Java Spring Boot

Unidad 4. Interacción entre microservicios

Índice

Introducción

Los microservicios son piezas de software que son consumidos por otras aplicaciones. Estas aplicaciones pueden ser a su vez otros microservicios. De hecho, la interacción entre microservicios es algo habitual en este tipo de arquitectura.

Spring hace posible realizar invocaciones desde un microservicio Spring a otro microservicio creado con cualquier otra tecnología. Este acceso puede realizar tanto de forma libre como de forma segura, pues Spring facilita la inserción de los credenciales en la cabecera de la petición.

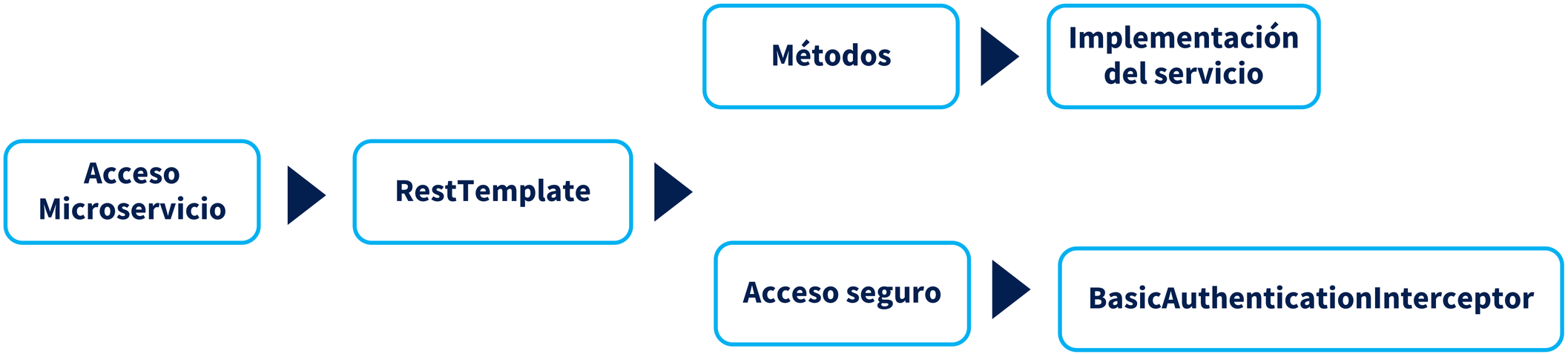
Objetivos

Comprender la necesidad de interacción entre microservicios.

Utilizar el objeto RestTemplate para acceder a un microservicio.

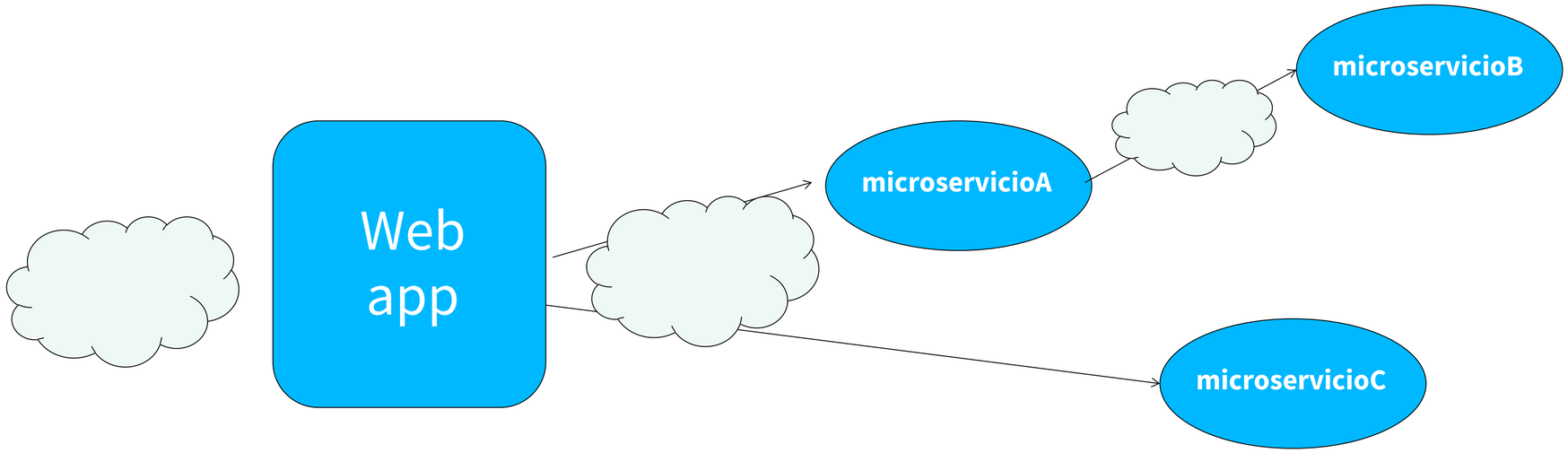
Aplicar las técnicas para acceder de forma segura a un microservicio.

