Conceptos importantes

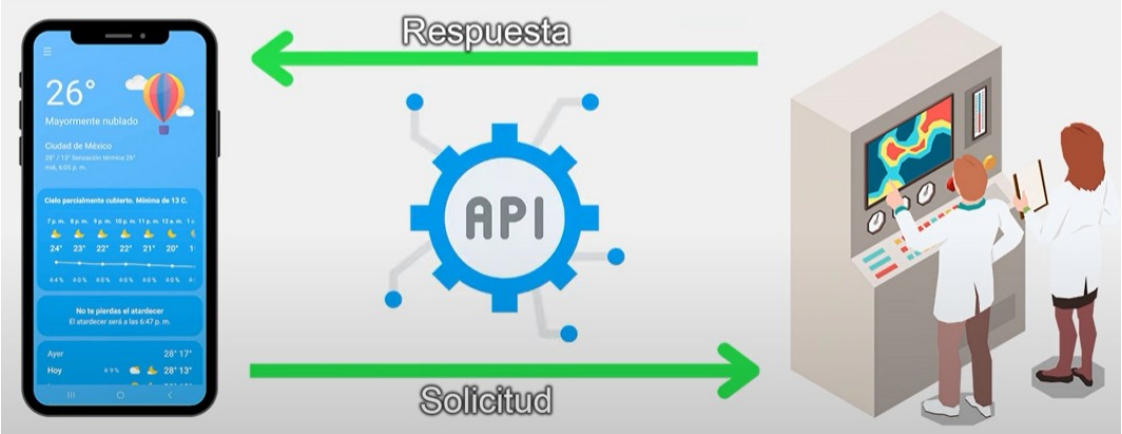
API: Application Programming Interface: Interfaz de programación de aplicaciones.

Es un conjunto de reglas y protocolos usados en el desarrollo de aplicaciones y que permiten que dos o más aplicaciones diferentes se comuniquen entre sí mediante solicitudes y respuestas.

Sirve para que los desarrolladores de software utilicen funciones y datos de otras aplicaciones en las suyas propias sin necesidad de conocer el código fuente o estructura de dichas aplicaciones. Permite reutilizar código y tener una mayor flexibilidad.

Por ejemplo:

- Twitter tiene una Api que permite que otras aplicaciones se conecten a ella y pueda ver determinada información. Nosotros como desarrolladores externos podríamos acceder a funciones de dicha Api sin saber cómo están implementadas.
- En tiendas online, se acceden a Apis que permiten validar los datos bancarios ofrecidos para realizar el pago.
- Consulta de meteorología:



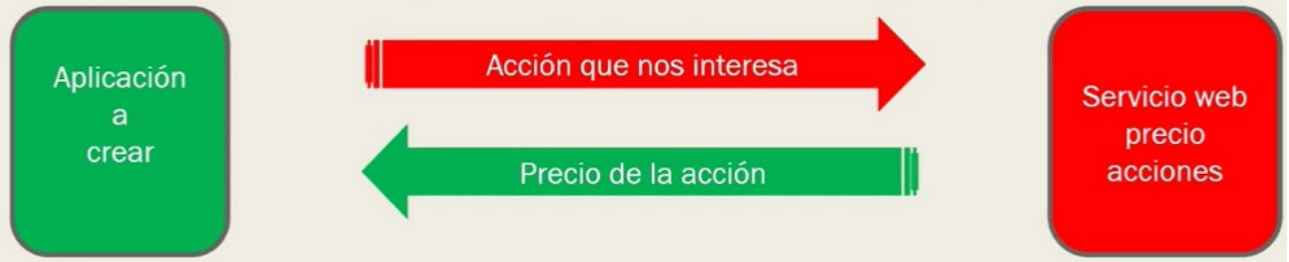
La aplicación que envía la solicitud o petición se llama **cliente** y la que envía la respuesta se llama **servidor**.
Las arquitecturas más comunes de una API son: SOAP, RPC, WebSocket y REST.

