

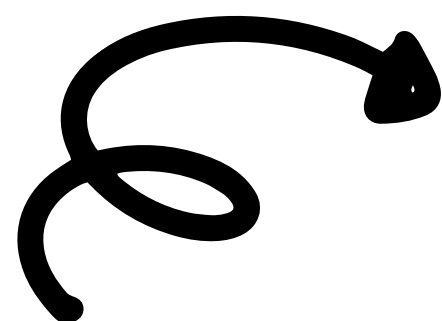
CASO: PERFULANDIA



IMPLEMENTACIÓN DE MICROSERVICIOS

PERFUSMART

Grupo 1: Gonzalo Navarrete - Carla Prado - Fernando Zárate



INTRODUCCIÓN

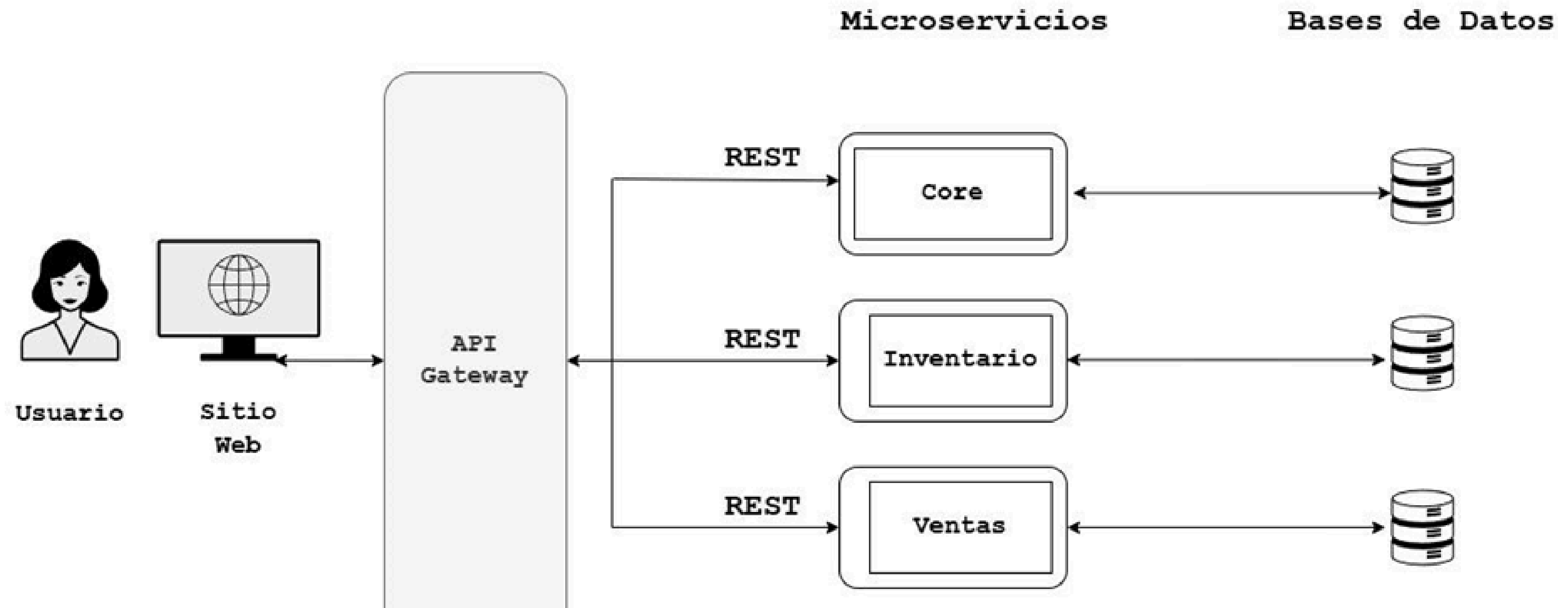
- Proyecto backend [Java + Framework Spring Boot con Maven](#).
- Enfoque [RESTful](#) (Comunicación por HTTP) y arquitectura tipo [microservicios](#).
- Motor Base de datos: [MySQL](#)
- API Testing: [Postman](#)
- Control de versiones: [Git](#), [GitHub](#)



ARQUITECTURA DE MICROSERVICIOS

El siguiente diagrama se diseñó pensando en el estándar en una arquitectura de microservicios. Siguiendo el principio de modularidad, cada microservicio tiene su propia base de datos y pueden ser desarrollados por separado, sin afectar a los otros.

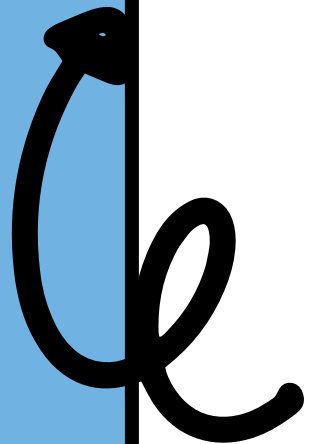
ARQUITECTURA DE MICROSERVICIOS





ESTRUCTURA DEL PROYECTO

DIAGRAMA DE CLASES



El trabajo de esta unidad se enfocó en la fase de testing de las APIs REST, verificando la correcta comunicación HTTP entre las entidades Spring Boot y la base de datos con sus endpoints asignados. En esta instancia sólo se trabajó con una base de datos, en la que se simula cada microservicio como una entidad y una tabla en dicha base.

