



5

# UNIDAD 5: API REST - SPRING BOOT

# 1. API REST

## 1.1. Introducción

Imagina que de repente te transportan a una ciudad extranjera donde no hablas el idioma; de hecho, cada persona que encuentras habla un idioma diferente, y ni siquiera estás seguro de cuál es. Esa es la situación que enfrentan muchos desarrolladores y usuarios hoy en día cuando intentan integrar diferentes software y sistemas.

Uno de los mayores desafíos de la informática moderna es su complejidad. Con millones de aplicaciones de software, servicios y sistemas diferentes actualmente en uso, cada uno habla su propio "idioma". ¿Qué pasa entonces cuando abordamos el problema del intercambio de información? La respuesta, para muchas organizaciones, es usar una API REST. Pero, ¿qué es exactamente una API REST y por qué es necesaria una?

## 1.2. ¿Qué es una API REST?

Una API, o interfaz de programación de aplicaciones, es un conjunto de reglas que definen cómo las aplicaciones o dispositivos pueden conectarse y comunicarse entre sí. Una API REST es una API que se ajusta a los principios de diseño de REST, o estilo arquitectónico de transferencia de estado de representación. Por esta razón, las API REST a veces se denominan API RESTful.

Definido por primera vez en 2000 por Roy Fielding en su tesis doctoral, REST proporciona un nivel relativamente alto de flexibilidad y libertad para los desarrolladores. Esta flexibilidad es solo una de las razones por las que las API REST han surgido como un método común para conectar componentes y aplicaciones.

## 1.3. Principios de diseño REST

En el nivel más básico, una API es un mecanismo que permite que una aplicación o servicio acceda a un recurso dentro de otra aplicación o servicio. La aplicación o servicio que realiza el acceso se denomina cliente, y la aplicación o servicio que contiene el recurso se denomina servidor.

Algunas API, como SOAP o XML-RPC, imponen un framework estricto a los desarrolladores. Pero las API REST se pueden desarrollar utilizando prácticamente cualquier lenguaje de programación y admiten una variedad de formatos de datos. El único requisito es que se alineen con los siguientes seis principios de diseño REST, también conocidos como restricciones arquitectónicas:

- 1. Interfaz uniforme.** Todas las solicitudes de API para el mismo recurso deben tener el mismo aspecto, sin importar de dónde provenga la solicitud. La API REST debe garantizar que el mismo dato, como el nombre o la dirección de correo electrónico de un usuario, pertenezca a un solo identificador uniforme de recursos (URI). Los recursos no deben ser demasiado grandes, pero deben contener toda la información que el cliente pueda necesitar.

2. **Desacoplamiento cliente-servidor.** En el diseño de API REST, las aplicaciones de cliente y servidor deben ser completamente independientes entre sí. La única información que debe conocer la aplicación cliente es el URI del recurso solicitado; no puede interactuar con la aplicación del servidor de ninguna otra manera. De manera similar, una aplicación de servidor no debe intervenir en la aplicación de cliente más allá de pasarle los datos solicitados a través de HTTP.
3. **Sin Estado.** Las API REST no tienen estado, lo que significa que cada solicitud debe incluir toda la información necesaria para procesarla. En otras palabras, las API REST no requieren ninguna sesión del lado del servidor. Las aplicaciones de servidor no pueden almacenar ningún dato relacionado con una solicitud de cliente.
4. **Capacidad de almacenamiento en caché.** Cuando sea posible, los recursos deben almacenarse en caché en el lado del cliente o del servidor. Las respuestas del servidor también deben contener información sobre si se permite el almacenamiento en caché para el recurso entregado. El objetivo es mejorar el rendimiento del lado del cliente, al tiempo que aumenta la escalabilidad del lado del servidor.
5. **Arquitectura del sistema en capas.** En las API REST, las llamadas y las respuestas pasan por diferentes capas. Como regla general, no se debe asumir que las aplicaciones cliente y servidor se conectan directamente entre sí. Puede haber varios intermediarios diferentes en el ciclo de comunicación. Las API REST deben diseñarse de modo que ni el cliente ni el servidor puedan saber si se comunica con la aplicación final o con un intermediario.
6. **Código bajo demanda (opcional).** Las API REST generalmente envían recursos estáticos, pero en ciertos casos, las respuestas también pueden contener código ejecutable (como applets de Java). En estos casos, el código solo debe ejecutarse bajo demanda.

#### 1.4. Cómo funcionan las API REST

Las API REST se comunican a través de solicitudes HTTP para realizar funciones de base de datos estándar, como crear, leer, actualizar y eliminar registros (también conocido como CRUD) dentro de un recurso. Por ejemplo, una API REST usaría una solicitud GET para recuperar un registro, una solicitud POST para crearlo, una solicitud PUT para actualizar un registro y una solicitud DELETE para eliminarlo.

Todos los métodos HTTP se pueden usar en llamadas API. Una API REST bien diseñada es similar a un sitio web que se ejecuta en un navegador web con funcionalidad HTTP integrada.

El estado de un recurso en cualquier instante particular, o marca de tiempo, se conoce como representación del recurso. Esta información se puede entregar a un cliente en prácticamente cualquier formato, incluida la notación de objetos de JavaScript (JSON), HTML, XML, Python, PHP o texto sin formato. JSON es popular porque es legible tanto por humanos como por máquinas, y es independiente del lenguaje de programación.

Los encabezados y parámetros de las solicitudes también son importantes en las llamadas a la API REST porque incluyen información de identificación importante, como metadatos, autorizaciones, identificadores uniformes de recursos (URI), almacenamiento en caché, cookies y más. Los encabezados de solicitud y los encabezados de respuesta, junto con los códigos de estado HTTP convencionales, se utilizan dentro de API REST bien diseñadas.

## 2. Spring

### 2.1. Introducción

Tal y como indica Wikipedia, "Spring Framework es un framework y contenedor de inversión de control para la plataforma Java". Cualquier aplicación Java puede utilizar las funciones principales del framework, pero existen extensiones para crear aplicaciones web sobre la plataforma Java EE (Enterprise Edition). Pero ¿Qué es la inversión de control (IoC)?

La inversión de control trata de eliminar las dependencias en el código. ¿Y qué es una dependencia? Supongamos que, en una parte del código, necesitamos usar una clase (por ejemplo, una de las EntidadEmpleado que definimos en la unidad anterior. Para el correcto funcionamiento del código debe haber entonces, en alguna parte, una inicialización de ese objeto:

```
EntidadEmpleado entidadEmpleado = new EntidadEmpleado();
```

Esta inicialización constituye la *dependencia*. Y es lo que se trata de evitar con la inversión de control, no depender de que el programador no olvide realizar determinadas tareas. Para ello, se realiza lo que se llama *inyección de dependencias*, esto es, será el propio Spring el que se encargue de la inicialización de todos los objetos que lo requieran.

### 2.2. Spring Boot

Spring Boot es la solución de Spring para crear aplicaciones basadas en Spring de desarrollo rápido. Está preconfigurado para que se pueda hacer uso de la plataforma Spring y las bibliotecas de terceros necesarias con el mínimo esfuerzo. La mayoría de las aplicaciones de Spring Boot necesitan muy poca configuración de Spring.

