

Autenticación y autorización en spring boot

La seguridad es uno de los aspectos mas importantes que se deben contemplar en un proyecto, es clave conocer opciones para llevar a cabo dicha seguridad y mantener los datos e información restringida correctamente.

Autenticación y autorización en spring-boot sobre una arquitectura de microservicios

Introducción

“

La seguridad de nuestros servicios como empresa es de las cosas mas importantes y de las que deben ser tratadas con más cuidado, la seguridad se refiere a proteger la información y el procesamiento de la misma, evitar el acceso a datos a los cuales un usuario no debería tener acceso, la autorización de usuarios y la autenticación de los mismos.

La seguridad de la información debe trabajarse para evitar fallas, no dejar ningún dato expuesto, podemos encontrar algunos pilares claves como por ejemplo:

Confidencialidad: La información se mantiene privada, no es expuesta y solo se mantiene para el usuario dueño de la misma o procesos internos.

Integridad: Se refiere a datos coherentes, datos sobre los cuales haya consistencia, referencias correctas etc. Garantiza una información correcta, y que los datos no serán alterados sin una justificación y una autorización (no se cambian si no es con su debido proceso).

Disponibilidad: Los datos están para cuando se necesitan.

Desde el nivel de desarrollo las aplicaciones suelen estar protegidas por un factor, un login, que generalmente contiene el usuario y la contraseña, esta información es enviada al servidor para ser validada y poder darle acceso a dicho usuario a la aplicación.

En seguridad de internet encontramos también lo que se conoce como autenticación de doble factor o múltiples factores, pero analicemos el de doble factor, en esta manera el primer factor sería el mencionado login, y otro segundo factor podría ser un sms, un correo, un google authenticator, que asegura que además de saberse los datos básicos de acceso también tiene otro segundo factor, algo como un sms, una cuenta de gmail un correo, en esencia consiste en saber algo (contraseña) y poseer algo (teléfono por ejemplo para el sms).

Pero bueno en este caso únicamente manejamos autenticación de 1 factor y por ahora una autenticación con una autorización básica.

Autenticación

La autenticación está definida como confirmar que el usuario es quien dice ser y no se está intentando hacer un fraude, se debe verificar esta identidad con los diferentes métodos existentes, tan pronto el usuario es verificado que efectivamente es quien dice ser ya tiene acceso a los recursos del sistema, aunque estos recursos deberán estar protegidos para no mostrar de más y garantizar la seguridad, este paso adicional se conoce como autorización.

Autorización

La autorización consiste en proteger los recursos para permitir su acceso únicamente a usuarios que tienen autoridad para consumirlos, esta autoridad comúnmente se trabaja en función de roles, permisos o cualquier tipo de credencial que me garantice que tengo acceso a lo que deseo consultar.

JWT

Para manejar la seguridad de las aplicaciones hay muchas maneras de hacerlo y muchos estándares que nos facilitan esta tarea, en esta sección revisaremos que es jwt, como funciona y demás.

Jwt o json web token es un estándar para la comunicación entre dos servicios o dos partes, entiéndase por dos partes como quien ofrece un servicio y quien lo consume, es la manera de garantizar la comunicación segura cliente servidor.

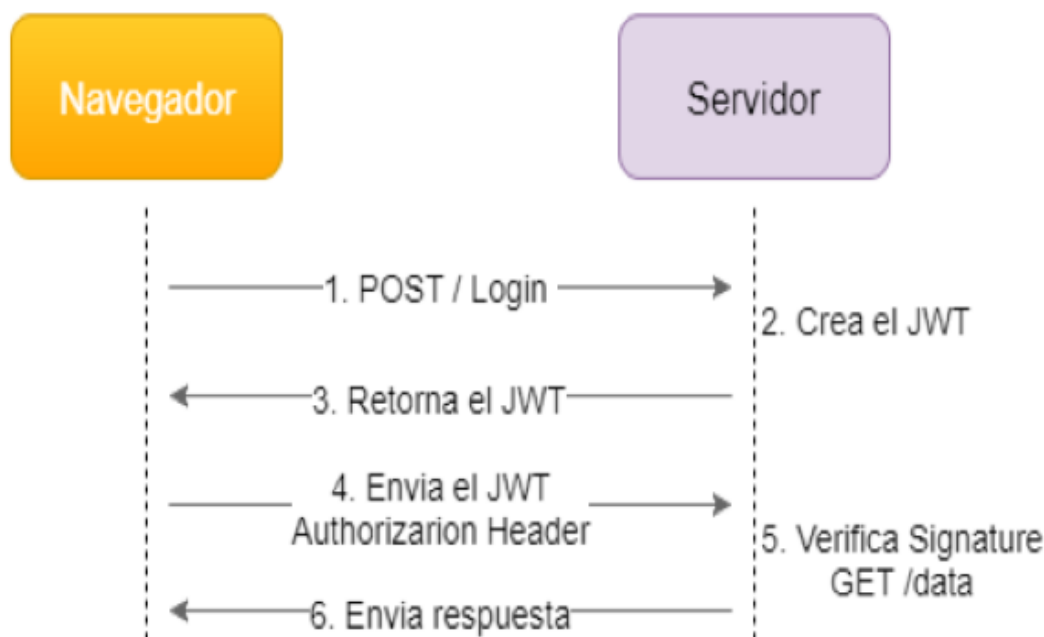
En contraste con otro estándar como es el de las cookies, jwt se está usando cada vez más en diversas aplicaciones de internet que requieren más flexibilidad, si bien las cookies funcionan para gestionar el estado de una petición http puede ser un poco menos flexible y sencillo de manejar, ahora bien ni jwt ni las cookies son la autenticación en si, mas bien son el estandar para comunicación que es ampliamente usado en la autenticación y autorización de servicios (seguridad).