DIAGRAMA DE CLASES

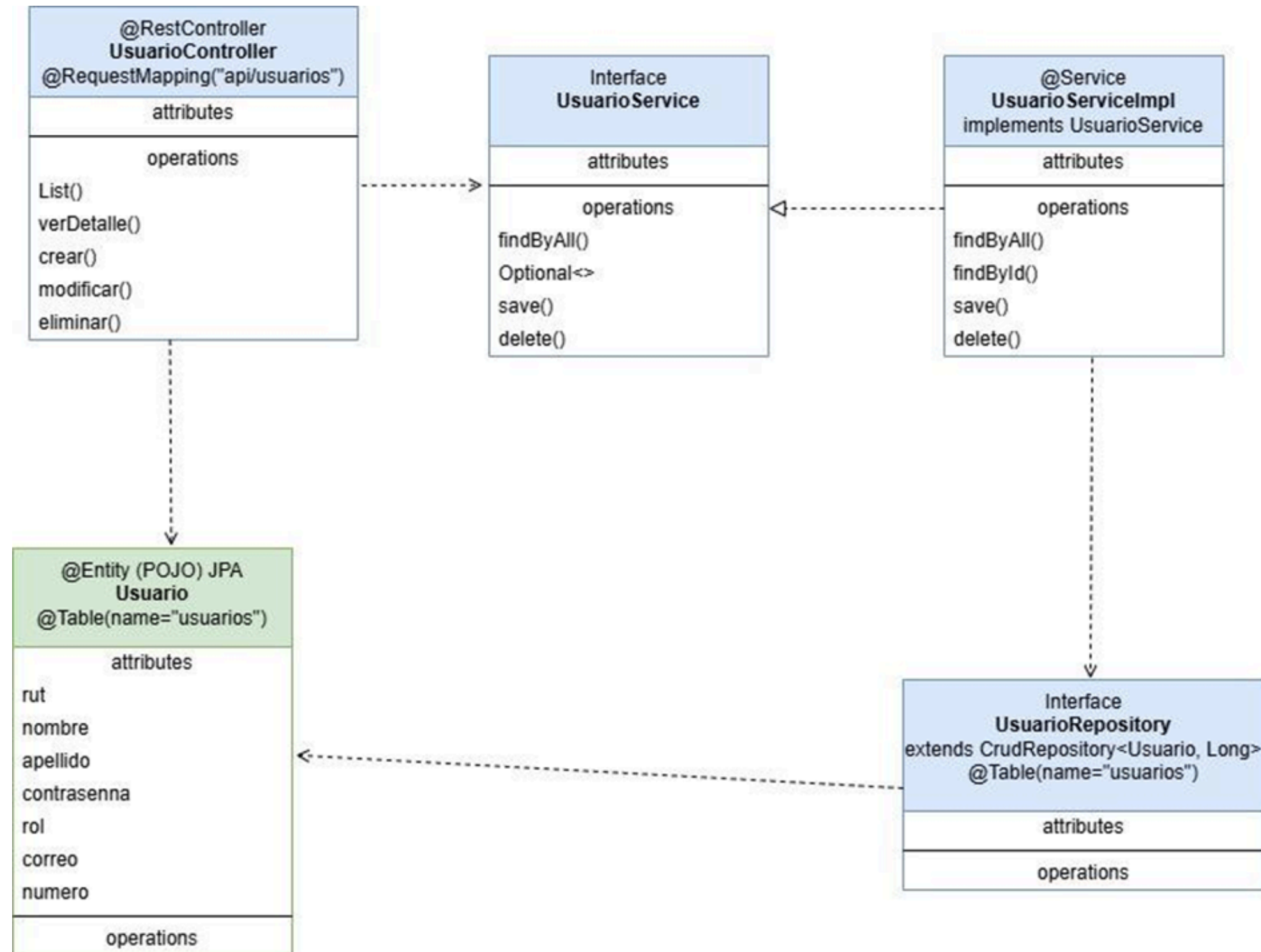
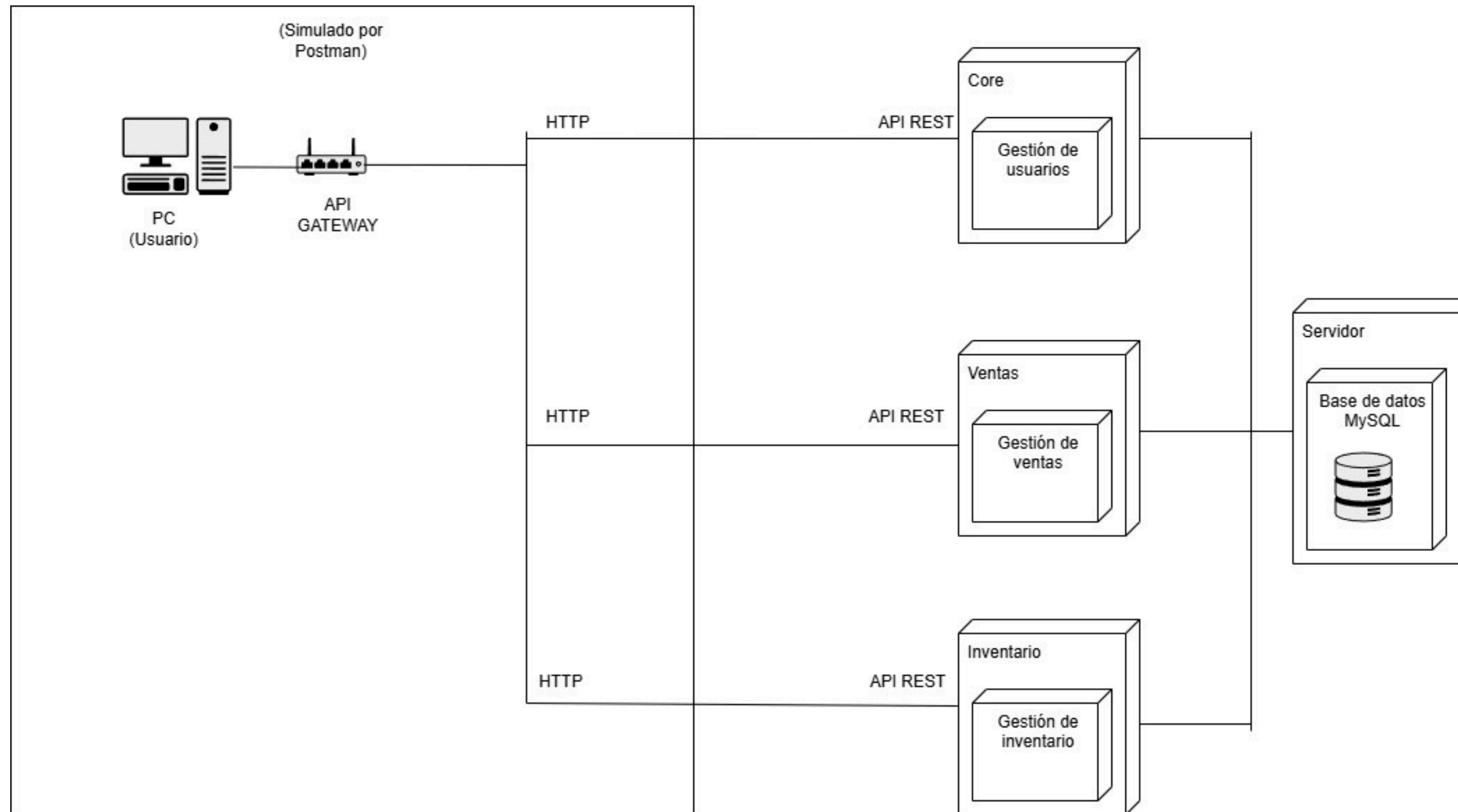


