Sebastian Castillo Rodríguez ene 8, 2019
(https://blog.softtek.com/es/author/sebastian-castillo-rodríguez)

Autenticación de APIs basada en tokens con Spring y JWT

En este post vamos a explicar cómo autenticar una API mediante tokens, para poder garantizar que los usuarios que consumen nuestros servicios tienen permisos para hacerlo y son quien dicen ser.

1. Crear un API REST con Spring Boot.
2. Proteger recursos publicados en el API.
3. Implementar un controlador para autenticar usuarios y generar un token de acceso.
4. Implementar un filtro para autorizar peticiones a recursos protegidos de nuestro API.

Conceptos básicos

Antes de comenzar, es importante entender algunos conceptos básicos que aparecerán a lo largo de este post:

- Servidor: Aplicación que contiene los recursos protegidos mediante API REST.
- Cliente: Aplicación que hace las peticiones a servidor para interacturar con los recursos protegidos.
- Autenticación: Proceso a través del cual un cliente garantiza su identidad. El ejemplo más sencillo sería el uso de usuario y contraseña.
- Autorización: Proceso a través del cual se determina si un cliente tiene autoridad, o autorización, para acceder a ciertos recursos protegidos.

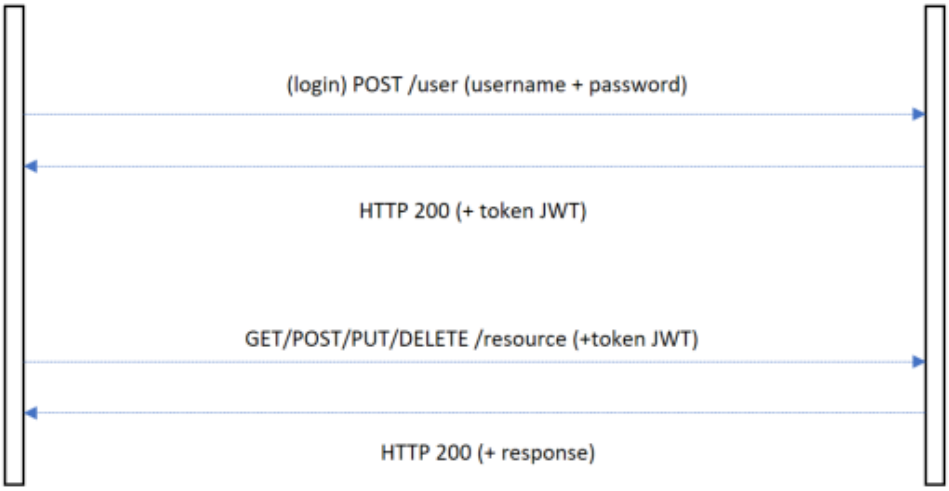
¿Que es JWT?

JSON Based Token (JWT, https://jwt.io/ (https://jwt.io/)) es un estándar de código abierto basado en JSON para crear tokens de acceso que nos permiten securizar las comunicaciones entre cliente y servidor

¿Cómo funciona?

1. El cliente se autentica y garantiza su identidad haciendo una petición al servidor de autenticación. Esta petición puede ser mediante usuario contraseña, mediante proveedores externos (Google, Facebook, etc) o mediante otros servicios como LDAP, Active Directory, etc.
2. Una vez que el servidor de autenticación garantiza la identidad del cliente, se genera un token de acceso (JWT).
3. El cliente usa ese token para acceder a los recursos protegidos que se publican mediante API.
4. En cada petición, el servidor desencripta el token y comprueba si el cliente tiene permisos para acceder al recurso haciendo una petición al servidor de autorización.

Disponemos pues, de hasta tres servidores: el servidor de nuestra API, el servidor de autenticación y el servidor de autorización. Sin embargo, como veremos en este post, podemos implementar las tres funcionalidades en una única aplicación.



Composición del token

Estos token están compuestos por tres partes:

