

SPRING SECURITY: **AUTENTICACIÓN BASADA EN JWT** **(JSON WEB TOKEN)**

Contenido

SPRING SECURITY: AUTENTICACIÓN BASADA EN JWT (JSON WEB TOKEN)	1
Introducción a JSON Web Token (JWT)	3
Algo más sobre los JWT	4
Creando proyecto JWT y configuraciones	7
Agregar las dependencias de JWT:	8
Proteger rutas en nuestro API REST	9
Creando la clase filtro JWTAuthenticationFilter	10
Información importante sobre los filtros (Filter)	12
Actualización SignWith deprecated utilizando últimas versiones de jjwt.....	13
Generando el JWT.....	15
Agregando más datos en el token JWT	17
Recibiendo los datos del login en estructura JSON	18
Manejando errores de autenticación.....	19
Creando una segunda clase filtro JWTAuthorizationFilter	19
Validando el token JWT con parse	22
Realizando autenticación con el token JWT enviado por el cliente.....	22
Implementando la clase Mixin GrantedAuthority	23
Creando la clase de servicio JWT	24
Implementando y optimizando la clase de servicio JWT	25
JWTServiceImpl	25
JWTService	26
JWTAuthenticationFilter	26
JWTAuthorizationFilter	26
SpringSecurityConfig	27
Constantes en el servicio JWT	27

Introducción a JSON Web Token (JWT)

Trabajar con sesiones es lo más sencillo para Spring MVC.

También existen los tokens, son más escalables, se usan cuando el proyecto consta de varias partes, por ejemplo: tenemos un backend que entrega servicios a diferentes clientes que son otras aplicaciones, como un frontend con angular, react...etc

Tokens:

El usuario envía un código alfanumérico al servidor y éste se encarga de descifrar, lo valida, y va a ver si el usuario está registrado en nuestro sistema, y los permisos que tiene en base a los roles.

Esto permite evitar tener que guardar sesiones, aligerando el servidor.

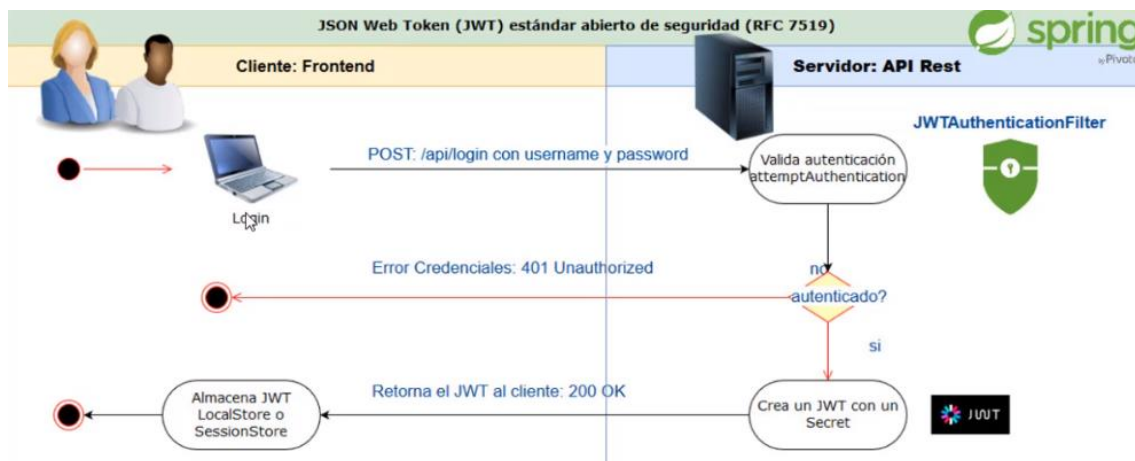
¿Qué es JWT?

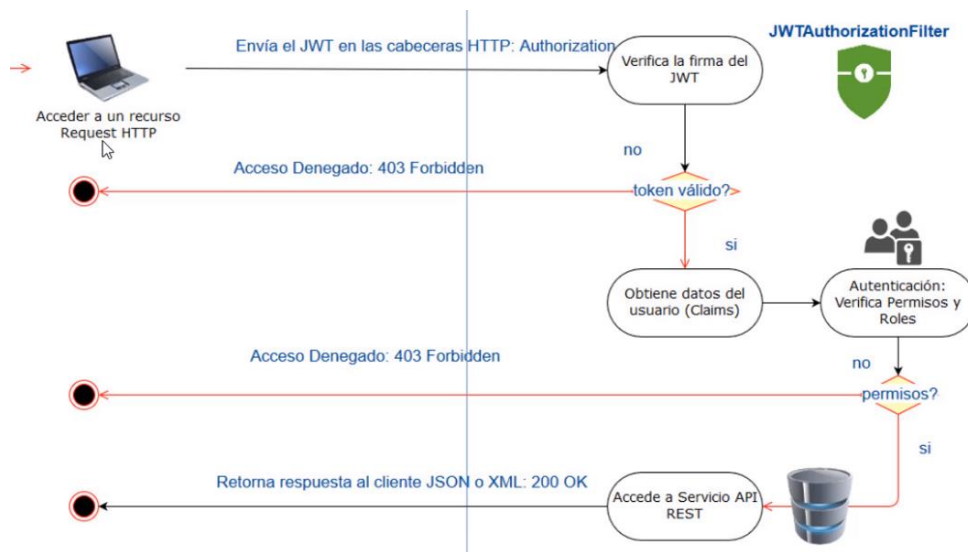
Es un standard abierto para implementar seguridad en nuestras aplicaciones API REST, basado en Tokens, (standard RF 7519), permite decodificar el código entrado por el cliente al servidor y verificar su validez.

Características:

- **Muy compacto:** debido a su pequeño tamaño permite una transmisión ágil y liviana. Se pueden enviar a través de una petición Web. (post)
- **Buena escalabilidad:** Es completamente autónomo, JWT contiene toda la información necesaria, no necesita realizar consultas al servidor ni alojar en él ninguna información.
- **Tiempo:** Si no se especifica, tiene carácter ilimitado, aunque como buena práctica se recomienda establecer un tiempo de caducidad. (1-4 horas)
- **Reversible:** El código (base64) se puede decodificar, si sabes cómo, por eso se añade una capa extra de seguridad. (verificación de firma)

¿Cómo Funciona?





Algo más sobre los JWT

Ejemplo simple: En la [web](#)

The image shows the JWT.io web interface. The 'Encoded' section displays a long string of characters. The 'Decoded' section shows the token's structure, including the header, payload, and signature. Arrows point from the encoded string to the decoded sections.