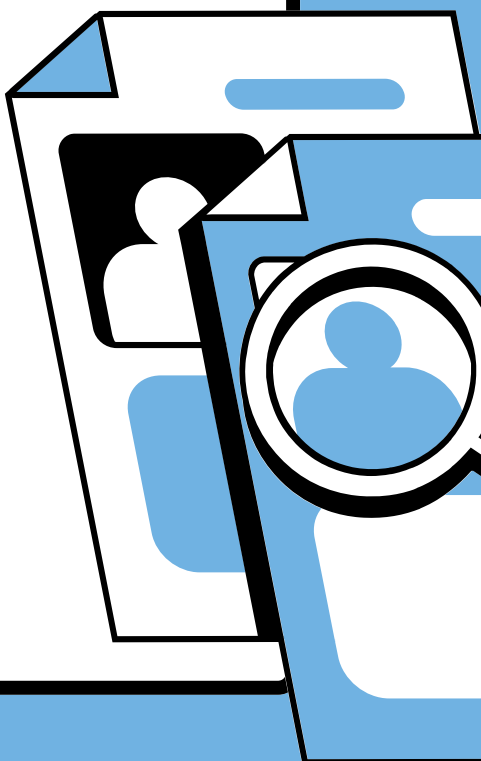
DIAGRAMA DE DESPLIEGUE



SPRING BOOT

Spring Boot es un framework Java que facilita la creación rápida de aplicaciones web y backend, reduciendo la configuración manual.

- **Dependencias:** Bibliotecas externas necesarias para que la aplicación funcione, gestionadas con herramientas como Maven.
- **Componentes:** Partes de la aplicación (como controladores o servicios) que Spring Boot detecta y gestiona automáticamente mediante anotaciones como `@Component`, `@Service` o `@Controller`.
- **Pom.xml:** es el archivo de configuración principal de Maven, una herramienta de gestión de proyectos en Java.



POM.XML DE MAVEN

Se ubica en el directorio principal. Maneja tags.

Gestiona:

- Dependencias <dependency>
- Configuraciones parent (configuraciones heredadas desde otro proyecto o POM base) <parent>
- Versión de Java <properties> <java.version>

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.4.5</version> Upgrade to the Latest Patch
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```



DEPENDENCIAS MAVEN

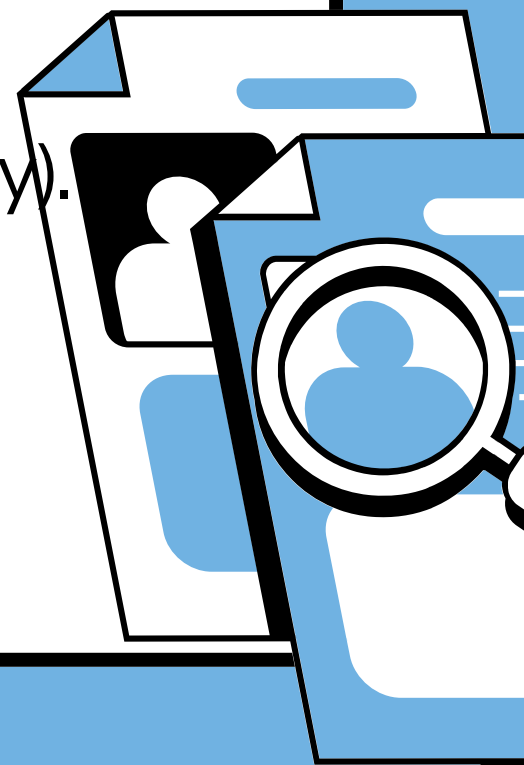
1. [Spring Web](#): Permite crear servicios REST y manejar peticiones HTTP, facilitando el desarrollo de APIs web
2. [Spring Data JPA](#): Facilita el acceso a bases de datos usando JPA y Hibernate, permitiendo realizar operaciones CRUD con poco código y de manera eficiente.
3. [Springboot Dev Tools](#): Mejora la experiencia de desarrollo recargando automáticamente la aplicación al detectar cambios en el código.
4. [MySqlDriver](#): Es el conector necesario para establecer la conexión entre la aplicación y una base de datos MySQL.



COMPONENTES SPRING BOOT

Anotaciones de componente o anotaciones de propósito

1. **@SpringBootApplication**: Es la anotación principal de arranque.
2. **@RestController**: Indica que la clase se encarga de recibir peticiones HTTP del cliente (como GET, POST, PUT, DELETE) y devolver respuestas en formato JSON. Es la forma de crear APIs web.
3. **@Repository**: Marca una clase que accede a la base de datos, y se encarga de operaciones como guardar, buscar, actualizar o eliminar datos (CRUD).
4. **@Service**: Define una clase que contiene la lógica de negocio. Actúa como intermediario entre el controlador (@RestController) y el repositorio (@Repository).
5. **@Entity**: Indica que la clase representa una tabla en la base de datos. Cada atributo de la clase corresponde a una columna.



