

PROYECTO DE CRUD DE DOS TABLAS USANDO SPRING CLOUD

INDICE:

<u>1. Introducción</u>	<u>2</u>
<u>2. Microservicio Procedimientos</u>	<u>3</u>
<u>2.1. Validación de los datos de Procedimientos</u>	<u>6</u>
<u>3. Microservicio Intervinientes</u>	<u>8</u>
<u>3.1. Validación de los datos de Intervinientes</u>	<u>12</u>
<u>4. Relación entre microservicios. OpenFeign</u>	<u>14</u>
<u>5. Test</u>	<u>24</u>



1. Introducción:

El presente proyecto consiste en la creación de una aplicación que genere el CRUD de dos tablas de dos bases de datos distintas: una tabla llamada Intervinientes y otra tabla llamada Procedimientos. Cada tabla se encontrará en una base de datos independiente, y se comunicarán ambas bases de datos mediante la aplicación Spring de Java usando un cliente Feign: cada una de las tablas tendrá asociado un microservicio que se comunicará con el otro.

El código del proyecto se encuentra en el siguiente enlace:

[JuanSalvadorFructuosoCampoy/proyectoSpringCloud \(github.com\)](https://github.com/JuanSalvadorFructuosoCampoy/proyectoSpringCloud)

Cada microservicio es un proyecto de Spring, y ambos microservicios estarán implementados dentro de un proyecto padre. Esto queda reflejado en sus archivos pom.xml.

Las bases de datos son de tipo H2, las cuales usarán los puertos 8001 (tabla de procedimientos) y 8002 (tabla de intervinientes) respectivamente.

También se ha añadido la dependencia de OpenFeign y Validation. La primera es para la comunicación entre microservicios y la segunda para realizar una validación de los datos cuando modifiquemos o insertemos los mismos.

Se ha añadido también la dependencia de Lombok para reducir la cantidad de líneas de código en las entidades.

He usado ModelMapper ya que los datos que se introducen en el API no deben incluir fechas de modificación ni de creación. Se introducirán IntervinientesDTO y ProcedimientosDTO y se devolverán Intervinientes y Procedimientos (que tienen todos los campos de IntervinientesDTO y ProcedimientosDTO pero añadiendo las fechas y el usuario).



2. Microservicio Procedimientos:

El microservicio de los procedimientos se encarga de gestionar la tabla Procedimientos, alojada en una base de datos H2, la cual es la siguiente:

SELECT * FROM PROCEDIMIENTOS;						
ID	NUMERO_PROCEDIMIENTO	ANNO	FECHA_CREACION	USUARIO_CREACION	FECHA_MODIFICACION	USUARIO_MODIFICACION
1	PR0001	2020	2024-01-01	JUANSA	null	null
2	PR0002	2021	2024-01-01	JUANSA	null	null
3	PR0003	2022	2024-01-01	JUANSA	null	null
4	PR0004	2023	2024-01-01	JUANSA	null	null
5	PR0005	2024	2024-01-01	JUANSA	null	null
6	PR0011	2024	2024-04-16	JUANSA	null	null

Como podemos comprobar, los atributos son los siguientes:

- ID.
- Número de procedimiento.
- Año.
- Fecha de creación.
- Usuario que lo ha creado (este usuario es el usuario que se ha conectado a la base de datos, mediante la función SQL USER()).
- Fecha de modificación.
- Usuario de modificación, que se obtiene de igual manera que el usuario de creación.

Mediante solicitudes HTTP usando Postman, podemos realizar un CRUD a dicha tabla:

(NOTA: en los cuerpos de la solicitud se ha respetado el formato y color de Postman para poder copiar directamente desde aquí los cuerpos de la solicitud y pegarlos directamente en Postman para hacer las comprobaciones).

- Solicitud GET:
URL: <http://localhost:8001>
GET de un único registro: <http://localhost:8001/{id}> (Siendo el id del registro que queramos ver su detalle)
- Solicitud POST:
URL: <http://localhost:8001>
Cuerpo de la solicitud:



```
{  
  "numeroProcedimiento": "PR0020",  
  "anno": 2022  
}
```

– Solicitud PUT:

URL: <http://localhost:8001/{id}> (Siendo el id del registro que queremos modificar)

Cuerpo de la solicitud:

```
{  
  "numeroProcedimiento": "PR0020",  
  "anno": 2023  
}
```

– Solicitud DELETE:

URL: <http://localhost:8001/{id}> (Siendo el id del registro que queremos eliminar)

Cuando hacemos un cambio en un registro de la base de datos, es importante recalcar que la fecha y el usuario se introducen automáticamente. La fecha actual y el nombre del usuario no están en las solicitudes. Esto es así porque en el repositorio de los procedimientos hemos creado una consulta SQL que nos permite generar el nombre del usuario conectado a la base de datos:

```
@Query(value = "SELECT USER()", nativeQuery = true)  
String getUsuario();
```

Y dicha consulta la establecemos en la implementación del servicio, de tal manera que cuando añadamos un nuevo registro se guardará en `usuarioCreacion` y cuando modifiquemos un registro lo pondremos en `usuarioModificacion`:

```
procedimiento.setUsuarioCreacion(repositorio.getUsuario());  
procedimiento.setUsuarioModificacion(repositorio.getUsuario());
```

La fecha actual de modificación o creación, la hacemos con un `LocalDate.now()`:

```
procedimiento.setFechaModificacion(LocalDate.now());  
procedimiento.setFechaCreacion(LocalDate.now());
```

Por ejemplo, cuando hay cualquier modificación en un registro, automáticamente se



añaden los valores de fechaModificacion y usuarioModificacion:

Veamos el valor original sin modificar:

```
{
  "id": 1,
  "numeroProcedimiento": "PR0001",
  "anno": 2020,
  "fechaCreacion": "2024-01-01",
  "usuarioCreacion": "JUANSÁ",
  "fechaModificacion": null,
  "usuarioModificacion": null,
  "intervinientes": []
},
```

Al modificar el año, de 2020 a 2024 mediante una solicitud PUT, vemos cómo, además de modificar el año, también se añade la fecha actual como fecha de modificación y el usuario que lo ha modificado, siendo el nombre del usuario que está conectado a la base de datos (mediante las credenciales de acceso a la base de datos):

```
{
  "id": 1,
  "numeroProcedimiento": "PR0001",
  "anno": 2024,
  "fechaCreacion": "2024-01-01",
  "usuarioCreacion": "JUANSÁ",
  "fechaModificacion": "2024-04-16",
  "usuarioModificacion": "JUANSÁ",
  "intervinientes": []
}
```



2.1. Validación de los datos de Procedimientos:

Mediante el Hibernate Validation de Spring Boot, se ha realizado una validación de los datos, los cuales impiden insertar un número de procedimiento vacío o duplicado:

```
PUT http://localhost:8001/1

{
  "numeroProcedimiento": "PR0002",
  "anno": 2024
}
```

```
{
  "error": "Ya existe un procedimiento con ese número."
}
```

Status: 400 Bad Request

```
PUT http://localhost:8001/1

{
  "numeroProcedimiento": "",
  "anno": 2024
}
```

```
{
  "error": "El campo numeroProcedimiento no puede quedar vacío ni con espacios en blanco"
}
```

Status: 400 Bad Request

También se ha validado que el JSON introducido sea correcto, mediante un `ExceptionHandler` en el `HttpMessageNotReadableException`:

```
PUT http://localhost:8001/1

{
  "numeroProcedimiento": "PR0002",
  "anno":
}
```