Los **JWT** están compuestos de 3 partes separadas por punto y hacen referencia a

- ▶ La primera sección (**header**) nos indica el algoritmo de cifrado del token y el tipo de token, codificado en base64.
- ▶ La segunda sección (**payload**) nos indica la información presente en el token tanto identificadores, fechas, expiración, codificado en base64.
- ▶ La tercera sección es la firma del token (**Signature**) No es decodificable puesto que está cifrada y además generada en base a un secreto que se conoce únicamente dentro de la aplicación. Esta firma es generada en base a un **secreto** y **firma** el token valga la redundancia, de modo que **cualquier cambio en el token hará que este no sea valido pues la firma garantiza que el token sea inmutable**.

A continuación podemos ver como funcionan los **JWT**, el usuario al cargar la pagina y loguearse envia una petición de autenticación al servidor, este valida las credenciales y en caso de exito genera un **JWT** que

contiene cierta información útil para nuestra aplicación, las siguientes peticiones que haga el usuario no sean a **/login** serán verificadas por el token y no por sus credenciales de logueo.



La aplicación envía en un header de autorización el token obtenido en el logueo para que cada petición sea verificada y el backend pueda dar acceso a los recursos que el usuario está solicitando.

En esencia es esto, el backend con su secreto revisa si el token está bien formado, revisa si ha sido firmado con el mismo secreto, si no ha expirado o si no ha sido modificado, en caso de que algo no cumpla no se procesarán las solicitudes y el cliente tendrá un mensaje de error, los tokens tienen por lo general una fecha de expiración para que no estén vigentes mucho tiempo y sea mas difícil para atacantes llegar a nuestro sistema.

Filtros

Para validar la autenticación del usuario ante cada petición y garantizar que es quien dice ser, se debe validar el token, el servidor debe hacerlo en función del secreto y garantizar la seguridad y aspectos mencionados previamente.

En spring boot podemos hacerlo mediante filtros, un filtro puede entenderse como una autoridad que procesa cada solicitud tan pronto alcanza el servidor y decide que hacer con ella, si dejarla pasar o bloquearla, spring boot maneja una cadena de filtros (filterchain) los cuales están ordenados por prioridad y la solicitud recorre esta cadena antes de ser procesada internamente por la capa controladora que es quien la ejecuta.

Para autenticar una petición lo que hacemos es registrar un bean para el filtro que extiende de la interface `OncePerRequestFilter`, de modo que se ejecutará para cada solicitud,

Este filtro se encarga de revisar y validar el token `validateClaims`, también se encarga de manejar las excepciones y errores en caso de que la autenticación no sea correcta.

```
1  /**
2   * Filtra todas las peticiones según lo definido en urls a interceptar, excluye
3   * las públicas
4   *
5   */
6  @Bean
7  public Filter filterRequests() {
8      return new OncePerRequestFilter() {
9          @Override
10         protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response)
11             throws ServletException, IOException {
12             try {
13                 if ((!enablePreflight && request.getMethod().equals("OPTIONS"))
14                     || (jwtSecret != null && jwtSecret.equals(DEFAULT_SECRET))) {
15                     chain.doFilter(request, response);
16                 } else if (checkJWTToken(request)) {
17                     validateClaims(request, response, chain);
18                 } else {
19                     responseRequest(response, null);
20                 }
21             } catch (MalformedJwtException | UnsupportedJwtException | SignatureException e) {
22                 responseRequest(response, e);
23             } catch (ExpiredJwtException ee) {
24                 responseRequest(response, null);
25             }
26         }
27
28         @Override
29         protected boolean shouldNotFilter(HttpServletRequest request) throws ServletException {
30             return matchesUrls().match(API_URI_PUBLIC_PATTERN, request.getRequestURL().toString());
31         }
32     };
33 }
34 }
```