COMPONENTES SPRING BOOT

Anotaciones y funciones por capa.

- 1.-@RequestMapping: define la ruta base que un controlador o método debe manejar.
- 2.-@GetMapping: se utiliza para obtener datos del servidor sin hacer ninguna modificación.
- 3.-@PostMapping: se utiliza para ingresar datos al servidor.
- 4.-@PutMapping: Se utiliza para modificar datos del servidor.
- 5.-@DeleteMapping: Se utiliza para eliminar datos del servidor.
- 6.-@Autowired: se utiliza para inyectar automáticamente dependencias en los componentes de la aplicación.
- 7.-@Table: se utiliza para especificar el nombre de la tabla de base de datos a la que estará vinculada la entidad.



COMPONENTES SPRING BOOT

8.-@Id: define el atributo como identificador primario.

9.-@GeneratedValue: indica que el atributo tendrá un valor generado automáticamente.

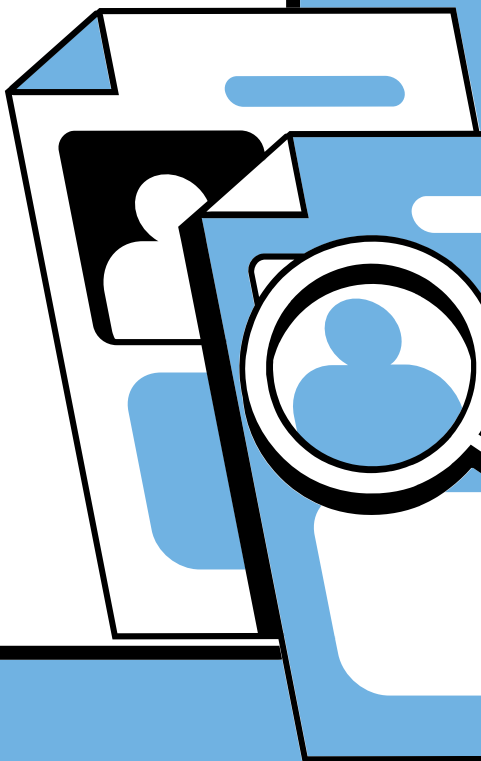
10.-@Column:indica el nombre que tendrá el atributo en la tabla de la base de datos.

11.-@Override:se utiliza para indicar que se está sobreescribiendo un método definido previamente en otra clase o interfaz

12.-@Transactional: gestiona las transacciones con la base de datos, si todo sale bien genera un commit, si algo falla, hace un rollback para mantener la integridad de los datos.

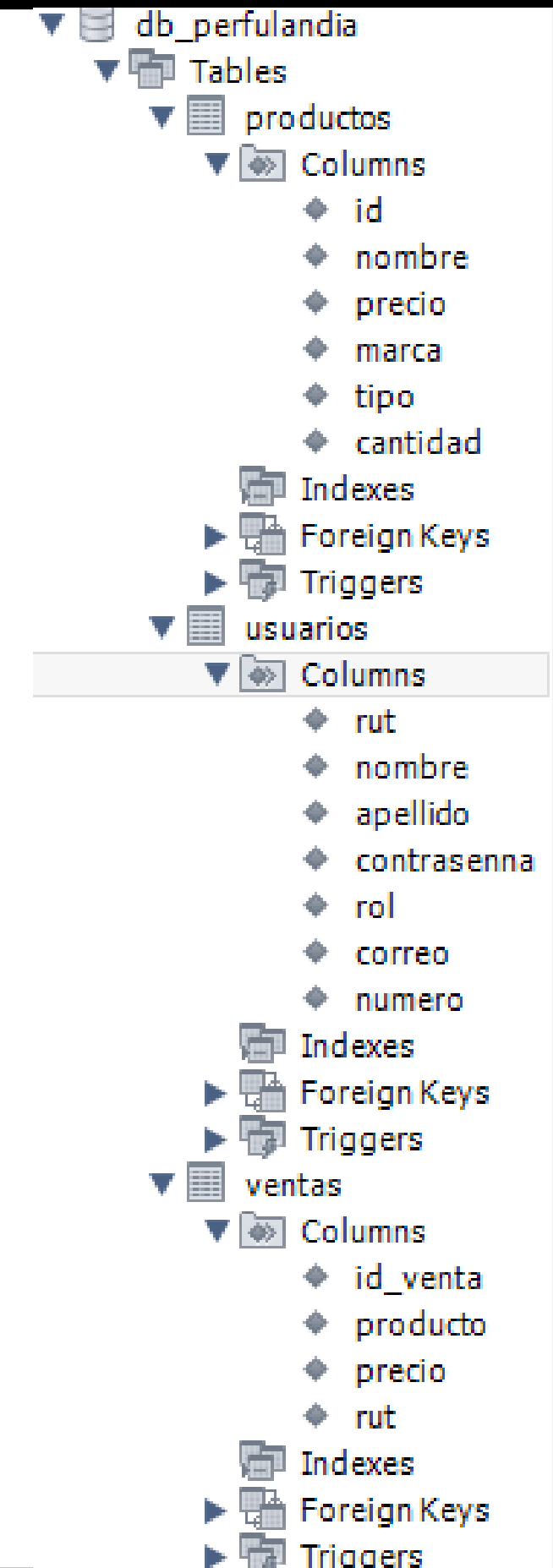
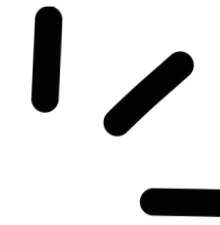
13.-@PathVariable: en Spring Boot sirve para extraer un valor de la URL y pasarlo como argumento al método del controlador.

14.-@RequestBody: recibir datos en formato JSON desde el cuerpo de una solicitud HTTP (por postman en este caso)y convertirlos automáticamente en un objeto Java.