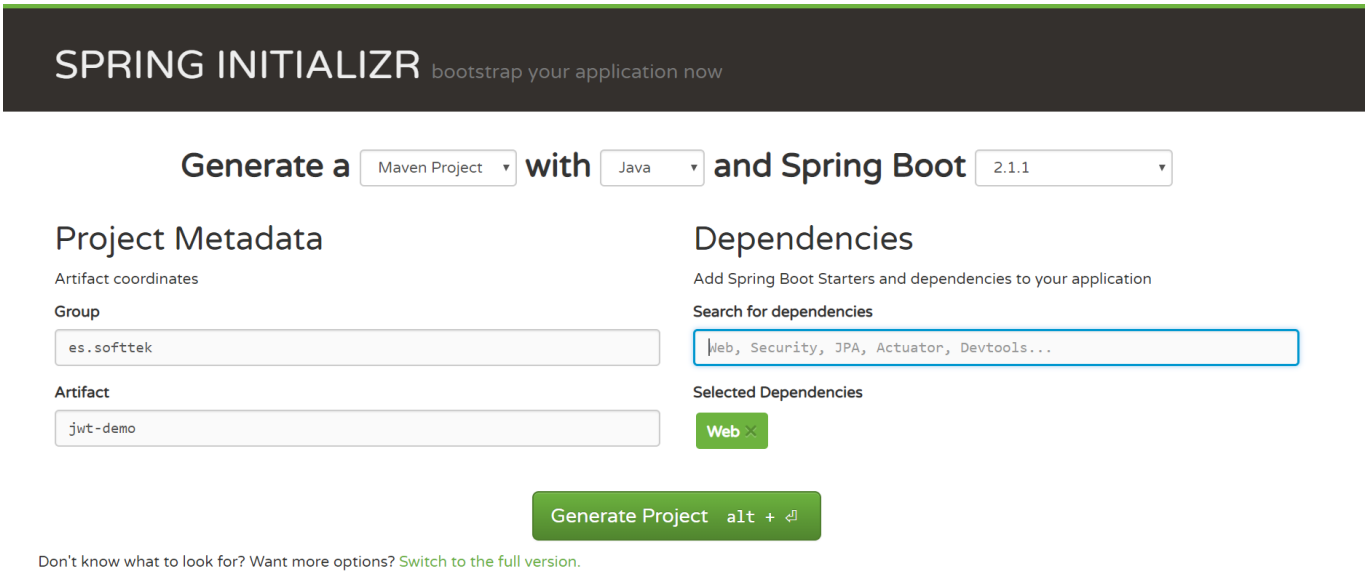
- **Header:** contiene el hash que se usa para encriptar el token.
- **Payload**: contiene una serie de atributos (clave, valor) que se encriptan en el token.
- **Firma**: contiene header y payload concatenados y encriptados (Header + "." + Payload + Secret key).

Manos a la obra...

La mejor forma de entenderlo es verlo funcionando, asi que... iiallá vamos!! Lo primero sería crear una aplicación Spring Boot para implementar nuestro API. Esto ya lo hicimos en el webinar "Construyendo un API REST con Spring Boot" (/webinar-construyendo-un-api-rest-con-spring-mongodb-heroku).

Crear aplicación Spring Boot

Accedemos a la web de Spring Initializr (https://start.spring.io) y generamos un proyecto Maven con Java y Spring Boot 2.1.1, rellenamos el *group*, el *artifact* de nuestro proyecto (en este caso "*es.softtek*" y "*jwt-demo*") y añadimos *dependencias* para Web:



Una vez generado, descomprimimos el zip y lo construimos ejecutando el comando *"mvn package"* sobre la raíz de nuestro proyecto. A continuación, generamos nuestro *eclipse project* ejecutando el comando *"mvn eclipse:eclipse"* desde la misma ubicación. Una vez compilado y construido, importamos el proyecto en nuestro IDE.

Vamos a crear un controlador REST para responder a todas las invocaciones del endpoint */hello,* que simplemente devuelva un mensaje de bienvenida a todos los clientes que tengan autorización para acceder al servicio (por defecto, todos).

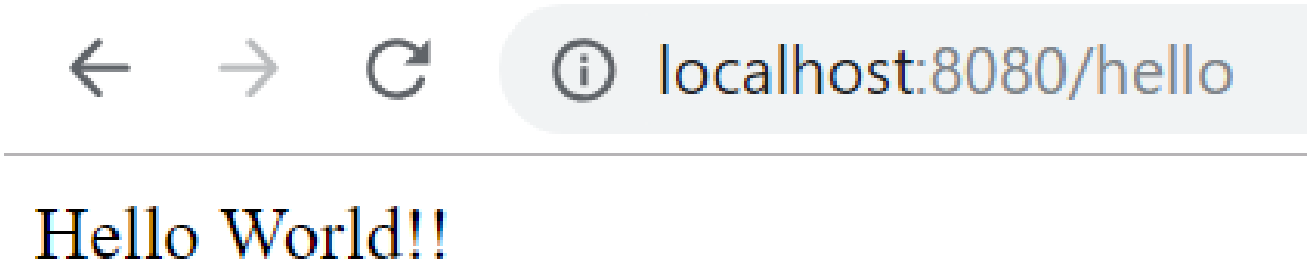
```
1  package es.softtek.jwtDemo.controller;
2
3  import org.springframework.web.bind.annotation.RequestMapping;
4  import org.springframework.web.bind.annotation.RequestParam;
5  import org.springframework.web.bind.annotation.RestController;
6
7  @RestController
8  public class HelloWorldController {
9
10     @RequestMapping("hello")
11     public String helloWorld(@RequestParam(value="name", defaultValue="World") String name) {
12         return "Hello "+name+"!!";
13     }
14 }
```