Mapa Conceptual



Cliente de un microservicio

Un microservicio está pensado para ser consumido por otra aplicación. Esta otra aplicación es en muchas ocasiones otro **microservicio, o una aplicación Web tradicional que se encarga de realizar varias llamadas a distintos microservicios** para conseguir ofrecer cierta funcionalidad al cliente final.



A continuación, estudiaremos el **componente RestTemplate** proporcionado por Spring, que nos va a permitir realizar llamadas a microservicios desde cualquier aplicación Java.

RestTemplate

Fundamentos

La clase RestTemplate del paquete org.springframework.web.client, proporciona una serie de métodos para realizar llamadas a un servicio Rest desde una aplicación Java, independientemente de que se trate de un servicio Rest estándar o un microservicio.

Si vamos a utilizarlo en una aplicación Spring Boot, bastará con incluir el starter Web en el archivo pom.xml

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

RestTemplate

Instanciación

Un objeto RestTemplate se puede instanciar directamente desde el bloque de código en donde lo vamos a utilizar, empleando el constructor por defecto.

No obstante, lo habitual es **definir un método de configuración en el que se realice la instanciación del objeto e inyectarlo después** en los distintos puntos donde vaya a ser utilizado.

En el caso de un microservicio cliente, podemos definir el método de configuración en la propia clase principal, dado que esa clase ya es una clase de configuración (recuerda que @SpringBootApplication incluye la anotación @Configuration):

```
@Bean
   public RestTemplate getTemplate() {
      return new RestTemplate();
   }
```

Después, en la clase donde vayamos a utilizarlo, por ejemplo el controlador, lo inyectaremos en una variable atributo:

@Autowired

RestTemplate template;

RestTemplate

Métodos

La clase RestTemplate proporciona una amplia variedad de métodos con los que podemos realizar llamadas a servicios Rest remotos. Vamos a presentar los más habituales:

- **T getForObject(String url, Class<T> responseType, Object... uriVariables).** Realiza una petición GET a la url indicada en el primer parámetro, devolviendo el resultado en un objeto del tipo indicado en el segundo parámetro. El tercer parámetro representa la lista de variables que le pasamos en la URL. En el siguiente código de ejemplo lanzamos una llamada a un servicio Rest que nos devuelve un array JSON de objetos persona. **Gracias a las librerías Jackson** incluidas en el starter Web, **Spring se encarga del mapeo de objeto JSON a Java**:

```
String url="http://localhost:8080/personas";
Persona[] personas=template.getForObject(url, Persona[].class);
```

En este otro ejemplo, lanzamos una petición GET a un recurso que requiere un par de variables en la URL. En este caso, podemos concatenar los valores de dichas variables en la URL o, de forma más profesional, **utilizar el argumento uriVariables**:

```
String url="http://localhost:8080/personas/{dni}/{codigo}";
Persona[] personas=template.getForObject(url, Persona[].class,”3333R”,345);
```