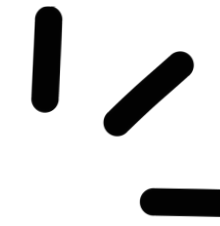


BASE DE DATOS

- Motor: MySQL (via Workbench).
- Entidades → Tablas:
 - Usuario
 - Producto
 - Venta



BASE DE DATOS



Configuración en application.properties.

application.properties ✕

springboot_crud > src > main > resources > application.properties

```
1  spring.application.name=springboot_crud
2
3  spring.datasource.url=jdbc:mysql://localhost:3306/db_perfulandia?serverTimezone=UTC&useSSL=false
4  spring.datasource.username=root
5  spring.datasource.password=
6
7  spring.jpa.hibernate.ddl-auto=update
8  spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
9
10 spring.jpa.show-sql=true
11 spring.jpa.properties.hibernate.format_sql=true
```

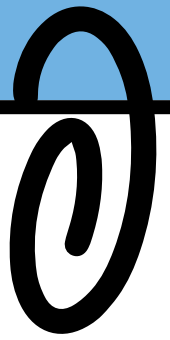


IMPLEMENTACIÓN DE LOS SERVICIOS

Implementar servicios en Spring Boot es crear rutas (endpoints) que responden a peticiones HTTP.

Postman se usa para probar esos servicios enviando solicitudes.

SERVICIOS CRUD CON POST MAN



Ejemplo: Servicio Producto

- GET /productos – Listar productos
- GET /productos/{id} – Buscar producto
- POST /productos – Crear
- PUT /productos/{id} – Actualizar
- DELETE /productos/{id} – Eliminar

MICROSERVICIO: VENTAS-ENTIDAD

```
11  @Entity
12  @Table(name="ventas")
13  public class Venta {
14      @Id
15      @GeneratedValue(strategy = GenerationType.IDENTITY)
16      @Column(name="id_venta")
17      private Long idVenta;
18
19      private String producto;
20      private int precio;
21      private String rut;
```

MICROSERVICIO: VENTAS-REPOSITORIO

```
1 package com.carla.springboot.crud.springboot_crud.repository;
2
3 import org.springframework.data.repository.CrudRepository;
4
5 import com.carla.springboot.crud.springboot_crud.entities.Venta;
6
7 public interface VentaRepository extends CrudRepository <Venta,Long> {
8
9 }
10
```

MICROSERVICIO: VENTAS-SERVICIOS

```
package com.carla.springboot.crud.springboot_crud.services;

import java.util.List;
import java.util.Optional;

import com.carla.springboot.crud.springboot_crud.entities.Venta;

public interface VentaService {

    List<Venta> findByAll();

    Optional<Venta> findById(Long idVenta); //optional validador e

    Venta save(Venta unVenta);

    Optional<Venta> delete(Venta unVenta);

}
```

```
@Service
public class VentaServiceImpl implements VentaService{

    @Autowired
    private VentaRepository repository;

    @Override
    @Transactional(readOnly= true) //es read only porque es solo para verlo c:
    public List<Venta> findByAll() {
        return (List<Venta>) repository.findAll();
    }

    @Override
    @Transactional(readOnly= true)
    public Optional<Venta> findById(Long id) {
        return repository.findById(id);
    }

    @Override
    @Transactional
    public Venta save(Venta unVenta) {
        return repository.save(unVenta);
    }

    @Override
    @Transactional
    public Optional<Venta> delete(Venta unVenta) {
        Optional<Venta> ventaOptional= repository.findById(unVenta.getIdVenta());
        ventaOptional.ifPresent(ventaDb->{
            repository.delete(unVenta);
        });
        return ventaOptional;
    }
}
```

MICROSERVICIO VENTAS- CONTROLADOR

```
20 @RestController
21 @RequestMapping("api/ventas")
22 public class VentaController {
23     @Autowired
24     private VentaService service;
25
26     @GetMapping
27     public List<Venta> List(){
28         return service.findAll();
29     }
30     @GetMapping("/{idVenta}")
31     public ResponseEntity<?> verDetalle(@PathVariable Long idVenta){
32         Optional <Venta> ventaOptional=service.findById(idVenta);
33         if (ventaOptional.isPresent()){
34             return ResponseEntity.ok(ventaOptional.orElseThrow());
35         }
36         return ResponseEntity.notFound().build();
37     }
38
39     @PostMapping
40     public ResponseEntity<Venta> crear (@RequestBody Venta unVenta){
41         return ResponseEntity.status(HttpStatus.CREATED).body(service.save(unVenta));
42     }
43     @PutMapping("/{idVenta}")
44     public ResponseEntity<?> modificar(@PathVariable Long idVenta, @RequestBody Venta unVenta){
45         Optional<Venta> ventaOptional=service.findById(idVenta);
46         if (ventaOptional.isPresent()){
47             Venta ventaexistente=ventaOptional.get();
48             ventaexistente.setProducto(unVenta.getProducto());
49             ventaexistente.setPrecio(unVenta.getPrecio());
50             ventaexistente.setRut(unVenta.getRut());
51             Venta ventamodificado= service.save(ventaexistente);
52             return ResponseEntity.ok(ventamodificado);
53         }
54         return ResponseEntity.notFound().build();
55     }
56     @DeleteMapping("/{idVenta}")
57     public ResponseEntity<?> eliminar(@PathVariable Long idVenta){
58         Venta unVenta=new Venta();
59         unVenta.setidVenta(idVenta);
60         Optional<Venta> ventaOptional=service.delete(unVenta);
61         if(ventaOptional.isPresent()){
62             return ResponseEntity.ok(ventaOptional.orElseThrow());
63         }
64         return ResponseEntity.notFound().build();
65     }
}
```

EJECUCIÓN POSTMAN: VENTAS

Get

localhost:8080/api/ventas

GET localhost:8080/api/ventas

Params Authorization Headers (8) Body ● Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "idVenta": 1,
4     "producto": "Lancôme - La Vie Est Belle Eau de Parfum 100 ml",
5     "precio": 72900,
6     "rut": "21861087-0"
7   }
8 ]
```

Status: 200 OK Time: 5 ms Size: 277 B Save Response

Get(id)

localhost:8080/api/ventas/1

GET localhost:8080/api/ventas/1

Params Authorization Headers (8) Body ● Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary JSON

1

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "idVenta": 1,
3   "producto": "Lancôme - La Vie Est Belle Eau de Parfum 100 ml",
4   "precio": 72900,
5   "rut": "21861087-0"
6 }
```

