

# Índice

---

## Índice

### Servicios

- ¿Qué es un servicio?
- Creando un servicio
- Añadir funcionalidades al servicio
- Servicios en Spring Boot
- Microservicios

### REST

- ¿Qué es REST?
- REST en Spring Boot
- Clase Artículo
- Lombok
- @ResponseBody
- @RestController
- Práctica 6

### Métodos de petición HTTP

- GET
- POST
- PUT
- DELETE
- PATCH
- Usando los métodos HTTP en Spring Boot
  - @DeleteMapping
  - @PostMapping
  - @PutMapping
  - @PatchMapping

### Conclusión

---

## Servicios

### ¿Qué es un servicio?

---

Los servicios dentro de una aplicación son clases normales, que se usan para separar la lógica de negocio de nuestra aplicación en una capa distinta, separado estos procesos del controlador o de la capa de datos. Así estamos facilitando separando responsabilidades, mejorando la escalabilidad y facilitando la detección de errores.

La idea es que **la lógica de negocio la hagan los servicios**, y los **controladores consuman el servicio**. Esto es que serán los encargados de hacer las llamadas a los métodos de la clase servicio.

Podríamos hacer toda la lógica de, por ejemplo, conectarnos con la base de datos, extraer la información y llevarla hasta la vista, todo en el controlador. Sería mejor separar toda esa lógica en una clase normal a la llamaremos servicio y que tendrá tantos métodos como trabajos queramos que realicen. De esta forma también estamos encapsulando el trabajo realizado, abstrayéndonos del CÓMO ha sido realizado.

Una cocina sería un servicio. El camarero le pide a la cocina un plato de comida de una determinada forma, la cocina se lo devuelve y el camarero tiene lo que ha pedido. El camarero no tiene que preocuparse del procedimiento que se realizó para obtener el plato. Lo pide y se lo dan. Y ya lo puede servir, o hacer lo que quiera con él.

Un patrón de diseño que se suele usar para crear servicios es el **patrón fachada** (*facade*). Consiste en crear una interfaz con los métodos que queremos que tenga el servicio, y después crear una clase que implemente dicha interfaz. Así nos aseguraremos que el servicio tenga obligatoriamente todos los métodos que se han definido en la interfaz. Esto también permite crear especificaciones y que un mismo servicio pueda implementar varias interfaces, lo cual flexibiliza las opciones. Ejemplo: Podemos tener una interfaz que tenga métodos sólo disponibles para un rol concreto, y un servicio podría implementar interfaces de rol usuario y de rol administrador, teniendo los métodos por separado.

## Creando un servicio

---

Veamos un servicio creado usando dicho patrón.

### Diseño de la interfaz del ArticulosService

```
1 public interface ArticulosService {
2     public Articulo getArticuloAleatorio();
3     public Articulo getArticuloById(Integer id);
4     public List<Articulo> getArticulos(int numero);
5     public List<Articulo> getArticulosBy(String descripcion);
6 }
```

### Implementación de la interfaz

```
1 public class ArticulosServiceImpl implements ArticulosService {
2
3     @Override
4     public Articulo getArticuloAleatorio() {
5         Random r = new Random();
6         int id = Math.abs(r.nextInt());
7         int cantidadRandom = r.nextInt(10)+1;
8         double precioRandom = r.nextDouble(50);
9         boolean congeladoRandom = r.nextBoolean();
10
11         return new Articulo(id, cantidadRandom, "Artículo nº" + id,
12             precioRandom, congeladoRandom);
13     }
14
15     @Override
```

```

15     public Artículo getArticuloById(Integer id) {
16         //Generamos uno al azar y le ponemos la id recibida
17         Artículo articuloCreado = this.getArticuloAleatorio();
18         articuloCreado.setId(id);
19         articuloCreado.setDescripcion("Artículo nº" + id);
20
21         return articuloCreado;
22     }
23
24     @Override
25     public List<Artículo> getArticulos(int numero) {
26         //Generamos una lista con tantos artículos nos hayan pedido
27         List<Artículo> lista = new ArrayList<Artículo>();
28         for (int i=0; i<=numero-1; i++) {
29             lista.add(this.getArticuloAleatorio());
30         }
31         return lista;
32     }
33
34     @Override
35     public List<Artículo> getArticulosByDescripcion(String descripcion) {
36         // TODO Auto-generated method stub
37         return null;
38     }
39 }

```

Vemos como al **implementar** la interfaz `ArticulosService`, Java nos obliga a sobrescribir los métodos abstractos heredados, teniendo que desarrollar lo que hace cada uno. El último método `.getArticulosByDescripcion()` lo hemos dejado intacto tal cual nos lo deja el IDE, y ya lo implementaremos cuando nos haga falta.

Ahora mismo el servicio de artículos, no los está recuperando de una base de datos, los está creando aleatoriamente. Pero al tener toda la lógica de negocio separada por capas, en el momento que usemos el servicio que REALMENTE SI devuelva los artículos obtenidos de una base de datos, los controladores seguirán funcionando exactamente igual, ya que se limitan a hacer llamadas a los servicios, uno falso como ahora (*mock*), o reales (más adelante).

## Añadir funcionalidades al servicio

Si queremos añadir funcionalidades al servicio, habría que añadirle un nuevo método abstracto a la interfaz `ArticulosService`

```

1     public interface ArticulosService {
2         ...
3         public boolean deleteArticulo(int id); //Añadimos un nuevo método para
           borrar un artículo
4     }

```

Y automáticamente el IDE nos pedirá que `ArticulosServiceImpl` debe implementar ese método abstracto heredado.

💡 Recordemos que en java, todos los métodos definidos en una interfaz son implícitamente `public abstract`.

```
public class ArticulosServiceImpl implements ArticulosService {  
    @Override  
    public Articulo deleteArticulo(int id) {  
        Random r = new Random();  
        int id = 1;  
        double precioRandom = r.nextDouble(50);  
        boolean congeladoRandom = r.nextBoolean();  
    }  
}
```

The type ArticulosServiceImpl must implement the inherited abstract method ArticulosService.deleteArticulo(int)

2 quick fixes available:

- Add unimplemented methods
- Make type ArticulosServiceImpl abstract

Al hacer clic en la opción de “Add unimplemented methods” nos heredará automáticamente los métodos nuevos, creando la estructura del mismo, y ahí es donde tendremos que desarrollar la implementación del nuevo método, programando la nueva funcionalidad.

```
1 public class ArticulosServiceImpl implements ArticulosService {  
2     ...  
3     @Override  
4     public boolean deleteArticulo(int id) {  
5         // TODO Auto-generated method stub  
6         return false;  
7     }  
8 }
```

Nosotros decidimos como funcionan los métodos, lo argumentos que reciben, lo que hace en su interior y los valores que devuelven. Por ejemplo, aquí hemos decidido devolver un `boolean` que indicará si pudo borrar el artículo o no. Otra opción sería devolver una instancia del artículo borrado (para poder mostrar información del mismo) o un `null` en caso de que no hubiese sido posible borrar el artículo.