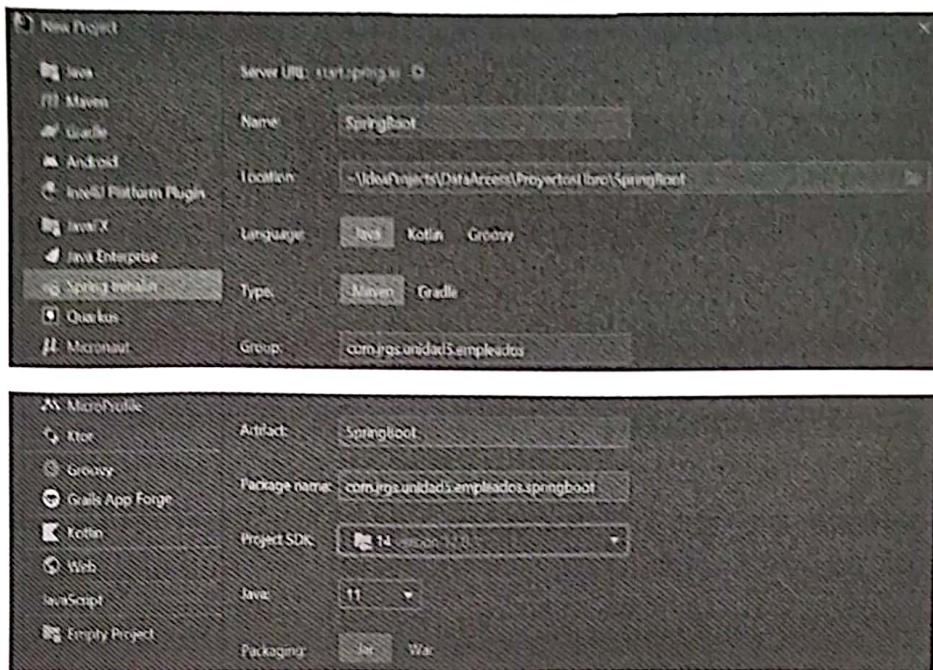
Entre sus características tenemos:

- Se pueden crear aplicaciones Spring totalmente independientes
- Se pueden incrustar directamente en Tomcat o Jetty (no es necesario implementar archivos WAR)
- Uso de ficheros POM para simplificar la configuración de Maven/Gradle
- Proporciona funciones listas para producción, como métricas, controles de estado y configuración externalizada
- No requiere generación de código ni requisitos para la configuración de XML.

## 3. Creación de una API REST con Spring Boot

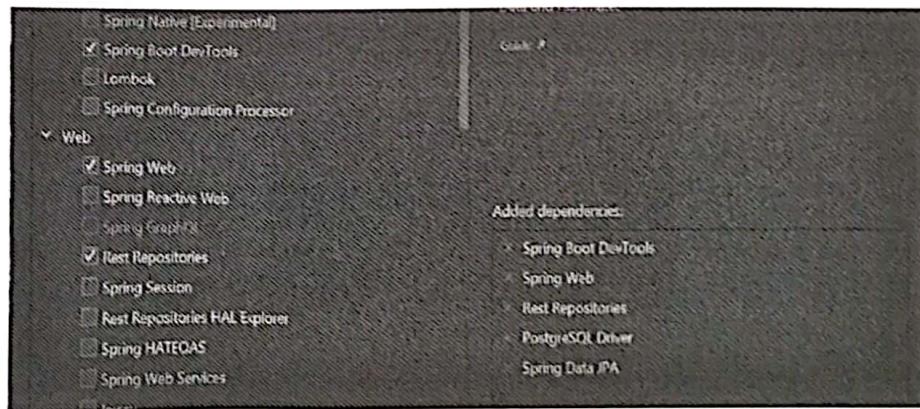
### 3.1. Creando el proyecto

La forma más fácil de crear un nuevo proyecto Spring Boot es usando Spring Initializr (<https://start.spring.io/>), que genera automáticamente un proyecto básico de Spring Boot. Se puede usar la página web o, alternativamente, la misma funcionalidad integrada en IntelliJ IDEA:



En la siguiente pantalla se deben seleccionar las dependencias a incluir en el proyecto. Para poder crear una API REST, debemos marcar las siguientes dependencias:

- Spring Web: para incluir Spring MVC y Tomcat integrado en su proyecto
- Spring Data JPA: API de persistencia de Java e Hibernate
- Spring Boot DevTools: herramientas de desarrollo muy útiles
- Controlador PostgreSQL: Controlador JDBC.
- Rest repositories: para crear de manera sencilla los objetos de acceso a datos (DAO's).



En el último paso aparecerá un tutorial en IntelliJ. Se puede cerrar con seguridad (o también leerlo, por supuesto).

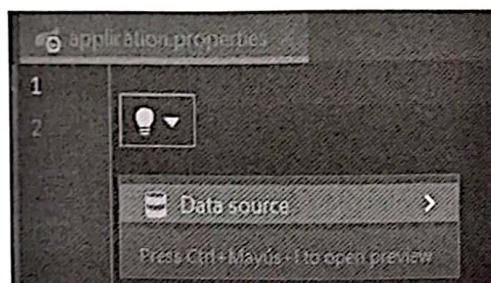
### 3.2. Conexión de Spring Boot a la base de datos

Antes de comenzar a trabajar en la aplicación, es conveniente configurar los parámetros de conexión a la base de datos. Esto se puede hacer fácilmente a través de Spring Data JPA, que nos permite configurar la conexión con solo un par de parámetros. En nuestro caso, vamos a utilizar nuestra base de datos de empleados, ya usada en anteriores unidades.

Para indicarle a Spring cómo conectar a la base de datos, en el archivo application.properties se debe agregar la información básica de la conexión:

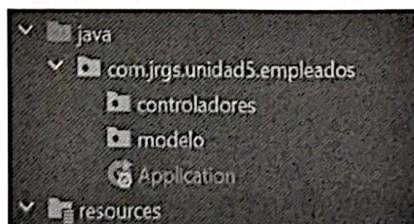
```
spring.datasource.url=jdbc:postgresql://localhost:5432/Empleados  
spring.datasource.username=postgres  
spring.datasource.password=postgres  
spring.datasource.driverClassName=org.postgresql.Driver  
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect  
spring.data.rest.base-path=/api-rest|
```

IntelliJ IDEA también puede agregar esta información automáticamente si el origen de datos se ha configurado previamente en la ventana *Database*:



### 3.3. Configuración del patrón MVC

Las aplicaciones Spring siguen el patrón MVC (arquitectura de modelo-vista-controlador) pero, de forma predeterminada, no se realizan en IntelliJ IDEA algunas operaciones necesarias requeridas por dicho patrón. Así que tendremos que crear una carpeta tanto para los controladores como para las clases del modelo. Las clases dentro la carpeta controladores tendrán el mismo sufijo. Las clases del modelo (POJOS) se pueden generar con Hibernate, agregando la faceta requerida, como vimos en la unidad 4.