Status: 200 OK Time: 6 ms Size: 275 B Save Response

EJECUCIÓN POSTMAN: VENTAS

Post

The screenshot displays the Postman interface for a POST request. At the top, the URL bar shows 'localhost:8080/api/ventas' with a 'Save' icon to its right. Below the URL bar, the 'POST' method is selected, and the same URL is repeated. A 'Send' button is located to the right of the URL bar. The interface includes tabs for 'Params', 'Authorization', 'Headers (8)', 'Body', 'Pre-request Script', 'Tests', and 'Settings'. The 'Body' tab is active, showing a JSON body with the following content:


```
1 {  
2   "producto": "Giorgio Armani - Acqua di Giò Profondo Eau de Parfum 75 ml ",  
3   "precio": 89990,  
4   "rut": "19233201-K"  
5 }
```


Below the body tab, there are radio buttons for 'none', 'form-data', 'x-www-form-urlencoded', 'raw', and 'binary', with 'JSON' selected. To the right of the body tab, there are links for 'Cookies' and 'Beautify'. The response section at the bottom shows a status of '201 Created', a time of '11 ms', and a size of '292 B'. It includes a 'Save Response' button and tabs for 'Body', 'Cookies', 'Headers (5)', and 'Test Results'. The 'Body' tab is active, showing the response JSON:

```
1 {  
2   "idVenta": 2,  
3   "producto": "Giorgio Armani - Acqua di Giò Profondo Eau de Parfum 75 ml ",  
4   "precio": 89990,  
5   "rut": "19233201-K"  
6 }
```

Ejecución Postman: Ventas

Put

 localhost:8080/api/ventas/2

 Save

PUT

localhost:8080/api/ventas/2

Send

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

● none

● form-data

● x-www-form-urlencoded

● raw

● binary

JSON

Beautify

1

2

3

4

5

1

2

3

4

5

```
{
  "producto": "Giorgio Armani - Acqua di Giò Profondo Eau de Parfum 75 ml ",
  "precio": 79990,
  "rut": "19233201-K"
}
```

Body

Cookies

Headers (5)

Test Results

⌐

Status: 200 OK

Time: 11 ms

Size: 287 B

Save Response

Pretty

Raw

Preview

Visualize

JSON

≡

🔍

1

2

3

4

5

6

1

2

3

4

5

6

```
{
  "idVenta": 2,
  "producto": "Giorgio Armani - Acqua di Giò Profondo Eau de Parfum 75 ml ",
  "precio": 79990,
  "rut": "19233201-K"
}
```


EJECUCIÓN POSTMAN: VENTAS

Delete

The screenshot shows the Postman interface for a DELETE request. The URL bar shows `localhost:8080/api/ventas/2`. The method is set to `DELETE`. The request body is empty. The response is a JSON object with status `200 OK`, time `12 ms`, and size `215 B`. The response body is displayed in the 'Body' tab, showing a JSON object with the following structure:

```
{  "idVenta": 2,  "producto": null,  "precio": 0,  "rut": null}
```

The interface includes tabs for Params, Authorization, Headers (8), Body, Pre-request Script, Tests, and Settings. The 'Body' tab is selected, and the 'JSON' format is chosen. The response status is `200 OK`, and the response time is `12 ms`. The response size is `215 B`. The response body is displayed in the 'Body' tab, showing a JSON object with the following structure:

```
{  "idVenta": 2,  "producto": null,  "precio": 0,  "rut": null}
```

IMPLEMENTACIÓN: PRODUCTOS

Producto:

@Entity

@Table(name="productos")

@Id

@GeneratedValue

ProductoRepository:

(Interface) extends

CrudRepository<Producto, Long>

ProductoService:

(Define métodos CRUD)

ProductoServiceImpl:

implements ProductoService

@Service

@Autowired

@Override

@Transactional

EJECUCIÓN POSTMAN: PRODUCTOS

Get

localhost:8080/api/productos

GET localhost:8080/api/productos

Params Authorization Headers (8) Body • Pre-request Script Tests Settings Cookies

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "id": 1,
4     "nombre": "Numero 5",
5     "precio": 5000,
6     "marca": "Chanel",
7     "tipo": "De mujer",
8     "cantidad": 500
9   }
10 ]
```

Status: 200 OK Time: 9 ms Size: 258 B Save Response

Get(id)

localhost:8080/api/productos/1

GET localhost:8080/api/productos/1

Params Authorization Headers (8) Body • Pre-request Script Tests Settings Cookies

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "id": 1,
3   "nombre": "Numero 5",
4   "precio": 5000,
5   "marca": "Chanel",
6   "tipo": "De mujer",
7   "cantidad": 500
8 }
```

Status: 200 OK Time: 9 ms Size: 256 B Save Response

EJECUCIÓN POSTMAN: PRODUCTOS

Post

The screenshot displays the Postman application interface. At the top, the URL bar shows 'localhost:8080/api/productos' with a 'Save' button. Below this, the 'POST' method is selected, and the same URL is entered in the request bar, accompanied by a 'Send' button. The 'Body' tab is active, showing a JSON payload:

```
{  "nombre": "Acqua di Giò Profondo Eau de Parfum 75 ml",  "precio": 74990,  "marca": "Giorgio Armani",  "tipo": "De Hombre",  "cantidad": 600}
```

. The bottom section shows the response, with a status of '201 Created', a time of '29 ms', and a size of '305 B'. The response body is displayed in 'Pretty' format:

```
{  "id": 2,  "nombre": "Acqua di Giò Profondo Eau de Parfum 75 ml",  "precio": 74990,  "marca": "Giorgio Armani",  "tipo": "De Hombre",  "cantidad": 600}
```

localhost:8080/api/productos Save

POST ▼ localhost:8080/api/productos Send ▼

Params Authorization Headers (8) Body ● Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● raw ● binary JSON ▼