Encoded: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG91IiwiaWF0IjoxNTE2MzE2MjY0Lm51LnR5cCI6IkpXVCJ9.SMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5dI

Decoded:

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239822
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) secret base64 encoded
```

Header: La cabecera, contiene información sobre el algoritmo en que se va a codificar nuestro token y el tipo.

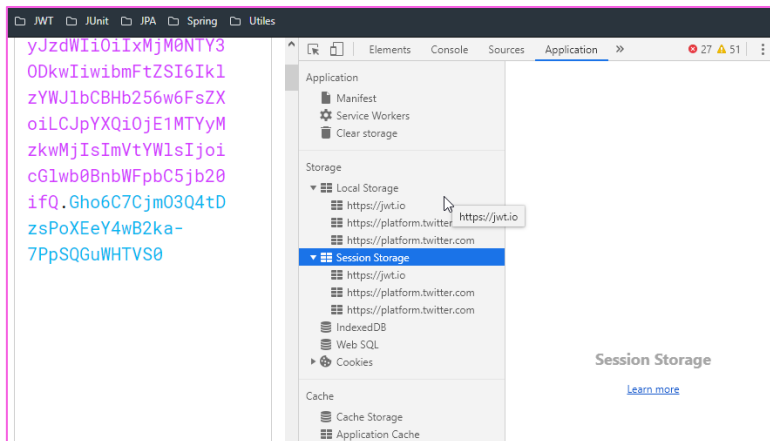
Payload: Los datos, la información del usuario. (privilegios, fecha de modificación, email del usuario...)

- Sub: Identificador, típicamente el username.
- Name: Nombre del usuario.
- Iat: fecha de creación.

Verify Signature: Verifica la firma, requiere un código secreto, que se encuentra en el servidor, por eso una aplicación cliente jamás va a tener acceso a este código.

Esto detecta si ha sido modificado algún campo, y automáticamente lo marcará como un token no válido.

Este token se almacena en el lado del cliente, normalmente en el SessionStorage, o el LocalStorage:



- Session Storage: La información permanece persistente mientras esté activo. Para un tiempo de expiración de unas horas.
- Local Storage: La información permanece en el equipo de forma permanente, por lo tanto, aunque cerremos el navegador, seguirá estando almacenada. Ideal para cuando la caducidad del token es muy larga.

Ejemplo de decodificación de un token:

The screenshot shows the JWT.io website with two panels for JWT validation. The left panel is for 'io.jsonwebtoken' and the right panel is for 'fusionauth-jwt'. Both panels show a list of checks with green checkmarks for successful validations and red X marks for failed ones. The left panel has all checks successful, while the right panel has failures for PS256, PS384, and PS512.

Check	io.jsonwebtoken	fusionauth-jwt
Sign	✓	✓
Verify	✓	✓
iss check	✓	✓
sub check	✓	✓
aud check	✓	✓
exp check	✓	✓
nbfi check	✓	✓
iat check	✓	✓
jti check	✓	✓
PS256	✓	✗
PS384	✓	✗
PS512	✓	✗
EdDSA	?	?

Below the panels, the Maven coordinates are shown:

- Left panel: `maven: io.jsonwebtoken / jjwt / 0.9.0`
- Right panel: `maven: io.fusionauth / fusionauth-jwt / 3.1.0`

Creando proyecto JWT y configuraciones

Clonamos el proyecto que teníamos para tener algo con lo que trabajar.

Y en el pom actualizamos:

```

spring-boot-jwt/pom.xml
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
5   <modelVersion>4.0.0</modelVersion>
6   <parent>
7     <groupId>org.springframework.boot</groupId>
8     <artifactId>spring-boot-starter-parent</artifactId>
9     <version>2.1.8.RELEASE</version>
10    <relativePath /> <!-- lookup parent from repository -->
11  </parent>
12  <groupId>com.isapruedas.springboot.jwt.app</groupId>
13  <artifactId>spring-boot-jwt</artifactId>
14  <version>0.0.1-SNAPSHOT</version>
15  <name>spring-boot-jwt</name>
16  <description>Demo project for Spring Boot</description>
17
18  <properties>
19    <java.version>1.8</java.version>

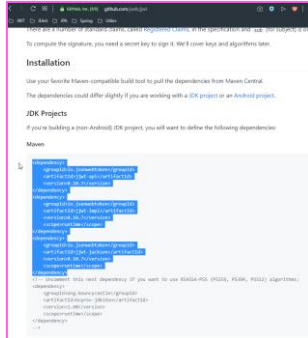
```

Luego guardamos y en Maven/Update project.

Ya tenemos un proyecto completo.

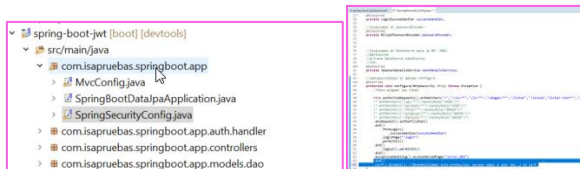
Agregar las dependencias de JWT:

[io.jsonwebtoken / jjwt](https://mvnrepository.com/artifact/io.jsonwebtoken/jjwt)

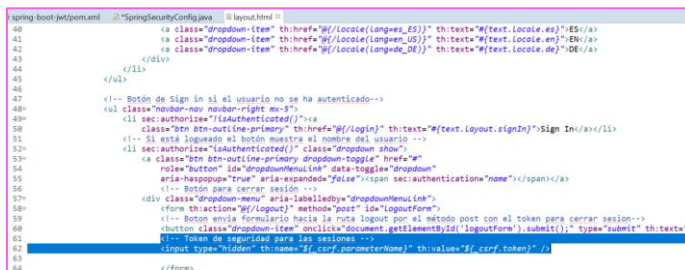


Y lo copiamos a nuestro pom.

Ahora hay que ir a la configuración de springSecurity, para adaptarla a JWT:

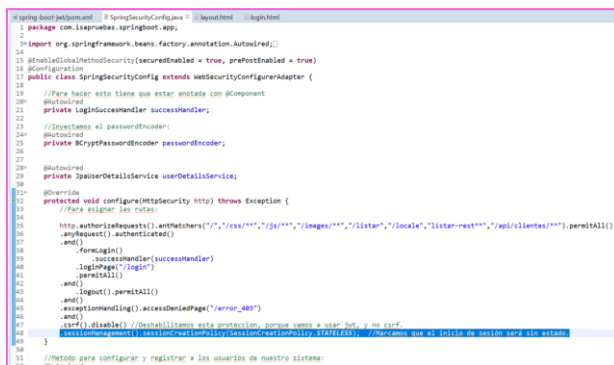


Esto influye en los inputs de las vistas:



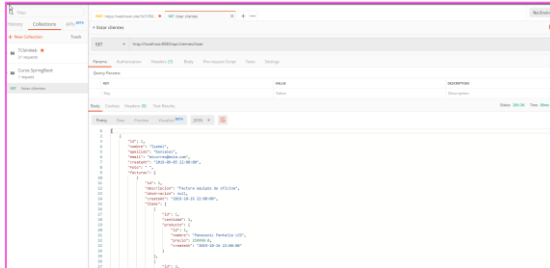
Ahora tenemos que eliminarlo.

Así quedaría nuestra clase de SpringSecurity:



Proteger rutas en nuestro API REST

Con Postman podemos hacer una petición a nuestro rest controller:



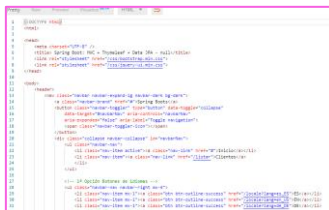
Y nos lista todos los clientes.

La idea es proteger esta página.

Por lo que vamos a la clase de configuración de la seguridad de Spring y eliminamos el acceso a esta url de la lista de permitidos:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    //Para asignar las rutas:
    http.authorizeRequests().antMatchers("/", "/css/**", "/js/**", "/images/**", "/listar", "/locale", "listar-rest").permitAll()
    .anyRequest().authenticated()
    .and()
    formLogin();
}
```

Volvemos a probar:



Nos muestra el código fuente del formulario de login.