## Servicios en Spring Boot

Ya que sabemos que es un servicio, como crearlos y usarlos en cualquier aplicación JavaEE, veremos como integrarlos en una aplicación con Spring Boot.

Spring nos facilita el uso de los servicios con la anotación `@Service`. Así le estamos diciendo a Spring que la clase va a poder ser inyectada.

```
1 @Service  
2 public class ArticulosServiceImpl implements ArticulosService {  
3     ...  
4 }
```

Ahora necesitamos en el controlador donde usemos el servicio, una instancia del objeto de la clase `ArticulosServiceImpl` y se hace usando la inyección de dependencias. Podemos hacerlo usando la anotación `@Autowired` sobre un atributo o sobre un constructor. Veamos las dos formas:

### Inyección por atributo (desaconsejada)

```

1  @Controller
2  @RequestMapping("/api")
3  public class APIController {
4
5      @Autowired
6      private ArticulosService articuloService;
7
8      @GetMapping("/articulo/{id}")
9      public String getArticuloPorId(
10         @PathVariable Integer id,
11         Model model
12     ) {
13         Artículo a = articuloService.getArticuloById(id);
14         model.addAttribute("articulo", a);
15         return "ficha-articulo";
16     }
17     ...
18 }

```

Ya podemos usar de una forma simple y eficiente los servicios de `ArticuloService` en todos los métodos del controlador, sin tener que inyectarlo método por método.

### Inyección por constructor (recomendada)

Se recomienda hacer la **inyección por constructor**, ya que se considera una buena práctica puesto que garantiza que el servicio esté disponible desde el momento en que se crea la instancia del controlador. Además hace que la clase sea más fácil de testear y reduce el acoplamiento entre el controlador y el servicio.

```

1  @Controller
2  @RequestMapping("/api")
3  public class APIController {
4
5      private final ArticulosService articuloService;
6
7      @Autowired
8      public APIController(ArticuloService articuloService) {
9          this.articuloService = articuloService;
10     }
11
12     @GetMapping("/articulo/{id}")
13     public String getArticuloPorId(
14         @PathVariable Integer id,
15         Model model
16     ) {
17         Artículo a = articuloService.getArticuloById(id);
18         model.addAttribute("articulo", a);
19         return "ficha-articulo";
20     }
21     ...
22 }

```

💡 En el caso de la inyección de dependencias, al declarar el atributo como `final`, estamos asegurándonos de que el objeto asignado por el contenedor de Spring no será reemplazado por otro en ningún momento, lo que puede ser importante para el correcto funcionamiento de la aplicación. Además, nos obliga a asignar el valor del atributo en el constructor, lo que hace que el código sea más legible y fácil de entender.

🤖 La anotación `@Autowired` la cambiamos del atributo al constructor. Aunque en las últimas versiones de Spring es opcional, ya que se considera implícitamente.

⚠️ **Atención:** Un “error” muy común es querer inyectar un objeto de la clase con la implementación de la interfaz. Hay que **inyectar la interfaz del servicio**, y Spring hará todo el trabajo de crear una única instancia del objeto que implementa esa interfaz. En nuestro ejemplo lo correcto sería inyectar `ArticulosService`, y no `ArticulosServiceImpl`. De hecho, funcionará de ambas maneras, pero es una buena práctica codificar las interfaces en general, por el mismo motivo por el que se hace

```
List<Articulo> lista = new ArrayList<Articulo>().
```

🤖 Técnicamente, el alcance de todas las anotaciones de Spring (`@Service`, `@Controller`, etc.) es un *Singleton*. Eso es otro patrón de diseño que consiste en crear una única instancia del objeto, que es la que se inyecta. De forma que no estamos creando múltiples instancias del mismo objeto en cada método.

## Microservicios

Los microservicios son un enfoque arquitectónico para el desarrollo de aplicaciones que consiste en dividir una aplicación monolítica en múltiples servicios pequeños e independientes, cada uno de los cuales se enfoca en una función específica y se ejecuta de manera autónoma. Cada microservicio se puede implementar, escalar y actualizar de forma independiente, lo que permite una mayor flexibilidad y agilidad en el desarrollo y despliegue de la aplicación.

**La principal diferencia** entre los servicios tradicionales y los microservicios **radica en su tamaño y alcance**. Los servicios tradicionales, también conocidos como servicios monolíticos, suelen ser componentes más grandes que encapsulan múltiples funciones y características de una aplicación. Estos servicios monolíticos suelen estar altamente acoplados y dependen de una única base de código y una única base de datos.

Por otro lado, los **microservicios son unidades más pequeñas y autónomas** de funcionalidad dentro de una aplicación. Cada microservicio se desarrolla y despliega de forma independiente, y puede tener su propia base de código y base de datos. Los microservicios se comunican entre sí a través de mecanismos como APIs (interfaces de programación de aplicaciones) o mensajes, lo que permite una mayor flexibilidad y escalabilidad.

Algunas diferencias clave entre los servicios tradicionales y los microservicios son:

1. **Tamaño y alcance:** Los servicios tradicionales son componentes más grandes y abarcan múltiples funciones de una aplicación, mientras que los microservicios se centran en una única funcionalidad específica.
2. **Acoplamiento:** Los servicios tradicionales suelen estar altamente acoplados, lo que significa que un cambio en una parte del servicio puede afectar otras partes. En contraste, los microservicios son más independientes y tienen un bajo acoplamiento.

3. **Despliegue y escalado:** Los servicios tradicionales se despliegan y escalan como una sola unidad, mientras que los microservicios se pueden desplegar y escalar de forma independiente, lo que permite un mayor nivel de flexibilidad y eficiencia.
4. **Mantenimiento y actualización:** Los servicios tradicionales requieren actualizaciones y mantenimiento más complejos debido a su tamaño y dependencias internas. En cambio, los microservicios permiten actualizaciones y mantenimiento más granulares y específicos para cada microservicio.

Es importante destacar que la diferencia clave entre un servicio tradicional y un microservicio no radica en las anotaciones o el código en sí, sino en la arquitectura y la organización del sistema en su conjunto. **El código necesario o anotaciones usadas para realizar un microservicio es exactamente el mismo que para realizar un servicio.**

En un enfoque de microservicios, los componentes se desarrollan, despliegan y escalan de forma independiente, mientras que en un servicio tradicional, todos los componentes están agrupados en una única base de código y se despliegan y escalan juntos.