### 3.4. Creación de clases de repositorio

Los repositorios de Spring intentan simplificar el acceso a nuestros datos. La forma más sencilla de comenzar es implementando `CrudRepository`, una interfaz que ofrece la funcionalidad básica para acceder a cada entidad de datos de nuestro modelo. En nuestro ejemplo, al tener dos entidades de datos (dos POJOS), tendremos dos repositorios. Estos repositorios también se denominan DAO (Data Access Objects, objetos de acceso a datos).

La forma más fácil de crear nuestros DAO es a partir de los POJOs de Hibernate, por lo que, en este punto (si no lo hemos hecho ya), necesitaremos agregar la faceta de Hibernate y crear automáticamente los POJOs (como vimos en la Unidad 4).

El objetivo principal de un DAO es realizar operaciones CRUD en nuestras entidades de datos. Para hacer esto, especificaremos una interfaz que extienda `CrudRepository` y la anotaremos con `@Repository`.

```
@Repository  
public interface IEmpleadosDAO extends CrudRepository<EntidadEmpleados, Integer> {  
}
```

Nuestro DAO será una interfaz que extiende el `CrudRepository`, indicándole a este último la entidad –clase– de datos (`EntidadEmpleados`) y el tipo de datos del campo clave (`Integer`).

`CrudRepository` declara métodos como `findAll()`, `findOne()` y `save()` que constituyen la funcionalidad CRUD básica de un repositorio genérico.

Se puede usar este `IEmpleadosDAO` tal cual, para realizar operaciones CRUD sobre la tabla `empleados`, sin necesidad de ninguna configuración adicional. Pero también es posible crear nuevos métodos sin apenas teclear una línea de código. Supongamos, por ejemplo, que además de localizar a un empleado por su `Id`, quisieramos localizarlo por el puesto de trabajo. Bastaría declarar el siguiente método:

```
EntidadEmpleados findByPuesto(String puesto);
```

Sin necesidad de declarar el cuerpo del método (Spring lo resolverá por nosotros usando JPA). También podríamos saber los empleados pertenecientes a departamentos por encima de un valor dado:

```
EntidadEmpleados findByDepnoGreaterThanOrEqual(int depno);
```

En este caso se han usado dos palabras clave (`GreaterThanOrEqual`) que indican que el valor debe ser igual o superior al especificado. Podemos encontrar la lista completa de modificadores que se pueden utilizar en el siguiente enlace:

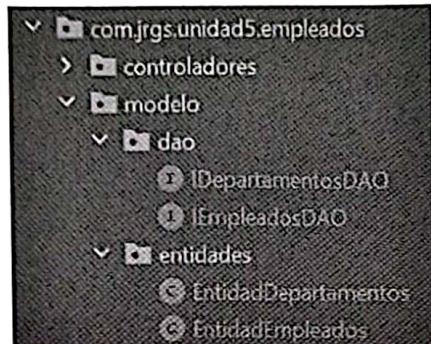
<https://docs.spring.io/spring-data/jpa/docs/2.2.2.RELEASE/reference/html/#jpa.query-methods.query-creation>

Por último, si no fuera posible construir usando estos modificadores la consulta que deseamos, se puede especificar directamente:

```
@Query("select e from EntidadEmpleados e where e.nombre like %:patron%")  
EmployeeEntity findByName(@Param("patron") String patron);
```

Aquí hemos usado la anotación `@Query` para indicar exactamente la consulta que queremos (búsqueda de nombre que contengan un patrón que pasamos como parámetro). Usamos también un parámetro con nombre (`:patron`) para pasar el valor a la consulta.

Una vez hayamos terminado de definir nuestro DAO, el aspecto actual de nuestro proyecto debería ser algo así (notar que en el modelo hemos creado carpetas separadas para DAO y entidades, aunque no es estrictamente necesario):



## Actividades

1. Agregar una opción de búsqueda por nombre a los departamentos.
2. Agregar una opción de búsqueda por ubicación a los departamentos.
3. Modificar los métodos anteriores (cambiando sólo el nombre del método) para ignorar mayúsculas y minúsculas.
4. Agregar un método de búsqueda de empleador por nombre, especificando una cadena como prefijo.
5. Agregar un método de búsqueda de empleado por puesto. El puesto ha de contener una subcadena que se suministrará como parámetro.

### 3.5. Creando el controlador

Finalmente, debemos crear un controlador, donde implementamos la lógica real de procesamiento de la información, y usamos los componentes de la capa de persistencia (repositorios) junto con el modelo para almacenar datos.

Nuestro objetivo es crear una API REST, por lo que marcaremos nuestro controlador como `@RestController`, y le agregaremos un `@RequestMapping`. `@RestController` es solo una combinación de otras dos anotaciones, `@Controller` y `@ResponseBody`, lo que significa que nuestro controlador, en lugar de mostrar páginas web, solo responderá con los datos que se hayan solicitado en el cuerpo de la página. Esto es la forma natural de comportamiento para las API REST: devolver información una vez que se ha alcanzado un *endpoint* de la API.

Avancemos y echemos un vistazo al controlador:

```
@RestController
@RequestMapping("/api-rest/empleados")
public class controladorEmpleados {

    @Autowired
    IEmpleadosDAO empleadosDAO;

    @GetMapping
    public List<EntidadEmpleados> buscarEmpleados() {
    }

    @GetMapping("/{id}")
    public ResponseEntity<EntidadEmpleados> buscarEmpleadoPorId(@PathVariable("value = "id") int id) {
    }

    @PostMapping
    public EntidadEmpleados guardarEmpleado(@Validated @RequestBody EntidadEmpleados empleado) {
    }
}
```

Hemos marcado nuestro `IEmpleadosDAO` con la anotación `@Autowired`. Esta anotación se usa para inyección de dependencia, ya que la clase de repositorio (el DAO) es una dependencia aquí (es decir, nuestro controlador depende del DAO para poder realizar su función). Spring se encargará automáticamente de inicializar (crear) aquellas clases que son dependencias (se puede apreciar en el código que no hemos creado el objeto asociado al DAO, lo hará Spring).

También hemos usado las anotaciones `@GetMapping` y `@PostMapping` para especificar qué tipos de solicitudes HTTP aceptan y manejan nuestros métodos. Estas son variantes derivadas de la anotación `@RequestMapping`, con un `method=RequestMethod.METHOD` establecido para cada uno de los tipos respectivos.

Comencemos con la implementación del *endpoint* `findAll()`:

```
@GetMapping("findAll")
public List<EntidadEmpleados> buscarEmpleados() {
    return (List<EntidadEmpleados>) empleadosDAO.findAll();
}
```

Este método simplemente llama a `empleadosDAO.findAll()` para encontrar todos los empleados y devuelve la lista completa como respuesta.