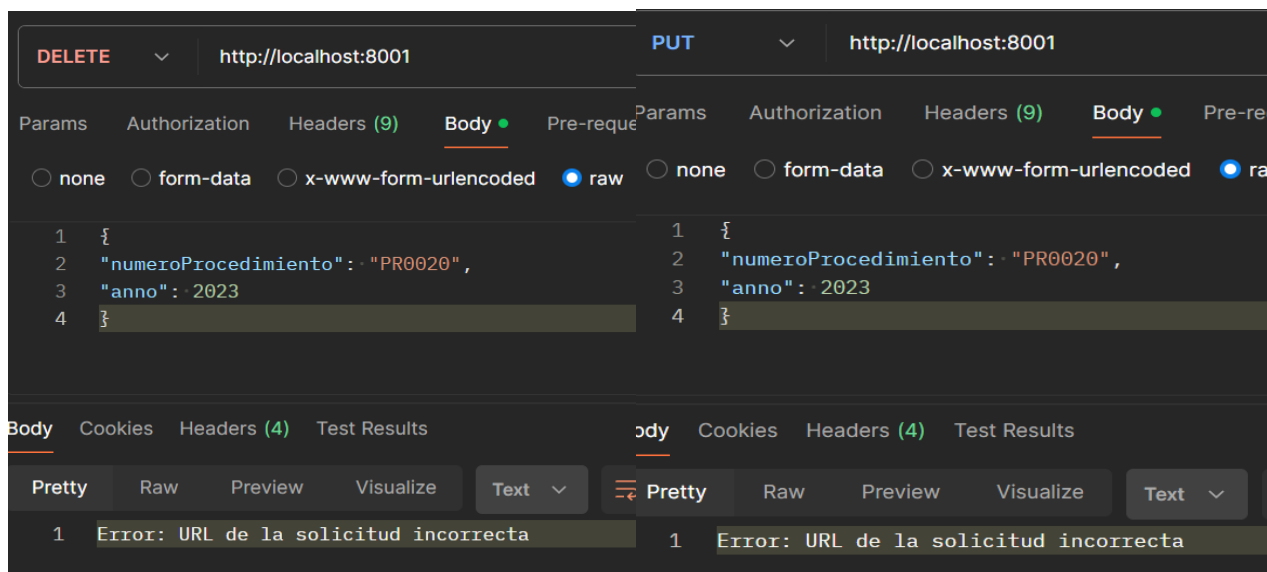
```
{
  "error": "El JSON proporcionado es incorrecto: JSON parse error: Unexpected character ('}' (code 125)): expected a value"
}
```

Status: 400 Bad Request Time: 6 ms Size: 266 B

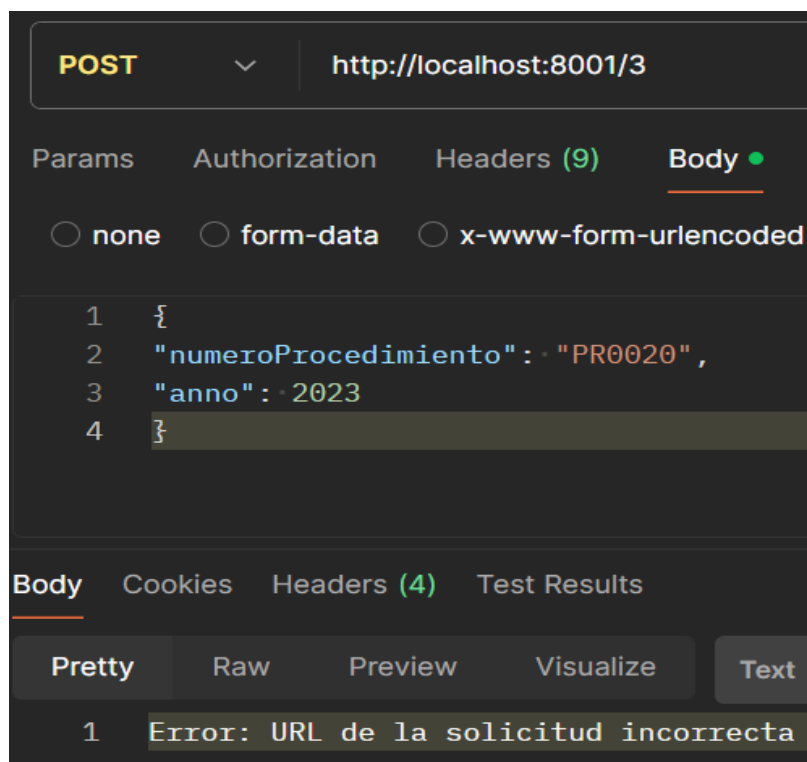
También hay que mencionar que se ha realizado una verificación de la URL, de tal manera



que al introducir un PUT o un DELETE sin especificar la id en la URL, se mandará un mensaje de error:



De la misma forma, si realizamos un POST con una id en la URL, también se mandará un mensaje de error:



3. Microservicio Intervinientes:

El microservicio intervinientes tiene como objetivo realizar el CRUD de la tabla de Intervinientes, que también se encuentra en una base de datos H2, no obstante esta base de datos es diferente a la de Procedimientos. La tabla es la siguiente:

SELECT * FROM INTERVINIENTES;							
ID	NOMBRE	TIPO_INTERVENCION	FECHA_CREACION	USUARIO_CREACION	FECHA_MODIFICACION	USUARIO_MODIFICACION	PROCEDIMIENTO_ID
1	Juan Antonio	Tipo 1	2024-01-01	JUANSÁ	null	null	1
2	Ricardo Martínez	Tipo 2	2024-01-01	JUANSÁ	null	null	1
3	Astrid González	Tipo 3	2024-01-01	JUANSÁ	null	null	2
4	Martín Gutiérrez	Tipo 4	2024-01-01	JUANSÁ	null	null	3
5	Maribel Matías	Tipo 5	2024-01-01	JUANSÁ	null	null	4

Vemos que los atributos son los siguientes:

- ID.
- Nombre.
- Tipo de intervención.
- Fecha de creación.
- Usuario de creación
- Fecha de modificación.
- Usuario de modificación.
- Procedimiento ID.

Al tratarse de una relación ManyToOne desde el lado de intervinientes (un procedimiento puede tener varios intervinientes, pero un interviniente sólo puede estar vinculado a un procedimiento), en la tabla de intervinientes se ha añadido una nueva columna: `procedimiento_id`, que se encargará de almacenar la id del procedimiento al que pertenece este interviniente. Normalmente se establece una Foreign Key para establecer esta relación, pero al estar en dos bases de datos diferentes, no se indica.

Las solicitudes HTTP, usando Postman, son las siguientes:

- Solicitud GET:
URL: <http://localhost:8002>
GET de un único registro: <http://localhost:8002/{id}> (Siendo el id del registro que queramos ver su detalle)
- Solicitud POST:
URL: <http://localhost:8002>



Cuerpo de la solicitud:

```
{  
  "nombre": "Antonio Campillo",  
  "tipoIntervencion": "Tipo 1"  
}
```

– Solicitud PUT:

URL: <http://localhost:8002/{id}> (Siendo el id del registro que queramos modificar)

Cuerpo de la solicitud:

```
{  
  "nombre": "Antonio Campos",  
  "tipoIntervencion": "Tipo 5"  
}
```