Básicamente hemos usado las siguientes anotaciones:

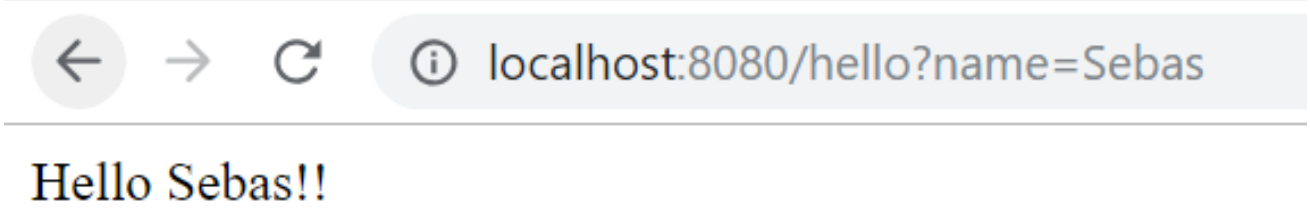
- *@RestController* para habilitar esta clase como un controlador REST y que pueda interceptar peticiones al servidor.
- *@RequestMapping* para habilitar a este método para interceptar una llamada al servidor, en este caso, a */hello.*
- *@RequestParam* para habilitar este argumento como parámetro del servicio.

Si arrancamos la aplicación (ejecutamos comando *"mvn spring-boot:run"* desde la raíz de nuestra aplicación) podemos testear nuestro servicio. Aún no hemos añadido ninguna configuración de seguridad, por lo que podemos invocar al servicio sin restricciones:

Si invocamos al endpoint de nuestro servicio (*http://localhost:8080/hello* (http://localhost:8080/hello)) nos devolverá el mensaje por defecto:



También podemos indicar un nombre invocando al mismo servicio, pero añadiendo nuestro parámetro (*http://* (http://localhost:8080/hello?name=Sebas)*localhost:8080/hello?**name=Sebas*** (http://localhost:8080/hello?name=Sebas)):



A continuación, vamos a añadir configuración de seguridad a nuestra aplicación para proteger el endpoint */hello* que acabamos de implementar.

Dependencias

Lo primero que necesitamos es añadir las dependencias para Spring Security y de JWT:

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

Autenticación

Vamos a crear otro controlador REST para implementar el proceso de autenticación mediante un login usuario/contraseña:

```
1  package es.softtek.jwtDemo.controller;
2
3  import java.util.Date;
4
5  import org.springframework.web.bind.annotation.PostMapping;
6  import org.springframework.web.bind.annotation.RequestParam;
7  import org.springframework.web.bind.annotation.RestController;
8
9  import es.softtek.jwtDemo.dto.User;
10 import io.jsonwebtoken.Jwts;
11 import io.jsonwebtoken.SignatureAlgorithm;
12
13 @RestController
14 public class UserController {
15
16     @PostMapping("user")
17     public User login(@RequestParam("user") String username, @RequestParam("password") String pwd) {
18
19         String token = getJWTToken(username);
20         User user = new User();
21         user.setUser(username);
22         user.setToken(token);
23         return user;
24
25     }
26
27     private String getJWTToken(String username) {
28         String secretKey = "mySecretKey";
29         List grantedAuthorities = AuthorityUtils
30                 .commaSeparatedStringToAuthorityList("ROLE_USER");
31
32         String token = Jwts
33                 .builder()
34                 .setId("softtekJWT")
35                 .setSubject(username)
36                 .claim("authorities",
37                         grantedAuthorities.stream()
38                                 .map(GrantedAuthority::getAuthority)
39                                 .collect(Collectors.toList()))
40                 .setIssuedAt(new Date(System.currentTimeMillis()))
41                 .setExpiration(new Date(System.currentTimeMillis() + 600000))
42                 .signWith(SignatureAlgorithm.HS512,
43                         secretKey.getBytes()).compact();
44
45         return "Bearer " + token;
46     }
47 }
```

El método *login(...)* interceptará las peticiones POST al endpoint */user* y recibirá como parámetros el usuario y contraseña. Como se puede observar, para este ejemplo no se realiza ninguna validación de usuario y contraseña, por lo que para cualquier valor de dichos parámetros dejaremos paso. Obviamente, para un proyecto real, en este punto deberíamos autenticar el usuario contra nuestra base de datos o contra cualquier proveedor externo.