RestTemplate

Métodos

- **T postForLocation(String url, Object request, Object... uriVariables)**. Lanza una petición POST a la url indicada en el primer parámetro. Dentro del cuerpo de la petición se envía en formato JSON el objeto indicado en el segundo parámetro. Al igual que en el método anterior, se pueden enviar variables en la URL, indicando los valores de las mismas a partir del tercer parámetro. El siguiente código de ejemplo, lanza una petición POST a un recurso encargado de dar de alta personas.

```
String url="http://localhost:8080/personas";
Persona p=new Persona(“94848T”,”pepito”,22);
template.posForLocation(url,p);
```

- **void put(String url, Object request, Object... uriVariables)**. Lanza una petición PUTa la url indicada en el primer parámetro. Dentro del cuerpo de la petición se envía en formato JSON el objeto indicado en el segundo parámetro. Como vemos, el método devuelve void, por lo que **si la petición PUT devuelve algún resultado como respuesta no podemos utilizar este método**.
- **void delete(String url, Object... uriVariables)**. Lanza una petición HTTP de tipo DELETE a la url indicada como parámetro. Al igual que put, se utilizará en casos en los que la llamada no devuelva resultados.

RestTemplate

Métodos

- **ResponseEntity<T> exchange(String url, HttpMethod method, HttpEntity<?> requestEntity, Class<T> responseType, Object... uriVariables).** Permite lanzar cualquier tipo de petición HTTP a la url indicada como primer parámetro. El método de envío se especificará en el segundo parámetro, mientras que **el objeto a enviar en el cuerpo será encapsulado en un objeto HttpEntity** y suministrado como tercer parámetro. El cuarto parámetro debe indicar el tipo de la respuesta, que es devuelto en un objeto ResponseEntity**.** Este método deberá ser **utilizando en aquellos casos en los que**, por las características del recurso solicitado, **no resultan apropiados los métodos anteriores**. Es el caso, por ejemplo, de una llamada a PUT en la que se espera un valor de respuesta; dado que el método *put()* devuelve void, no podemos hacer uso del mismo, por lo que habrá que utilizar *exchange()*.

El siguiente código lanza una petición PUT a un recurso que, a su vez, devuelve el objeto modificado:

```
Persona p=new Persona (“94848T”,”juanito”,22);
ResponseEntity<Persona> entity=template.exchange(url, HttpMethod.PUT, new HttpEntity<Persona>(p),
Persona.class);
Persona datos=entity.getBody(); //obtenemos el cuerpo de la respuesta, que contiene el objeto devuelto
System.out.println(datos.getEdad());
```

El objeto **ResponseEntity representa toda la respuesta recibida**. A través de su método *getBody()* recuperamos el cuerpo de la misma.

Ejemplo práctico

Servicio empleados

A continuación, vamos a crear un microservicio cliente del servicio de empleados creado en el capítulo anterior. En primer lugar, nos crearemos una **copia de este servicio** porque haremos algunas modificaciones sobre el mismo.

Lo primero que haremos sobre esta copia del servicio es **desactivar la seguridad**, para ello, quitaremos la dependencia al starter de seguridad del pom.xmo y eliminaremos la clase de configuración de seguridad, de forma que se pueda acceder libremente a los recursos del servicio.