1
2 "nombre": "Acqua di Giò Profondo Eau de Parfum 75 ml",
3 "precio": 74990,
4 "marca": "Giorgio Armani",
5 "tipo": "De Hombre",
6 "cantidad": 600
7
8

Body Cookies Headers (5) Test Results 🌐 Status: 201 Created Time: 29 ms Size: 305 B Save Response ▼

Pretty Raw Preview Visualize JSON ▼ 🔍

1
2 "id": 2,
3 "nombre": "Acqua di Giò Profondo Eau de Parfum 75 ml",
4 "precio": 74990,
5 "marca": "Giorgio Armani",
6 "tipo": "De Hombre",
7 "cantidad": 600
8

EJECUCIÓN POSTMAN: PRODUCTOS

Put

The screenshot displays the Postman interface for a PUT request. The URL bar shows 'localhost:8080/api/productos/2'. The request method is 'PUT'. The 'Body' tab is selected, showing a JSON payload:

```
{  "nombre": "Acqua di Giò Profondo Eau de Parfum 75 ml",  "precio": 70000,  "marca": "Giorgio Armani",  "tipo": "De Hombre",  "cantidad": 250}
```

. The response status is '200 OK' with a time of '15 ms' and size of '300 B'. The response body is shown in the 'Pretty' format:

```
{  "id": 2,  "nombre": "Acqua di Giò Profondo Eau de Parfum 75 ml",  "precio": 70000,  "marca": "Giorgio Armani",  "tipo": "De Hombre",  "cantidad": 250}
```

localhost:8080/api/productos/2

PUT localhost:8080/api/productos/2

Params Authorization Headers (8) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary JSON

```
1  {
2    "nombre": "Acqua di Giò Profondo Eau de Parfum 75 ml",
3    "precio": 70000,
4    "marca": "Giorgio Armani",
5    "tipo": "De Hombre",
6    "cantidad": 250
7  }
```

Body Cookies Headers (5) Test Results

Status: 200 OK Time: 15 ms Size: 300 B Save Response

Pretty Raw Preview Visualize JSON

```
1  {
2    "id": 2,
3    "nombre": "Acqua di Giò Profondo Eau de Parfum 75 ml",
4    "precio": 70000,
5    "marca": "Giorgio Armani",
6    "tipo": "De Hombre",
7    "cantidad": 250
8  }
```

EJECUCIÓN POSTMAN: PRODUCTOS

Delete

The screenshot displays the Postman interface for a DELETE request. The URL bar shows `localhost:8080/api/productos/2`. The request method is set to **DELETE**. The request body is a JSON object with the following fields: `"nombre": "Acqua di Giò Profondo Eau de Parfum 75 ml", "precio": 70000, "marca": "Giorgio Armani", "tipo": "De Hombre", "cantidad": 250`. The response status is **200 OK** with a time of 12 ms and a size of 235 B. The response body is a JSON object: `"id": 2, "nombre": null, "precio": 0, "marca": null, "tipo": null, "cantidad": 0`.

HTTP `localhost:8080/api/productos/2` Save

DELETE `localhost:8080/api/productos/2` Send

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary **JSON** Beautify

```
1  {
2    "nombre": "Acqua di Giò Profondo Eau de Parfum 75 ml",
3    "precio": 70000,
4    "marca": "Giorgio Armani",
5    "tipo": "De Hombre",
6    "cantidad": 250
7  }
```

Body Cookies Headers (5) Test Results Status: 200 OK Time: 12 ms Size: 235 B Save Response

Pretty Raw Preview Visualize **JSON** Copy

```
1  {
2    "id": 2,
3    "nombre": null,
4    "precio": 0,
5    "marca": null,
6    "tipo": null,
7    "cantidad": 0
8  }
```

IMPLEMENTACIÓN: USUARIO

Usuario:

@Entity

@Table(name="usuarios")

@Id

UsuarioRepository:

(Interface) extends

CrudRepository<Usuario, Long>

UsuarioService:

(Define métodos CRUD)

UsuarioServiceImpl:

implements UsuarioService

@Service

@Autowired

@Override

@Transactional

EJECUCIÓN POSTMAN: USUARIO

Get

GET localhost:8080/api/usuarios Send

Params Authorization Headers (8) Body • Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Bulk Edit
Key	Value	

Body Cookies Headers (5) Test Results 200 OK 18 ms 337 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   {
3     "rut": 218610870,
4     "nombre": "Gonzalo",
5     "apellido": "Navarrete",
6     "contrasenna": "guajajaxd",
7     "rol": "Administrador",
8     "correo": "gonzalonavarreteb@gmail.com",
9     "numero": "+56 983022268"
10  }
```

Get(id)

GET localhost:8080/api/usuarios/218610870 Send

Params Authorization Headers (8) Body • Pre-request Script Tests Settings Cookies

Query Params

Key	Value	Bulk Edit
Key	Value	

Body Cookies Headers (5) Test Results 200 OK 32 ms 335 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "rut": 218610870,
3   "nombre": "Gonzalo",
4   "apellido": "Navarrete",
5   "contrasenna": "guajajaxd",
6   "rol": "Administrador",
7   "correo": "gonzalonavarreteb@gmail.com",
8   "numero": "+56 983022268"
9 }
```


EJECUCIÓN POSTMAN: USUARIO

Post

The screenshot displays the Postman application interface for a POST request. At the top, the URL bar shows 'localhost:8080/api/usuarios'. Below the URL bar, the 'POST' method is selected, and the 'Body' tab is active. The body is formatted as JSON and contains the following data:

```
{
  "rut": 19233201,
  "nombre": "Carla",
  "apellido": "Prado",
  "contrasenna": "apirestTest",
  "rol": "Empleado de ventas",
  "correo": "carlaprado@gmail.com",
  "numero": "+56 912345678"
}
```

Below the body editor, the 'Body' tab is selected in the response section, showing the same JSON data. The status bar at the bottom indicates a successful response: 'Status: 200 OK', 'Time: 22 ms', and 'Size: 328 B'. A 'Save Response' button is also visible.