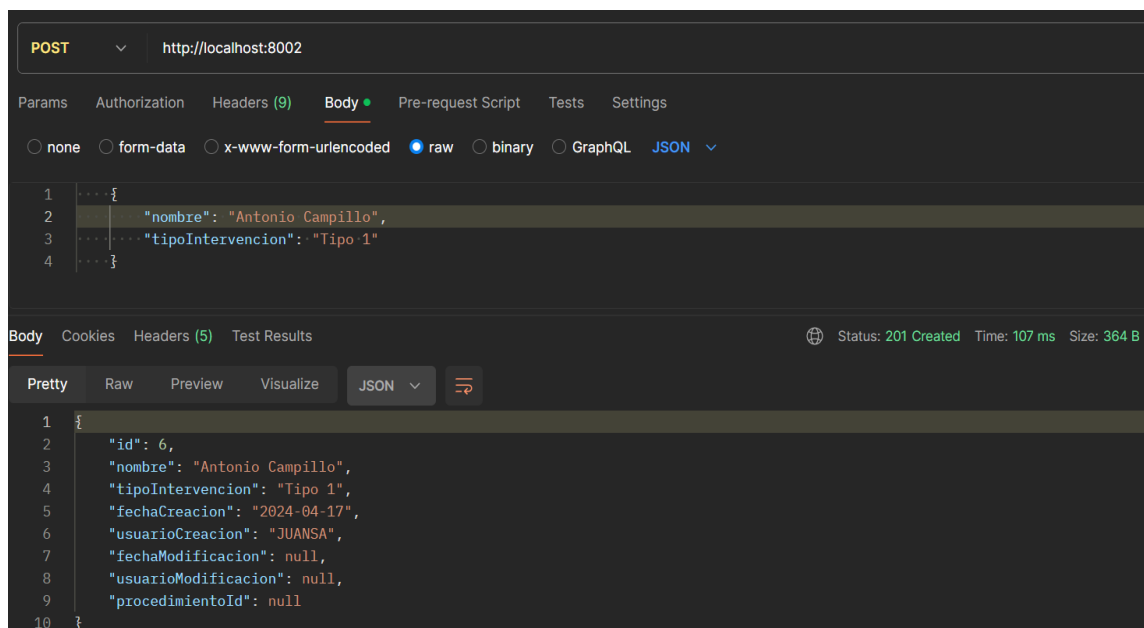
– Solicitud DELETE:

URL: <http://localhost:8002/{id}> (Siendo el id del registro que queramos eliminar)

De la misma manera que el microservicio Procedimientos, en intervinientes tenemos datos de auditoría que se insertan automáticamente cuando creamos registros y/o los modificamos. Dichos datos los obtenemos del método SQL USER() y de LocalDate(), para obtener el nombre del usuario conectado a la base de datos y la fecha actual, respectivamente.

Por ejemplo, al hacer una solicitud POST con un body, obtenemos el siguiente registro:

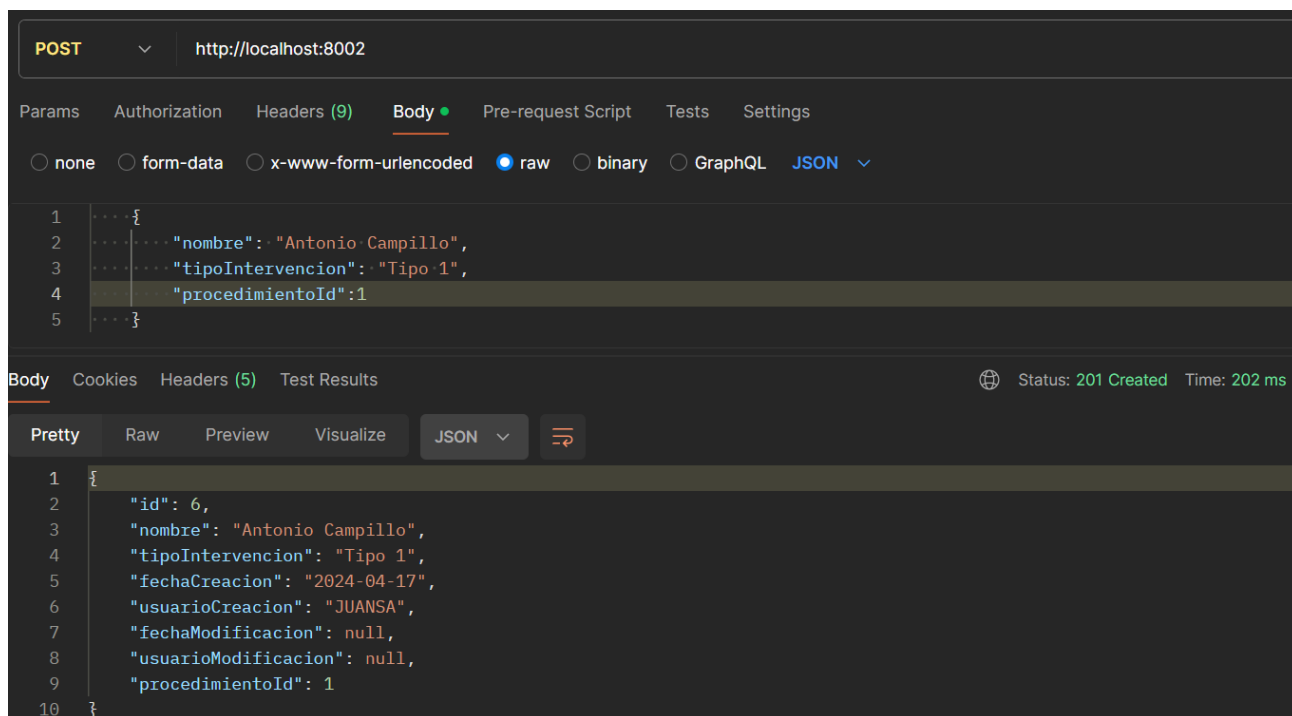




El `procedimiento_id` es opcional. Si no lo especificamos, entonces el interviniente que se creará no tendrá un procedimiento asociado, pero si lo especificamos, entonces se le asignará el procedimiento en el mismo momento en el que se crea. Más adelante veremos cómo hacer este proceso de asignación y desasignación desde el microservicio de procedimientos.

Repetimos el POST anterior, pero esta vez con un procedimiento asociado:





The screenshot displays a REST client interface with a POST request to `http://localhost:8002`. The request body is a JSON object with the following fields: `nombre` (Antonio Campillo), `tipoIntervencion` (Tipo 1), and `procedimientoId` (1). The response status is 201 Created, and the response body is a JSON object with the following fields: `id` (6), `nombre` (Antonio Campillo), `tipoIntervencion` (Tipo 1), `fechaCreacion` (2024-04-17), `usuarioCreacion` (JUANSA), `fechaModificacion` (null), `usuarioModificacion` (null), and `procedimientoId` (1).

```
POST http://localhost:8002

{
  "nombre": "Antonio Campillo",
  "tipoIntervencion": "Tipo 1",
  "procedimientoId": 1
}
```

Status: 201 Created Time: 202 ms

```
{
  "id": 6,
  "nombre": "Antonio Campillo",
  "tipoIntervencion": "Tipo 1",
  "fechaCreacion": "2024-04-17",
  "usuarioCreacion": "JUANSA",
  "fechaModificacion": null,
  "usuarioModificacion": null,
  "procedimientoId": 1
}
```

Podemos comprobar que ahora sí tenemos un procedimiento asociado al crearse el nuevo interviniente.