Seguidamente, añadiremos un par de recursos más que permitan dar de alta nuevos empleados (post) y eliminar empleados a partir de su id. El controlador al completo deberá quedar como se indica en el siguiente listado:

```
@RestController
public class EmpleadosController {
   private List<Empleado> empleados;
   @PostConstruct
   public void init() {
      empleados=new ArrayList<>();
      empleados.add(new Empleado(2003,"Javier López",1780));
      empleados.add(new Empleado(1000,"María Sánchez",1900));
      empleados.add(new Empleado(4000,"David Martín",1600));
   }
   @GetMapping(value="empleados",produces=MediaType.APPLICATION_JSON_VALUE)
   public List<Empleado> recuperarEmpleados(){
      return empleados;
   }
```

Ejemplo práctico

Servicio empleados

```
@GetMapping(value="empleados/{id}",produces=MediaType.APPLICATION_JSON_VALUE)
   public Empleado buscarContacto(@PathVariable("id") int id ){
      for(Empleado emp:empleados) {
         if(emp.getIdEmpleado()==id) {
            return emp;
         }
      }
      return null;
   }
   @PostMapping(value="empleados",consumes=MediaType.APPLICATION_JSON_VALUE)
   public void altaContacto(@RequestBody Empleado emp){
      empleados.add(emp);
   }
   @DeleteMapping(value="empleados/{id}")
   public void eliminarContacto(@PathVariable("id") int id ){
      for(int i=0;i<empleados.size();i++) {
         if(empleados.get(i).getIdEmpleado()==id) {
            empleados.remove(i);
            return;
         }
      }
      return;
   }
}
```

Ejemplo práctico

Servicio cliente

De lo que se trata ahora es de realizar un microservicio que, accediendo como cliente al servicio de empleados, proporcione un recurso que **ante una petición GET que reciba en la URL los datos de un nuevo contacto, añada el nuevo contacto en caso de que no exista** y nos devuelva siempre como resultado la lista de contactos que queden.

Para realizar esta tarea, tendrá que realizar una serie de llamadas al microservicio de empleados, primero para comprobar la existencia del Empleado cuyo id reciba en URL y, en caso de que no exista, realizará una llamada al recurso POST que hemos incluido anteriormente en el servicio.

Para crear el servicio cliente, crearemos como en los casos anteriores un proyecto "Spring Starter Project", y añadiremos como única dependencia Spring Web. En la clase principal, incluiremos el método que crea el RestTemplate. Así deberá quedar la clase:

```
@ComponentScan(basePackages= {"controllers"})
@SpringBootApplication
public class Application {
   public static void main(String[] args) {
      SpringApplication.run(Application.class, args);
   }
   @Bean
   public RestTemplate getTemplate() {
      return new RestTemplate();
   }
}
```

Ejemplo práctico

Servicio cliente

Como vamos a probar ambos servicios en la misma máquina, **no podrán estar utilizando el mismo puerto**, con lo que cambiaremos el puerto del servicio cliente, incluyendo la siguiente entrada en el application.properties:

```
server.port=5000
```

En cuanto a la implementación del controlador, se muestra en el siguiente listado:

```
@RestController
public class ClienteEmpleadosController {
   //url base del servicio cliente
   private static final String url="http://localhost:8080/empleados";
   @Autowired
   RestTemplate template;
   @GetMapping(value="procesado/{id}/{nombre}/{salario}",produces=MediaType.APPLICATION_JSON_VALUE)
   public List<Empleado> empleados(@PathVariable("id") int id,
         @PathVariable("nombre") String nombre,
         @PathVariable("salario") double salario){
      //si no existe el empleado lo añade
      if(template.getForObject(url+"/"+id, Empleado.class)==null) {
         Empleado emp=new Empleado(id,nombre,salario);
         template.postForLocation(url, emp);
      }
      //en cualquier caso, devuelve un array con los empleados existentes
      return template.getForObject(url, List.class);
   }
}
```

Ejemplo práctico

Servicio cliente

Para probar el funcionamiento, ponemos en ejecución los dos servicios y desde el navegador o la aplicación postman lanzaríamos una petición del servicio cliente. Por ejemplo, si lanzamos una petición como la siguiente, en la que enviamos como id de empleado un valor ya existente, el servicio deberá respondernos con la lista de empleados existentes:

http://localhost:5000/procesado/1000/emp_new/20

Pero si enviamos un valor de **id no registrado**, obtendremos como respuesta la **lista de empleados incluyendo al nuevo que acabamos de añadir**:

http://localhost:5000/procesado/5600/emp_new/20

Acceso a microservicio securizado

En el capitulo anterior estudiamos como proteger un servicio para permitir el acceso a sus recursos solamente a determinados usuarios. De cara a poder acceder a un microservicio de estas características, el cliente, en nuestro caso otro microservicio, deberá proporcionar unos credenciales en la petición a fin de que el servidor remoto pueda autenticarlo y proporcionarle o no acceso a los recursos.