Ahora bien esta manera funciona y nos permite tener rutas publicas que no necesitan autenticación pues son expuestas de manera general para cualquier persona sin distinción de usuario, rol etc... Ahora bien el siguiente paso en la securización de nuestra aplicación consiste en la autorización, donde mediante el uso de roles no solo quedarnos con autenticar al usuario y ver que la petición es valida sino además corroborar que tiene acceso a la información porque lo hemos definido con unos roles o permisos y garantizar esa

confidencialidad, e integridad mencionada anteriormente.



Para la autorización se plantean (pues aún no está definida) 3 opciones:

1. **Autorización custom definida por base de datos:** En esta alternativa la base de datos almacena los endpoints o rutas base (a la capa controller) a los que tiene acceso un usuario con sus respectivos roles asociados.
2. **Autorización a nivel de metodos o controllers definida por el programador:** En esta alternativa al realizar la autenticación se almacena en el contexto del hilo de ejecución de la solicitud los datos que autentican al usuario, así como sus roles o authorities para que sea el programador quien decida que roles permisos o de que forma será accedido dicho metodo o dicha clase controladora.
3. **Autorización a nivel generico basado en roles :** Se define una configuración general a nivel del proyecto similar al filtro de autenticación donde se evalúan las rutas que requieren ser securizadas, en función de los roles y demás.

RBAC (Role Based Access Control)

El control de acceso basado en roles permite asignar roles y funciones desde nuestra arquitectura, la cual será un reflejo de como funciona la organización y el modelo de negocios, este modelo se basa en roles, un modelo de rol y una jerarquía que nos indica el orden de autoridad y por lo tanto los privilegios que tendrán los usuarios dentro de nuestra aplicación, cada usuario tendrá un rol asociado con el que se autorizará, y en función de este rol podrá acceder a ciertas funcionalidades mientras que otras serán bloqueadas porque no cumple con la autoridad para acceder a ellas.

1) Control de acceso custom usando lógica de base de datos

La primera forma consiste en una autorización administrada totalmente desde RSI por medio de la base de datos, donde el schema de autorización definirá los endpoints o controllers a los que un rol puede acceder, recordando además que un usuario puede tener varios roles .

Para esta implementación es necesario tan pronto se validan las claims del token incluir un apartado para la autorización la cual viene despues de la autenticación, entonces sería tan sencillo como incluir una función `checkAuthorization` la cual procesaría la request en función del usuario previamente autenticado, **veamos el ejemplo**

```
1 | ...filtro
2 |     ...autenticacion
3 |     -> autorización: checkAuthorizationControllerLevel(HttpServletRequest request req
4 |         List<String> controllerPaths;
5 |         var idRol= request.getHeader(IconstantesAdministracionPersonal.ID_I
6 |         if(Objects.equals(idRol, "48")){
```

```

7         controllerPaths = getControllersUser(); // obtenemos los contro
8     }else{
9         controllerPaths = getControllersAdmin();
10    }
11    var url = request.getRequestURL().toString();
12    var controller = url.split("/api/")[1].split("/")[0]; // se puede p
13    //para que sea mas eficiente en sql y verificar si existe el contro
14    return controllerPaths.stream().anyMatch(controller::contains); //
15    //controlador en la lista, es decir un booleano que autoriza o der
16    }

```

Acá tenemos un ejemplo donde obtenemos los controladores en base al rol y el usuario, donde logicamente dejamos un rol quemado para hacerlo mas facil pero la logica sería: enviarle a la base de datos el path solicitado, en el ejemplo sería `var controller` junto con el `idUsuario` y por medio de joins y un select case la base de datos debería devolver un booleano que represente si tiene o no acceso ese usuario al controlador solicitado.

