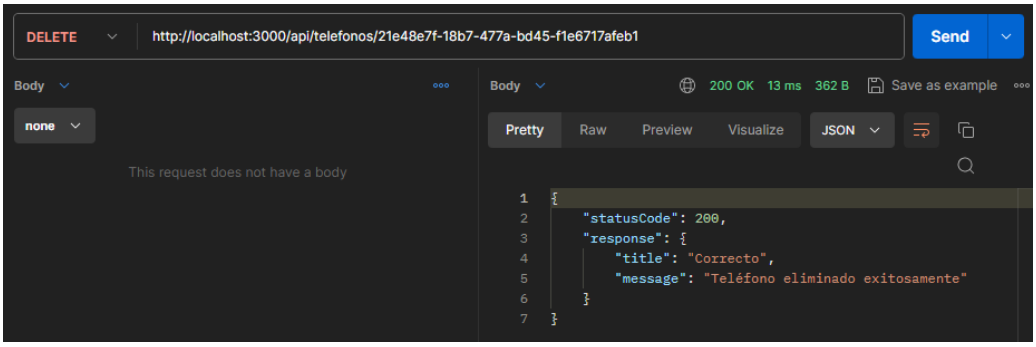


Figura 11: DELETE teléfono

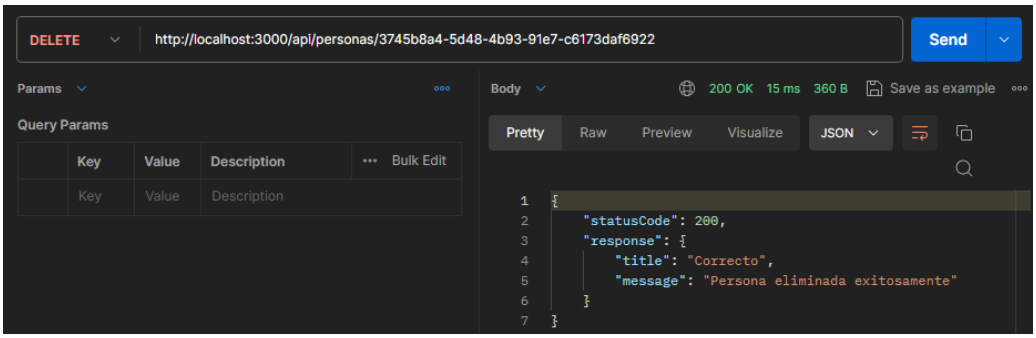


Figura 12: DELETE persona