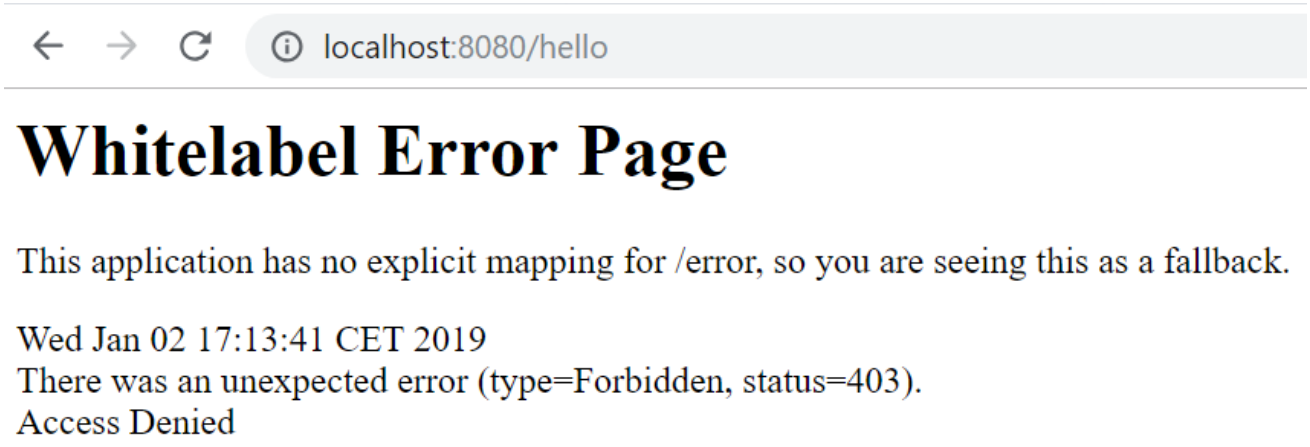
Utilizamos el método *getJWTToken(...)* para construir el token, delegando en la clase de utilidad *Jwts* que incluye información sobre su expiración y un objeto de *GrantedAuthority* de Spring que, como veremos más adelante, usaremos para autorizar las peticiones a los recursos protegidos.

Por último, editaremos nuestra clase de arranque *JwtDemoApplication* para añadir la siguiente configuración:

```
1
2  package es.softtek.jwtDemo;
3
4  import org.springframework.boot.SpringApplication;
5  import org.springframework.boot.autoconfigure.SpringBootApplication;
6  import org.springframework.context.annotation.Configuration;
7  import org.springframework.http.HttpMethod;
8  import org.springframework.security.config.annotation.web.builders.HttpSecurity;
9  import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
10 import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
11 import org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter;
12
13 import es.softtek.jwtDemo.security.JWTAuthorizationFilter;
14
15 @SpringBootApplication
16 public class JwtDemoApplication {
17
18     public static void main(String[] args) {
19         SpringApplication.run(JwtDemoApplication.class, args);
20     }
21
22     @EnableWebSecurity
23     @Configuration
24     class WebSecurityConfig extends WebSecurityConfigurerAdapter {
25
26         @Override
27         protected void configure(HttpSecurity http) throws Exception {
28             http.csrf().disable()
29                 .addFilterAfter(new JWTAuthorizationFilter(), UsernamePasswordAuthenticationFilter.class)
30                 .authorizeRequests()
31                 .antMatchers(HttpMethod.POST, "/user").permitAll()
32                 .anyRequest().authenticated();
33         }
34     }
35
36 }
```

La clase interna *WebSecurityConfig,* decorada con *@EnableWebSecurity y @Configuration*, nos permite especificar la configuración de acceso a los recursos publicados. En este caso se permiten todas las llamadas al controlador */user,* pero el resto de las llamadas requieren autenticación.

En este momento, si reiniciamos la aplicación y hacemos una llamada a *http://* (http://localhost:8080/hello)*localhost:8080/hello (http://localhost:8080/hello),* nos devolverá un error 403 informando al usuario de que no está autorizado para acceder a ese recurso que se encuentra protegido:



Autorización

Por último, necesitamos implementar el proceso de autorización, que sea capaz de interceptar las invocaciones a recursos protegidos para recuperar el token y determinar si el cliente tiene permisos o no.