Ahora para el caso de autorización a nivel de endpoint es necesario tener la consideración de las `@PathVariable`, estas son una manera de incluir en la ruta de la petición variables ya sean numericas, alfanumericas, comodines etc, entonces es necesario tener en cuenta que para mapear este contenido variable los endpoints deberan ser registrados como expresiones regulares, esto no es un problema realmente pues el procedimiento es el mismo, y sql soporta dichas expresiones regulares, **veamos el ejemplo**

```

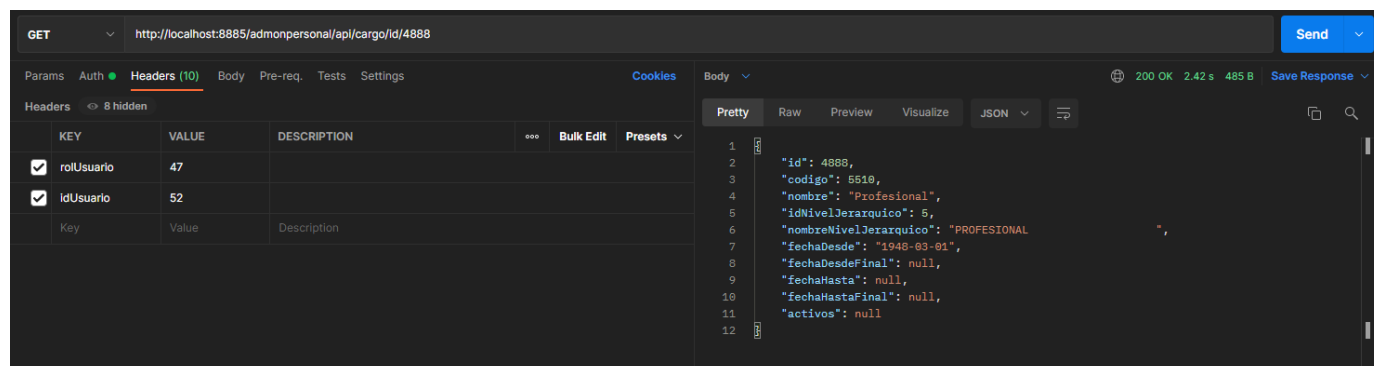
1    ...filtro
2    ...autenticacion
3    -> autorización: checkAccessEndpoint(HttpServletRequest request) {
4        List<String> endpoints;
5        var idRol = request.getHeader(IConstantesAdministracionPersonal.ID_ROL);
6        if(Objects.equals(idRol, "48")){
7            endpoints = getEndpointsUser (); // obtenemos los endpoints en base
8        }else{
9            endpoints = getEndpointsAdmin();
10       }
11       var endpointRequest = request.getRequestURL().toString().split("/admin");
12       return endpoints.stream().anyMatch(endpointRequest::matches);
13       // para el ejemplo se usa la siguiente expresión regular : "cargo+\\/[a-z0-9]+/"
14       // si hace match con la forma devolverá true
15       //se debe pasar la logica a la consulta para que sea mas eficiente
16       //y en el mismo select utilizar la reg ex que está ubicada en el campo
17
18    }

```

De nuevo se utilizan datos constantes para ejemplificar pero la logica recaería en la base de datos y haría que fuera mucho más veloz la autorización, se devuelve un booleano que nos indica si continuar con la cadena de filtros o abortar la solicitud con un mensaje de error **UNAUTHORIZED**.

Podemos corroborar esto probándolo, si enviamos el rol 48 que corresponde a un usuario base obtenemos **UNAUTHORIZED**, mientras que si usamos cualquiera diferente al 48 obtenemos acceso puesto que somos administradores.

Ejemplo admin request:



GET http://localhost:8885/admonpersonal/api/cargo/id/4888

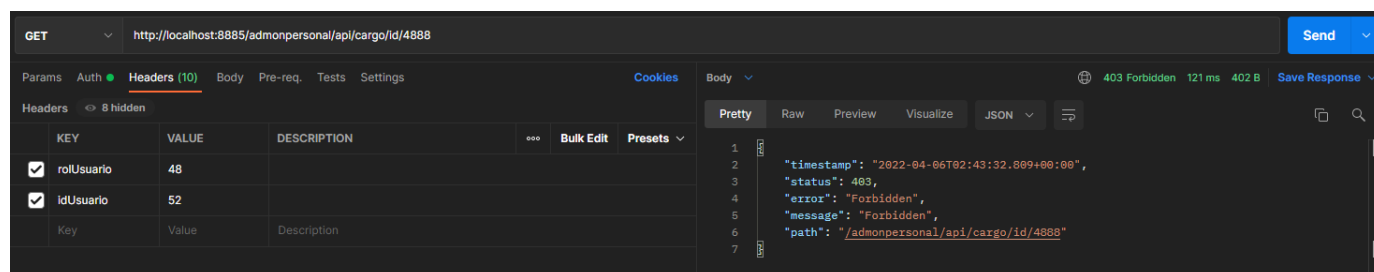
Headers (10):