3.1. Validación de los datos de Intervinientes:

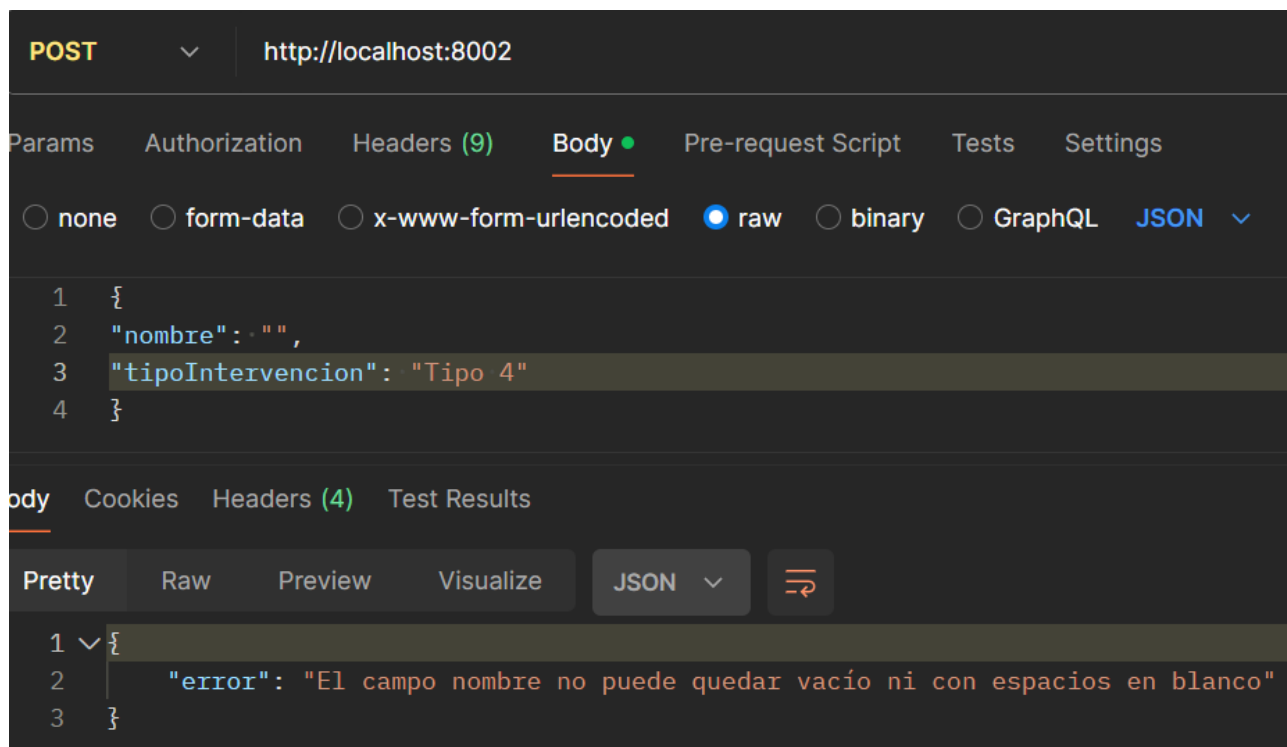
Al igual que con Procedimientos, en Intervinientes también tenemos una verificación de los datos para asegurarnos de que los datos introducidos son correctos:

```
@Column(name = "nombre")
@NotBlank(message = "no puede quedar vacío ni con espacios en blanco")
private String nombre;

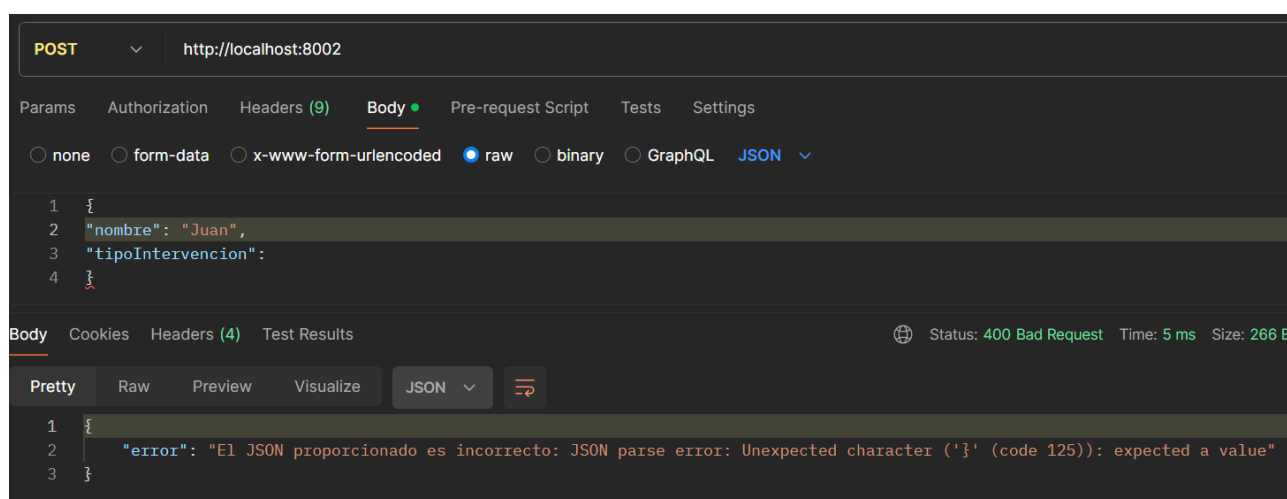
@Column(name = "tipo_intervencion")
@NotBlank(message = "no puede quedar vacío ni con espacios en blanco")
private String tipoIntervencion;
```

The screenshot displays a REST client interface. At the top, a POST request is configured to `http://localhost:8002`. The 'Body' tab is selected, showing a JSON payload: `{ "nombre": "Juan Antonio", "tipoIntervencion": "" }`. Below the request, the 'Test Results' tab shows the response body in 'Pretty' format: `{ "error": "El campo tipoIntervencion no puede quedar vacío ni con espacios en blanco" }`. The response is returned with a status code of 400.





También se ha realizado una comprobación de que el JSON introducido no tiene errores de formato:



En cuanto a las comprobaciones de la id en la URL para las solicitudes PUT, DELETE y POST, sigue los mismos mecanismos que en el microservicio de Procedimientos.



4. Relación entre microservicios. OpenFeign:

Ambos microservicios deben estar relacionados mediante un cliente Feign. Para ello, debemos realizar una serie de cambios en los archivos pom.xml de los microservicios. Lo primero que se ha hecho ha sido comprobar que tengan esta dependencia:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
<dependency>
```

La dependencia de OpenFeign es la encargada de realizar la conexión entre microservicios. A continuación, hemos añadido lo siguiente:

```
<parent>
  <groupId>com.juansa.proyectospringcloud</groupId>
  <artifactId>proyectospringcloud</artifactId>
  <version>0.0.1-SNAPSHOT</version>
</parent>
```

Como se comentó en la introducción, el proyecto consta de dos microservicios (proyectos Spring) alojados dentro de otro proyecto (un proyecto padre), por lo que en el archivo pom.xml de ambos microservicios debemos especificar que el proyecto padre será el proyecto que los va a alojar.

Por su parte, en el archivo pom.xml del proyecto padre, debemos asegurarnos de que se encuentre esto en el apartado parent:

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.2.4</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

Y también, debemos especificar ambos microservicios dentro del archivo pom.xml del proyecto padre mediante lo siguiente:



```
<modules>|
<module>msvc-intervinientes</module>
<module>msvc-procedimientos</module>
</modules>
```

Así, definimos los microservicios dentro del proyecto padre.

Una vez terminadas las modificaciones a los archivos pom.xml, debemos modificar el archivo de configuración y asegurarnos de que tiene lo siguiente:

```
spring.application.name=msvc-intervinientes
server.port=8002
```

Este es el ejemplo del microservicio intervinientes, el cual debe tener especificado el nombre del microservicio y el puerto en el que se encuentra. Esto es para poder configurar la conexión entre microservicios.

Después de establecer los parámetros de configuración, vamos a la clase principal de ambos microservicios y añadimos la anotación `@EnableFeignClients`:

```
@EnableFeignClients  Juan Salvador Fructuoso Campoy
@SpringBootApplication
public class MsvcIntervinientesApplication {

>     public static void main(String[] args) { Spri
```

Como en los requisitos del proyecto se indica que se debe *“utilizar Feign Client para realizar operaciones relacionadas con la gestión de intervinientes desde el microservicio de Procedimiento”*, en el microservicio de procedimientos crearemos un paquete llamado “clients”, que alojará en su interior una interface de Intervinientes llamada `IntervinienteClientRest`:



```

@FeignClient(name = "msvc-intervinientes", url = "localhost:8002") 6 usages Juan Salvador Fructuoso Campoy
public interface IntervinienteClientRest {
    @GetMapping("/{id}") 9 usages Juan Salvador Fructuoso Campoy
    Optional<Interviniente> porId(@PathVariable Long id);