#### **Ejemplo de uso:**

Imagina que estás desarrollando una aplicación de comercio electrónico que consta de varios microservicios. Cada microservicio se encarga de una funcionalidad específica de la aplicación y se comunica con otros microservicios para realizar tareas relacionadas.

1. Microservicio de autenticación: Se encarga de gestionar la autenticación y autorización de los usuarios. Proporciona endpoints (son las distintas url de un API REST, que verás en el siguiente apartado) para registrar usuarios, iniciar sesión, generar tokens de acceso, etc.
2. Microservicio de catálogo de productos: Se ocupa de gestionar el catálogo de productos de la tienda en línea. Proporciona endpoints para obtener información de los productos, agregar nuevos productos, actualizar información, etc.
3. Microservicio de carrito de compras: Se encarga de gestionar el carrito de compras de los usuarios. Proporciona endpoints para agregar productos al carrito, eliminar productos, calcular el total, etc.
4. Microservicio de procesamiento de pagos: Se encarga de procesar los pagos de los pedidos realizados por los usuarios. Se comunica con servicios externos de pago para procesar transacciones y actualizar el estado de los pagos.

En este escenario, cada microservicio tiene su propia base de código, su propio conjunto de endpoints y su propia lógica de negocio. Cada microservicio se desarrolla, despliega y escala de forma independiente, lo que permite un mayor grado de flexibilidad y modularidad en el desarrollo y mantenimiento de la aplicación.

La comunicación entre los microservicios generalmente se realiza a través de API RESTful (lo verás en el siguiente apartado) o mediante el uso de mensajería asíncrona, como colas de mensajes. Por ejemplo, el microservicio de carrito de compras puede comunicarse con el microservicio de catálogo de productos para obtener información sobre los productos, y el microservicio de procesamiento de pagos puede comunicarse con el microservicio de carrito de compras para obtener los detalles de los pedidos y procesar los pagos.

En resumen, el uso de microservicios en esta aplicación de comercio electrónico permite dividir la funcionalidad en componentes más pequeños y especializados, lo que facilita el desarrollo, la escalabilidad y el mantenimiento del sistema en general. Cada microservicio se desarrolla y despliega como una aplicación independiente, y se comunican entre sí para ofrecer una funcionalidad completa y coherente a los usuarios.

---

# REST

---

## ¿Qué es REST?

---

REST es un acrónimo de **RE**presentational **State** **T**ransfer. En pocas palabras, si HTTP es transferencia de archivos, REST se basa en transferencia de recursos. Aunque ambos siguen usando el mismo protocolo, el HTTP, lo que cambia es la respuesta ofrecida.

Mientras que una respuesta HTTP estándar, es texto que crea otra página web que representa el navegador, una respuesta REST tiene el formato de un archivo XML o JSON. Se usan principalmente para el intercambio de datos, de una manera ligera y legible.

Es una forma de separar el cliente del servidor y hacer esa separación de forma independiente de la plataforma y tecnología usadas. Por ejemplo, podemos tener el backend en Java usando Spring y emitir las respuestas en JSON, de forma que el frontend en Angular haga las peticiones al backend e interprete esas respuestas en JSON. Si el día de mañana cambiamos de Angular a React o bien, el backend lo cambiamos a PHP, todo seguirá funcionando igual, siempre el API REST siga funcionando bajo las mismas rutas y devolviendo los objetos con la misma estructura.

## REST en Spring Boot

---

Podemos construir sin mucho esfuerzo un controlador REST en Spring Boot, de forma que una petición a una URL nos devuelva casi de forma automática una respuesta en formato JSON:

**Ejemplo:** Una petición a <http://localhost:8080/proyecto/api/articulos>, en lugar de redireccionar a una vista con Thymeleaf, podemos emitir una respuesta con la lista completa de artículos que tengamos en nuestra base de datos:

```
1  [
2    {
3      "id": 1,
4      "cantidad": 9,
5      "descripcion": "Bolsa Patatas Prefritas 1kg",
6      "precio": 0.79,
7      "congelado": true
8    },
9    {
10     "id": 2,
11     "cantidad": 5,
12     "descripcion": "Garbanzos precocidos 500ml",
13     "precio": 0.80,
14     "congelado": false
15   },
```



```

16     {
17         "id": 3,
18         "cantidad": 9,
19         "descripcion": "Leche en polvo 1l",
20         "precio": 1.00,
21         "congelado": false
22     },
23     {
24         "id": 4,
25         "cantidad" : 3,
26         "descripcion": "Pollo asado 700gr",
27         "precio": 4.30,
28         "congelado": false
29     },
30     {
31         "id": 5,
32         "cantidad": 2,
33         "descripcion": "Nuggets de pollo 500gr",
34         "precio": 2.50,
35         "congelado": true
36     }
37 ]

```

💡 Para que la conversión no de errores, el objeto a convertir deberá ser un bean (tener un constructor por defecto, tener todos los *getters* y *setters* públicos bien construidos, etc.)

Vamos a ver todas la anotaciones que debemos usar para crear un controlador REST con muy poco código.

## Clase Artículo

En los siguientes ejemplos, usaremos la siguiente clase **Articulo.java**, por lo que exponemos el código completo.

```

1  public class Articulo {
2      //Atributos
3      private long id;
4      private int cantidad;
5      private String descripcion;
6      private double precio;
7      private boolean congelado;
8
9      //Constructores
10     public Articulo() {}
11
12     public Articulo(long id, int cantidad, String descripcion, double precio,
13                     boolean congelado) {
14         super();
15         this.id = id;
16         this.cantidad = cantidad;
17         this.descripcion = descripcion;
18     }
19 }

```


```
17         this.precio = precio;
18         this.congelado = congelado;
19     }
20
21     public Artículo(int cantidad, String descripcion, double precio) {
22         super();
23         this.id = 0;
24         this.cantidad = cantidad;
25         this.descripcion = descripcion;
26         this.precio = precio;
27         this.congelado = false;
28     }
29
30     //Métodos
31     public long getId() {
32         return id;
33     }
34
35     public void setId(long id) {
36         this.id = id;
37     }
38
39     public int getCantidad() {
40         return cantidad;
41     }
42
43     public void setCantidad(int cantidad) {
44         this.cantidad = cantidad;
45     }
46
47     public String getDescripcion() {
48         return descripcion;
49     }
50
51     public void setDescripcion(String descripcion) {
52         this.descripcion = descripcion;
53     }
54
55     public double getPrecio() {
56         return precio;
57     }
58
59     public void setPrecio(double precio) {
60         this.precio = precio;
61     }
62
63     public boolean isCongelado() {
64         return congelado;
65     }
66
67     public void setCongelado(boolean requiereFrio) {
68         this.congelado = requiereFrio;
```

```

69     }
70
71     @Override
72     public String toString() {
73         return "Articulo {id=" + id +
74             ", cantidad=" + cantidad +
75             ", descripcion=" + descripcion +
76             ", precio=" + precio +
77             ", congelado=" + congelado +
78             "}";
79     }
80 }

```

## Lombok

 **Nota:** La documentación de Lombok de este tutorial es experimental. No está testeada. Usar con precaución.

Lombok es una librería que posee muchas anotaciones que nos ahorra trabajos repetitivos de muchos tipos.