A continuación, veamos el *endpoint* para obtener un empleado por su identificador:

```
@GetMapping("findById/{id}")
public ResponseEntity<EntidadEmpleados> buscarEmpleadoPorId(@PathVariable(value = "id") int id) {
    Optional<EntidadEmpleados> empleado = empleadosDAO.findById(id);

    if(empleado.isPresent()) {
        return ResponseEntity.ok().body(empleado.get());
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

Se puede ver que la ruta del *endpoint* es la misma que para `buscarEmpleado`, pero agregando el `/ {id}`. En el caso de la búsqueda por clave, es posible que un empleado con la identificación dada no esté presente en la base de datos, por lo que envolvemos la `EntidadEmpleados` devuelta en un `Optional`. Luego, si `empleado.isPresent()` –es decir, el valor devuelto no es nulo–, devolvemos una respuesta HTTP 200 OK y establecemos la instancia de `EntidadEmpleados` como el cuerpo de la respuesta. De lo contrario, devolvemos una respuesta `ResponseEntity.notFound()` -HTTP 404-.

Finalmente, veamos el *endpoint* para guardar un nuevo empleado en la base de datos:

```
@PostMapping("guardar")
public EntidadEmpleados guardarEmpleado(@Validated @RequestBody EntidadEmpleados empleado) {
    return empleadosDAO.save(empleado);
}
```

El método `save()` del repositorio de empleados guarda un nuevo empleado si aún no existe. Si el empleado con la identificación dada ya existe, lanza una excepción. Si, por el contrario, la operación tiene éxito, devuelve el empleado insertado.

La anotación `@Validated` es un validador de los datos que proporcionamos sobre el empleado y aplica una validación básica. Si la información del empleado no es válida, los datos no se guardan. Además, la anotación `@RequestBody` asigna el cuerpo de la solicitud POST enviada al *endpoint* a la instancia de `EntidadEmpleados` que queremos guardar. En el apartado 3.8 veremos de manera más amplia la validación de datos.

## Actividades

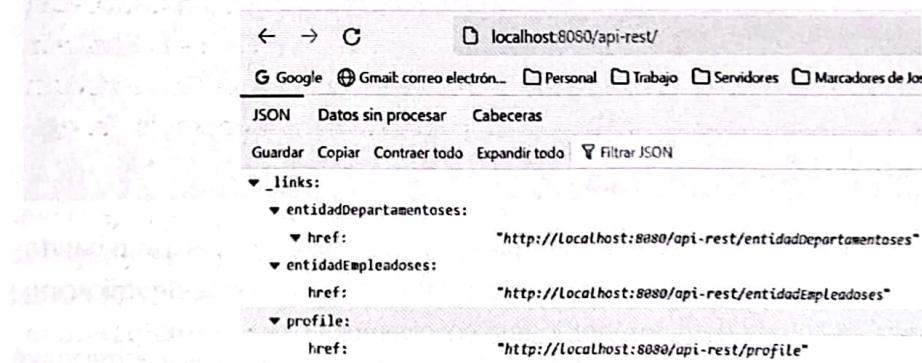
6. Crear el controlador correspondiente para los departamentos.
7. Partiendo de la base de datos de biblioteca/librería que desarrollamos con Hibernate en la unidad 4, realizar un API REST para la misma.

### 3.6. Ejecutar y probar la API REST

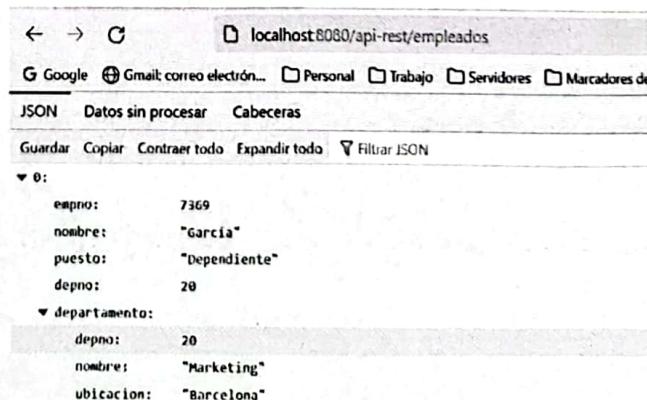
Una vez realizadas las tareas anteriores, deberíamos poder ejecutar con éxito nuestra API REST. Si todo está bien, IntelliJ IDEA informará que hay un servidor Tomcat escuchando en el puerto 8080:

```
: Initialized JPA EntityManagerFactory for persistence unit 'default'
: spring.jpa.open-in-view is enabled by default. Therefore, database qu...
: LiveReload server is running on port 35729
: Tomcat started on port(s): 8080 (http) with context path ''
: Started EmployeesApplication in 5.729 seconds (JVM running for 6.841)
```

Se puede comprobar abriendo un navegador web y escribiendo la URL: `localhost:8080/{nombre-de-la-api}`.



Vemos todos los *endpoints* disponibles en nuestra API REST, se pueden usar los enlaces proporcionados o escribirlos directamente:



En este punto, si observamos más detenidamente el conjunto JSON devuelto, notaremos que estamos ante un problema: como los empleados pertenecen a un departamento (dentro de la clase EntidadDepartamentos existe una lista empleados), al mostrar la información de un departamento, SPRING intenta mostrar también los empleados que pertenecen a él, entrando en un bucle infinito (ya que el empleado, a su vez, tiene el departamento al que pertenece).

Para solucionar esto, usamos la anotación `@JsonIgnoreProperties`. Con `@JsonIgnoreProperties`, le indicamos a Spring que no incluya una propiedad específica en el conjunto JSON final devuelto. Si queremos resolver el problema del bucle infinito, podemos hacer lo siguiente en la EntidadDepartamentos:

```
@OneToMany(mappedBy = "departamento")
@JsonIgnoreProperties("departamento")
public List<EntidadEmpleados> getEmpleados() { return empleados; }
```

Y también en la EntidadEmpleados:

```
@ManyToOne
@JoinColumn(name = "depno", referencedColumnName = "depno")
@JsonIgnoreProperties("empleados")
public EntidadDepartamentos getDepartamento() { return departamento; }
```

### 3.7. Completando nuestro CRUD

Hasta ahora, hemos implementado la opción Crear y Leer de nuestro repositorio CRUD. Para completarlo, también debemos implementar las opciones Actualizar y Eliminar. Para ello añadiremos la siguiente funcionalidad a nuestro controlador (agregando los *endpoints* correspondientes). Veamos en primer lugar el endpoint correspondiente al borrado. En este caso, se usará una petición DELETE:

```
@DeleteMapping("/{id}")
public ResponseEntity<?> borrarEmpleado(@PathVariable(value = "id") int id) {
    Optional<EntidadEmpleados> Employeexml = empleadosDAO.findById(id);
    if(Employeexml.isPresent()) {
        empleadosDAO.deleteById(id);
        return ResponseEntity.ok().body("Borrado");
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

La estructura básica es muy similar a la de la búsqueda por clave, ya que, en este caso, también es necesario especificar el empleado concreto a borrar.