Para ello implementaremos el siguiente filtro, *JWTAuthorizationFilter*.

```java
package es.softtek.jwtDemo.security;

import java.io.IOException;
import java.util.List;
import java.util.stream.Collectors;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.filter.OncePerRequestFilter;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.ExpiredJwtException;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.MalformedJwtException;
import io.jsonwebtoken.UnsupportedJwtException;

public class JWTAuthorizationFilter extends OncePerRequestFilter {

    private final String HEADER = "Authorization";
    private final String PREFIX = "Bearer ";
    private final String SECRET = "mySecretKey";

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain) throws
        try {
            if (existeJWTToken(request, response)) {
                Claims claims = validateToken(request);
                if (claims.get("authorities") != null) {
                    setUpSpringAuthentication(claims);
                } else {
                    SecurityContextHolder.clearContext();
                }
            }
            chain.doFilter(request, response);
        } catch (ExpiredJwtException | UnsupportedJwtException | MalformedJwtException e) {
            response.setStatus(HttpServletResponse.SC_FORBIDDEN);
            ((HttpServletResponse) response).sendError(HttpServletResponse.SC_FORBIDDEN, e.getMessage());
            return;
        }
    }

    private Claims validateToken(HttpServletRequest request) {
        String jwtToken = request.getHeader(HEADER).replace(PREFIX, "");
        return Jwts.parser().setSigningKey(SECRET.getBytes()).parseClaimsJws(jwtToken).getBody();
    }

    /**
     * Metodo para autenticarnos dentro del flujo de Spring
     *
     * @param claims
     */
    private void setUpSpringAuthentication(Claims claims) {
        @SuppressWarnings("unchecked")
        List authorities = (List) claims.get("authorities");

        UsernamePasswordAuthenticationToken auth = new UsernamePasswordAuthenticationToken(claims.getSubject(), null,
                authorities.stream().map(SimpleGrantedAuthority::new).collect(Collectors.toList()));
        SecurityContextHolder.getContext().setAuthentication(auth);

    }

    private boolean existeJWTToken(HttpServletRequest request, HttpServletResponse res) {
        String authenticationHeader = request.getHeader(HEADER);
        if (authenticationHeader == null || !authenticationHeader.startsWith(PREFIX))
```

```
71              return false;
72          return true;
73      }
74
75  }
```


Este filtro intercepta todas las invocaciones al servidor (extiende de *OncePerRequestFilter*) y:

1. **Comprueba la existencia del token (*existeJWTToken(...)*).**
2. **Si existe, lo desencripta y valida (*validateToken(...)*).**
3. **Si está todo OK, añade la configuración necesaria al contexto de Spring para autorizar la petición (*setUpSpringAuthentication(...)*).**

Para este último punto, se hace uso del objeto *GrantedAuthority* que se incluyó en el token durante el proceso de autenticación.

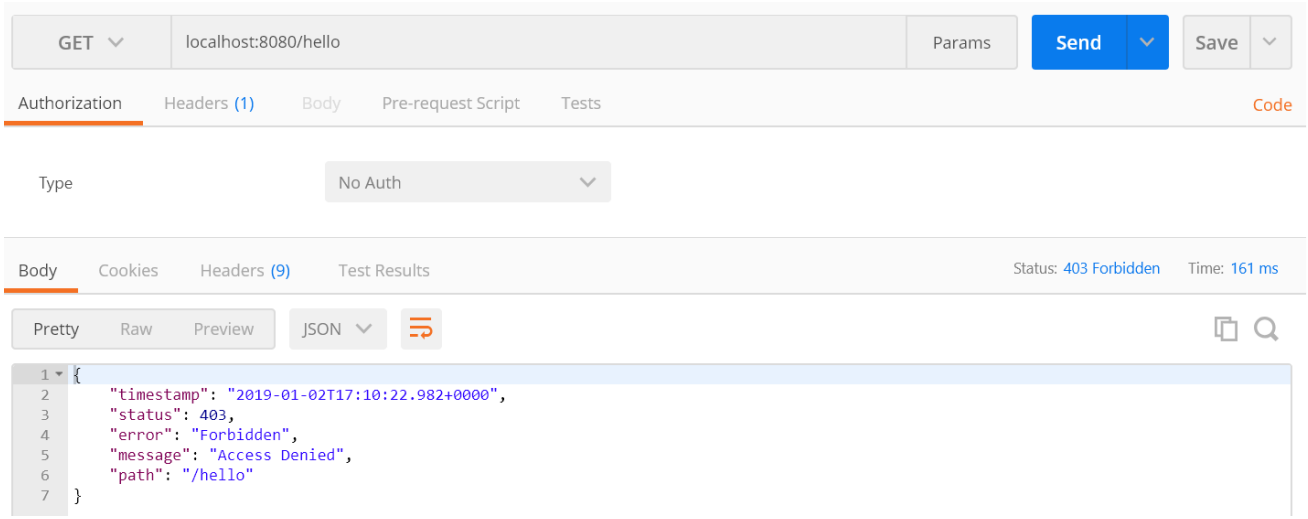
Probando nuestra API

Una vez que hemos implementado la lógica de autenticación y autorización, vamos a volver a probar nuestro API.