KEY	VALUE	DESCRIPTION
rolUsuario	47	
IdUsuario	52	
Key	Value	Description

Body (Pretty):

```
{  "id": 4888,
  "codigo": 6510,
  "nombre": "Profesional",
  "idNivelJerarquico": 5,
  "nombreNivelJerarquico": "PROFESIONAL",
  "fechaDesde": "1948-03-01",
  "fechaDesdeFinal": null,
  "fechaHasta": null,
  "fechaHastaFinal": null,
  "activos": null}
```

Ejemplo user request:



GET http://localhost:8885/admonpersonal/api/cargo/id/4888

Headers (10):

KEY	VALUE	DESCRIPTION
rolUsuario	48	
IdUsuario	52	
Key	Value	Description

Body (Pretty):

```
{  "timestamp": "2022-04-06T02:43:32.009+00:00",
  "status": 403,
  "error": "Forbidden",
  "message": "Forbidden",
  "path": "/admonpersonal/api/cargo/id/4888"}
```

Spring security (security dependency)

Bueno para entender un poco más y profundizar en la seguridad spring boot nos ofrece funcionalidades ya desarrolladas e implementables, spring security es una de esas dependencias que podemos incluir en nuestro proyecto y nos facilitan el manejo de la seguridad.



Spring Security is a framework that provides authentication, authorization, and protection against common attacks.

Con spring security podemos crear una configuración global que maneje nuestra seguridad, ya sea mediante clases de configuración anotadas correctamente o desde la clase principal haciendo uso de los @Bean, entre las dos opciones faltantes basadas en roles, está la configuración a nivel de metodos o capa controladora

manejada por el desarrollador, y la segunda es una autenticación y autorización a nivel global, sin ser necesario entrar a cada controlador, cabe resaltar que ambas opciones pueden complementarse es decir manejar una configuración básica y además en la capa controlador especificar un poco más... veamos la primera opción

2) Seguridad a nivel de métodos o capa controladora (similar a un interceptor)

Para securizar nuestra aplicación puede ser muy seguro tener esta seguridad justo antes de activar cada método o en el controlador directamente debido a que el backend queda super específico para saber quien y solo quien puede acceder al servicio, esto tiene la desventaja de que es tedioso y requiere un manejo cuidadoso y pensado de cada funcionalidad que se desarrolla. se dice similar a un interceptor porque el intercepto se ejecuta posterior a los filtros y se puede manejar en fase precontroller ...

@Preauthorize , httpSecurity & authorities

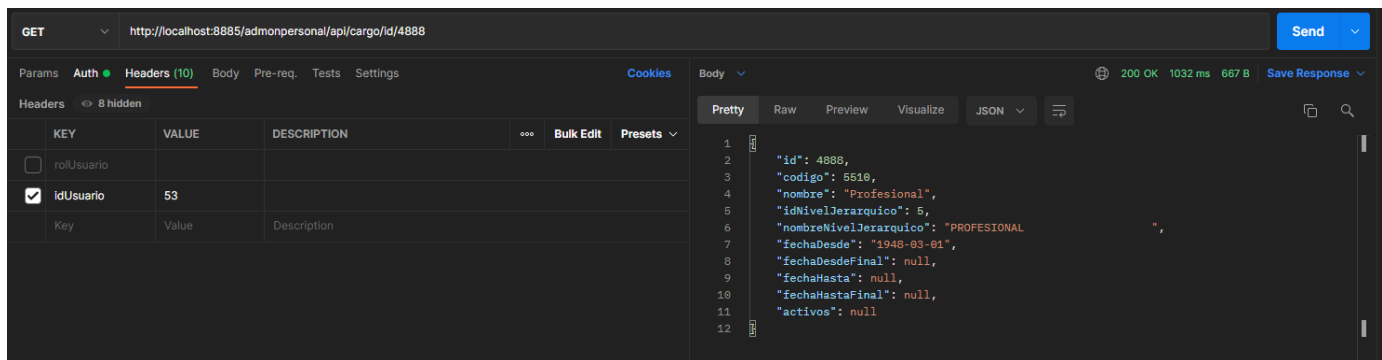
Para implementar la seguridad a nivel de metodos y restringir cualquier endpoint o controlador debemos realizar lo siguiente:

1. Habilitar la seguridad y la seguridad a nivel de metodo con las anotaciones `@EnableWebSecurity` `@EnableGlobalMethodSecurity(prePostEnabled = true)` respectivamente.
2. Definir un `@Bean` o una clase configuración para extender de la interface `WebSecurityConfigurerAdapter`