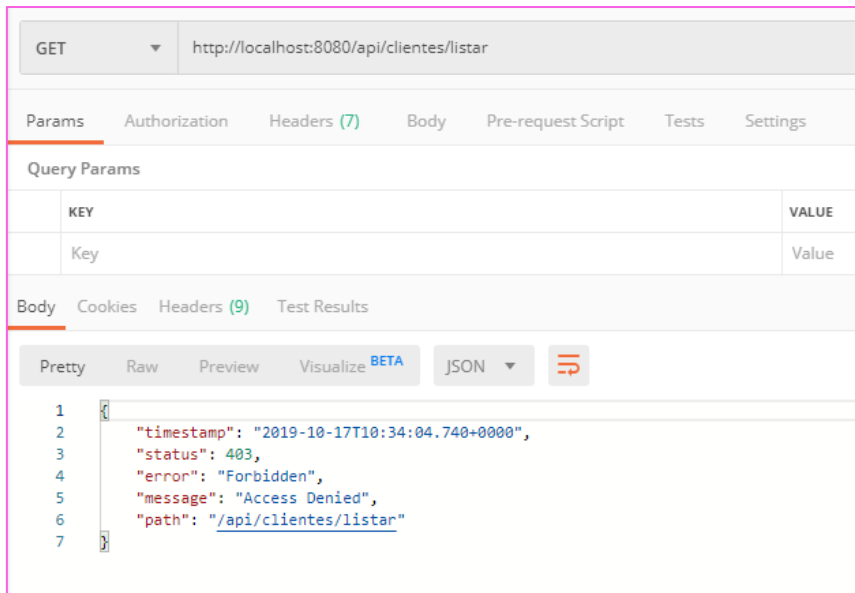
Por lo que tenemos que deshabilitar el formulario de login:

```
@Autowired
private JpaUserDetailsService userDetailsService;

@Override
protected void configure(HttpSecurity http) throws Exception {
    //Para asignar las rutas:
    http.authorizeRequests().antMatchers("/", "/css/**", "/js/**", "/images/**", "/listar", "/locale", "listar-rest").permitAll()
    .anyRequest().authenticated()
    .and()
    .csrf().disable() //Deshabilitamos esta proteccion, porque vamos a usar jwt, y no csrf.
    .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS); //Marcamos que el inicio de sesión será sin esta
}

//Metodo para configurar y registrar a los usuarios de nuestro sistema:
@Autowired
public void configureGlobal(AuthenticationManagerBuilder builder) throws Exception{

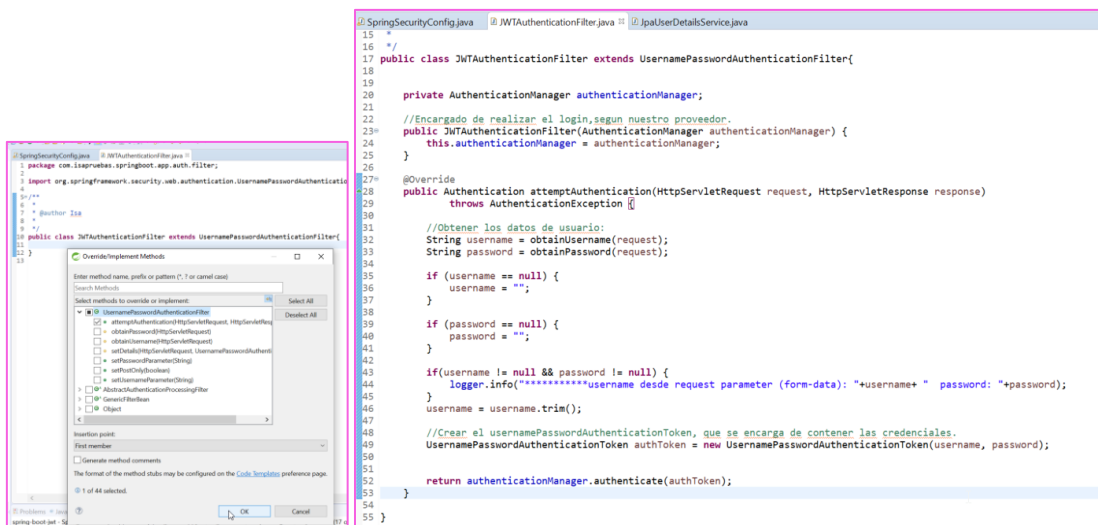
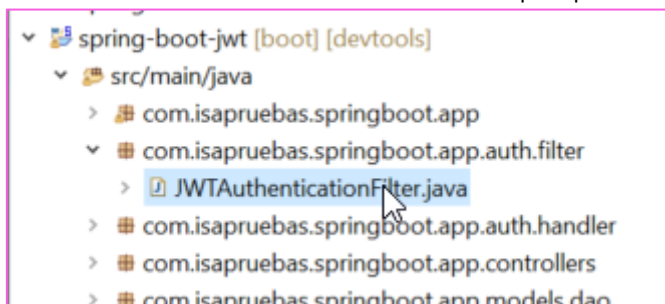
    //Configuramos para JPA:
    builder.userDetailsService(userDetailsService)
    .passwordEncoder(passwordEncoder);
}
}
```



Creando la clase filtro JWTAuthenticationFilter

Para que todo esto funcione necesitamos un filtro que se encargue de realizar la autenticación.

Primero nos creamos nuestro nuevo paquete:



Para que este filtro funcione, lo debemos registrar.

```

1 package com.isapuebas.springboot.app;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4
5 @EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled = true)
6 @Configuration
7 public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {
8
9     //Para hacer esto tiene que estar anotado con @Component
10
11     @Autowired
12     private LoginSuccessHandler successHandler;
13
14     //Inyectamos el passwordEncoder:
15     @Autowired
16     private BCryptPasswordEncoder passwordEncoder;
17
18     @Autowired
19     private JpaUserDetailsService userDetailsService;
20
21
22     @Override
23     protected void configure(HttpSecurity http) throws Exception {
24         //Para asignar las rutas:
25
26         http.authorizeRequests().antMatchers("/","/css/**","/js/**","/images/**","/listar","/local","/listar-rest").permitAll()
27             .anyRequest().authenticated()
28             .and()
29             .csrf().disable(); //Desactivamos esta protección porque vamos a usar jwt, y no csrf.
30             .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS); //Marcamos que el inicio de sesión será sin ses
31     }
32
33     //Método para configurar y registrar a los usuarios de nuestro sistema:
34     @Autowired
35     public void configureGlobal(AuthenticationManagerBuilder builder) throws Exception {
36         //Configuramos para JPA:
37         builder.userDetailsService(userDetailsService)
38             .passwordEncoder(passwordEncoder);
39     }
40 }

```

Ahora ya probamos en postman:

Postman interface showing a POST request to `http://localhost:8080/login` with form-data body containing `username` and `password`. The response is a JSON array of user details.