El *endpoint* para actualización será una mezcla entre la búsqueda por clave y la inserción. En este caso, el tipo de petición será PUT.

```

@PutMapping("{" + id + "}")
public ResponseEntity<?> actualizarEmpleado(@RequestBody EntidadEmpleados nuevoEmpleado,
                                             @PathVariable(value = "id") int id) {
    Optional<EntidadEmpleados> empleado = empleadosDAO.findById(id);
    if(empleado.isPresent()) {
        empleado.get().setNombre(nuevoEmpleado.getNombre());
        empleado.get().setPuesto(nuevoEmpleado.getPuesto());
        empleado.get().setDepartamento(nuevoEmpleado.getDepartamento());
        empleadosDAO.save(empleado.get());
        return ResponseEntity.ok().body("Updated");
    } else {
        return ResponseEntity.notFound().build();
    }
}

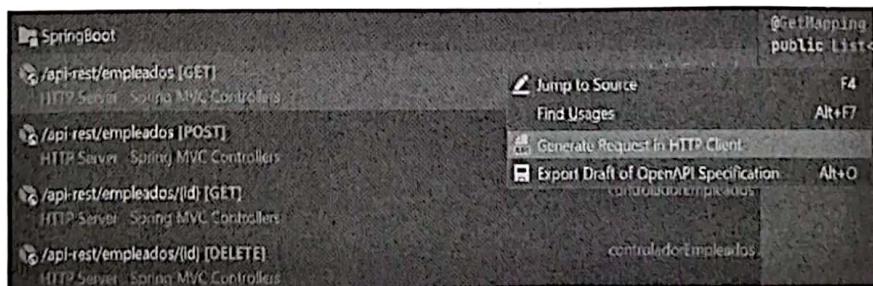
```

Podemos ver que, en el caso de la actualización, el *endpoint* consta de dos parámetros: uno que suministramos en la propia URL (el *id*) y otro que se envía, como en el caso de la inserción, en el cuerpo de la petición en formato JSON. Podemos ver también que en este caso, el único valor que no se actualiza es la clave primaria (lógico, por otra parte).

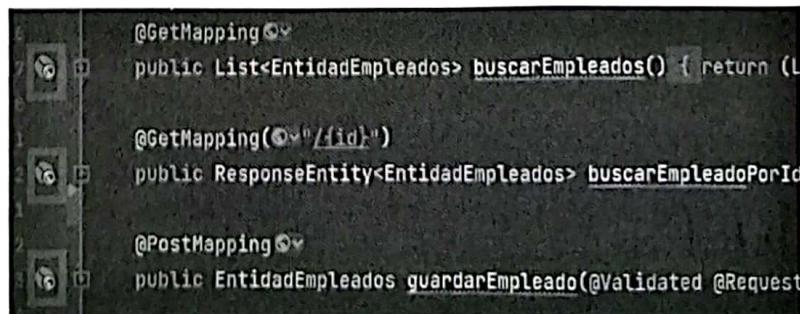
Para probar estos *endpoints*, tenemos dos opciones: usar la funcionalidad integrada de IntelliJ IDEA o usar una herramienta específica diseñada para probar la funcionalidad de un API REST, Postman. Se puede descargar Postman desde el siguiente enlace: <https://www.postman.com/downloads/>.

### Uso de la funcionalidad integrada de IntelliJ IDEA

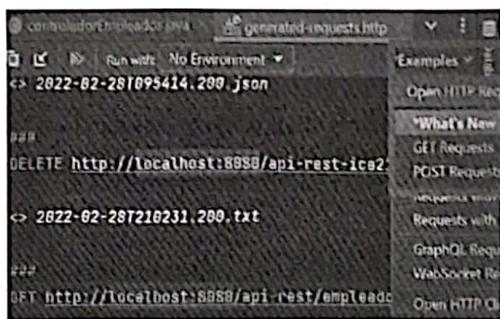
IntelliJ IDEA tiene todas las herramientas necesarias para probar nuestra API REST, aunque no es tan potente como Postman. Se puede verificar cuántos *endpoints* tiene disponibles su API REST en la pestaña *endpoints*.



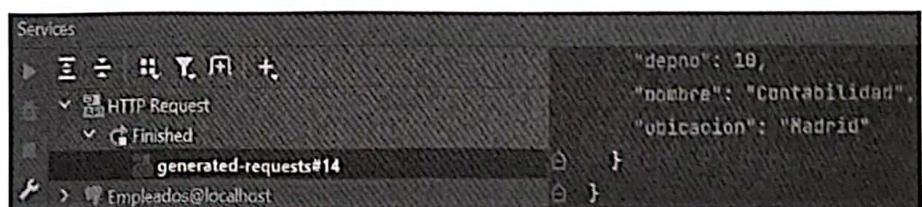
Como se puede ver en la imagen superior, las solicitudes HTTP se pueden generar directamente desde IntelliJ IDEA, ya sea usando la pestaña *endpoints* o directamente, desde el ícono de *endpoint* que se muestra en el código.



Las solicitudes se generan en un archivo llamado `generated-requests.http`. Este archivo es de fácil acceso (simplemente haciendo clic en los iconos) y permite al usuario ejecutar individualmente cada *endpoint*. A partir de la versión 2022.3 de IntelliJ IDEA, existe una opción *Examples* en esta ventana que permite generar la estructura básica de, por ejemplo, una petición POST que envía un JSON.

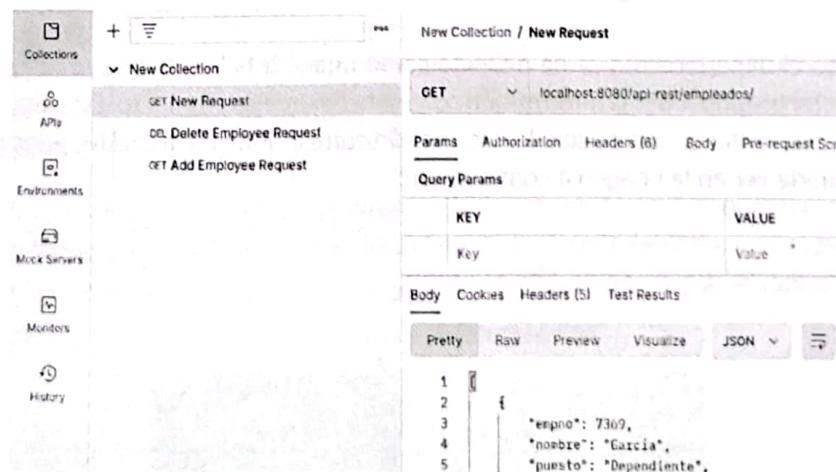


Los resultados de la ejecución de las peticiones se muestran en la pestaña *Services*.



### Usando Postman

Si se opta por utilizar Postman (es una herramienta muy popular para esta tarea), primero se debe crear (o acceder, si ya existe) un espacio de trabajo y, dentro de este, crear (e ingresar) una nueva colección. Dentro de una colección podemos crear tantas solicitudes como deseemos para probar nuestra API REST.