REST: Es una interfaz para conectar varios sistemas basados en el protocolo HTTP. La idea es que los sistemas puedan interactuar de manera estándar. Se basa en unos principios de RESTful:

- Utiliza URLs para acceder a recursos,
- Utiliza instrucciones HTTP (GET, POST, PUT y DELETE) que identifica la acción a realizar
- Utiliza formato estándar para intercambiar información, como **JSON o XML**.

API REST: Es un estilo de arquitectura de software que permite diseñar servicios web de manera estándar. Estas Apis usan el protocolo HTTP para poder realizar operaciones CRUD (Create, Read, Update y Delete) utilizando instrucciones HTTP (GET, POST, PUT y DELETE).

Ejemplo: crear una aplicación que proporcione el precio de una acción en tiempo real conectándose a un servicio externo.



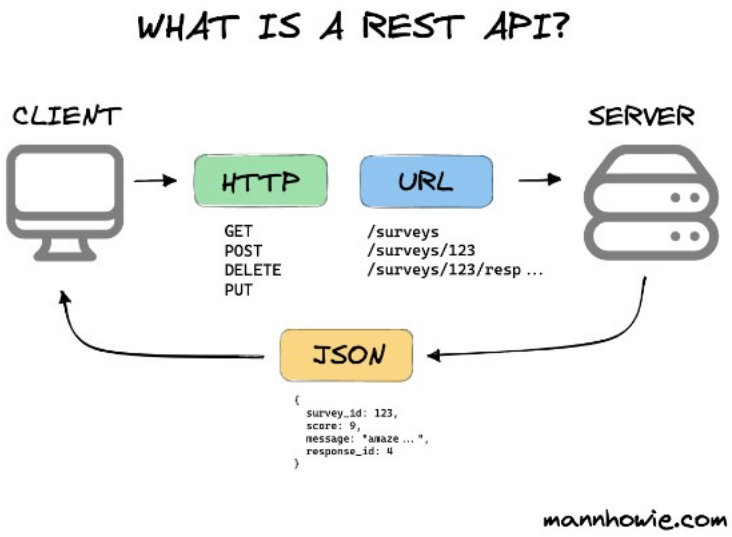
- ¿Cómo conectamos con el servicio web externo? Creando una API REST que comunique via HTTP.
- ¿Qué lenguaje de programación usamos? REST es independiente del lenguaje, tanto cliente como servidor pueden usar cualquier lenguaje.
- ¿En que formato obtenemos la información? Lo más común en XML y **JSON (más usado)**