EJECUCIÓN POSTMAN: USUARIO

Put

The screenshot displays the Postman interface for a PUT request. The URL bar shows `localhost:8080/api/usuarios/19233201`. The request method is set to **PUT**. The request body is a JSON object with the following fields: `"rut": 19233201`, `"nombre": "Carla"`, `"apellido": "Prado"`, `"contrasenna": "apiRESTTest"`, `"rol": "Empleado"`, `"correo": "carlaprado@duocuc.cl"`, and `"numero": "+56 912345678"`. The response status is **200 OK** with a time of 15 ms and a size of 318 B. The response body is displayed in the 'Pretty' format, showing the same JSON object as the request body.

```
1 {
2   "rut": 19233201,
3   "nombre": "Carla",
4   "apellido": "Prado",
5   "contrasenna": "apiRESTTest",
6   "rol": "Empleado",
7   "correo": "carlaprado@duocuc.cl",
8   "numero": "+56 912345678"
9 }
```

```
1 {
2   "rut": 19233201,
3   "nombre": "Carla",
4   "apellido": "Prado",
5   "contrasenna": "apiRESTTest",
6   "rol": "Empleado",
7   "correo": "carlaprado@duocuc.cl",
8   "numero": "+56 912345678"
9 }
```

EJECUCIÓN POSTMAN: USUARIO

Delete

The screenshot shows the Postman interface for a DELETE request. The URL bar shows `localhost:8080/api/usuarios/19233201`. The method is set to **DELETE**. The request body is empty. The response is displayed in the bottom panel, showing a **200 OK** status with a response time of 31 ms and a size of 268 B. The response body is a JSON object with the following fields:

```
{  "rut": 19233201,  "nombre": null,  "apellido": null,  "contrasenna": null,  "rol": null,  "correo": null,  "numero": null}
```

CONTROL DE VERSIONES CON GIT Y GITHUB

Desde la terminal de git, inicializamos el repositorio:

```
MINGW64/c:/Users/Chalo/Desktop/PerfusmartSpringbootCrud
Chalo@DESKTOP-2106TE3 MINGW64 ~/Desktop/PerfusmartSpringbootCrud
$ git init
Initialized empty Git repository in C:/Users/Chalo/Desktop/PerfusmartSpringbootCrud/.git/
```

Configuramos el repositorio con nuestras credenciales:

```
Chalo@DESKTOP-2106TE3 MINGW64 ~/Desktop/PerfusmartSpringbootCrud (master)
$ git config user.name GonzaloNavarrete
```

```
Chalo@DESKTOP-2106TE3 MINGW64 ~/Desktop/PerfusmartSpringbootCrud (master)
$ git config user.email gon.navarrete@gmail.com
```

Posteriormente, generamos un token en nuestro github, con el scope "repo", para luego utilizarlo como password:

```
Chalo@DESKTOP-2106TE3 MINGW64 ~/Desktop/PerfusmartSpringbootCrud (master)
$ git config user.password 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
```

Vinculamos la carpeta local de nuestro proyecto, con nuestro repositorio remoto en github:

CONTROL DE VERSIONES CON GIT Y GITHUB

```
Chalo@DESKTOP-2106TE3 MINGW64 ~/Desktop/PerfusmartSpringbootCrud (master)
$ git remote add origin https://github.com/Gon7w7r/Perfusmart
```

Luego preparamos los archivos locales de nuestro proyecto para ser subidos al repositorio remoto:

```
Chalo@DESKTOP-2106TE3 MINGW64 ~/Desktop/PerfusmartSpringbootCrud (master)
$ git add .
```

Realizamos control de versión:

```
*****
```

```
Chalo@DESKTOP-2106TE3 MINGW64 ~/Desktop/PerfusmartSpringbootCrud (master)
$ git commit -m "Implementación de servicios REST"
```

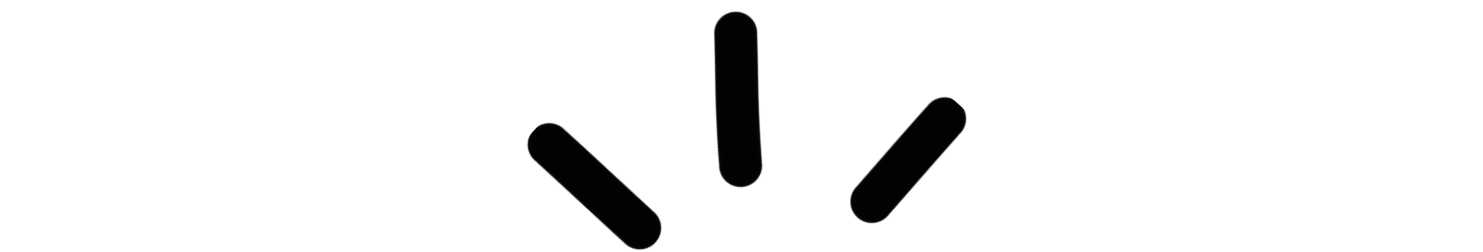
```
*****
```

Subimos los archivos al repositorio:

```
Chalo@DESKTOP-2106TE3 MINGW64 ~/Desktop/PerfusmartSpringbootCrud (master)
$ git push -u origin master
Enumerating objects: 52, done.
Counting objects: 100% (52/52), done.
Delta compression using up to 12 threads
Compressing objects: 100% (39/39), done.
Writing objects: 100% (52/52), 14.72 KiB | 1.23 MiB/s, done.
Total 52 (delta 10), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (10/10), done.
To https://github.com/Gon7w7r/Perfusmart
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.
```

CONCLUSIÓN

- Spring Boot + Maven permiten crear un backend ordenado.
- Simulación de microservicios funcional para CRUD, y el uso de herramientas modernas (Git, Postman, JPA) , presenta una base sólida para entender la lógica del lado servidor en aplicaciones Full Stack.
- Constituye un paso importante para abordar sistemas más complejos en entornos reales



MUCHAS

GRACIAS

www.unsitiogenial.es