KEY	VALUE	DESCRIPTION
username	isa	
password	1234	

Response Body (JSON):

```

1 id,nombre,apellido,email,createAt
2 1,Isabel,Gonzalez,micorreo@mola.com,2019-09-06
3 2,Pipo,Pipo@mola.com,2019-10-10
4 3,Ane,Gonzalez,micorreo@mola.com,2019-09-06
5 4,Paco,Pii,pipo@mola.com,2019-10-10
6

```

Consola:

```

*****username desde request parameter (form-data): isa password: 1234
select usuario0_.id as id1_5_, usuario0_.enabled as enabled2_5_, usuario0_.password as password3_5_, usu
select roles0_.user_id as user_id3_0_0_, roles0_.id as id1_0_0_, roles0_.id as id1_0_1_, roles0_.authorit
Role: ROLE_USER
Utilizando forma estática SecurityContextHolder.getContext().getAuthentication(): Hola usuario autentica
hola anonymousUser NO tienes acceso :^
Forma usando SecurityContextHolderAwareRequestWrapper: hola anonymousUser NO tienes acceso
Forma usando HttpServletRequest: hola anonymousUser NO tienes acceso
select cliente0_.id as id1_1_, cliente0_.apellido as apellido2_1_, cliente0_.create_at as create_a3_1_, c
select count(cliente0_.id) as col_0_0_ from clientes cliente0

```

Ahora nos falta que nuestro filtro nos retorne un json con el token y ese token lo guardamos en nuestro cliente.



Nota: para personalizar la ruta sólo hay que añadir:

```

public class JWTAuthenticationFilter extends UsernamePasswordAuthenticationFilter{

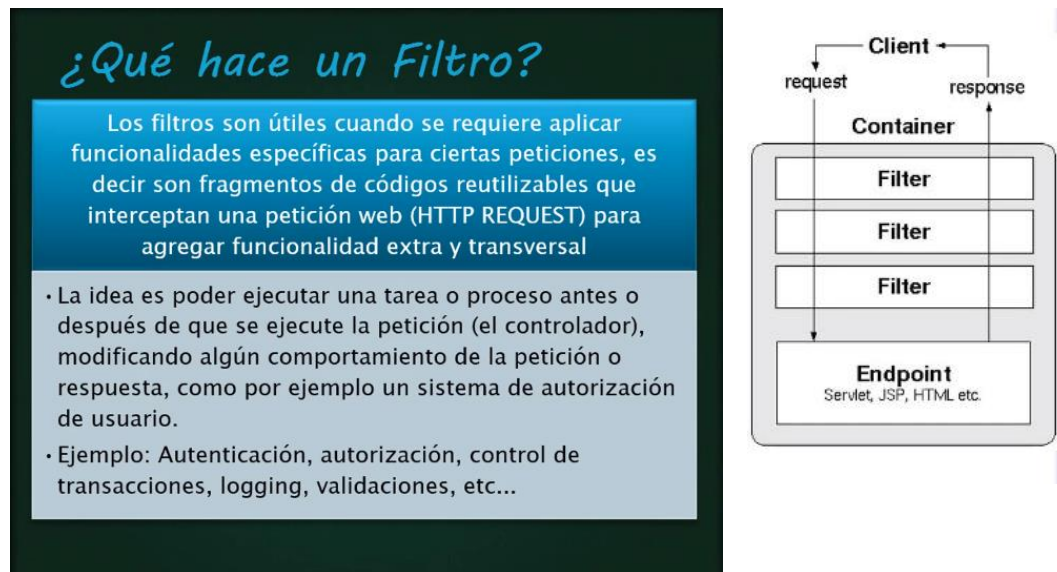
    private AuthenticationManager authenticationManager;

    //Encargado de realizar el login,segun nuestro proveedor.
    public JWTAuthenticationFilter(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
        setRequiresAuthenticationRequestMatcher(new AntPathRequestMatcher("/api/login", "POST")); //para personalizar la ruta
    }

    @Override

```

Información importante sobre los filtros (Filter)



Los Filtros son útiles cuando se requiere aplicar una funcionalidad específica antes o después de una petición web

Es parte de la especificación Java Servlet (JSR 340)

Autenticación

Autorización

Logging

Transacción

En Spring en general se aplican en métodos handler del controlador.

¿Qué hace un Filtro?

Los Filtros deben implementar la interfaz Filter o extender de la clase abstracta GenericFilterBean

- Método dofilter(): es un método de una clase Filter (API servlets) que nos permite implementar alguna tarea que necesitemos invocar antes o después de cada request.
- Cuando se invoca al chain.doFilter(...), continúa con la ejecución del controlador y si tiene más filtros asociados continúa con la ejecución en cadena.

Ejemplo Filtro

Algún filtro importante

```
public class FilterBean implements Filter {

    private static final Logger logger = LoggerFactory.getLogger(FilterBean.class);

    public void init(FilterConfig filterConfig) throws ServletException {}

    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {

        logger.info("ALGÚN PROCESO ANTES");
        chain.doFilter(request, response);
        logger.info("ALGÚN PROCESO DESPUÉS");
    }

    public void destroy() {}
}
```

El método init es para inicializar el filtro, por ejemplo, recursos, bases de datos.

El método dofilter(), con el argumento chain, que lo obtenemos como argumento del método.

Filter → doFilter

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

    @Autowired
    private IClienteService clienteService;

    @GetMapping(value = "/listar")
    public Clientelist listar() {
        return new Clientelist(clienteService.findAll());
    }
}
```



Ejecutar justo antes del request o método handler



Ejecutar después del request o método handler

Actualización SignWith deprecated utilizando últimas versiones de jjwt

Con respecto a las últimas versiones de `io.jsonwebtoken (jjwt)`:

Si instalamos la última versión del `io.jsonwebtoken`, la `0.10.5` o superior vamos a obtener un warning:

`"signWith is deprecated"`

Cambió un poco, en ese caso hay que usar la llave secreta de forma automática, básicamente debería ser así:

```
1.         SecretKey secretKey = Keys.secretKeyFor(SignatureAlgorithm.HS512);
2.
3.         String token = Jwts.builder()
4.             .setSubject(username)
5.             .signWith(secretKey)
6.             .compact();
```

Hay diferentes formas, la idea es inicializarlo una sola vez, por ejemplo, puede ser en una constante de la clase:

```
public static final Key SECRET_KEY =
Keys.secretKeyFor(SignatureAlgorithm.HS512);
```

Luego ocupas la constante tanto para generar el token con `signWith(SECRET_KEY)` como para validarlo con `setSigningKey(SECRET_KEY)`:

Generar:

```
1.         String token = Jwts.builder()
2.             .setClaims(claims)
3.             .setSubject(username)
4.             .signWith(SECRET_KEY)
5.             .setExpiration(new Date(System.currentTimeMillis() + 3600000*4)).compact(
);
```

Validar:

```
1.         try {
2.             Claims claims = Jwts.parser()
3.                 .setSigningKey(SECRET_KEY)
4.                 .parseClaimsJws(resolve(token)).getBody();
5.         } catch (JwtException | IllegalArgumentException e) ...
```

También se puede implementar una clave secreta pero de forma estática, es decir que uno mismo le asigne el valor, y no que se genere de forma automática, para eso puedes usar el método `hmacShaKeyFor(byte[] bytes)`, por ejemplo:

```
SecretKey secretKey = Keys.hmacShaKeyFor("algunaLlaveSecreta".getBytes())
```

O bien crear una instancia de la clase `SecretKeySpec` con el operador `new` para generar el `SecretKey`:

```
SecretKey secretKey = new SecretKeySpec("algunaLlaveSecreta".getBytes(),
SignatureAlgorithm.HS256.getJcaName());
```

De dependencias pom.xml:

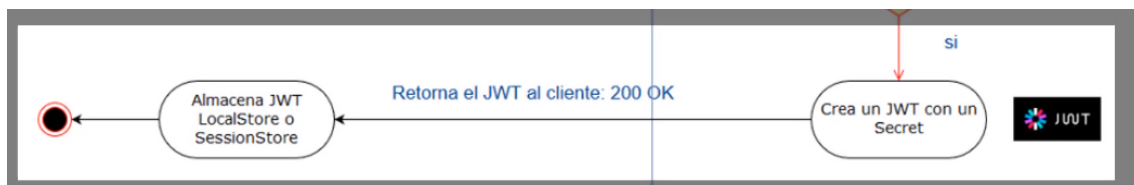
Ahora en las últimas versiones `0.10.5 o superior` son 3:

```
1.      <!-- jjwt -->
2.      <dependency>
3.          <groupId>io.jsonwebtoken</groupId>
4.          <artifactId>jjwt-api</artifactId>
5.          <version>0.10.5</version>
6.      </dependency>
7.      <dependency>
8.          <groupId>io.jsonwebtoken</groupId>
9.          <artifactId>jjwt-impl</artifactId>
10.         <version>0.10.5</version>
11.         <scope>runtime</scope>
12.     </dependency>
13.     <dependency>
14.         <groupId>io.jsonwebtoken</groupId>
15.         <artifactId>jjwt-jackson</artifactId>
16.         <version>0.10.5</version>
17.         <scope>runtime</scope>
18.     </dependency>
```

Generando el JWT

Hasta ahora solo tenemos implementado el método de autenticación a través del método `AttemptAuthentication`.

Si se ha realizado con éxito:



Este paso es vital, ahora tenemos que crear el token y enviárselo al cliente como respuesta, en las cabeceras o en formato json.

Así que nos vamos a la clase de filtro que estamos implementando:



Login con form data

POST <http://localhost:8080/api/login>

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings

☐ none
 ☒ form-data
 ☐ x-www-form-urlencoded
 ☐ raw
 ☐ binary
 ☐ GraphQL [BETA](#)

KEY	VALUE
<input checked="" type="checkbox"/> username	isa
<input checked="" type="checkbox"/> password	1234
Key	Value











Body Cookies Headers (10) Test Results

[Pretty](#)
[Raw](#)
[Preview](#)
[Visualize \[BETA\]\(#\)](#)
[JSON](#)
[iP](#)

```

1  {
2    "mensaje": "Hola isa,has iniciado sesi3n con 3xito",
3    "user": {
4      "password": null,
5      "username": "isa",
6      "authorities": [
7        {
8          "authority": "ROLE_USER"
9        }
10     ],
11     "accountNonExpired": true,
12     "accountNonLocked": true,
13     "credentialsNonExpired": true,
14     "enabled": true
15   },
16   "token": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJpc2EifQ._3yfEoYMaS7hvvbCGo3d5Dba5yCOaiY_Tfq86rH-HELKuu20sUmdsv_-fS3DQ6pIbEqqltuIHEIr1810ksA"
17 }
  
```


Las cabeceras son las que hemos implementado en el backend:

KEY	VALUE
Authorization 	Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpzZW50b3R5IiwiaWF0Ij0yOTY1MjM5OTUwMjE5In0=
X-Content-Type-Options 	nosniff
X-XSS-Protection 	1; mode=block
Cache-Control 	no-cache, no-store, max-age=0, must-revalidate
Pragma 	no-cache
Expires 	0
X-Frame-Options 	DENY
Content-Type 	application/json; charset=ISO-8859-1
Content-Length 	366
Date 	Thu, 17 Oct 2019 13:28:25 GMT

Agregando más datos en el token JWT

```

@Override
protected void successfulAuthentication(HttpServletRequest request, HttpServletResponse response, FilterChain chain,
    Authentication authResult) throws IOException, ServletException {

    //Obtenemos el usuario:
    String username = ((User) authResult.getPrincipal()).getUsername();

    //Obtenemos los roles:
    Collection<? extends GrantedAuthority> roles = authResult.getAuthorities();
    Claims claims = Jwts.claims();
    claims.put("authorities", new ObjectMapper().writeValueAsString(roles)); //tenemos que pasar un json

    //Generamos la clave secreta de forma automática:
    SecretKey secretKey = Keys.secretKeyFor(SignatureAlgorithm.HS512);

    //Creamos el token para retornar al cliente:
    String token = Jwts.builder()
        .setClaims(claims)
        .setSubject(username) //Nombre de usuario
        .signWith(secretKey) //Clave secreta para firmar el token
        .setIssuedAt(new Date()) //Fecha de creación
        .setExpiration(new Date(System.currentTimeMillis() + 3600000L)) //Fecha de expiración para 1 hora
        .compact();

    //Pasamos el token en la cabecera de la respuesta al usuario:
    response.setHeader("Authorization", "Bearer " + token); //El token se genera con el prefijo Bearer

    //Para pasarlo en formato JSON- 1º Generamos los parámetros:
    Map<String, Object> body = new HashMap<String, Object>();
    body.put("token", token);
    body.put("user", (User) authResult.getPrincipal());
    body.put("mensaje", String.format("Hola %s, has iniciado sesión con éxito", username));

    //2º Pasarlo a la respuesta- En formato JSON:
    response.getWriter().write(new ObjectMapper().writeValueAsString(body)); //toma el map y lo convierte en un objeto JSON

    //3º indicar el estado de la petición:
    response.setStatus(200); //éxito

    //4º retornamos un json, así que lo indicamos:
    response.setContentType("application/json");
}

```

```

1 {
2   "mensaje": "Hola isa,has iniciado sesión con éxito",
3   "user": {
4     "password": null,
5     "username": "isa",
6     "authorities": [
7       {
8         "authority": "ROLE_USER"
9       }
10    ],
11    "accountNonExpired": true,
12    "accountNonLocked": true,
13    "credentialsNonExpired": true,
14    "enabled": true
15  },
16  "token":
17    "eyJhbGciOiJIUzI1NiIsInR5cGU6ImFpbGljYXN0ZXQ6bnVlcnJpdCIpc1kiPTExOTYyZWZmVFUwIiwiaWF0IjE1LTczdml0Ij0pc2UiLCJ0eXQiOiJlbnRlcnRkX3Q0ODIxWV44C10TU3RlbiIsImVudCI6ImFkbG86LjY3KXxkdG90dD0tFu6PX0S4L1lgIJTB8vxx3FKFXzhgH2CEjaAcUl...qkw23uvTS-Q3SPfe7fNATCKF6

```

Vamos a copiar el token y comprobar si incluye las fechas y los roles en la página oficial de JWT:

Encoded	Decoded
<pre>eyJhbGciOiJIUzUxMiJ9.eyJhdXRo b3JpdGllcyI6Ilt7XCJhdXRob3Jpd HlcIjpcIlJPTEVfVFNFUlwiV0iLC JzdWIiOiJpc2EiLCJpYXQ0IjE1NzE zMTk3ODQsImV4cCI6MTU3MTMyMzM4 NH0.6JyKtxGBoG-fu6PXD54- 1lg1j7Bk6vxJFKFXhdH20CEJaAcw1 --qKw2JuvIS- Q35P6fINATckFGFVgQLsBwbeQ</pre>	<div>HEADER:</div> <pre>{ "alg": "HS512" }</pre> <div>PAYLOAD:</div> <pre>{ "authorities": " [{\"authority\": \"ROLE_USER\"}], \"sub\": \"isa\", \"iat\": 1571319784, \"exp\": 1571323384 }</pre> <div>VERIFY SIGNATURE</div> <pre>HMACSHA512(base64UrlEncode(header) + "." + base64UrlEncode(payload), your-256-bit-secret) <input type="checkbox"/> secret base64 encoded</pre>

Nos dirá que el formato es inválido, porque tenemos que colocar nuestro código secreto.

Recibiendo los datos del login en estructura JSON

En vez de enviar los datos como hasta ahora a través del header, también se pueden mandar en bruto mediante un json:

```

57
58
59 @Override
60 public Authentication attemptAuthentication(HttpServletRequest request, HttpServletResponse response)
61     throws AuthenticationException {
62     //Obtener los datos de usuario
63     String username = obtainUsername(request);
64     String password = obtainPassword(request);
65
66     if(username != null && password != null) {
67         logger.info("*****username desde request parameter (form-data): " + username + " password: " + password);
68     } else { //para casos de envío json
69         Usuario user = null;
70         try {
71             user = new ObjectMapper().readValue(request.getInputStream(), Usuario.class);
72         } catch (IOException e) {
73             logger.info("*****username desde request Inputstream (raw): " + username + " password: " + password);
74             e.printStackTrace();
75         } catch (JsonMappingException e) {
76             e.printStackTrace();
77         } catch (IOException e) {
78             e.printStackTrace();
79         }
80     }
81     e.printStackTrace();
82
83     username = username.trim();
84
85     //Crear el UsernamePasswordAuthenticationToken, que se encarga de contener las credenciales.
86     UsernamePasswordAuthenticationToken authToken = new UsernamePasswordAuthenticationToken(username, password); //Este token NO es el de JWT
87
88     return authenticationManager.authenticate(authToken);
89 }

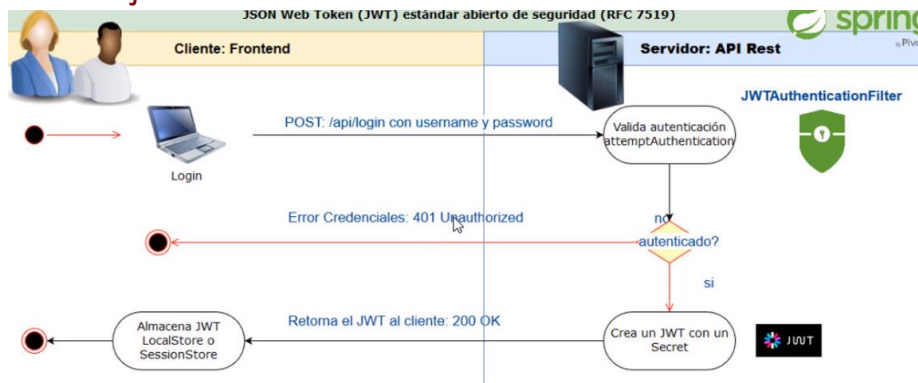
```

```

1
2
3 {
4   "username": "isa",
5   "password": "123456"
6 }

```

Manejando errores de autenticación



Cuando la autenticación falla, porque el usuario o la contraseña son erróneos debería lanzar un error de credenciales un 401.

Eso hay que programarlo, en el filtro, vamos a implementar el método **Unsuccessful**:

```
@Override
protected void unsuccessfulAuthentication(HttpServletRequest request, HttpServletResponse response,
    AuthenticationException failed) throws IOException, ServletException {

    //Preparamos el mensaje de error:
    Map<String, Object> body = new HashMap<String, Object>();
    body.put("mensaje", "Error de autenticación: Username o password incorrectos");
    body.put("error", failed.getMessage());

    //Convertir y pasarlo a la respuesta:
    response.getWriter().write(new ObjectMapper().writeValueAsString(body)); //toma el map y lo convierte en un objeto JSON
    response.setStatus(401); //Acceso NO autorizado
    response.setContentType("application/json");

}
```

Probamos en postman con datos incorrectos

La imagen muestra una captura de pantalla de Postman. En la parte superior, se configura una solicitud **POST** a **/api/login** con un cuerpo JSON: `{ "username": "lsavvv", "password": "1234567" }`. Al ejecutar la solicitud, se muestra el resultado en la pestaña **Body** con un estado **401 Unauthorized** y un tiempo de ejecución de **272ms**. El cuerpo de la respuesta es un JSON: `{ "mensaje": "Error de autenticación: Username o password incorrectos", "error": "Bad credentials" }`.

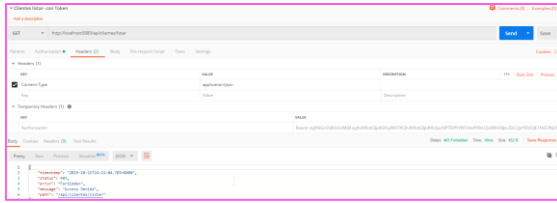
Creando una segunda clase filtro JWTAuthorizationFilter

Ahora que ya nos podemos autenticar, vamos a continuar con el flujo que hemos visto antes.

- 1º/ Autenticación: copiamos el token que acabamos de recibir.
- 2º/ Acceder al recurso: acceder desde POSTMAN, añadiendo a la configuración la opción de Authentication

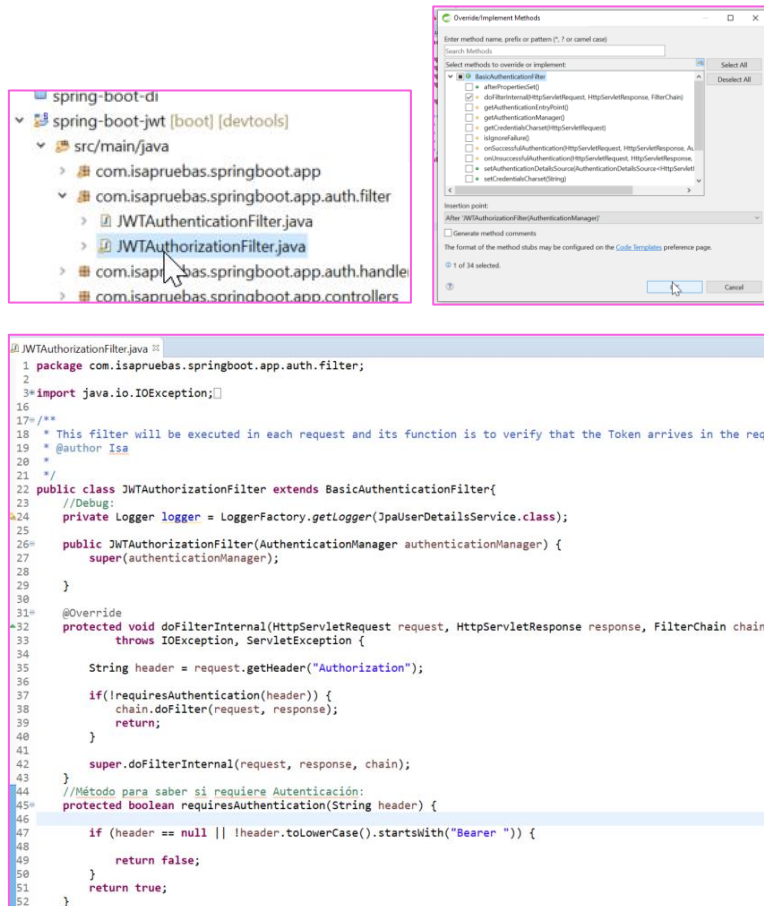
[illegible]

Aunque ahora enviamos el token, seguimos sin tener acceso:

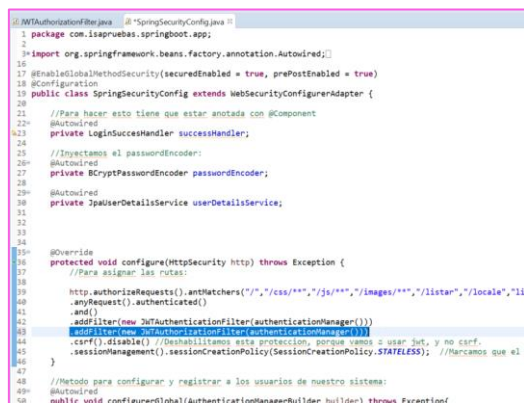


Porque hay que crear el filtro, y éste, al contrario que el otro, se ejecutará con cada request, con cada petición, siempre que enviemos un Bearer.

Así que creamos un nuevo filtro en el package correspondiente:



Importante: Registrar el filtro en la configuración de SpringSecurity.



Validando el token JWT con parse

Ahora nos queda implementar la validación del token.