ENDPOINT

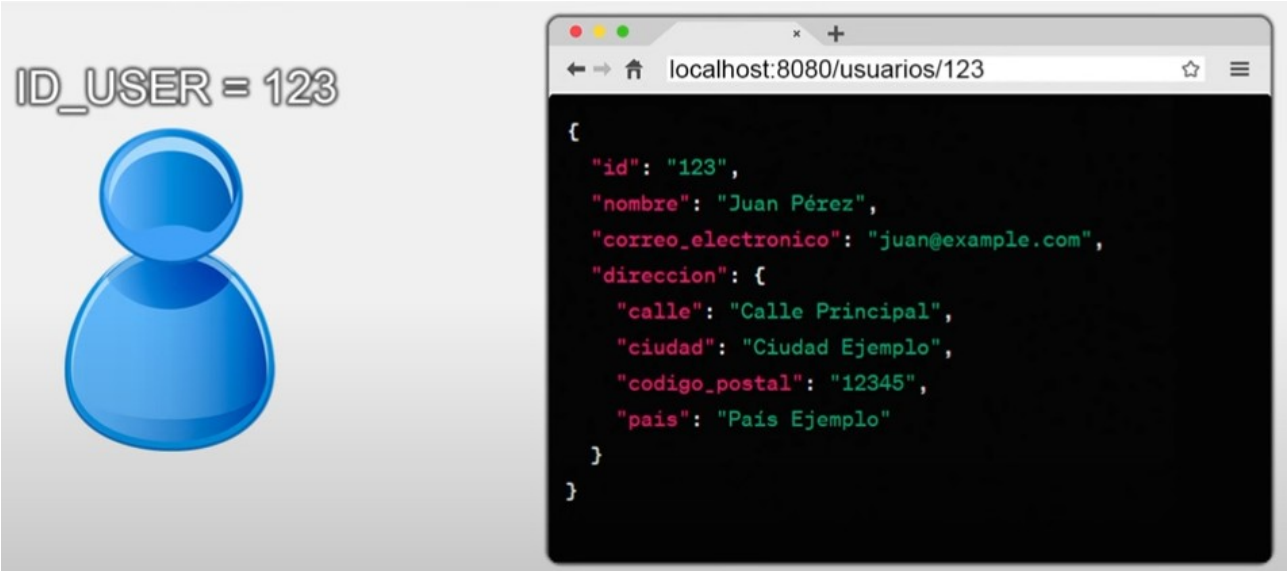
Un endpoint en una API REST es una dirección de una API que responde a una petición a través de una interfaz, enviando información, si procede, en formatos JSON o XML.

Ejemplos:

/clientes/12
/clientes



JSON: JavaScript Object Notation. Es un formato de intercambio de datos ligero y fácil de leer y escribir. JSON se basa en un subconjunto de JavaScript, aunque es independiente del lenguaje, y es ampliamente utilizado para el intercambio de datos en aplicaciones web.



Ejemplo JSON:

```
{
    "status": "success",
    "data": {
            "user": {
                "id": 1,
                "name": "Juan",
                "email": "juan@example.com"
            }
    }
}
```

Anotaciones

- **@SpringBootApplication**: Esta es la anotación principal que se utiliza para marcar la clase principal de una aplicación Spring Boot.

- **@RestController**: Se utiliza para marcar una clase Java como un controlador REST y generan respuestas HTTP para APIs RESTful.

  Cuando una clase está anotada con @RestController, cada método de esa clase se mapea automáticamente a una ruta URL y responde a las solicitudes HTTP en función de las anotaciones: @GetMapping, @PostMapping, @PutMapping, @DeleteMapping, ...

- **@GetMapping**: mapea las solicitudes **HTTP GET**. Spring Boot asigna la URL especificada en la anotación a un método en el controlador y maneja automáticamente las solicitudes HTTP GET que llegan a esas URLs, invocando el método correspondiente y devolviendo el resultado como respuesta.

  @GetMapping necesita tener el atributo que indica la URL de la solicitud GET.
  Ejemplo: @GetMapping("/alta")

  También pueden tener más de una URL a la vez:
  @ GetMapping({"/alta", "/nueva" , "/adios"})

Paso de parámetros por URL

- **@PathVariable:** Se usa para mapear partes de la URL de una solicitud a parámetros de un controlador de Spring. Los parámetros dinámicos que vienen por la URL los usamos en nuestro endpoint.

```
@GetMapping("/hola/{name}")
public String helloName(@PathVariable String name){
    return "Hello " + name;
}
```

Ejemplo de implementación de un Rest Service

1.- Crear un proyecto con Spring Starter Project. Incluir dependencias a Spring Boot DevTools y Spring Web.
Se creará una clase con el método main que es el punto de entrada para la aplicación.

```
@SpringBootApplication
public class RestServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(RestServiceApplication.class, args);
    }

}
```

SpringApplication.run(RestServiceApplication.class, args): Esta línea es la que arranca la aplicación Spring Boot. Es crucial porque se encarga de inicializar el contexto de Spring, configura los componentes automáticamente y arranca el servidor web embebido, todo a partir de una sola línea de código. Esto simplifica enormemente el proceso de inicio de una aplicación Spring.