En el caso de una autenticación básica, los credenciales deben proporcionarse en la cabecera de la petición. Para realizar el proceso, podemos utilizar un **objeto BasicAuthorizationInterceptor** de Spring, que nos permite **añadir de forma sencilla la cabecera de autenticación en una petición Http**, a través del objeto RestTemplate.

Acceso a microservicio securizado

Supongamos que queremos acceder a la versión securizada del servicio de empleados. Dado que nuestro servicio cliente accede a la url empleados/{id}, necesita ser miembro del rol ADMIN, así pues, tendremos que proporcionar a RestTemaplate un **BasicAuthorizationInterceptor con las credenciales admin/admin**, que pertenecen al usuario miembro de ese rol.

Para realizar esta tarea, tendremos que modificar el método de creación del RestTemaplate, quedando como se indica en el siguiente listado:

```
@Bean
   public RestTemplate getTemplate() {
      RestTemplate template=new RestTemplate();
      //crea interceptor y lo añade a RestTemaplate
      BasicAuthenticationInterceptor itercep=new BasicAuthenticationInterceptor("admin",
"admin");
      template.getInterceptors().add(itercep);
      return template;
   }
```

Si ahora ejecutamos la versión segura del microservicio de empleados y nuestro microservicio cliente. Cuando accedamos a la dirección: http://localhost:5000/procesado/5600/emp_new/20, no añadirá el empleado por ya existir el id, pero nos devolverá correctamente la lista de los existentes. Si modificamos el método anterior, cambiando los credenciales por user1/user1, obtendremos un error 403, usuario no autorizado.

Acceso a microservicio securizado

Hay que tener en cuenta que BasicAuthenticationInterceptor, **encripta la contraseña con la codificación** predeterminada de Spring 5, esto es, utilizando el objeto **BCryptPasswordEncoder**. Esta es la **misma encriptación que se aplica en el servidor cuando almacenamos las contraseñas con {noop}.**

Ejercicio Práctico

Se trata de crear un microservicio que acceda de forma segura al microservicio de cursos.

Expondrá un recurso que se ejecute con una petición POST y reciba en el cuerpo los datos de un nuevo curso a dar de alta. Si el curso existe, lo elimina y luego lo vuelve a añadir, haciendo las pertinentes llamadas al servicio de cursos. Probar el funcionamiento con usuarios autorizados y no autorizados, de modo que solo funcione con usuarios del rol PROF.

Solución ejercicio

- En el siguiente video tutorial podrás ver la solución al ejercicio propuesto, con la explicación detallada de los componentes desarrollados.

VER TUTORIAL

Recuerda

- Un microservicio puede acceder a cualquier otro microservicio, independientemente de la tecnología con la que ambos estén implementados.
- Desde un microservicio creado con Spring Boot, es posible acceder a otros microservicios utilizando la clase RestTemplate. Esta clase proporciona una amplia variedad de métodos para lanzar peticiones HTTP a servicios externos.
- El objeto RestTemplate suele crearse con Spring desde un método anotado con @Bean en la clse de configuración e inyectarlo después en una atributo del controller que lo va a utilizar.
- Para acceder a un microservicio securizado desde otro servicio Spring Boot, mediante autenticación básica, podemos incluir los credenciales en la cabecera de la petición utilizando un objeto BasicAuthenticationInterceptor.