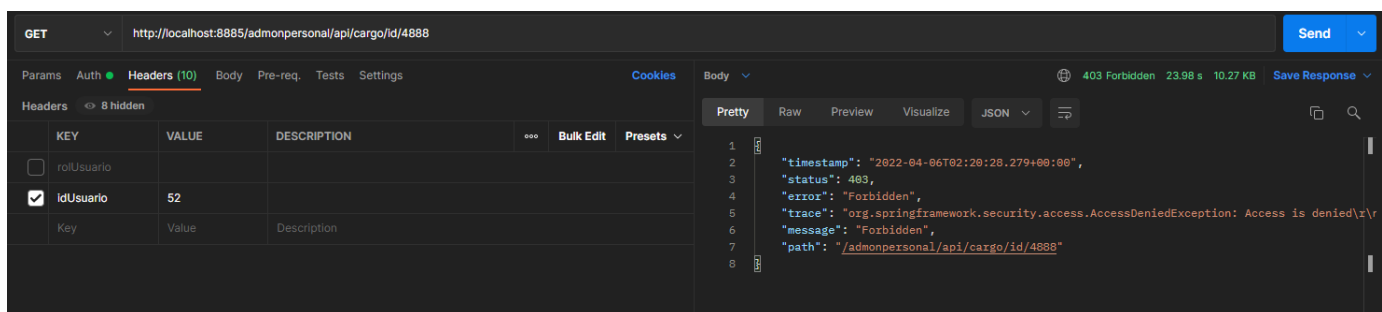
```
1 //configuramos el bloqueo de las rutas y definimos una autorizacion requerida,
2 @Bean
3 public WebSecurityConfigurerAdapter configurerAdapter(){
4     return new WebSecurityConfigurerAdapter() {
5         @Override
6         protected void configure(HttpSecurity http) throws Exception {
7             //empty by default that mean that all requests are allowed.
8         }
9     };
10 }
```

En este caso no hay ninguna configuración para la seguridad esto significa que todas las solicitudes son permitidas y se dejan procesar.

3. La validación la hacemos en el controlador o el método, utilizando la anotación `preauthorize` de la siguiente manera: `@PreAuthorize("hasAuthority('ROLE_Administrador')")` esto lo que hace es proteger el metodo para que solo sea accesible si el usuario en contexto tiene dicha autoridad, para el ejemplo el usuario 53 es el administrador y vemos como si realiza una petición al metodo definido esta es resuelta sin ningun problema.



Ahora bien si analizamos la misma petición pero con un usuario que no posee el rol de administrador vemos como spring boot automáticamente devuelve una excepción con status 403 y error **Forbidden**, esto significa que el servidor deniega la acción por permisos, además para tener en cuenta si realizamos la prueba en debug vemos como el metodo nunca llega a ejecutarse.



Spring Expression Language, cors, csrf

Algo importante a tener en cuenta dentro de la seguridad son los **cors, el csrf y las spel**.

Las spel son un conjunto de expresiones soportadas por spring boot, desde operadores logicos, operadores condicionales , aritmeticos y lo interesante es que nos permite trabajar de manera dinamica las anotaciones **@PreAuthorize** incluyendo en ellas estas operaciones e incluso funciones para adaptarlo a la logica que necesita nuestra aplicación.

los cors significan **Intercambio de Recursos de Origen Cruzado** o **Cross-origin resource sharing**, y nos permiten definir de donde son bienvenidas las solicitudes, que metodos soporta nuestro servidor, y es una manera de asegurar que las peticiones son de un origen seguro o un origen al menos conocido que al tenerlo bien configurado podemos evitar ataques.

Los csrf significan **Cross-Site Request Forgery** y basicamente nos protege de que algún atacante malicioso se aproveche de nuestro navegador y envíe información al servidor como si fuéramos nosotros, si no se tuviera el servidor bloquea las opciones diferentes de get, pero afortunadamente el jwt es una técnica que permite sobreponerse a este ataque y podemos deshabilitarlo.

3) Seguridad configurada de manera genérica

Para la seguridad de manera generica habilitaremos la configuración http de clase

`WebSecurityConfigurerAdapter` y junto con el filtro de autenticación manejaremos la autorización por roles, entre los aspectos importante a tener en cuenta está el desarrollo y configuración a nivel local, debemos buscar una configuración que no afecte el entorno local donde se hacen pruebas y **se debe evitar** tener que generar tokens a nivel local o limitar funcionalidades porque para algo es el entorno de desarrollo, para esto debemos manejar una **configuración condicional** que respete los diferentes entornos.

Para hacerlo vamos a crear un archivo de configuración donde pondremos la clase que extienda la interfaz y sobrescriba los metodos de interés, será distinto de la versión anterior pues esta opción es compatible tanto a nivel genérico como usando `@PreAuthorize` a nivel de controlador o método y mantendrá los entornos local y servidor bien diferenciados.