2.- Crear un paquete controllers y dentro, una clase controlador: HolaController. La clase debe tener la anotación **@RestController**.

En la clase vamos a crear un método que imprima un mensaje. Para mapear la solicitud del cliente de tipo GET con dicho método, tenemos que utilizar la anotación **@GetMapping** indicando la URL con el que vamos a acceder a él.

```
@RestController
public class HolaController {

    @GetMapping("/hola")
    String saludo(){
    return "Bienvenido!!!";
    }

}
```

Cuando recibe una petición GET en /hola, el método saludo escribe el resultado en el cuerpo de la respuesta.

Ejecutamos la aplicación Spring Boot y visualizamos en el navegador: localhost:8080/hola

Enrutamiento en Spring Boot

Es el proceso mediante el cual Spring Boot decide qué método del controlador ejecutar en respuesta a una solicitud HTTP específica. Para ello se usan las siguientes anotaciones:

- ✔ **@GetMapping**: Mapea las solicitudes HTTP GET
- ✔ **@PostMapping**: Mapea las solicitudes HTTP POST
  La solicitud lleva datos incluidos en el cuerpo de la petición. Es necesaria la anotación:
  - ✔ @**RequestBody** indica que un objeto se construirá a partir de la información enviada en el body del request.

```
@PostMapping("/clientes")
public Cliente postCliente(@RequestBody Cliente cliente) {
```

- ✔ **@PutMapping**: Mapea las solicitudes HTTP PUT
  Se usa para actualizar un recurso con los datos proporcionados en el cuerpo de la solicitud.

```
@PutMapping("/clientes")
public Cliente putCliente(@RequestBody  Cliente modif) {
```

- ✔ **@DeleteMapping**: Mapea las solicitudes HTTP DELETE
  Se utiliza para eliminar un recurso existente.

```
@DeleteMapping("/clientes/{id}")
public Cliente deleteCliente(@PathVariable int id) {
```

- ✔ **@PatchMapping:** Mapea solicitudes HTTP de tipo PATCH a métodos que se encargan de realizar la modificación parcial de recursos sin necesidad de enviar de manera completa el recurso. A diferencia de PUT, que reemplaza por completo el recurso, PATCH permite actualizar sólo algunos campos.

- ✔ **@RequestMapping:** Se puede aplicar tanto a nivel de clase como a nivel de método.

  - ✔ Si se usa a nivel de clase, proporciona una ruta base común para todos los métodos. Unifica el prefijo para todos los endpoints.

```
@RestController
@RequestMapping("/clientes")
public class ClienteRequestMappingController {
```

  - ✔ Si se usa a nivel de método, especifica la ruta y el método de la petición en cada uno.

```
@RequestMapping(method= RequestMethod.GET)
public List<Cliente> getClientes(){
    return clientes;
}
```

  Si el endpoint necesita algún parámetro hay que especificarlo con el atributo value y además el tipo de petición:

```
@RequestMapping(value="/{username}", method= RequestMethod.GET)
public Cliente getCliente (@PathVariable String username) {
```

¿Cómo identifica Spring Boot si mapeamos dos endpoints con la misma ruta base?
Utiliza patrones adiciones a la ruta de acceso base, como parámetros de ruta o extensiones.
Ejemplos:

localhost:8080/**clientes**
localhost:8080/**clientes**/username

Para hacer las pruebas de los endpoints: Postman

La clase ResponseEntity

Se utiliza para representar toda la respuesta HTTP, incluyendo el cuerpo, los encabezados y el estado. Es muy conveniente para poder controlar cómo se construyen y devuelven las respuestas HTTP.

Esta clase nos permite devolver tanto el contenido generado por el servicio así como dar información adicional como el código de estado de HTTP.