Para añadir la librería, añadiremos las siguientes líneas al `pom.xml` de Maven.

```

1  <!-- https://projectlombok.org/setup/maven -->
2  <dependency>
3      <groupId>org.projectlombok</groupId>
4      <artifactId>lombok</artifactId>
5      <version>1.18.26</version>
6      <scope>provided</scope>
7  </dependency>

```

Una vez actualizadas las dependencias y descargadas las librerías, ya podemos usar todas sus anotaciones.

Las más usadas son:

- `@Getter` y `@Setter`, nos ahorramos tener que crear los métodos getters y setters respectivamente, de cada atributo.
- `@ToString` en la clase, nos creará el método `toString()`.
- `@EqualsAndHashCode`, nos creará ambos métodos a la vez.
- `@NoArgsConstructor`, creará un constructor por defecto (que no recibe parámetros).
- `@AllArgsConstructor`, creará un constructor que recibe todos los valores de los atributos por parámetros.
- `@Data`, es un anotación que incluye las ventajas de `@ToString`, `@EqualsAndHashCode`, `@Getter` / `@Setter` y `@RequiredArgsConstructor`. En otras palabras, genera todo el código repetitivo que normalmente se hace en todos los POJOs y Beans, sin tener que usar todas las anotaciones una por una.

- La lista completa la puedes encontrar en su documentación oficial en la página <https://projectlombok.org/features/all>.

Usando todas las anotaciones anteriores, la clase `Articulo` creada anteriormente se podría quedar con el siguiente código:

```
1  @AllArgsConstructor @NoArgsConstructor
2  @ToString @EqualsAndHashCode
3  @Getter @Setter
4  public class Articulo {
5      private long id;
6      private int cantidad;
7      private String descripcion;
8      private double precio;
9      private boolean congelado;
10
11     public Articulo(int cantidad, String descripcion, double precio) {
12         super();
13         this.id = 0;
14         this.cantidad = cantidad;
15         this.descripcion = descripcion;
16         this.precio = precio;
17         this.congelado = false;
18     }
19 }
```

💡 De 80 líneas ha pasado a 19. Y se podrían quedar en menos usando la anotación `@Data`.

💡 Podéis encontrar más información en <https://javatodev.com/lombok-spring-boot/>.

## @ResponseBody

La anotación `@ResponseBody` le indica a un controlador que el objeto que retornemos se debe convertir automáticamente en JSON y eso será lo que se envíe como respuesta, en lugar de ir a una vista con Thymeleaf.

```
1  @Controller
2  @RequestMapping("/api")
3  public class APIController {
4      public final ArticulosService articulosService;
5
6      public APIController(ArticulosService articulosService){
7          this.articulosService = articulosService;
8      }
9
10     ...
11
12     @ResponseBody
13     @GetMapping("/articulo/{id}")
14     public Articulo getArticulo(@PathVariable Integer id) {
```

```

15         return articulosService.getArticuloById(id);
16     }
17 }

```

Tenemos un controlador `APIController`, que será el encargado de recibir todas las peticiones REST. Destacamos:

- Le inyectamos el servicio `ArticulosService`, que será el encargado de usar la lógica de negocio para obtener la información que se le pide (obtener artículos por id, por precio, por descripción, todos los artículos, etc.)
- Con la anotación `@ResponseBody` le indicamos al controlador que no vamos a redireccionar a una vista HTML, si no que lo que vamos a devolver es un objeto, el cual será convertido a JSON por Spring Boot. Por eso tenemos que indicar en la firma del método la clase del objeto que vamos a retornar.
- El método `.getArticulo()` será llamado cada vez que realicemos una consulta a `/api/articulo/{id}`, donde `{id}` es la id del artículo a consultar. El servicio llamará al método que ya tiene para tal fin `articulosService.getArticuloById()`, el cual retornará el artículo cuya id sea `{id}` o bien `null` si esa id no se encuentra.

Obtendríamos la siguiente respuesta en JSON al hacer la petición GET a la URL `/api/articulo/3`

```

1 {
2     "id": 3,
3     "cantidad": 9,
4     "descripcion": "Leche en polvo 1l",
5     "precio": 1.00,
6     "congelado": false
7 }

```

Con esa petición obtendremos UN único artículo. Si queremos obtener un array de artículos, tan sólo deberemos retornar una colección de objetos y Spring Boot hará el trabajo de conversión a JSON.

```

1 @ResponseBody
2 @GetMapping("/articulos")
3 public List<Articulo> getArticulos() {
4     return articulosService.getAllArticulos();
5 }

```

```

1 [
2     {
3         "id": 1,
4         "cantidad": 9,
5         "descripcion": "Bolsa Patatas Prefritas 1kg",
6         "precio": 0.79,
7         "congelado": true
8     },
9     {
10        "id": 2,
11        "cantidad": 5,

```

```
12         "descripcion": "Garbanzos precocidos 500ml",
13         "precio": 0.80,
14         "congelado": false
15     },
16     {...},
17     {...}
18 ]
```

## @RestController

---

Hemos visto que para hacer que los métodos de un controlador devuelvan objetos JSON, debemos usar `@ResponseBody`. Por otro lado, es muy frecuente que todos los métodos que devuelvan JSON estén agrupados en un mismo controlador, por lo que hay una anotación que se usa como combinación de `@Controller` + `@ResponseBody`, y es `@RestController`.

Si usamos la anotación especializada `@RestController` en lugar de `@Controller`, le estamos diciendo explícitamente que TODOS los métodos de ese controlador devolverán JSON, por lo que no es necesario indicarle `@ResponseBody`.

```
1  @RestController
2  @RequestMapping("/api")
3  public class APIController {
4      ...
5
6      @GetMapping("/articulo/{id}")
7      public Artículo getArticulo(@PathVariable Integer id) {
8          return articulosService.getArticuloById(id);
9      }
10 }
```

---

## Práctica 6

---

Hacer un proyecto Spring Boot, con un REST de usuarios funcional como el siguiente. El servicio será un mock que actuará sobre una colección.