La clase es la siguiente:

```

1  @Configuration
2  @EnableWebSecurity
3  public class SecurityConfig extends WebSecurityConfigurerAdapter {
4
5      @Value("${authorization.enable.preflight}")
6      boolean securityEnabled;
7
8      @Qualifier("authenticationFilter")
9      @Autowired Filter filterRequests;
10
11
12      @Bean
13      CorsConfigurationSource corsConfigurationSource(){
14          var corsConfiguration = new CorsConfiguration();
15          corsConfiguration.setAllowedOrigins(Collections.unmodifiableList(Colle
16          corsConfiguration.setAllowedMethods(List.of("GET", "POST", "PUT", "PAT
17          corsConfiguration.setAllowedHeaders(Collections.unmodifiableList(Colle
18
19          var source = new UrlBasedCorsConfigurationSource();
20          source.registerCorsConfiguration("/**", corsConfiguration);
21          return source;
22      }
23
24      /**
25       * Ignorar rutas publicas de la cadena de seguridad, security chain
26       */
27
28      @Override
29      public void configure(WebSecurity web) throws Exception {
30          web.ignoring().antMatchers("/public/api/**");
31      }
32
33      /**

```

```

34     * definimos como manejar las rutas gracias al httpsecurity, donde definim
35     * y autorizaciones que debe manejar nuestro sistema. cambia de acuerdo a
36     * @throws Exception
37     */
38
39     @Override
40     protected void configure(HttpSecurity http) throws Exception {
41         //habilitamos cors y desactivamos la protección csrf de modo que cualc
42         if (securityEnabled) {
43             http
44                 .cors().and().csrf().disable()
45                 .authorizeRequests()
46                 .antMatchers("/api/cargo/**").hasRole("Administrador")
47                 .antMatchers("/api/**").hasRole("Pruebas1")
48                 .anyRequest().authenticated().and().addFilterBefore(
49             }else{
50                 http.cors().and().csrf().disable();
51             }
52         }
53         //los @preauthorize también deben excluirse en local y para esa parte deber
54         @ConditionalOnProperty(prefix = "authorization.enable",name = "preflight",l
55         @EnableGlobalMethodSecurity(prePostEnabled = true)
56         static class GlobalMethodSecurity { }
57
58     }

```

Como vemos en el código tenemos que crear una clase `@Configuration` por medio de la cual habilitaremos la seguridad con `@EnableWebSecurity`, como nosotros tenemos un filtro de autenticación en la clase principal inyectamos el bean correspondiente para luego ser usado en la configuración http, definimos un bean para el cors y este se activa tan pronto hacemos `http.cors()`, luego desactivamos la protección `csrf()` tanto en local como en el servidor y en el servidor además establecemos una autorización de ejemplo para las rutas de cargo y la ruta base con roles administrador y pruebas1 respectivamente luego decimos que cualquier otra solicitud debe como mínimo estar autenticada en el contexto de seguridad de spring, el security context, y por último y super importante añadimos un filtro (el de autenticación creado en la clase principal) el cual se añadirá a la cadena de filtros de seguridad y se encarga de autenticar y añadir el usuario con sus roles al contexto para que se active la autorización.

Es importante también el `configure` de `WebSecurity` que ignora las rutas públicas y las salta de la cadena de spring security, aunque podríamos tentarnos a eliminar la configuración del filtro para no tenerlas duplicadas, el tener un filtro como bean en la clase principal hace que ese sea ejecutado y es necesaria la discriminación individual.

Por último tenemos una declaración de una subclase estática, esto es importante para poder utilizar lo que son beans condicionales, para trabajar con `@PreAuthorize` necesitamos habilitar

`@EnableGlobalMethodSecurity` y esto activa la anotación, pero como queremos cargarlo de manera dinamica en base a una variable en este caso preflight debemos utilizar esta subclase para lograrlo, al estar dentro de una clase configuración estas clases son escuchadas y añadidas al contenedor de objetos de spring por lo que garantizamos que el "Switch" se hace tan pronto se levanta la aplicación y dependiendo del entorno se activarán o desactivarán las anotaciones **"Preflight"** sin tener que remover o añadirlas en el controler o metodo deseado.

Esta variable puede ser cambiada desde desarrollo ya sea para manejar solo las anotaciones y no la configuración de seguridad general o como se prefiera.

Con esto ya obtenemos el comportamiento deseado, en local no necesitamos token ni headers, mientras que en servidor si son requeridos.