Los códigos de respuesta son códigos numéricos estándar que los servidores web envían a los clientes para indicar el estado de la solicitud. Ayudan a entender si la solicitud fue exitosa, tuvo errores o necesita alguna acción adicional.

Códigos de respuesta:

Código HTTP	Descripción	Uso típico
200	OK	Respuesta exitosa
201	Created	Recurso creado exitosamente
204	No Content	Respuesta exitosa sin contenido
400	Bad Request	Solicitud no válida
401	Unauthorized	Falta de autenticación o credenciales inválidas
403	Forbidden	Acceso denegado
404	Not Found	Recurso no encontrado
405	Method Not Allowed	Método no permitido para el recurso
409	Conflict	Conflicto en la solicitud
422	Unprocessable Entity	Entidad no procesable
500	Internal Server Error	Error interno del servidor

Para usarlos, los endpoints deben devolver un objeto de tipo ResponseEntity.

- **ResponseEntity.ok()**: Si el endpoint devuelve la información con éxito: 200
- **ResponseEntity.notFound().build()** : Si devuelve un 404, recurso no encontrado.
- **ResponseEntity.noContent().build() :** Si el endpoint es exitoso pero no debe devolver datos, por ejemplo en el caso de las peticiones PUT, POST, DELETE, PATCH.

RECORDATORIO

List:

```
List<String> listaCadenas = new ArrayList<>();
```

Métodos de List:

```
listaCadenas.add("Hola");  //añade elementos al final de la lista
listaCadenas.add(2, "Hola"); // añade la cadena "Hola" en la posición 2 de la lista

listaCadenas.set(1,"Adios"); // sustituye y coloca la cadena "Adios" en la posición 1

String cad= listaCadenas.get(0); // obtiene el elemento de la posición 0

listaCadenas.remove(1); //Elimina el elemento que están en la posición 1 de la lista
listaCadenas.remove(objeto); //Elimina el primer elemento objeto de la lista

listaCadenas.clear(); //borra toda la lista.

Integer tam = listaCadenas.size(); // tamaño de la lista
Boolean esVacia= listaCadenas.isEmpty(); // si esta vacia
Boolean existe= listaCadenas.contains(objeto);  // si la lista contiene ese objeto
Integer primer = listaCadenas.indexOf(objeto); // primera posición del objeto en la lista
Integer ult = listaCadenas.lastIndexOf(objeto); // última posición del objeto en la lista
```

IMPORTANTE: Si queremos borrar elementos de una lista a la vez que la estamos recorriendo, es obligatorio utilizar un iterador. Si no, tendremos errores.

Set:

```
Set<String> conjunto = new HashSet<>();
```

Como no hay posiciones, no puedo pedirle que me dé el objeto que está en la posición i-ésima. Sólo podré obtener los objetos de un conjunto recorriéndolo.

Map:

```
Map<String, Integer> mapa = new HashMap<>();
mapa.put("A" , 1);    // inserto en el mapa
mapa.remove("A"); //borro el par cuya clave es "A"
mapa.get("A");        //obtiene el valor de la clave "A"
```

También podemos usar el método getOrDefault() al que le pasamos dos parámetros: por un lado la clave del valor que queremos obtener; y, por otro, el valor por defecto que queremos que nos devuelva si la clave no está en el mapa.

Métodos de Map:

Saber si el mapa está vacío: isEmpty()
Vaciar el mapa: clear()
Saber si el mapa contiene una clave: containsKey()
Saber si el mapa contiene un valor determinado: containsValue()

Recorrer Mapa:

```
Set<String> keys = mapa.keySet();
for (String key : keys) {
    System.out.println("Clave : "  + key);
    System.out.println("Valor: " + mapa.get(key));
}
```

```
Collection<Alumno> values = mapa.values();
for (Alumno alumno : values) {
    System.out.println(alumno);
}
```

```
Set<Entry<String, Alumno>> pares = mapa.entrySet();
for (Entry<String, Alumno> par : pares) {
    System.out.println("Clave: " + par.getKey());
    System.out.println("Valor: " + par.getValue());
}
```