# Servidor REST

## RestController de usuarios

URL	Prueba	Ubicación	Descripción
/api/usuarios	/api/usuarios	APIController	Devuelve todos los usuarios
/api/usuario/id/{id}	/api/usuario/id/5	APIController	Devuelve el usuario cuya id es 5
/api/usuario/email/{email}	/api/usuario/email/prueba3@correo.com	APIController	Devuelve el usuario cuyo correo es 'prueba3@correo.com'
/api/usuarios/email/{email}	/api/usuarios/email/prueba	APIController	Devuelve los usuarios cuyo correo empiece por 'prueba'
/crear/usuarios/{numero}	/crear/usuarios/5	HomeController	Vacía la lista, y la rellena con n usuarios.
/crear/usuario/{email}	/crear/usuario/usuario@test.com	HomeController	Crea un usuario con el email recibido y lo añade a la lista

En <https://github.com/borilio/curso-spring-boot/tree/master/assets/clases/practica-6> encontrarás los siguientes recursos para reutilizar:

### Lo que ya está hecho:

- `vistas/home.html` -> Es la página principal de la aplicación (la captura de arriba). En ella se encuentran las url que debemos satisfacer en nuestro Rest Controller y la descripción de lo que debe hacer cada una.
- `controllers/HomeController.java` -> El controlador principal que nos lleva a `home.html`. Ya encontrarás definidos e implementados en su interior formas para crear usuarios manualmente.
- `users/User.java` -> Una clase POJO que representa un usuario. Examina los constructores que tiene.
- `users/UserService.java` -> La interfaz `UserService` donde están definidos los métodos que deberemos desarrollar en `UserServiceImpl`.
- **⚠ Aviso:** En los archivos proporcionados, deberás cambiar el nombre del paquete por el definido en tu proyecto.

### Lo que hay que hacer:

1. Primero deberemos completar el servicio. En orden haremos:

1. Creamos en el paquete `users`, una clase llamada `UserServiceImpl`.
2. Convertimos la clase en un servicio, usando la anotación correspondiente.
3. La clase implementará la interfaz `UserService`, sobrescribiendo todos los métodos abstractos heredados por la interfaz.
4. La clase tendrá un sólo atributo privado, `private List<User> listaUsuarios;`. En el constructor de la clase inicializaremos ese atributo a un nuevo `ArrayList<User>` vacío.
5. Cada método heredado actuará sobre el atributo de la clase `listaUsuarios`, como si de una base de datos se tratara. De forma que nuestro servicio “imitará” el funcionamiento de una base de datos. Desde fuera del servicio, parecerá que está tratando internamente con una base de datos.

6. En `UserService`, cada método tiene un comentario explicando lo que debería hacer cada método del servicio. Esto deberá implementarse en `UserServiceImpl`. Son acciones simples, entre 1 y 4 líneas cada uno como máximo. Si tenéis dudas en alguna preguntar y se aclarará.
2. Crear el controlador `ApiController`. Aquí es donde se definirán las url que se pueden ver en `home.html` (o en la captura).
  1. Usar las anotaciones oportunas para crear un REST.
  2. Inyectar el servicio.
  3. Hacer los 4 métodos necesarios para satisfacer las necesidades del API definidas, haciendo uso del servicio `UserService`.
3. Comprobar que todo lo indicado en `home.html` funciona.

💡 Verás que haciendo uso del servicio, las acciones para interactuar con la “base de datos” se resumen a UNA LINEA DE CÓDIGO. Y lo mejor es que este servicio *mock*, puede ser fácilmente sustituido por uno real que sí acceda a una base de datos real, dejando el código del controlador intacto.

---

## Métodos de petición HTTP

---

En la práctica anterior, no hemos borrado nada de nuestra “base de datos” falsa, a pesar de tener un método en el servicio.

```
1 public interface UserService {
2     ...
3     //Limpia todo el contenido de la lista de usuarios, dejándola vacía
4     public void deleteAll();
5 }
```

Su implementación era simple, era limpiar el `ArrayList`.

```
1 @Override
2 public void deleteAll() {
3     this.listaUsuarios.clear();
4 }
```

Podríamos haber añadido una acción más, que por ejemplo al ir a `GetMapping` de `/borrar/usuarios`, hacer una llamada al método del servicio y se borraba la lista completa. TODO lo hacemos con `GET`.

Pero las peticiones de tipo `GET` deberían usarse sólo para recuperar datos. Veamos los métodos más comunes de HTTP y sus aplicaciones (hay más, pero mostraremos los básicos).

## GET

---

El método `GET` solicita una representación de un recurso específico. Las peticiones que usan el método `GET` sólo deben recuperar datos.



```
1 | GET localhost:8080/api/usuarios
```

## POST

---

El método `POST` se utiliza para enviar una entidad a un recurso específico. Aunque se pueda usar `GET` para enviar datos, tiene muchas limitaciones que `POST` no tiene. Su fin es más genérico, envía información para que quién la reciba haga lo que estime con ella. `POST` no es idempotente. Una nueva petición `POST` tendría un efecto distinto que la primera llamada (por ejemplo, dos peticiones seguidas insertarían 2 usuarios en la base de datos).

```
1 | POST localhost:8080/api/usuario/user
```

El backend crearía un nuevo objeto `user` y le asignaría una id, creando el objeto `/api/usuario/123`. Por eso cada llamada puede tener efectos distintos. Otra llamada crearía otro usuario con otra id distinta, creando otro recurso `/api/usuario/124`.

## PUT

---

El modo `PUT` reemplaza todas las representaciones actuales del recurso de destino con la carga útil de la petición. Se suele usar para actualizar contenidos o bien para crear nuevos. Pone un contenido en un recurso, si no existe lo crea, y si existe lo reemplaza. `PUT` es idempotente, es decir, siempre tendrá el mismo resultado cuantas veces se realice la petición.

```
1 | PUT localhost:8080/api/usuario/5
```

Se actualizará el recurso indicado con la información que lleve en la petición, o se creará uno nuevo. Si repetimos la petición, se volvería a realizar, pero dejándolo en el mismo estado. No duplicaría nada.

## DELETE

---

El método `DELETE` borra un recurso en específico.

```
1 | DELETE localhost:8080/api/usuario/5
```

Borraría el recurso (usuario) cuya id sea 5.