Antes de probar la solicitud `DELETE`, crearemos una solicitud para (también) probar la función de guardar. Necesitaremos agregar una solicitud `POST`, enviando la información en formato JSON:

New Collection / Add Employee Request

Sa

POST

localhost:8080/api-rest/empleados

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

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

```

1  {
2   ... "empno": 1234,
3   ... "nombre": "Diez",
4   ... "puesto": "Dependiente",
5   ... "departamento": [
6    ...   ... "depno": 20,
7    ...   ... "nombre": "Marketing",
8    ...   ... "ubicacion": "Barcelona"
9 }
```

Si hemos creado con éxito un nuevo empleado (lo sabremos si el cuerpo de la respuesta es el mismo JSON que hemos enviado), ahora podemos crear una nueva solicitud DELETE para probar si nuestra API puede ejecutar las eliminaciones con éxito:

DELETE

localhost:8080/api-rest/empleados/1234

Params Authorization Headers (6) Body Pre-request Script Tests Settings

Query Params

|  | KEY | VALUE |
|--|-----|-------|
|  | Key | Value |

Body Cookies Headers (5) Test Results

Pretty

Raw

Preview

Visualize

Text

1 Borrado

Se debe tener en cuenta lo siguiente:

- Como el departamento se ha definido como una entidad en la EntidadEmpleados, se debe proporcionar la información completa de un departamento. Este inconveniente se puede evitar enviando solo la información correspondiente a la clave principal, como se puede ver en la imagen a continuación:

```

2   ...
3   ...
4   ...
5   ...
6   ...
7   ...
8   ...
9   ...
10 }
```

- Para actualizar un registro usamos exactamente el mismo procedimiento que usamos para crear un nuevo registro, cambiando los datos deseados.

3. No es posible crear un nuevo departamento cuando creamos un empleado, el departamento debe estar previamente creado.

## Actividades

1. Probar la API REST de la biblioteca usando Postman

### 3.8. Uso de DTOs

Tal y como hemos visto en los puntos anteriores, si utilizamos la funcionalidad de Hibernate de usar, en lugar de las claves ajenas, una referencia a la clase que referencia dicha clave, podemos encontrarnos problemas tales como los bucles infinitos (que solucionamos mediante la anotación `@JsonIgnoreProperties`) o el tener que enviar en la petición, en vez de el valor de la clave ajena (el código del departamento, por ejemplo) la clase completa (aunque se pueda simplificar el envío dándole únicamente valor a la clave). Por ello es bastante habitual que las entidades sean un reflejo exacto de las tablas de la base de datos, sin incorporar ningún campo añadido, tal y como podemos ver a continuación.

```
@Entity  
@Table(name = "empleados", schema = "public", catalog = "Empleados")  
public class EntidadEmpleados {  
    private int empno;  
    private String nombre;  
    private String puesto;  
    private int depno;
```

Sin embargo, esto nos plantea otro problema ¿Qué hacemos en el caso de que necesitemos suministrar al usuario información de más de una tabla, o, simplemente, enviarle, no toda la tabla completa, sino sólo una parte de la misma? En Spring, para solucionar esto, disponemos de los DTOs (Data Transfer Object).

Un DTO no es más que un POJO (es decir, una clase con atributos y setters/getters) que dinámicamente se rellena con la información necesaria cuando se accede al *endpoint* correspondiente.

Imaginemos que estamos utilizando la entidad de la imagen anterior y que queremos incluir (tal y como hacímos previamente) la información de los departamentos con cada uno de los empleados (no sólo el número de departamento). Para ello, deberíamos definir, en primer lugar, el POJO que contendrá la información que necesitemos:

```
public class EmpleadosDTO {  
    private int empno;  
    private String nombre;  
    private String puesto;  
    private int depno;  
    private String departamentoNombre;  
    private String departamentoUbicacion;
```

Una vez tenemos el POJO creado (en este caso no son necesarias las anotaciones, ya que no equivale a ninguna tabla de la base de datos), crearemos un nuevo *endpoint*, con un mapeado concreto (`/dto/{id}`) por donde serviremos la información contenida en el DTO:

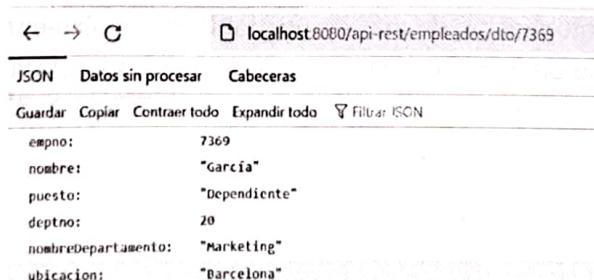
```
@GetMapping("/{id}")
public ResponseEntity<EmpleadosDTO> buscarEmpleadoDTOPorId(@PathVariable(value = "id") int id) {
    Optional<EntidadEmpleados> employee = empleadosDAO.findById(id);

    if(employee.isPresent()) {
        Optional<EntidadDepartamentos> departamento =
            departamentosDAO.findById(employee.get().getDepartamento().getDepno());

        EmpleadosDTO empleadosDTO = new EmpleadosDTO();
        empleadosDTO.setEmpno(employee.get().getEmpno());
        empleadosDTO.setNombre(employee.get().getNombre());
        empleadosDTO.setPuesto(employee.get().getPuesto());
        empleadosDTO.setDepno(employee.get().getDepartamento().getDepno());
        empleadosDTO.setDepartamentoNombre(departamento.get().getNombre());
        empleadosDTO.setDepartamentoUbicacion(departamento.get().getUbicacion());

        return ResponseEntity.ok().body(empleadosDTO);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

Como se puede ver, la estructura es similar a la del GET del empleado pero, una vez hemos recuperado la información del empleado, accedemos a la del departamento (mediante el DAO correspondiente que habremos añadido al controlador, `departamentosDAO`) y creamos un nuevo DTO con la información tanto del empleado como del departamento. Si ahora accedemos a nuestro nuevo *endpoint*, tendremos que obtener el siguiente resultado:



Sin embargo, en el caso de que nuestro DTO incluya muchos campos, el mapeado manual del mismo puede resultar un tanto tedioso. En este caso, puede ser buena idea hacer uso de la clase `ModelMapper`, la cual nos mapeará de manera automática los campos de la entidad en el DTO:

```
if(employee.isPresent()) {
    Optional<EntidadDepartamentos> departamento =
        departamentosDAO.findById(employee.get().getDepartamento().getDepno());

    ModelMapper mapper = new ModelMapper();
    EmpleadosDTO empleadosDTO = mapper.map(employee.get(), EmpleadosDTO.class);
    mapper.map(departamento.get(), empleadosDTO);

    return ResponseEntity.ok().body(empleadosDTO);
}
```

En este caso, y dado que nuestro DTO está formado por campos de dos tablas diferentes (empleados y departamentos), habremos de realizar dos mapeados, un primer mapeado de la entidad empleado, del cual obtenemos una instancia de EmpleadosDTO, y un segundo mapeado del departamento sobre dicha instancia. Dado que EntidadDepartamentos y EntidadEmpleados comparten un campo (el nombre del empleado y el nombre del departamento), puede producirse, al realizar el segundo mapeado el siguiente error:

|                        |               |
|------------------------|---------------|
| emno:                  | 7369          |
| nombre:                | "Marketing"   |
| puesto:                | "Dependiente" |
| depno:                 | 20            |
| departamentoNombre:    | "Marketing"   |
| departamentoUbicacion: | "Barcelona"   |

Como podemos ver, al tener el mismo nombre el campo empleados.nombre y departamento.nombre, el segundo mapeado provoca que se sobreescriba el nombre del empleado asignado en el primer mapeado. Esto se puede solucionar cambiando el nombre del campo en una de las dos entidades o agregando una regla de exclusión en el mapeado:

```
ModelMapper mapper = new ModelMapper();
EmpleadosDTO empleadosDTO = mapper.map(employee.get(), EmpleadosDTO.class);
mapper.typeMap(EntidadDepartamentos.class, EmpleadosDTO.class);
    addMappings( mapping -> mapping.skip(EmpleadosDTO::setNombre) );
mapper.map(departamento.get(), empleadosDTO);
```

Como podemos ver, después de realizar el primer mapeado, indicamos a la clase ModelMapper que no mapee el campo EmpleadosDTO.setNombre, evitando de esta manera la sobreescritura anterior.

### 3.9. Validando datos

Uno de los problemas a los que es muy probable que nos enfrentemos al desarrollar nuestra API REST es el relacionado con datos no válidos, es decir, un nombre vacío o un código o correo electrónico incorrecto, por ejemplo. ¿Cómo podemos solucionarlo?

Para afrontar la gestión de la validación de datos dentro de Spring, debemos añadir una dependencia a nuestro archivo pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Este módulo nos permitirá utilizar anotaciones para validar la información recibida por el API REST. Estas anotaciones están contenidas en el módulo javax.validation.constraints. Algun ejemplo de anotaciones podrían ser los siguientes:

```
@Basic
@NotEmpty(message="El nombre no puede estar vacío")
@Size(min = 2, max = 10, message = "El nombre tiene que tener entre 2 y 10 caracteres")
@Column(name = "nombre", nullable = true, length = 10)
public String getNombre() { return nombre; }
```

En el ejemplo anterior, estamos exigiendo que el campo nombre no sea vacío y que tenga una longitud mínima de 2 caracteres y máxima de 10. Algunas otras anotaciones útiles son:

- `@NotNull`: indica que un campo no debe ser nulo.
- `@NotEmpty`: para indicar que un campo de lista no debe estar vacío.
- `@NotBlank`: para decir que un campo de cadena no debe ser la cadena vacía (es decir, debe tener al menos un carácter).
- `@Min` y `@Max`: para indicar que un campo numérico solo es válido cuando su valor está por encima o por debajo de cierto valor.
- `@Pattern`: para decir que un campo de cadena solo es válido cuando coincide con una determinada expresión regular.
- `@Email`: para decir que un campo de cadena debe ser una dirección de correo electrónico válida.

La lista completa se puede encontrar aquí: <https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/package-summary.html>.

Una vez que hemos definido las restricciones a validar, debemos asegurarnos en nuestro controlador de que los valores proporcionados son realmente válidos. Este trabajo se realiza mediante el uso de las anotaciones `@Validated` y `@Valid`.

La anotación `@Validated` es una anotación a nivel de clase que podemos usar para decirle a Spring que valide los parámetros que se pasan a un método de la clase anotada. Podemos poner la anotación `@Valid` en los parámetros y campos del método para decirle a Spring que queremos que se valide un parámetro o campo de método concreto.

Nuestro método de guardar (punto 3.5, `guardarEmpleado`) se anotó previamente con `@Validated`, por lo que, si intentamos ejecutar la siguiente solicitud:

```
curl -X POST -H "Content-Type: application/json" -d '{ "empno": 1234, "nombre": "Agustín Diez de Ansuaga", "puesto": "Dependiente", "departamento": { "depo": 20, "nombre": "", "ubicacion": "" } }' http://localhost:8080/api/empleados
```

obtendremos una respuesta 400 (error) en formato JSON, dado que el nombre es demasiado largo, según nuestras restricciones:

```
{"defaultMessage": "El nombre tiene que tener entre 2 y 10 caracteres", "objectName": "claseEmpleados", "field": "nombre", "rejectedValue": "Agustín Diez de Ansuaga", "bindingFailure": false, "code": "Size"}
```

## Actividades

1. Teniendo en cuenta el diseño de la base de datos (tamaños de campo, etc), realizar las validaciones que se consideren necesarias para la base de datos de empleados y de biblioteca.

## 4. Consumir REST: lado del cliente

Hasta ahora hemos visto cómo configurar una API REST para entregar los datos en nuestro DBMS a través de un servidor web, utilizando solicitudes HTTP y JSON. Pero, ¿cómo consumimos estos datos?

### 4.1. formato de archivo JSON

JSON (JavaScript Object Notation) es un formato de archivo estándar abierto para compartir datos que utiliza texto legible por humanos para almacenar y transmitir datos. Los archivos JSON se almacenan con la extensión .json. JSON requiere menos formato y es una buena alternativa para XML. JSON se deriva de JavaScript, pero es un formato de datos independiente del lenguaje. Muchos lenguajes de programación modernos admiten la generación y el análisis de ficheros JSON. application/json es el tipo de medio utilizado para JSON.

Los datos JSON se escriben en pares clave/valor. La clave y el valor están separados por dos puntos (:) en el medio con la clave a la izquierda y el valor a la derecha. Los diferentes pares clave/valor están separados por una coma (,). La clave es una cadena entre comillas dobles, por ejemplo, "nombre". Los valores pueden ser de los siguientes tipos.

- Número
- Cadena: Secuencia de caracteres Unicode entre comillas dobles ("Coche").
- Booleano: Verdadero o Falso.
- Array (matriz): una lista de valores entre corchetes, por ejemplo

```
[ "Rojo verde azul" ]
```

- Objeto: una colección de pares clave/valor entre llaves, por ejemplo:

```
{"empno": 1234, "nombre": "Diez", "puesto" : "Dependiente"}
```

A continuación, se muestra el resultado JSON que debería devolver el endpoint departamentos/10:

```
{
  "depno":10,
  "nombre":"Contabilidad",
  "ubicacion":"Madrid",
  "empleados":[
    {
      "empno":7782,
      "nombre":" Sánchez",
      "puesto":" Responsable"
    },
    {
      "empno":7839,
      "nombre":" Peláez",
      "puesto":" Presidente"
    }
  ]
}
```

```
    },
    {
        "empno":7934,
        "nombre":"Hernández",
        "puesto":"Dependiente"
    }
]
```

En estos datos JSON, tenemos dos tipos de objetos (departamento y empleado) y un array (empleados).

## 4.2. Peticiones GET

Si nuestra API REST está integrada dentro de un sitio web, podemos desarrollar páginas dinámicas (usando, por ejemplo, AJAX) para interactuar con los usuarios. Sin embargo, también podemos usar aplicaciones de escritorio Java para consumir desde REST. En el siguiente ejemplo, tenemos una aplicación muy simple que recupera la lista de departamentos de nuestra API REST, utilizando bibliotecas Java tanto para realizar una solicitud HTTP como para manejar datos JSON.