LocalDate

Para crear un objeto que sea una fecha tenemos estas dos opciones:

1. Crear una fecha con la fecha de hoy:

```
LocalDate fecha = LocalDate.now();
```

2. Crear una fecha para un momento determinado:

```
LocalDate fecha = LocalDate.of(2021, Month.APRIL, 1); // 01/04/2021
```

También podríamos haber escrito el número del mes:

```
LocalDate fecha = LocalDate.of(2021, 4, 1); // 01/04/2021
```

Si indicamos unos valores que no son una fecha correcta, nos lanzará un error.

¿Cómo obtenemos el tiempo que hay entre dos fechas y compararlas?

Para ello usamos el método **until**() que nos devolverá un objeto de tipo **Period**. Será el periodo transcurrido entre las dos fechas. A ese periodo, podemos luego solicitarle que nos dé los días, meses o años. IMPORTANTE: al llamar a until(), debemos hacerlo sobre la fecha más antigua, si lo hacemos al revés, el periodo será negativo.

```
LocalDate fecha = LocalDate.now();
LocalDate fechaAnterior = LocalDate.of(2021, Month.APRIL, 1);

Period periodo = fechaAnterior.until(fecha);

Integer añosDiferencia = periodo.getYears();
Integer mesesDiferencia = periodo.getMonths();
Integer diasDiferencia = periodo.getDays();
```

Hay que tener en cuenta que los meses de diferencia, por ejemplo, no es el total de meses. Si comparo el 01/01/2000 con 02/02/2001, el periodo será 1 año + 1 mes + 1 día. Es decir, los meses que me devuelve el periodo son 1. Si quiero saber el total de meses de diferencia, tendrá que sumarle a esos meses la cantidad de años multiplicada por 12. Con los días totales habría que hacer algo similar.

Ejercicios

Ejercicio 1. Crear una aplicación utilizando Spring Boot (incluye la dependencia Spring Web) que permita a los usuarios verificar si una palabra ingresada es un palíndromo o no. Es decir, que se lee igual de derecha a izquierda o de izquierda a derecha, por ejemplo: radar es palíndroma.
Crear un paquete controller con una clase java PalindromoController.

El endpoint será GET: validar-palindromo/'palabra' ,donde 'palabra' es un parámetro que se pasa por URL.
Debe devolver un mensaje indicando si la palabra es un palíndromo o no. Nombra adecuadamente tus clases y métodos. Puedes comentar código cuando sea necesario.

Ejemplo de llamada:
localhost:8080/validar-palindromo/radar → La palabra radar es un palíndromo
localhost:8080/validar-palindromo/hola → La palabra radar no es un palíndromo

Ejercicio 2. Crear una Api REST que devuelva una lista de clientes a la siguiente petición de tipo GET: localhost:8080/clientes. Crear un paquete con el nombre controller y otro con el nombre modelo.
Partir con una lista de 4 clientes.

- Dentro del paquete modelo, crear una clase Cliente con los siguientes atributos: id, nombre, username y password.
- Dentro del paquete controller, añadir otro endpoint para obtener los datos de un cliente a partir del username. La petición es: localhost:8080/clientes/username

Ejercicio 3. Crear un nuevo endpoint en el controlador anterior para añadir un nuevo cliente a la lista de clientes. Utilizar una petición de tipo POST. El método debe devolver el nuevo cliente insertado. La petición es: localhost:8080/clientes

Ejercicio 4. Crear un nuevo endpoint en el controlador anterior para modificar los datos de un cliente de la lista de clientes. Utilizar una petición de tipo PUT. El método debe recibir el cliente y en caso de encontrar el id en la lista de clientes, actualizar sus datos. Devolver el nuevo cliente modificado. La petición es: localhost:8080/clientes
Crear otro endpoint para que reciba el id del cliente a modificar, en el path: /clientes/123

Ejercicio 5. Crear un nuevo endpoint para eliminar un cliente de la lista de clientes. El método debe recibir el id de un cliente. Una vez eliminado, debe devolver el cliente eliminado. La petición es: localhost:8080/clientes

Ejercicio 6. Crear un nuevo endpoint para actualizar sólo los datos que se envíen de un cliente. El método debe recibir un cliente sólo con los datos que queramos modificar. Una vez modificado, debe devolver el cliente. La petición es: localhost:8080/clientes.