## PATCH

---

El método `PATCH` es utilizado para aplicar modificaciones parciales a un recurso. A diferencia de `PUT` que lo reemplaza (o crea) completamente.

```
1 | PATCH localhost:8080/api/usuario/5
```

En la petición iría sólo la información que queremos modificar en el recurso. La que no se incluya se dejará como estaba.

---

## Usando los métodos HTTP en Spring Boot

Ya hemos visto que hay más métodos/verbos en las peticiones HTTP. Veamos como se aplicarían correctamente en la práctica anterior.

### @DeleteMapping

Para hacer una petición que borre un recurso, en lugar de hacerlo mediante el método `GET`, lo correcto sería hacerlo con el método `DELETE`, y eso sería cambiando el método soportado en la petición a `/borrar/usuarios`

```
1  @RestController
2  @RequestMapping("/api")
3  public class APIController {
4      public final ArticulosService articulosService;
5
6      public APIController(ArticulosService articulosService){
7          this.articulosService = articulosService;
8      }
9
10     @GetMapping("/usuarios")
11     public List<User> getAllUsers(){
12         return userService.getAll();
13     }
14     ...
15     @DeleteMapping("/usuarios")
16     public void deleteAllUsers() {
17         userService.deleteAll();
18     }
19 }
```

Al soportar cada url métodos distintos, podemos usar las mismas url, facilitando la escalabilidad del api.

Lo único que ahora para probar el método `.deleteAllUsers()` no podremos ir a la url `/api/usuarios` desde el navegador, ya que si no estaremos haciendo una petición de tipo `GET`, y ejecutaremos `.getAllUsers()`.

La url será la misma, pero dependiendo del tipo de petición que hagamos, hará una cosa u otra. Para poder probar esto, podemos usar una aplicación tipo [postman](#) o extensiones del navegador.

Podemos ver que al hacer la petición de tipo `GET` a `/api/usuarios`, obtenemos como respuesta el JSON con todos los usuarios.

localhost:8080/api/usuarios Save

GET ▼ localhost:8080/api/usuarios Send ▼

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Query Params

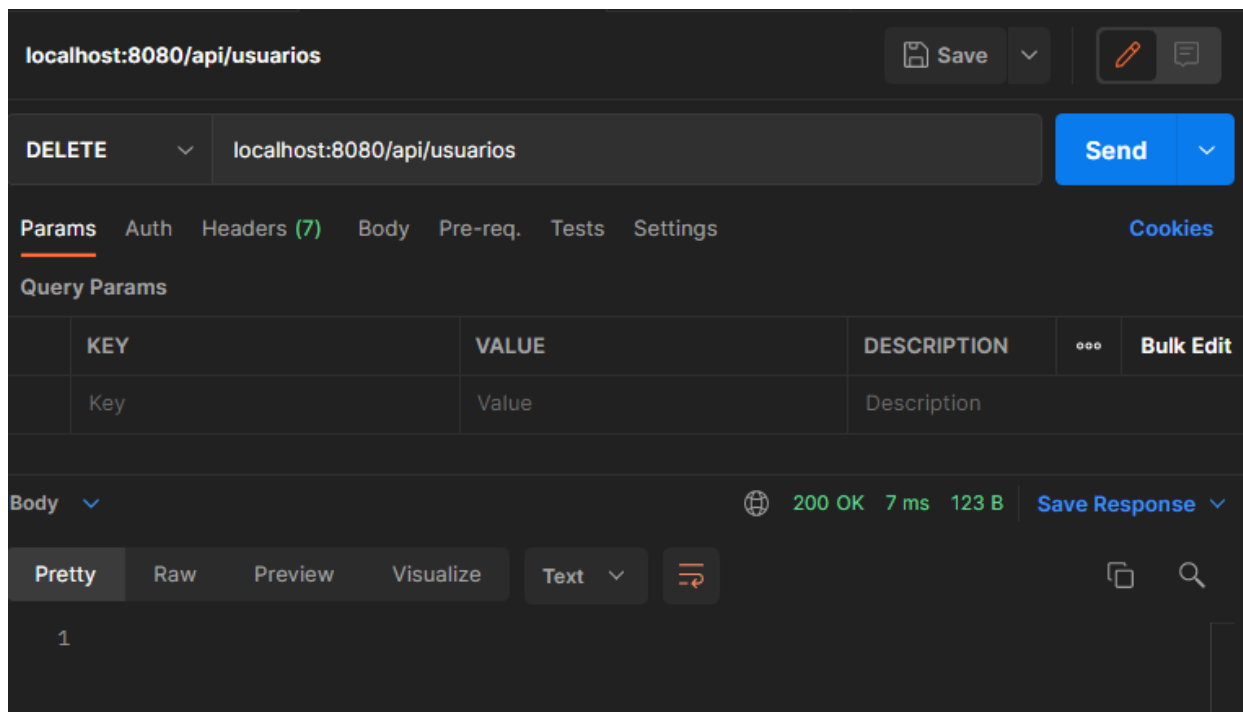
	KEY	VALUE	DESCRIPTION	...	Bulk Edit
	Key	Value	Description		