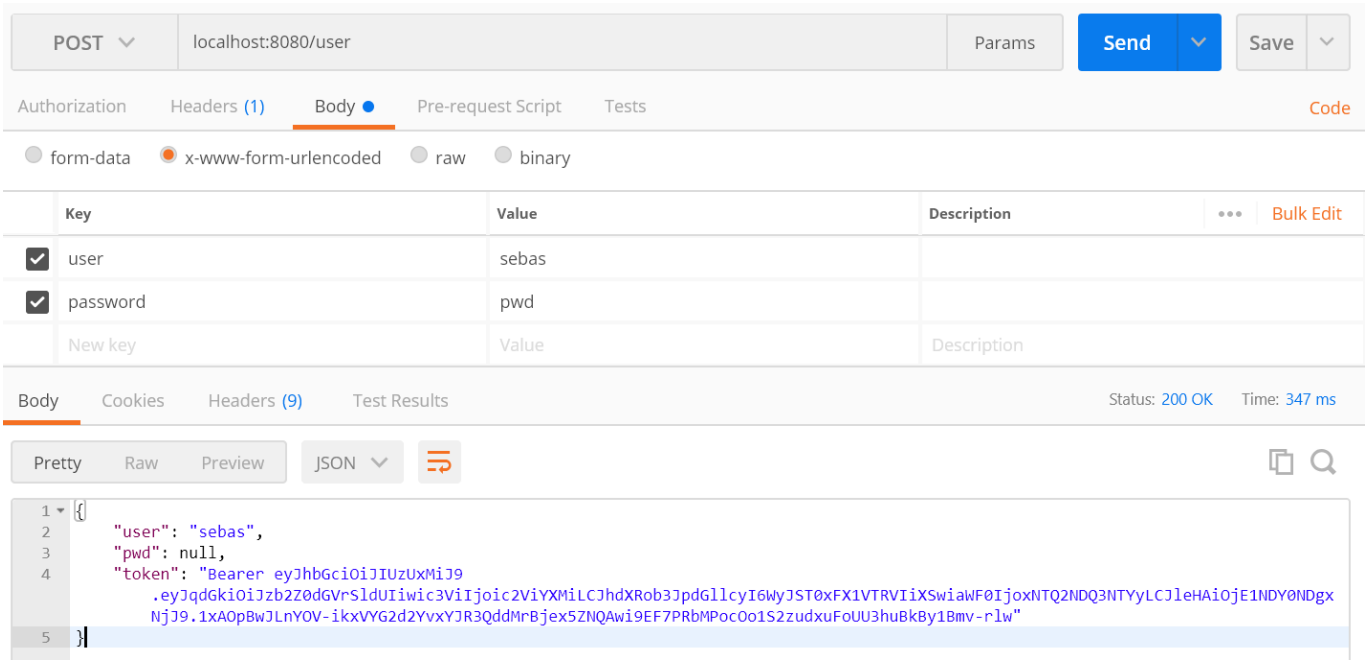
Para ello podemos usar Postman (https://www.getpostman.com/), una sencilla extensión de Chrome que nos permite ejecutar y monitorizar peticiones.

1) Arrancamos nuestro servidor ejecutando el comando *"mvn spring-boot:run"*.