```
import org.json.JSONArray;
import org.json.JSONObject;

import java.net.HttpURLConnection;
import java.net.URL;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        GetRequest();
    }

    public static void GetRequest() {
        HttpURLConnection conn = null;
        try {
            URL url = new URL("http://localhost:8080/" +
                               "api-rest/departamentos");
            conn = (HttpURLConnection) url.openConnection();
            conn.setRequestProperty("Accept", "application/json");

            if (conn.getResponseCode() == 200) {
                Scanner scanner = new Scanner(conn.getInputStream());
                String response = scanner.useDelimiter("\z").next();
                scanner.close();

                JSONArray jsonArray = new JSONArray(response);
                for (int i = 0; i < jsonArray.length(); i++) {

```

```
JSONObject jsonObject = (JSONObject)
                        jsonArray.get(i);
                    System.out.println(jsonObject.get("depno") + " "
                        + jsonObject.get("nombre"));
                }
            }
        else
            System.out.println("Fallo en la conexión.");
    }
    catch( Exception e ) {
        System.out.println( e.getMessage() );
    }
    finally {
        if ( conn != null )
            conn.disconnect();
    }
}
```

La función que se encarga de hacer la petición, `GetRequest()`, crea un objeto de tipo `HttpURLConnection` basándonos en el *endpoint* que queremos acceder. Si el acceso es correcto (código de respuesta 200), obtenemos el cuerpo de la página (`conn.getInputStream()`) y lo convertimos a cadena.

Las clases `JSONArray` y `JSONObject` permiten analizar los datos JSON devueltos. En el ejemplo anterior, obtenemos los datos obtenidos desde el *endpoint* `departments` (un array de departamentos), leyéndolos con un `JSONArray`. Cada departamento dentro del array se lee mediante un `JSONObject`. Como podemos deducir, debemos conocer previamente la estructura JSON que devuelve el *endpoint*. Si, por ejemplo, hubiéramos accedido al *endpoint* con identificador, directamente analizaríamos el resultado con un `JSONObject`.

Para poder utilizar estas clases, es necesaria la siguiente dependencia Maven:

```
<dependency>
    <groupId>com.vaadin.external.google</groupId>
    <artifactId>android-json</artifactId>
    <version>0.0.20131108.vaadin1</version>
</dependency>
```

### 4.3. Peticiones POST

Al realizar una solicitud POST, se debe enviar una información JSON a la API REST con nuestra solicitud. Por ejemplo, si queremos crear un nuevo empleado (como hicimos con Postman en el punto 3.7), tendremos que enviar la siguiente cadena JSON:

```
{
    "empno": 1234,
    "nombre": "Diez",
    "puesto": "Dependiente",
    "departamento": {
        "depno": 20,
        "nombre": "MARKETING",
```

```
        "ubicacion": "Barcelona"
    }
}
```

Para hacer esto, debemos agregar a nuestra conexión la cadena JSON, como se muestra a continuación:

```
public static void PostRequest() {
    HttpURLConnection conn = null;
    String jsonInputString = new JSONObject()
        .put("empno", 1234)
        .put("nombre", "Díez")
        .put("puesto", "Dependiente")
        .put("departamento", new JSONObject()
            .put("depno", 20)
            .put("nombre", "Marketing"))
        .put("ubicacion", "Barcelona")
        .toString()).toString();

    try {
        URL url = new URL("http://localhost:8080/api-rest/empleados");
        conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("POST");
        conn.setRequestProperty("Content-Type", "application/json; utf-8");
        conn.setRequestProperty("Accept", "application/json");
        conn.setDoOutput(true);

        try ( OutputStream os = conn.getOutputStream() ) {
            byte[] input = jsonInputString.getBytes("utf-8");
            os.write(input, 0, input.length);
        }

        if (conn.getResponseCode() == 200)
            System.out.println("Empleado insertado");
        else {
            System.out.println("Fallo en la conexión");

            Scanner scanner = new Scanner(conn.getErrorStream());
            String response = scanner.useDelimiter("\Z").next();
            scanner.close();

            JSONObject jsonObject = new JSONObject(response).
                getJSONArray("errors").getJSONObject(0);
            System.out.println(jsonObject.get("defaultMessage"));
        }
    } catch( Exception e ) {
        System.out.println( e.getMessage() );
    }
    finally {
        if ( conn != null )
            conn.disconnect();
    }
}
```

```
}
```

Como podemos ver, primero creamos un `JSONObject` con la información que queremos enviar al `endpoint`, y la cadena resultante la agregamos al `OutputStream` de la conexión, la cual, en este caso, tiene especificada de manera explícita que el tipo de conexión que se va a realizar es POST.

En caso de que se produjera un error (por un formato incorrecto, o por que no se haya podido validar la cadena JSON debido a que no cumple con las restricciones especificadas en los validadores) debería analizarse la respuesta (que, tal y como se ha comentado en el punto 3.9, será una cadena JSON) devuelta en el stream `conn.getErrorStream()`. En el ejemplo, simplemente se imprime el mismo error que hemos generado con el validador de Spring. A continuación se adjunta un ejemplo de objeto JSON de error devuelto por Spring (el campo `trace` se ha simplificado debido a su longitud):

```
{
  "timestamp": "2023-03-23T09:27:19.630+00:00",
  "status": 400,
  "error": "Bad Request",
  "trace": "org.springframework.web.bind.MethodArgumentNotValidException:  
Validation failed for argument [0] in public  
com.jrgs.unidad5.empleados.modelo.entidades.\r\n",
  "message": "Validation failed for object='entidadEmpleados'. Error count:  
1",
  "errors": [
    {
      "codes": [
        "Size.entidadEmpleados.nombre",
        "Size.nombre",
        "Size.java.lang.String",
        "Size"
      ],
      "arguments": [
        {
          "codes": [
            "entidadEmpleados.nombre",
            "nombre"
          ],
          "arguments": null,
          "defaultMessage": "nombre",
          "code": "nombre"
        },
        10,
        2
      ],
      "defaultMessage": "El nombre tiene que tener entre 2 y 10 caracteres",
      "objectName": "entidadEmpleados",
      "field": "nombre",
      "rejectedValue": "GARCIA SEVILLA",
      "bindingFailure": false,
    }
  ]
}
```

```
        "code": "Size"  
    },  
    "path": "/api-rest/empleados"  
}
```

#### 4.4. Petición DELETE

Finalmente, para poder eliminar usando nuestra API REST, podemos usar el siguiente código:

```
public static void DeleteRequest(String codeToDelete) {  
    HttpURLConnection conn = null;  
    try {  
        URL url = new URL("http://localhost:8080/" +  
                            "api-rest/empleados/" + codeToDelete);  
        conn = (HttpURLConnection) url.openConnection();  
        conn.setRequestMethod("DELETE");  
  
        if (conn.getResponseCode() == 200)  
            System.out.println("Empleado borrado");  
        else  
            System.out.println("Fallo en la conexión");  
    }  
    catch( Exception e ) {  
        System.out.println( e.getMessage() );  
    }  
    finally {  
        if ( conn != null )  
            conn.disconnect();  
    }  
}
```

Podemos apreciar que éste es el método más sencillo, ya que no tenemos que enviar más que el código del empleado a borrar, y basta con agregarlo directamente a la URL del *endpoint*.

#### 4.5. Actividades

1. Tomando como base los métodos anteriores, diseñar el método necesario para poder realizar una actualización (petición PUT).
2. Realizar una aplicación JAVA de terminal que, mediante un menú de opciones, permita realizar todas las operaciones CRUD que ofrece nuestra API REST, tanto con empleados como con departamentos.