Ejercicio 7. Actualizar el controlador ClienteController, utilizando @RequestMapping a nivel de clase y a nivel de método. ¿Qué ruta base tendrán todos los endpoints?
¿Qué otros cambios a nivel de métodos tenéis que hacer?

Ejercicio 8. Actualizar el controlador ClienteController, utilizando @RequestMapping sólo a nivel de clase. ¿Qué cambios a nivel de métodos tenéis que hacer?

Ejercicio 9. Desarrollar una aplicación utilizando Spring Boot que permita gestionar una lista de alumnos mediante un CRUD. La aplicación de proporcionar endpoints RESTful para:

- ✔ Mostrar todos los alumnos
- ✔ Consultar un alumno por su email
- ✔ Crear un nuevo alumno
- ✔ Modificar la información de un alumno tanto de manera parcial como total
- ✔ Eliminar un alumno por su id.

Las rutas de todos los endpoints deberán estar unificadas.
Tenéis que usar una lista inicialmente con 4 alumnos guardando para ellos: id, nombre, email, edad y curso (String).

Ejercicio 10. Crear otro controlador en el ejercicio anterior para que los endpoints devuelvan también información del estado de las peticiones utilizando la clase ResponseEntity.

Ejercicio 11. Añadir un campo a la clase Alumno que sea de tipo Direccion. Dirección es una clase en la que guardaremos los atributos calle, código postal y ciudad, todos de tipo String.
Prueba los métodos, ¿funcionan? Si tuvieras que adaptar algún o algunos métodos, cuales y cómo lo harías?

- ✓ Añade un método getDirecciones, que devuelva la lista de todas las direcciones de todos los alumnos.
- ✓ Añade un método obtenerDireccionesPorCodigoPostal, que devuelva la lista de todas las direcciones de los alumnos que coincidan con un código postal que se pasa por URL.
- ✓ Añade un método contarAlumnosPorCiudad, que dada una ciudad, devuelva el número de alumnos de dicha ciudad.

Ejercicio 12. Crear una aplicación que gestione una lista de libros. Los endpoints permitirán realizar las siguientes operaciones CRUD (Crear, Leer, Actualizar y Eliminar):

- ✓ Mostrar todos los libros
- ✓ Consultar un libro por su título
- ✓ Crear un nuevo libro
- ✓ Modificar la información de un libro tanto de manera parcial como total
- ✓ Eliminar un libro por su ID
- ✓ ObtenerNovelas: Devuelve una lista de los libros cuyo genero sea novela.
- ✓ ObtenerPorGenero: Dado un atributo género que se pasa por URL, devolver el listado de libros que sean de dicho género.
- ✓ ObtenerAutoresConMasDeXLibros: devuelve un mapa de <String,Integer>, donde la clave es el autor, y el valor, el numero de libros que ha escrito a partir del atributo numLibro que se pasará por la URL.

Los libros tendrán los siguientes atributos: id, titulo, autor, editorial, isbn, añoPublicacion y géneros. Donde géneros es una lista de String.

Ejercicio 13. Crear una aplicación Spring Boot que permita gestionar una lista de tareas. Cada tarea tendrá un identificador único, un título, una descripción, una fecha de vencimiento y un estado. Los estados pueden ser: "PENDIENTE", "EN_PROCESO" o "COMPLETA".