Ejemplo de seguridad acceso aceptado

GET `http://localhost:8885/admonpersonal/api/cargo/id/4888` Send

Params Auth **Headers (9)** Body Pre-req. Tests Settings Cookies

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> rolUsuario	53	
<input checked="" type="checkbox"/> idUsuario	53	
Key	Value	Description

Body Pretty Raw Preview Visualize JSON Save Response

```

1  {
2    "id": 4888,
3    "codigo": 5510,
4    "nombre": "Profesional",
5    "idNivelJerarquico": 5,
6    "nombreNivelJerarquico": "PROFESIONAL",
7    "fechaDesde": "1948-03-01",
8    "fechaDesdeFinal": null,
9    "fechaHasta": null,
10   "fechaHastaFinal": null,
11   "activos": null
12 }

```

Ejemplo de seguridad acceso rechazado por la capa de configuración

GET `http://localhost:8885/admonpersonal/api/cargo/id/4888` Send

Params Auth **Headers (9)** Body Pre-req. Tests Settings Cookies

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> rolUsuario		
<input type="checkbox"/> idUsuario	53	
Key	Value	Description

Body Pretty Raw Preview Visualize JSON Save Response

```

1  {
2    "timestamp": "2022-04-06T22:37:22.577+00:00",
3    "status": 403,
4    "error": "Forbidden",
5    "message": "Access Denied",
6    "path": "/admonpersonal/api/cargo/id/4888"
7  }

```

Ejemplo de seguridad acceso rechazado por la capa de controller, usando anotaciones `@PreAuthorize`



Configuración dinámica para invocar un servicio y aplicar una lógica a la consulta de rol en el `@PreAuthorize`

`@PreAuthorize("@LoadRole.hasAccesToMyController(principal)")`

En la anotación es necesario agregar un `@serviceName.methodName(parameters)` y con esta estructura

hacemos que un servicio valide el método en base al rol y el usuario o cualquier lógica deseada sin tener estáticos o quemados los roles.

```
1  @Service("LoadRole")
2  public class LoadRole {
3
4      @Autowired
5      private IRolRepository rolRepository;
6
7      @Autowired
8      private IUsuarioRepository usuarioRepository;
9
10
11     public boolean hasAccesToMyController(User user) {
12         var role = this.rolRepository.findById(49L).orElseThrow(RuntimeException:
13         if(user.getAuthorities() != null && !user.getAuthorities().isEmpty() && us
14             contains(new SimpleGrantedAuthority("ROLE_"+role.replace(' ', '_'))
15             return true;
16         }
17         return false;
18     }
```

Consideraciones y aspectos importantes a tener en cuenta

Actualmente no hay una política establecida para el manejo de los roles, los permisos, los grupos y cualquier consideración que pueda llegar a definir fácilmente como manejar la autorización, es por ello que es complicado manejar una seguridad en proyectos tan grandes donde no tiene una arquitectura claramente definida, ahora bien debemos adaptarnos a lo que hay y buscar maneras de solventar las necesidades para ello se analiza **primero** una implementación donde es necesario registrar todas las rutas en base de datos, asignar los roles a las mismas y constantemente estar monitoreando, actualizando y añadiendo datos a medida que el desarrollo crece y nos limita a estar muy atados.,la aplicación es menos autónoma.

Ahora bien para seleccionar una arquitectura, unos roles definidos y una serie de grupos de permisos que facilitaría mucho el manejo de la autorización, es algo complejo también pero que nos favorecería a largo plazo. En esta **segunda opción** implementando una arquitectura, roles, permisos, grupos de permisos se tendría mas claro como manejarlo y generalizarlo en una configuración general pocamente cambiante en el tiempo, teniendo claro que si además son requeridas algunas validaciones adicionales se permita a los líderes y desarrolladores con las anotaciones @PreAuthorize llevarlas a cabo para un control específico de los servicios ofrecidos.

Conclusiones

“

- ▶ Spring security nos ofrece varias maneras de trabajar la autenticación y la autorización, incluso estándares como la autenticación de dos factores y demás, con lo cual esta puede ir escalando. La tecnología está y nos ofrece las opciones para trabajarlo.
- ▶ Hace falta un análisis y un diseño de la estructura de roles, permisos, grupos de permisos o lo que sea conveniente para poder dinamizar los procesos y avanzar ágilmente y no tener que ir cambiando tanto sobre la marcha.
- ▶ Entre las opciones 1 y 2 la opción #1 es mas sencilla de manejar, es fácil de implementar pero llevaría mucho tiempo y limitaciones, además de que no maneja spring security sino únicamente filtros y lógica nuestra. Por su parte la opción #2 es bastante potente pero requiere tiempo de análisis, experimentación, capacitación y comunicación con lo cual es más complejo de iniciar y llevar a cabo pero con el tiempo y a largo plazo será mucho mejor tanto para el equipo como para los nuevos desarrolladores que deban entender el proceso y la lógica de negocio pertinente.