Body ▼ 200 OK 22 ms 851 B Save Response ▼

Pretty Raw Preview Visualize JSON ▼

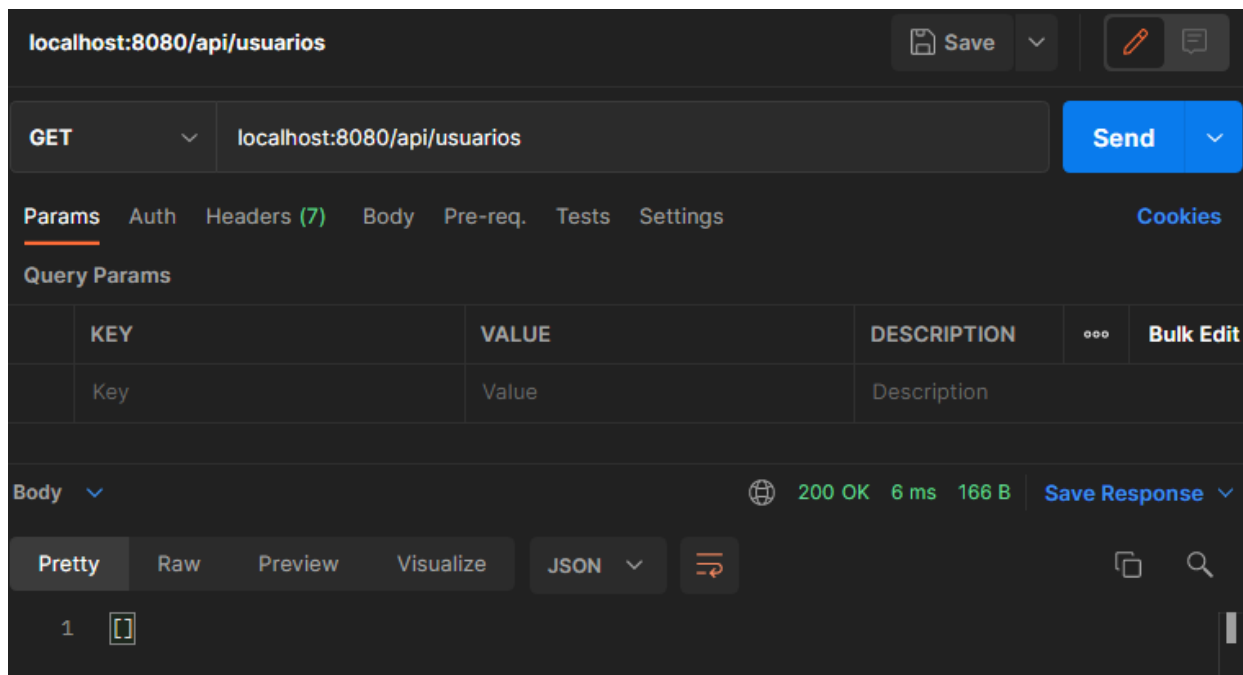
```
1 [
2   {
3     "id": 1488813158,
4     "email": "prueba1@correo.com",
5     "password": "10e20892"
6   },
7   {
8     "id": 902623064,
9     "email": "prueba2@correo.com",
10    "password": "1e25e4fe"
11  },
12  {
13    "id": 1805564868,
14    "email": "prueba3@correo.com",
15    "password": "1a1a63e2"
16  },
17  {
18    "id": 1406383380,
19    "email": "prueba4@correo.com",
20    "password": "5f18eec5"
```

Sin embargo si hacemos una petición `DELETE` a la misma url, obtenemos lo siguiente:



No obtenemos cuerpo de respuesta, pero nos devolvió un código 200. Eso significa que no hubo errores. Se procesó la petición bien y nos dirigió al método correcto que borró la “base de datos” de usuarios.

Si repetimos la petición `GET` a `/api/usuarios` para asegurarnos, obtendremos lo siguiente:



Si decidimos crear un nuevo método al servicio para borrar UN usuario concreto, haríamos lo siguiente:

Añadimos el método al servicio `UserService`.

```

1 public interface UserService {
2     ...
3     //Borra el usuario cuya id sea igual a la recibida, devolviendo el User si lo
    borró, o null si no se encontró
4     public User deleteById(int id);
5
6 }

```

Lo implementamos en `UserServiceImpl`.

```

1 @Service
2 public class UserServiceImpl implements UserService {
3     ...
4
5     @Override
6     public User deleteById(int id) {
7         User userBorrado = null;
8         for (User u: this.listaUsuarios) {
9             if (u.getId() == id) {
10                 userBorrado = u;
11                 listaUsuarios.remove(u);
12             }
13         }
14         return userBorrado;
15     }
16 }

```

Añadimos el método al `APIController`, usando el servicio anterior.

```

1 @RestController
2 @RequestMapping("/api")
3 public class APIController {
4     ...
5
6     @DeleteMapping("/usuario/{id}")
7     public void deleteUserById(@PathVariable Integer id) {
8         userService.deleteById(id);
9     }
10 }

```

Si tenemos 3 usuarios en nuestra “base de datos”, y hacemos la siguiente petición, obtendremos la siguiente respuesta, respectivamente:

```

1 GET localhost:8080/api/usuarios

```

```

1 [
2     {
3         "id": 2114061879,
4         "email": "prueba1@correo.com",

```

```

5      "password": "7e5ddfb1"
6    },
7    {
8      "id": 1610394191,
9      "email": "prueba2@correo.com",
10     "password": "4e31045"
11   },
12   {
13     "id": 295829159,
14     "email": "prueba3@correo.com",
15     "password": "64857c0"
16   }
17 ]

```

Comprobamos que ahí está la base de datos completa. Hacemos nueva petición y respuesta.

```

1 | GET localhost:8080/api/usuario/id/1610394191

```

```

1 | {
2 |   "id": 1610394191,
3 |   "email": "prueba2@correo.com",
4 |   "password": "4e31045"
5 | }

```

Y si en lugar de `GET`, usamos `DELETE`, se borrará ese usuario, en lugar de obtener su JSON.

```

1 | DELETE localhost:8080/api/usuario/id/1610394191

```

Comprobamos de nuevo la lista completa y obtenemos la respuesta:

```

1 | GET localhost:8080/api/usuarios

```

```

1 | [
2 |   {
3 |     "id": 2114061879,
4 |     "email": "prueba1@correo.com",
5 |     "password": "7e5ddfb1"
6 |   },
7 |   {
8 |     "id": 295829159,
9 |     "email": "prueba3@correo.com",
10    "password": "64857c0"
11   }
12 ]

```

El usuario cuya id era `1610394191` fue borrado de la base de datos, usando su correspondiente `DELETE` como método HTTP, en lugar de `GET`.

## @PostMapping

Para borrar hemos visto que se puede o bien no recibir nada (si lo quiero borrar todo) o bien se puede recibir por `@PathVariable` la id del recurso a borrar. Pero para crear un nuevo recurso, ¿como podría enviar un objeto a través de la petición para que el controlador lo recoja y lo guarde en la base de datos?

Esto ahora mismo sabríamos hacerlo. Podríamos:

- Añadiendo en la petición `GET` los parámetros en la misma url. Ej: `/crear/usuario?id=5&email=nuevo@test.com&pass=12345`. Esto es muy engorroso, seguridad nula, y para objetos complejos sería prácticamente inviable.
- Con un formulario, por `GET` o `POST`, y recogiendo los parámetros con `@RequestParam` por separado en variables (`id`, `email` y `pass` siguiendo nuestro ejemplo), creando un nuevo objeto de la clase `User`, pasándole esas variables al constructor. Casi los mismos problemas que en la opción anterior.