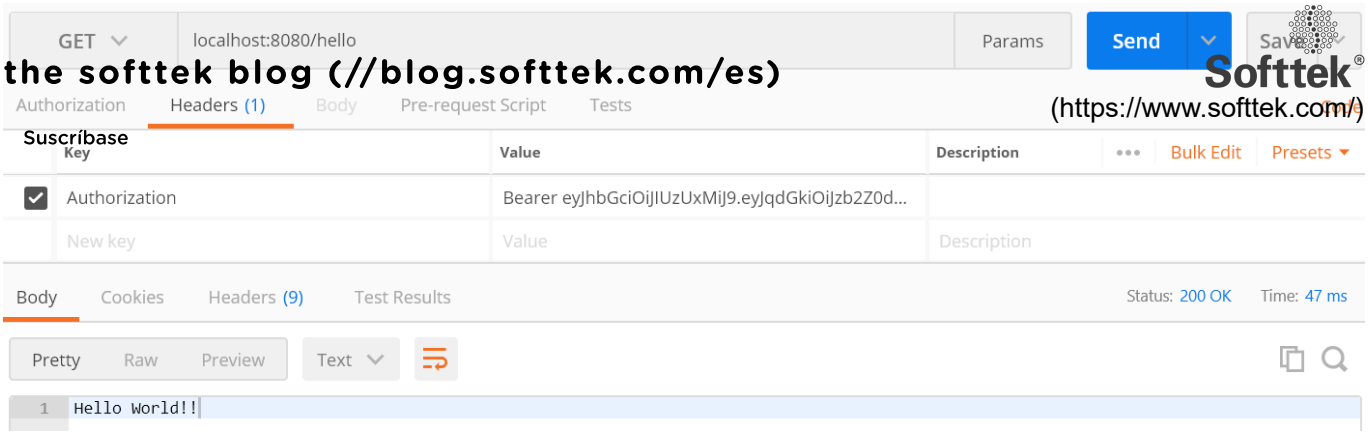
2) Desde Postman, hacemos una petición GET a */hello* y comprobamos que nos da un 403, pues el recurso está protegido:



3) Desde Postman, hacemos una petición POST a */user* para autenticarnos, incluyendo usuario y contraseña, y obtenemos un token de acceso:



4) Volvemos a hacer la petición GET del paso 2) incluyendo una cabecera *Authorization* con el token generado en el punto 3)



Conclusiones

Hemos visto una manera sencilla de autenticar y autorizar las peticiones a un API REST construido con Java y Spring Boot.

En posteriores publicaciones veremos cómo controlar el ciclo de vida de nuestros tokens, y generar excepciones, e implementaremos la lógica de autenticación para validar nuestro usuario y contraseña contra una base de datos.

El código de la aplicación está publicado en mi Github (https://github.com/sebascastillo89/jwtDemo).

¡¡Muchas gracias por leer este post, espero que haya sido de utilidad!! Para cualquier duda o consulta, podéis usar la sección de comentarios que aparece a continuación.

RUBEN CANO

Muy bueno, útil y facil de entender con un simple ejemplo. JWT es algo de lo que se está empezando a hablar en el banco y es bueno que con post como este vayamos adquiriendo la funcionalidad de los conceptos. Un saludo

Reply to *RUBEN CANO*

Sebastian Castillo

¡Gracias por leernos y comentar, Rubén! Me alegro que sea de utilidad ;)

Reply to *Sebastian Castillo*

Jorge Luis Arriaga Morales

Si quisiera poner en el configure que valide todos mis servicios que tengan en la url private como seria

Reply to *Jorge Luis Arriaga Morales*

Sebastian Castillo

Buenas Jorge Luis,

Tendrías que invertir el antMatchers y el anyRequest, algo similar a esto:

(...)
.antMatchers(HttpMethod.POST, "/private").authenticated()

.anyRequest().permitAll();
acerca (//blog.softtek.com/es/acerca)
Saludos y muchas gracias por comentar

Reply to *Sebastian Castillo*

Diego Marquia

Genial el tuorial, era justo lo que necesitaba gracias!!. Solo una cosa, actualiza la parte en JwtDemoApplication, no funciona, a diferencia de lo que esta en tu Github que si funciona completamente. Nuevamente gracias :D

Reply to *Diego Marquia*

Sebastian Castillo

Muchisimas gracias por leernos y por el feedback Diego. Actualizo el código ;)

Reply to *Sebastian Castillo*

Duglas Cañas

Muy Interesante.
En efecto encontré unos errores. Me puedes proporcionar la url de Gitub por favor.
Gracias.

Reply to *Duglas Cañas*

Duglas Cañas

Ya lo encontré.
Gracias.

Reply to *Duglas Cañas*

Adrian

Como solucionaste el problema de CORS?

Reply to *Adrian*

Sebastian Castillo

Buenas tardes Adrián,

Gracias por leer y comentar el post.

Esta pequeña demo de JWT se ejecuta en local por lo que no se configura nada a nivel de CORS.

En cualquier caso, bastaría con usar la anotación @CrossOrigin en el controlador REST. Puedes encontrar más info en la página oficial de Spring:

https://spring.io/guides/gs/rest-service-cors/
acerca (//blog.softtek.com/es/acerca)
¡Muchas gracias!

Reply to *Sebastian Castillo*

Ruben

Buenas.