    @PostMapping("/") 3 usages Juan Salvador Fructuoso Campoy
    Interviniente crear(@RequestBody Interviniente interviniante);

    @GetMapping("/intervinientes-por-procedimiento/{id}") 3 usages Juan Salvador Fructuoso Campoy
    List<Interviniente> obtenerIntervinientesPorProcedimiento(@PathVariable Long id);

    @PutMapping("/actualizacion") 2 usages Juan Salvador Fructuoso Campoy
    Interviniente actualizacion(Interviniente interviniante);
}

```

Podemos comprobar que, encabezando la interface, tenemos la anotación `@FeignClient` con los parámetros de conexión al otro microservicio (desde Procedimientos a Intervinientes), los parámetros son el nombre del microservicio y la URL con el puerto de acceso (de ahí que haya sido necesario especificarlo en `application.properties`).

Los métodos que hay en la interface se encuentran en el controlador del microservicio `intervinientes`, de tal manera que, desde el cliente, se llama al controlador de `Intervinientes` y realiza el método especificado. Es importante que la signatura del método deba coincidir o, al menos, ser muy similar al del controlador para evitar incompatibilidades.

Este cliente se inyectará en la implementación del servicio de `Procedimientos`:

```

private final ProcedimientoRepository repositorio; 19 usages
private final ModelMapper modelMapper; 2 usages
private final IntervinienteClientRest cliente; 5 usages

@Autowired 1 usage Juan Salvador Fructuoso Campoy +1
public ProcedimientoServiceImpl(ProcedimientoRepository repositorio, ModelMapper modelMapper, IntervinienteClientRest cliente) {
    this.repositorio = repositorio;
    this.modelMapper = modelMapper;
    this.cliente = cliente;
}

```

Por lo tanto, cuando queramos realizar una solicitud desde `procedimientos` que implique a los `intervinientes`, usaremos el cliente como puente de enlace entre microservicios. Por ejemplo, al hacer una solicitud GET a un procedimiento en específico, queremos tener un apartado de `Intervinientes` en donde veamos los datos de los `intervinientes` que se encuentran en dicho procedimiento. Para ello, en primer lugar debemos tener en la entidad `Procedimiento` el siguiente atributo:




```
public class Procedimiento {  
  
    @Column(name = "fecha_creacion")  
    private LocalDate fechaCreacion;  
  
    @Column(name = "usuario_creacion")  
    private String usuarioCreacion;  
  
    @Column(name = "fecha_modificacion")  
    private LocalDate fechaModificacion;  
  
    @Column(name = "usuario_modificacion")  
    private String usuarioModificacion;  
  
    @Transient  
    private List<Interviniente> intervinientes;  
  
    public Procedimiento() {  
        this.intervinientes = new ArrayList<>();  
    }  
}
```

Esa lista de Intervinientes no se encuentra en la tabla de la base de datos de Procedimiento, por eso tiene la anotación `@Transient` (indica que no permanece y no se refleja en la base de datos, sino que este atributo se controla desde la aplicación). Por otra parte, añadimos un constructor que inicializa un `ArrayList` vacío de intervinientes cada vez que creamos un nuevo objeto de tipo `Procedimiento`.

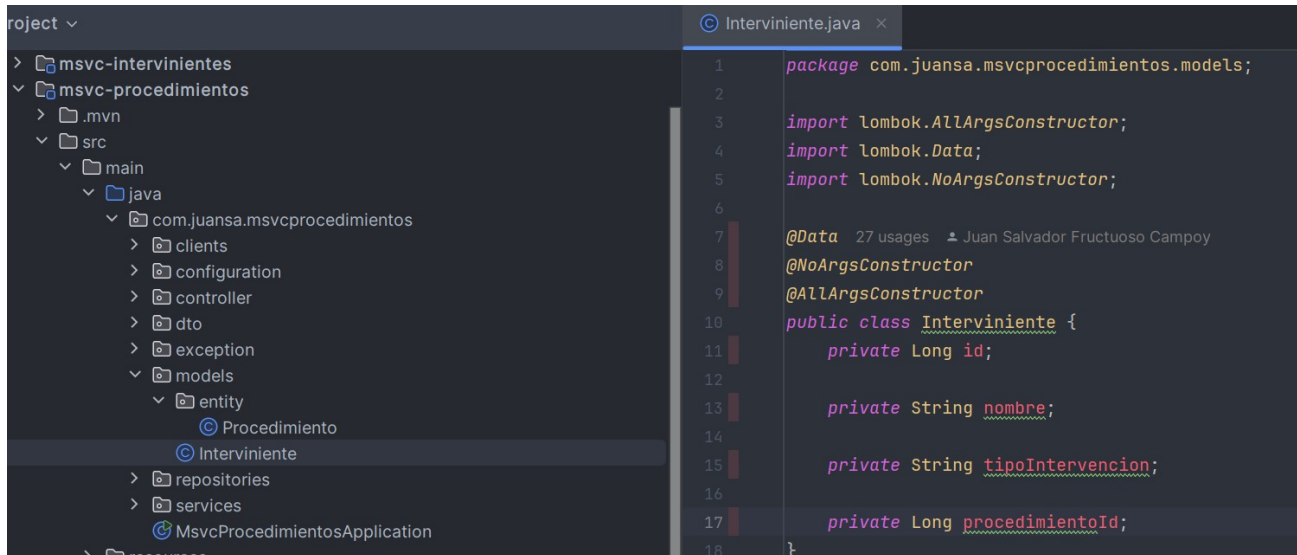
Es importante reflejar que, a la hora de establecer una lista de intervinientes, no podemos importar directamente la entidad interviniente desde su microservicio, por lo que debemos hacer lo siguiente:

- En el microservicio de Procedimiento, creamos un paquete llamado “models”, e



insertamos en su interior el paquete de “entity”.

- Dentro de la carpeta “models” pero fuera de “entity”, creamos una clase de Java llamada “Interviniente”, dentro de la cual creamos un POJO de intervinientes, con los atributos de intervinientes que nos interesa reflejar en el microservicio de Procedimientos:

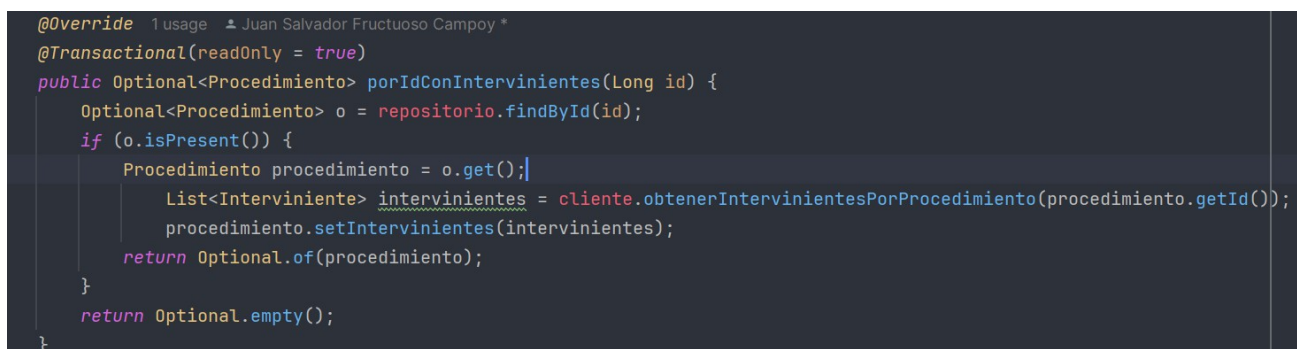


The screenshot shows an IDE with a project structure on the left and the code for the `Interviniente` class on the right. The project structure includes a package `com.juansa.msvcprocedimientos` with sub-packages `clients`, `configuration`, `controller`, `dto`, `exception`, `models`, `repositories`, `services`, and `Interviniente`. The `Interviniente` class is located in the `models` package. The code for the class is as follows:

```
1 package com.juansa.msvcprocedimientos.models;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6
7 @Data 27 usages Juan Salvador Fructuoso Campoy
8 @NoArgsConstructor
9 @AllArgsConstructor
10 public class Interviniente {
11     private Long id;
12
13     private String nombre;
14
15     private String tipoIntervencion;
16
17     private Long procedimientoId;
18 }
```

Este modelo será el que utilice el microservicio de Procedimientos para mostrar los intervinientes.

Si queremos que al hacer una solicitud GET de un procedimiento en concreto, veamos también los datos de los intervinientes que tiene vinculados, haremos lo siguiente en el servicio de procedimientos:



The screenshot shows the code for the `porIdConIntervinientes` method in the service layer. The code is as follows:

```
@Override 1 usage Juan Salvador Fructuoso Campoy *
@Transactional(readOnly = true)
public Optional<Procedimiento> porIdConIntervinientes(Long id) {
    Optional<Procedimiento> o = repositorio.findById(id);
    if (o.isPresent()) {
        Procedimiento procedimiento = o.get();
        List<Interviniente> intervinientes = cliente.obtenerIntervinientesPorProcedimiento(procedimiento.getId());
        procedimiento.setIntervinientes(intervinientes);
        return Optional.of(procedimiento);
    }
    return Optional.empty();
}
```

Como se puede ver, hacemos una llamada al cliente de intervinientes, el cual llama al controlador de Intervinientes:

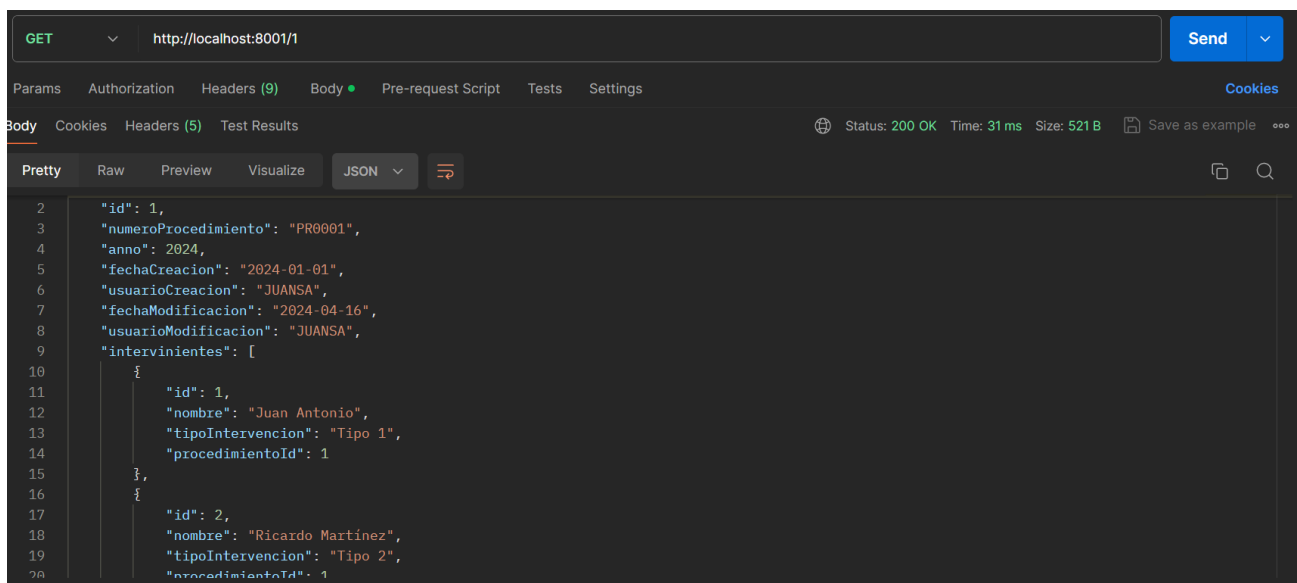


```
@GetMapping("/intervinientes-por-procedimiento/{id}") no usages Juan Salvador Fructuoso Campoy
public ResponseEntity<Object> obtenerIntervinientesPorProcedimiento(@PathVariable Long id) {
    return ResponseEntity.ok(servicio.listarPorProc(id));
}
```

Dicho controlador seguirá el flujo habitual de este tipo de aplicaciones (del controlador al servicio, del servicio al repositorio), y es en el repositorio donde hemos creado una consulta SQL personalizada:

```
@Query("SELECT i FROM Interviniente i WHERE i.procedimientoId = ?1")
List<Interviniente> buscarPorProcedimientoId(Long procedimientoId);
```

Al hacer esto, al hacer la ya mencionada solicitud GET al procedimiento, tenemos lo siguiente:



The screenshot shows a REST client interface with a GET request to `http://localhost:8001/1`. The response status is 200 OK, with a time of 31 ms and a size of 521 B. The response body is a JSON object representing a procedure with its details and a list of participants.

```
{
  "id": 1,
  "numeroProcedimiento": "PR0001",
  "anno": 2024,
  "fechaCreacion": "2024-01-01",
  "usuarioCreacion": "JUANSa",
  "fechaModificacion": "2024-04-16",
  "usuarioModificacion": "JUANSa",
  "intervinientes": [
    {
      "id": 1,
      "nombre": "Juan Antonio",
      "tipoIntervencion": "Tipo 1",
      "procedimientoId": 1
    },
    {
      "id": 2,
      "nombre": "Ricardo Mart\u00ednez",
      "tipoIntervencion": "Tipo 2",
      "procedimientoId": 1
    }
  ]
}
```

Podemos ver que se muestran los detalles de los intervinientes que participan en cada procedimiento, mientras que si hacemos una solicitud GET sin especificar ningún id de procedimiento, no se mostrarán los detalles de intervinientes.

A continuación, pasamos al método que permite asignar intervinientes a un procedimiento. Se hace mediante una petición PUT al microservicio de procedimientos.

El código dentro del servicio de procedimiento es el siguiente:



```
@Override 1 usage Juan Salvador Fructuoso Campoy
@Transactional
public Optional<Interviniente> asignarInterviniente(Interviniente interviniente, Long procedimientoId) {
    Optional<Procedimiento> opt = repositorio.findById(procedimientoId);
    if(opt.isPresent()) {
        Optional<Interviniente> intervinienteDb = cliente.findById(interviniente.getId());
        if (intervinienteDb.isEmpty()) {
            return Optional.empty();
        }
        Procedimiento procedimiento = opt.get();
        procedimiento.getIntervinientes().add(intervinienteDb.get());
        procedimiento.setFechaModificacion(LocalDate.now());
        procedimiento.setUsuarioModificacion(repositorio.getUsuario());
        intervinienteDb.get().setProcedimientoId(procedimientoId);
        repositorio.save(procedimiento);
        cliente.actualizacion(intervinienteDb.get());
        return Optional.of(intervinienteDb.get());
    }
    return Optional.empty();
}
```

La URL es la siguiente:

<http://localhost:8001/asignar-int/1?interId=5>

En este ejemplo, al procedimiento 1 le asignamos el interviniente cuya ID es 5. Al ser una relación OneToMany desde el lado de los procedimientos, el procedimiento_id del interviniente cambiará y pasará a ser 1:

The screenshot shows two Postman requests. The first request is a GET to `http://localhost:8002/5` with a raw body of `{}`. The response is a JSON object representing an intervention:

```
{
  "id": 5,
  "nombre": "Maribel Matías",
  "tipoIntervencion": "Tipo 5",
  "fechaCreacion": "2024-01-01",
  "usuarioCreacion": "JUANSÁ",
  "fechaModificacion": null,
  "usuarioModificacion": null,
  "procedimientoId": 4
}
```

The second request is a PUT to `http://localhost:8001/asignar-int/1?interId=5` with a raw body of `{}`. The response is a JSON object representing the same intervention, but with `procedimientoId` set to 1:

```
{
  "id": 5,
  "nombre": "Maribel Matías",
  "tipoIntervencion": "Tipo 5",
  "procedimientoId": 1
}
```

También podemos comprobar que se han insertado los valores de fecha de modificación y de usuario de modificación tanto en el procedimiento como en el interviniente:



```
1  {
2      "id": 5,
3      "nombre": "Maribel Matías",
4      "tipoIntervencion": "Tipo 5",
5      "fechaCreacion": "2024-01-01",
6      "usuarioCreacion": "JUANSÁ",
7      "fechaModificacion": "2024-04-16",
8      "usuarioModificacion": "JUANSÁ",
9      "procedimientoId": 1
10 }

{id": 1,
"numeroProcedimiento": "PR0001",
"anno": 2020,
"fechaCreacion": "2024-01-01",
"usuarioCreacion": "JUANSÁ",
"fechaModificacion": "2024-04-16",
"usuarioModificacion": "JUANSÁ",
"intervinientes": [
```

El siguiente método es uno que nos va a permitir crear un nuevo interviniente y asignarlo directamente a un procedimiento.

El código de este método, en el servicio de Procedimiento, es este:

```
@Override 1 usage Juan Salvador Fructuoso Campoy
public Optional<Interviniente> crearInterviniente(Interviniente interviniente, Long procedimientoId) {
    Optional<Procedimiento> opt = repositorio.findById(procedimientoId);
    if(opt.isPresent()) {
        Procedimiento procedimiento = opt.get();
        Interviniente interNuevo = cliente.crear(interviniente);
        interNuevo.setProcedimientoId(procedimientoId);
        procedimiento.getIntervinientes().add(interNuevo);
        procedimiento.setFechaModificacion(LocalDate.now());
        procedimiento.setUsuarioModificacion(repositorio.getUsuario());
        repositorio.save(procedimiento);
        cliente.crear(interNuevo);
        return Optional.of(interNuevo);
    }
    return Optional.empty();
}
```

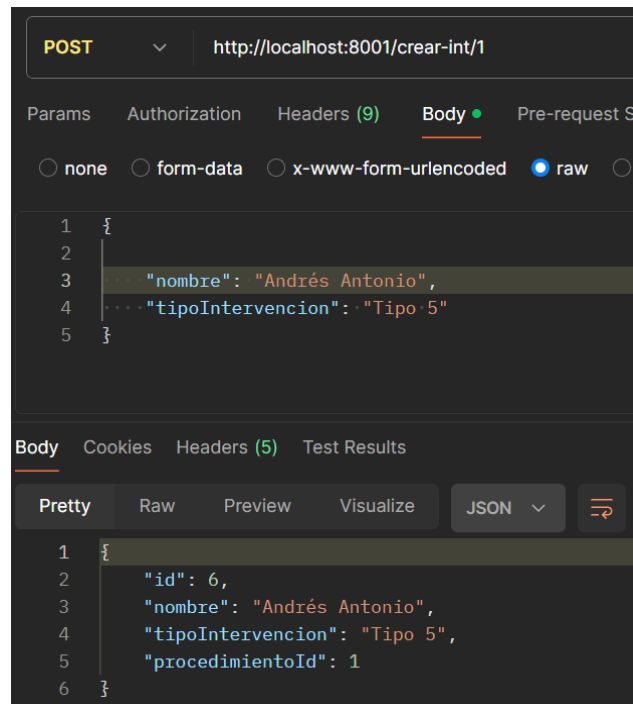
Este código es muy similar al anterior, pero con la diferencia de que, en lugar de buscar un interviniente por su Id, lo introducimos como parámetro y lo creamos usando el controlador del interviniente a través del cliente.

La URL a introducir en Postman es la siguiente:

<http://localhost:8001/crear-int/1>

Al introducir como cuerpo de la solicitud un JSON con la estructura de un interviniente, éste se crea en la tabla de intervinientes y se añade directamente al procedimiento:





Finalmente, pasamos al tercer método: el de desvincular a un interviniente de un procedimiento. Su código en el servicio es el siguiente:

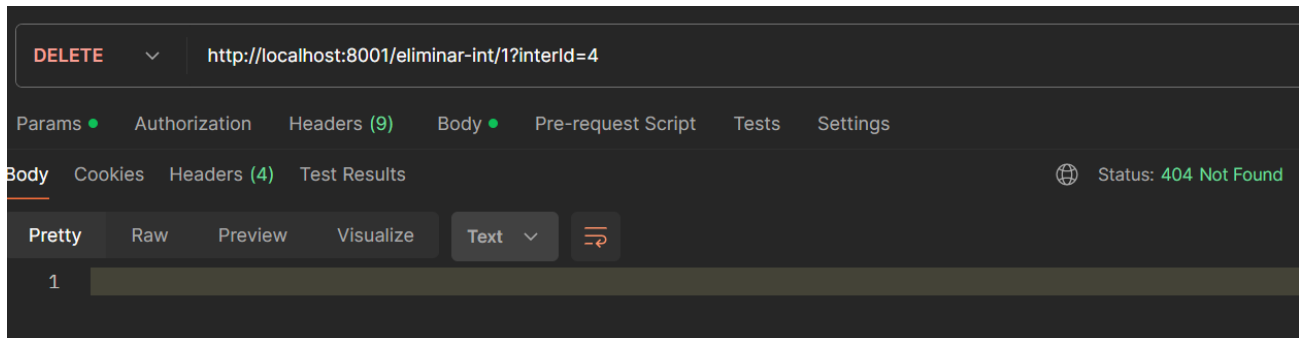
```
@Override 1 usage Juan Salvador Fructuoso Campoy *
public Optional<Interviniente> eliminarInterviniente(Interviniente interviniente, Long procedimientoId) {
    Optional<Procedimiento> opt = repositorio.findById(procedimientoId);
    if(opt.isPresent()) {
        Optional<Interviniente> intervinienteDb = cliente.findById(interviniente.getId());
        if(intervinienteDb.isEmpty()) {
            return Optional.empty();
        }
        if(!Objects.equals(interviniente.getProcedimientoId(), procedimientoId)){
            return Optional.empty();
        }
        Procedimiento procedimiento = opt.get();
        procedimiento.getIntervinientes().remove(intervinienteDb.get());
        procedimiento.setFechaModificacion(LocalDate.now());
        procedimiento.setUsuarioModificacion(repositorio.getUsuario());
        intervinienteDb.get().setProcedimientoId(null);
        repositorio.save(procedimiento);
        cliente.actualizacion(intervinienteDb.get());
        return Optional.of(intervinienteDb.get());
    }
    return Optional.empty();
}
```

En este código, establecemos como null el parámetro `procedimientoId` de ese interviniente y salvamos los cambios, de tal manera que el interviniente seguirá existiendo, pero no tendrá ningún procedimiento vinculado. La URL para realizar este método es la siguiente:



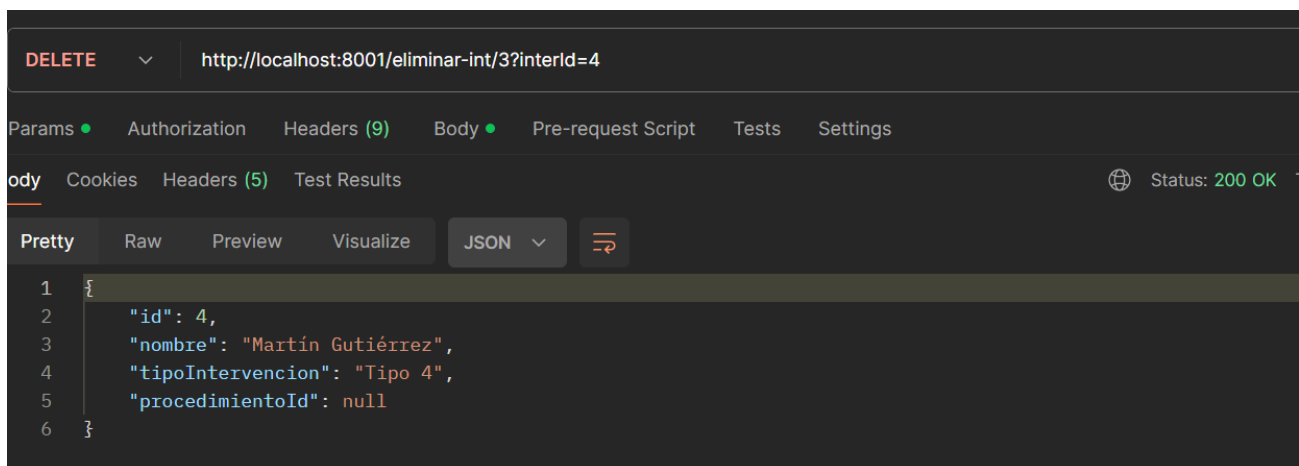
<http://localhost:8001/eliminar-int/1?interId=2>

El interId es el ID del interviniente que queremos desvincular del procedimiento, y el número después del “eliminar-int/” es el ID del procedimiento. Si intentamos desvincular un interviniente de un procedimiento al que no pertenece, nos devolverá un 404:



El interviniente 4 está vinculado al procedimiento 3, por lo tanto, al intentar desvincularlo del procedimiento 1 nos dará un 404.

Sin embargo, si seleccionamos correctamente el código de procedimiento y de interviniente, nos saldrá en el cuerpo los datos del interviniente con el código de procedimiento como null, indicando que ya no tiene procedimiento vinculado:



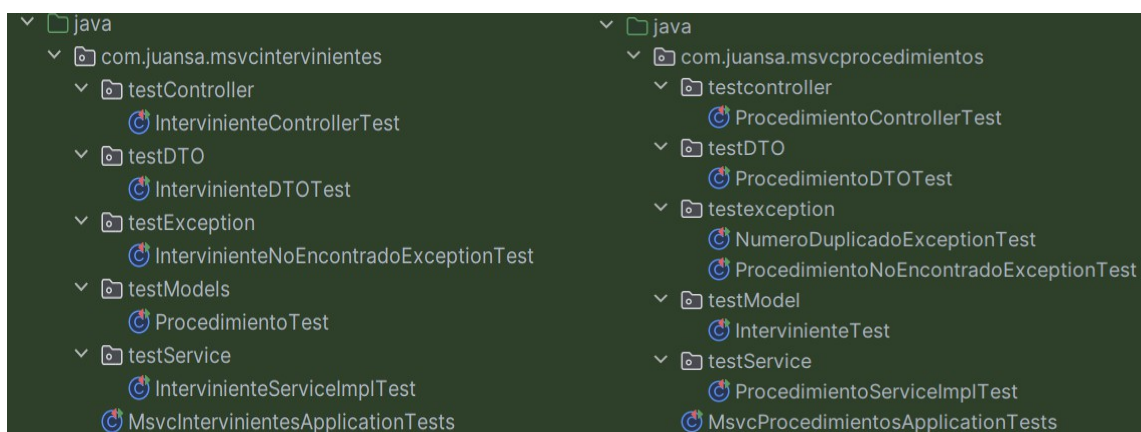
5. Test:

Dentro de cada microservicio (dentro del explorador de archivos y carpetas del proyecto), haciendo click derecho en la carpeta test/java y seleccionando "More Run/Debug" -> "Test in 'java' with Coverage", se ejecutarán los tests de los microservicios con cobertura, que nos va a indicar qué porcentaje de código cubren las pruebas.

Para la realización de las pruebas hemos utilizado Mockito, que es un marco de pruebas de Java para crear objetos simulados (llamados mocks) y objetos de prueba (llamados test doubles).

En los tests, básicamente lo que se ha hecho es usar el método when() para establecer el comportamiento del método que queramos testear, y usando mockMvc.perform hemos simulado una petición HTTP para que se ejecute el método. Si coincide lo esperado con el resultado, entonces la prueba tiene éxito. Si no es así, la prueba falla. Se ha testado también los casos en los que el método principal devuelve un valor vacío o los casos de error de la aplicación para comprobar todas las salidas de los métodos de los microservicios.

Se han creado paquetes y clases de pruebas para testear las distintas clases de los microservicios por separado, para mantener un cierto orden. La estructura de las clases de prueba en el microservicio de intervinientes y procedimientos es la siguiente:



Al ejecutar las pruebas, obtenemos los siguientes resultados, primero en procedimientos:

Element ^	Class, %	Method, %	Line, %	Branch, %
com.juansa.msvcprocedimientos	88% (8/9)	98% (53/54)	99% (173/174)	32% (56/170)
clients	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IntervinienteClientRest	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
configuration	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
ModelMapperConfig	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
controller	100% (1/1)	100% (16/16)	100% (72/72)	100% (22/22)
ProcedimientoController	100% (1/1)	100% (16/16)	100% (72/72)	100% (22/22)
dto	100% (1/1)	100% (5/5)	100% (5/5)	0% (0/24)
ProcedimientoDTO	100% (1/1)	100% (5/5)	100% (5/5)	0% (0/24)
exception	100% (2/2)	100% (2/2)	100% (2/2)	100% (0/0)
NumeroDuplicadoExcep	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
ProcedimientoNoEncont	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
models	100% (2/2)	100% (17/17)	100% (18/18)	12% (13/102)
entity	100% (1/1)	100% (10/10)	100% (11/11)	1% (1/64)
Procedimiento	100% (1/1)	100% (10/10)	100% (11/11)	1% (1/64)
Interviniente	100% (1/1)	100% (7/7)	100% (7/7)	31% (12/38)
repositories	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
ProcedimientoRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
services	100% (1/1)	100% (12/12)	100% (75/75)	95% (21/22)
ProcedimientoService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
ProcedimientoServiceImpl	100% (1/1)	100% (12/12)	100% (75/75)	95% (21/22)
MsvcProcedimientosApplic	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)

Y en intervinientes:

Element ^	Class, %	Method, %	Line, %	Branch, %
com.juansa.msvcintervinientes	87% (7/8)	97% (46/47)	98% (90/91)	10% (15/146)
configuration	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
ModelMapperConfig	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
controller	100% (1/1)	100% (15/15)	100% (45/45)	100% (12/12)
IntervinienteController	100% (1/1)	100% (15/15)	100% (45/45)	100% (12/12)
dto	100% (1/1)	100% (6/6)	100% (6/6)	0% (0/38)
IntervinienteDTO	100% (1/1)	100% (6/6)	100% (6/6)	0% (0/38)
exception	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
IntervinienteNoEncontra	100% (1/1)	100% (1/1)	100% (1/1)	100% (0/0)
models	100% (2/2)	100% (16/16)	100% (16/16)	1% (1/94)
entity	100% (1/1)	100% (10/10)	100% (10/10)	1% (1/70)
Interviniente	100% (1/1)	100% (10/10)	100% (10/10)	1% (1/70)
Procedimiento	100% (1/1)	100% (6/6)	100% (6/6)	0% (0/24)
repositories	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IntervinienteRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
services	100% (1/1)	100% (7/7)	100% (21/21)	100% (2/2)
IntervinienteService	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
IntervinienteServiceImpl	100% (1/1)	100% (7/7)	100% (21/21)	100% (2/2)
MsvcIntervinientesApplicat	0% (0/1)	0% (0/1)	0% (0/1)	100% (0/0)

Como podemos comprobar, tanto en procedimientos como en intervinientes, el porcentaje de cobertura es superior al 80%, cumpliendo el requisito del proyecto.