Se pide, obtener los siguientes CRUD endpoints:

- ✓ **Mostrar todas las tareas**: GET /tareas - Devuelve una lista de todas las tareas.
- ✓ **Consultar una tarea por ID**: GET /tareas/{id} - Devuelve una tarea específica por ID.
- ✓ **Crear una nueva tarea**: POST /tareas - Crea una nueva tarea y la añade a la lista.
- ✓ **Actualizar una tarea**: PUT /tareas/{id} - Actualiza completamente la tarea especificada por ID.
- ✓ **Actualizar parcialmente una tarea**: PATCH /tareas/{id} - Actualiza parcialmente la tarea especificada por ID.
- ✓ **Eliminar una tarea**: DELETE /tareas/{id} - Elimina la tarea especificada por ID.
- ✓ **Obtener tareas por estado**: GET /tareas/estado/{estado}
- ✓ **Obtener Tareas Próximas a Vencer**: GET proximas/{dias}: devuelve la lista de tareas próximas a vencer en los próximos X días.
- ✓ **Contar Tareas por Estado**: GET /contar-estado", que devuelve un mapa Map<String, Integer> donde la clave es el estado y el valor el número de tareas en dicho estado
- ✓ **Obtener Tareas por Palabra Clave en Descripción:** /buscar/{palabraClave}
- ✓ **Marcar tareas como completadas:** PATCH /marcar-completadas: Todas las tareas vencidas, aquellas cuya fecha ya ha pasado, deben marcarse como COMPLETA

Ejercicio 14 Crear una aplicación para gestionar una biblioteca de películas. Cada película consta de: un id, titulo, director, fecha de lanzamiento, duración (en minutos), y una lista de actores. De cada actor guardaremos su id, nombre y nacionalidad. Se pide obtener los siguientes CRUD endpoints:

- ✓ Mostrar todas las películas
- ✓ Consultar una película por su título
- ✓ Crear una nueva película
- ✓ Modificar la información de una película de manera parcial y total
- ✓ Eliminar una película por su id
- ✓ Obtener todas las películas por un director específico
- ✓ Obtener todas las películas de los últimos 5 años
- ✓ Obtener la película con la mayor duración
- ✓ Obtener un mapa con los directores con más de X películas, donde la clave es el nombre del director y el valor, el número de películas.
- ✓ Obtener la lista de todos los actores sin repetir ninguno. ¿Qué estructura de datos podemos usar?
- ✓ ObtenerPeliculasPorActor. Este método obtiene una lista de todas las películas en las que ha participado un actor específico, dado su nombre.
- ✓ ObtenerActoresPorNacionalidad. Este método obtiene todos los actores que tienen una nacionalidad específica dada por parámetro.

Ejercicio 15. Crear una aplicación para gestionar una tienda de música. Esta tienda tendrá álbumes y cada álbum contendrá varias canciones. Cada canción puede tener múltiples artistas y cada artista puede aparecer en múltiples canciones.
De cada álbum guardaremos el id, el título, el artista principal, el año de lanzamiento y una lista de las canciones del mismo. De cada canción, su id, titulo y lista de artistas. De cada artista conocemos su id, su nombre y su nacionalidad.

Se pide obtener los siguientes CRUD endpoints:

- ✓ Mostrar todos los álbumes.
- ✓ Consultar un álbum por su título.
- ✓ Crear un nuevo álbum.
- ✓ Modificar la información de un álbum de manera parcial y total.
- ✓ Eliminar un álbum por su id.
- ✓ Obtener todas las canciones de un álbum específico.
- ✓ Obtener todas las canciones de un artista específico.
- ✓ Obtener todos los artistas de una canción específica.
- ✓ Obtener la lista de todos los artistas sin repetir ninguno.
- ✓ ObtenerArtistasConNumeroDeCanciones. Debe devolver un mapa donde la clave es el nombre del artista y el valor el número de canciones en las que ha participado.