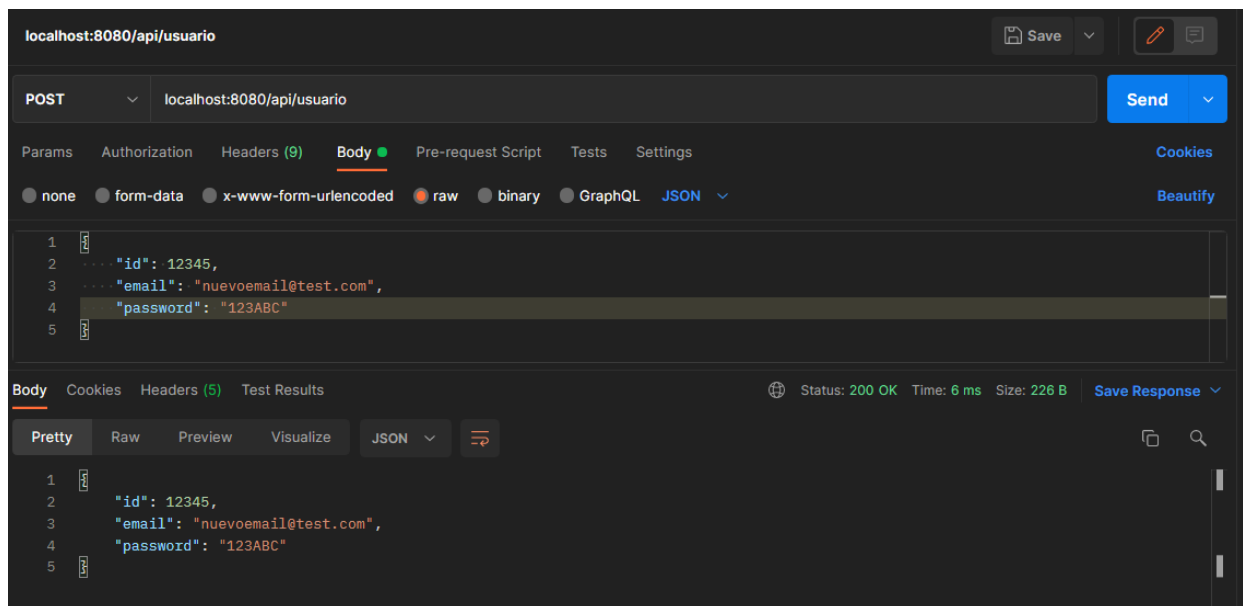
**Una mejor solución es enviar el objeto en JSON por la petición** `POST` o `PUT`, y en Spring Boot hará el trabajo de convertir ese JSON a un Objeto Java directamente.

Si cuando convertimos de Objeto a JSON para la respuesta, usamos la anotación `@ResponseBody`, ahora que estamos haciendo justo lo contrario, convertir JSON de la petición en un Objeto, la anotación que usaremos será `@RequestBody`.

En nuestro RestController `APIController`, le añadimos el siguiente método:

```
1 @RestController
2 @RequestMapping("/api")
3 public class APIController {
4     ...
5     @PostMapping("/usuario")
6     public User nuevoUsuario(@RequestBody User usuarioNuevo) {
7         return userService.add(usuarioNuevo);
8     }
9 }
```

Y si con *postman* hacemos la petición de tipo `POST` y en el *body* le indicamos *raw* y a la derecha del todo, el formato JSON. Y escribimos directamente el JSON en el cuerpo de la petición, la url `/api/usuario` recibirá el JSON indicado y gracias al `@RequestBody`, Spring lo convertirá a un objeto `User` y gracias al servicio lo insertaremos en la “base de datos”. La petición nos devolverá el objeto que ha sido guardado.



Y ahora comprobamos que el usuario ha sido guardado correctamente.

```
1 | GET localhost:8080/api/usuarios
```

```
1 | [  
2 |   {  
3 |     "id": 2114061879,  
4 |     "email": "prueba1@correo.com",  
5 |     "password": "7e5ddfb1"  
6 |   },  
7 |   {  
8 |     "id": 295829159,  
9 |     "email": "prueba3@correo.com",  
10 |    "password": "64857c0"  
11 |   },  
12 |   {  
13 |     "id": 12345,  
14 |     "email": "nuevoemail@test.com",  
15 |     "password": "123ABC"  
16 |   }  
17 | ]
```

## @PutMapping

Con todo lo que hemos visto anteriormente, ya sabemos que para actualizar un elemento concreto entero (o crear) usaremos `PUT`.

Definimos el método en la interfaz y lo implementaremos:



```

1 public interface UserService {
2     ...
3     public User update(User nuevo, int id);
4 }

```

```

1 @Service
2 public class UserServiceImpl implements UserService {
3     ...
4     @Override
5     public User update(User nuevo, int id) {
6         User actualizado = this.getUserB;
7         for (User u: this.listaUsuarios) {
8             if (u.getId() == id) {
9                 u.setEmail(nuevo.getEmail());
10                u.setPassword(nuevo.getPassword());
11                actualizado = u;
12            }
13        }
14        return actualizado;
15    }
16 }

```

Añadimos el método al `APIController`, usando el servicio anterior.

```

1 @RestController
2 @RequestMapping("/api")
3 public class APIController {
4     ...
5
6     @PutMapping("/usuario/id/{id}")
7     public User updateUser(
8         @PathVariable Integer id,
9         @RequestBody User userUpdated
10    ) {
11        return userService.update(userUpdated, id);
12    }
13 }

```

Y haciendo la siguiente petición `PUT`, tendremos la respuesta (suponiendo que existe un usuario cuya id es `1545713824`):

```
1 PUT /api/usuario/1545713824 HTTP/1.1
2 Host: localhost:8080
3 Content-Type: application/json
4
5 {
6   "id": 1545713824,
7   "email": "pruebados@correonuevo.com",
8   "password": "12345Laclavequepondríaunestúpidoensusmaletas"
9 }
```

```
1 {
2   "id": 1545713824,
3   "email": "pruebados@correonuevo.com",
4   "password": "12345Laclavequepondríaunestúpidoensusmaletas"
5 }
```

## @PatchMapping

Sería exactamente igual que `PUT` pero con la diferencia de que podemos omitir los campos que no queremos que se modifiquen. Por ejemplo si sólo queremos modificar el email, pues en la petición sólo pondríamos:

```
1 {
2   "email": "pruebados@correonuevo.com"
3 }
```

Y el resto de atributos quedarían tal y como estaban.

## Conclusión

Usando los métodos adecuados de HTTP para las peticiones podemos conseguir que nuestros frontend y backend se comuniquen mediante objetos JSON, de forma bidireccional. **Así nos aseguramos que uno no dependa del otro** y que trabajen de forma independiente.

El equipo de desarrollo de frontend puede usar un backend mock ( <https://my-json-server.typicode.com>, <https://www.mockable.io>, <https://get.mocklab.io>) y viceversa con Postman por ejemplo.

Cuando ambos funcionen y ya estén funcionales y testeados, podrán integrarse fácilmente cambiando uno por otro y todo funcionará perfectamente a la primera 🍌.