Dentro del controller, en el método "getJWTToken" tienes la línea:

.map(GrantedAuthority::getAuthority)

El error que obtengo constantemente es el de "Non-static method cannot be referenced from a static context". ¿Hay alguna solución o alguna otra manera de realizar la tarea de esa línea?

Gracias de antemano. Por lo demás, todo muy bien explicado.

Reply to *Ruben*

Sebastian Castillo

Hola Rubén,

Muchas gracias por leernos y tomar tu tiempo en contestar.

No consigo reproducir lo que comentas. ¿Se trata de un error o de un warning? ¿Que versión de Spring Security estás usando?

Si tienes tu código publicado en algún repositorio podría echarle un vistazo.

¡Saludos!

Reply to *Sebastian Castillo*

Ruben

Gracias por la respuesta!

Me he dado cuenta del error: Resulta que el error está en el código del post, en tu repositorio de GitHub está correcto y funciona perfectamente. Esta es la línea que difiere entre el post y el código del repositorio.

List grantedAuthorities = AuthorityUtils
.commaSeparatedStringToAuthorityList("ROLE_USER");

Se encuentra en el controlador, en el método privado getJWTToken.

Con eso resuelto, solo me queda resolver un problema de "Circular view path" que llevo bastante tiempo buscando información acerca de él.
El error se produce cuando intento hacer el POST desde Postman a la ruta /user para obtener el token generado.

javax.servlet.ServletException: Circular view path [user]: would dispatch back to the current handler URL [/rurentero/opendataranksapi/1.0.0/user] again. Check your ViewResolver setup! (Hint: This may be the result of an unspecified view, due to default view name generation.)

Si sirve de ayuda, en toda la traza, la única clase que es código como tal (de este ejemplo y no de librerías de Spring y demás) es la del filtro:

at io.swagger.configuration.JWTAuthorizationFilter.doFilterInternal(JWTAuthorizationFilter.java:40) ~[classes/:na]

Haciendo referencia a la siguiente línea donde se hace el doFilter: chain.doFilter(request, response);

Un saludo.

Reply to *Ruben*

Ruben

Solucionado el problema del "Circular path view", era porque en mi controlador tenía la etiqueta @Controller, que es la que uso en el resto de controladores. Al sustituirla por la etiqueta @RestController (tal y como tienes en el código), se evita ese problema.

Gracias.

Reply to *Ruben*

Sebastian Castillo

Me alegro que haya quedado resuelto Ruben.

Actualizaré el código del post con lo que hay en el repo porque esa línea ya ha dado más problemas (disculpa!)

Saludos

Reply to *Sebastian Castillo*

Miguel Mendoza

Muchas gracias, me soluciono la vida para mi proyecto de pasantia, pero tengo una duda que talvez ya resolvieron en otro post... Como hago el logout, o sea como inhabilito el token

Reply to *Miguel Mendoza*

Sebastian Castillo

Muchas gracias por comentar Miguel. Me alegra mucho que fuera de utilidad.

En cuanto a tu pregunta:

En el momento en que generamos un token, su contenido no se puede modificar por lo que no podemos aplicar ninguna dinámica para inhabilitarlo.

Podemos configurar una fecha de caducidad, una fecha tope a partir de la cual el token se considere inactivo, porque esa fecha va "incrustada" como un valor fijo en el token. Pero no podemos incrustar un flag (true/false) que nos indique si está activo o no, pues no podemos modificar el contenido del token.

Sin embargo, hay muchas formas alternativas para "inhabilitar" el token sin modificar su contenido, como crear una pequeña base de datos adicional donde volcar todos los token invalidados (hasta su fecha de caducidad). Tienes más detalle de este y otros mecanismos en esta página: https://stackoverflow.com/questions/28759590/best-practices-to-invalidate-jwt-while-changing-passwords-and-logout-in-node-js/28804067

Reply to *Sebastian Castillo*

Miguel Mendoza
acerca (//blog.softtek.com/es/acerca)
Gracias por tomarte el tiempo de responder, le dare una leida

Reply to *Miguel Mendoza*

the softtek blog (//blog.softtek.com/es)

Softtek®
(https://www.softtek.com/)

Suscríbase

Sebastian Castillo

A ti por leernos! ;)

Saludos

Nombre

Apellido

Correo electrónico

Comentario

Al completar este formulario, comprendo que mi información será procesada por Softtek como se describe en su Aviso de Privacidad (http://www.softtek.com/es/aviso-privacidad).

protección de reCAPTCHA
Privacidad - Condiciones

Submit Comment

(https://www.softtek.com/)