```

JWTAuthorizationFilter.java | SpringSecurityConfig.java | JWTAuthenticationFilter.java
31 public class JWTAuthorizationFilter extends BasicAuthenticationFilter {
32     // Debug:
33     private Logger logger = LoggerFactory.getLogger(JpaUserDetailsService.class);
34     public static final Key SECRET_KEY = Keys.secretKeyFor(SignatureAlgorithm.HS512);
35
36     public JWTAuthorizationFilter(AuthenticationManager authenticationManager) {
37         super(authenticationManager);
38     }
39
40
41     @Override
42     protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
43         throws IOException, ServletException {
44
45         String header = request.getHeader("Authorization");
46
47         if (!requiresAuthentication(header)) {
48             chain.doFilter(request, response);
49             return;
50         }
51         boolean validToken = false;
52         Claims token = null;
53         // Validamos el token:
54         try {
55             // pasamos el token sin el Bearer y el espacio
56             token = Jwts.parser()
57                 .setSigningKey(SECRET_KEY)
58                 .parseClaimsJws(header.replace("Bearer ", ""))
59                 .getBody();
60             validToken = true;
61         } catch (JwtException | IllegalArgumentException e) {
62             validToken = false;
63         }
64
65
66
67     }
68
69     // Método para saber si requiere Autenticación:
70     protected boolean requiresAuthentication(String header) {
71

```

Realizando autenticación con el token JWT enviado por el cliente

En este punto ya tenemos validado el token, sólo nos queda dar acceso o denegarlo en función de nuestros intereses:

```

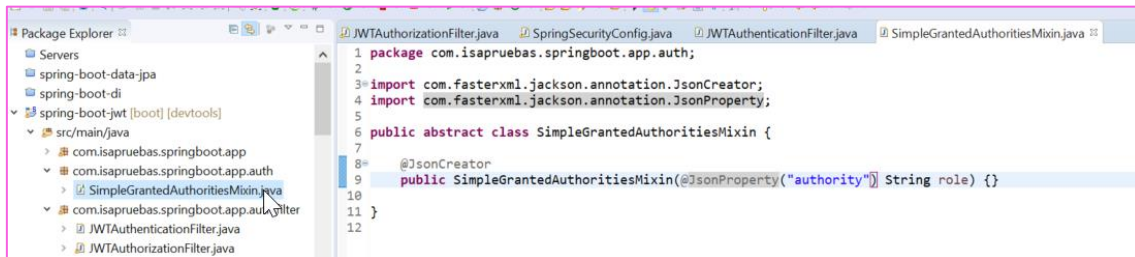
JWTAuthorizationFilter.java | SpringSecurityConfig.java | JWTAuthenticationFilter.java
50 protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
51     throws IOException, ServletException {
52
53     String header = request.getHeader("Authorization");
54
55     if (!requiresAuthentication(header)) {
56         chain.doFilter(request, response);
57         return;
58     }
59     boolean validToken = false;
60     Claims token = null;
61     // Validamos el token:
62     try {
63         // pasamos el token sin el Bearer y el espacio
64         token = Jwts.parser()
65             .setSigningKey(SECRET_KEY)
66             .parseClaimsJws(header.replace("Bearer ", ""))
67             .getBody();
68         validToken = true;
69     } catch (JwtException | IllegalArgumentException e) {
70         validToken = false;
71     }
72
73     //Cada vez que queramos acceder a un recurso tenemos que autenticarnos, definimos el token de autenticación
74     UsernamePasswordAuthenticationToken authentication = null;
75     //Obtenemos los roles
76     if (validToken) {
77         String username = token.getSubject();
78         Object roles = token.get("authorities");
79         //Cambiamos el tipo de roles, que era json, a una collection para poder pasarlo a la authentication:
80         Collection<GrantedAuthority> authorities = Arrays.asList( new ObjectMapper().readValue(roles.toString().getBytes(), SimpleGrantedAuthority[].class));
81         //Si el token es válido se asigna la autenticación
82         authentication = new UsernamePasswordAuthenticationToken(username, null, authorities);
83     }
84     //Asignamos la autenticación dentro del contexto, para autenticar al usuario dentro del request.
85     SecurityContextHolder.getContext().setAuthentication(authentication);
86     chain.doFilter(request, response); // seguimos con el flujo de los filtros.
87
88
89     // Método para saber si requiere Autenticación:
90     protected boolean requiresAuthentication(String header) {
91

```

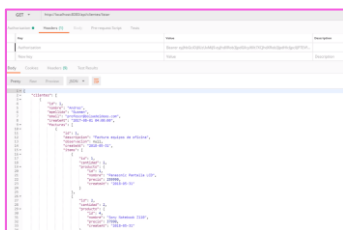
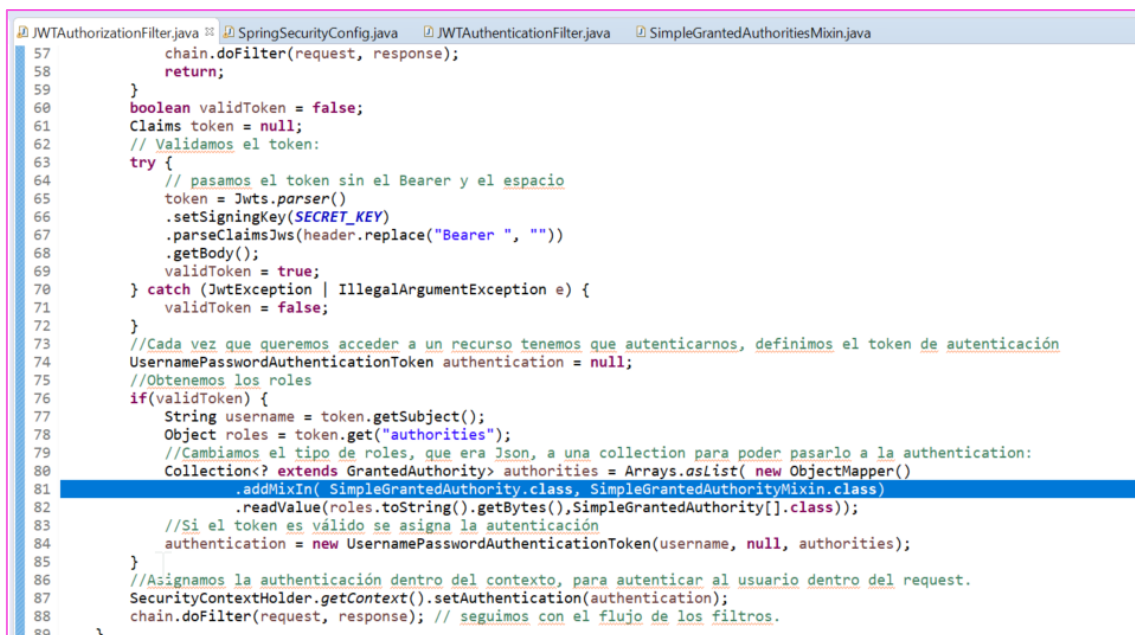
Si ejecutamos esto ahora va a dar error.

Implementando la clase Mixin GrantedAuthority

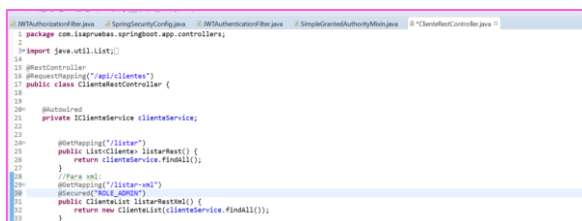
Unifica con SimpleGrantedAuthority, para agregar el constructor vacío que nos falta, y nos está dando problemas.



Volvemos al filtro para utilizar esta nueva clase:



Ahora para implementar esto habría que marcar los métodos así:



Creando la clase de servicio JWT

- Cambiamos el nombre de la clase SimpleGrantedAuthorities por su versión singular.
- Creamos una clase para gestionar los servicios JWT:

```
JWTAuthorizationFilter.java SpringSecurityConfig.java JWTAuthenticationFilter.java SimpleGrantedAuthorityMixin.java JWTService.java JWTServiceImpl.java
1 package com.isapuebas.springboot.app.auth.service;
2
3 import java.util.Collection;
4
5 import org.springframework.security.core.Authentication;
6 import org.springframework.security.core.GrantedAuthority;
7
8 import io.jsonwebtoken.Claims;
9
10 public interface JWTService {
11
12     //Crear el token:
13     public String create(Authentication auth);
14     //Validar el token:
15     public boolean validate(String token);
16     //Obtener los claims:
17     public Claims getClaims(String token);
18     //Obtener el username:
19     public String getUsername(String token);
20     //Obtener los roles
21     public Collection<? extends GrantedAuthority> getRoles(String token);
22     //Resolver el token
23     public String resolve(String token);
24
25 }
```

Y dejamos planteada la clase que implementa esta interfaz:

```
JWTAuthorizationFilter.java SpringSecurityConfig.java JWTAuthenticationFilter.java SimpleGrantedAuthorityMixin.java JWTService.java JWTServiceImpl.java
1 package com.isapuebas.springboot.app.auth.service;
2
3 import java.util.Collection;
4
5 public class JWTServiceImpl implements JWTService {
6
7     @Override
8     public String create(Authentication auth) {
9         // TODO Auto-generated method stub
10        return null;
11    }
12
13    @Override
14    public boolean validate(String token) {
15        // TODO Auto-generated method stub
16        return false;
17    }
18
19    @Override
20    public Claims getClaims(String token) {
21        // TODO Auto-generated method stub
22        return null;
23    }
24
25    @Override
26    public String getUsername(String token) {
27        // TODO Auto-generated method stub
28        return null;
29    }
30
31    @Override
32    public Collection<? extends GrantedAuthority> getRoles(String token) {
33        // TODO Auto-generated method stub
34        return null;
35    }
36
37    @Override
38    public String resolve(String token) {
39        // TODO Auto-generated method stub
40        return null;
41    }
42
43 }
```


Implementando y optimizando la clase de servicio JWT

Debemos mover el código a la nueva clase serviceImplementacion:

JWTServiceImpl:

```

1 package com.isapuebas.springboot.app.auth.service;
2
3 import java.io.IOException;
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25 @Component
26 public class JWTServiceImpl implements JWTService {
27     public static final Key SECRET_KEY = Keys.secretKeyFor(SignatureAlgorithm.HS512);
28
29     //Authentication
30     @Override
31     public String create(Authentication auth) throws IOException {
32         //Obtenemos el usuario:
33         String username = ((User) auth.getPrincipal()).getUsername();
34         //Obtenemos los roles:
35         Collection<? extends GrantedAuthority> roles = auth.getAuthorities();
36         Claims claims = Jwts.claims();
37         claims.put("authorities", new ObjectMapper().writeValueAsString(roles)); //tenemos que pasar un json
38
39         //Generamos la clave secreta de forma automática:
40         SecretKey secretKey = Keys.secretKeyFor(SignatureAlgorithm.HS512);
41
42         //Creamos el token para retornar al cliente:
43         String token = Jwts.builder()
44             .setClaims(claims)
45             .setSubject(username) //Nombre de usuario
46             .signWith(secretKey) //Clave secreta para firmar el token
47             .setIssuedAt(new Date()) //Fecha de creación
48             .setExpiration(new Date(System.currentTimeMillis()+ 3600000L)) //Fecha de expiración para 1 hora
49             .compact();
50
51         return token;
52     }
53
54     //Authorization
55     @Override
56     public boolean validate(String token) {
57         Claims claims = null;
58         // Validamos el token:
59         try {
60             getClaims(token);
61
62             return true;
63         } catch (JwtException | IllegalArgumentException e) {
64             return false;
65         }
66     }
67

```

```

78
79
80 @Override
81 public String getUsername(String token) {
82     return getClaims(token).getSubject();
83 }
84
85
86 @Override
87 public Collection<? extends GrantedAuthority> getRoles(String token) throws IOException {
88     Object roles = getClaims(token).get("authorities");
89     //Cambiamos el tipo de roles, que era json, a una collection para poder pasarlo a la authentication:
90     Collection<? extends GrantedAuthority> authorities = Arrays.asList( new ObjectMapper()
91         .addMixIn( SimpleGrantedAuthority.class, SimpleGrantedAuthorityMixin.class)
92         .readValue(roles.toString().getBytes(), SimpleGrantedAuthority[].class));
93
94     return authorities;
95 }
96
97
98 @Override
99 public String resolve(String token) {
100     if(token != null && token.startsWith("Bearer ")) {
101         return token.replace("Bearer ", "");
102     }
103     return null;
104 }
105

```

```

1 JWTAuthorizationFilter.java 2 SpringSecurityConfig.java 3 JWTAuthenticationFilter.java 4 SimpleGrantedAuthorityMixin.java
2 package com.isapruuebas.springboot.app.auth.service;
3 import java.io.IOException;
11 public interface JWTService {
12
13
14 //Crear el token:
15 public String create(Authentication auth) throws IOException;
16 //Validar el token:
17 public boolean validate(String token);
18 //Obtener los claims:
19 public Claims getClaims(String token);
20 //Obtener el username:
21 public String getUsername(String token);
22 //Obtener los roles
23 public Collection<? extends GrantedAuthority> getRoles(String token) throws IOException;
24 //Resolver el token
25 public String resolve(String token);
26
27 }
28

```

[illegible]

```

01 // 01Authenticating the user
02 // 02Implementing the logic
03 // 03Authenticating the user
04 // 04SimpleAuthenticating the user
05 // 05TestCases
06 // 06TestCases
07
08 // Import java.io.IOException
09
10 // 07+
11 // This filter will be executed in each request and its function is to verify
12 // that the token arrives in the request
13
14 // @author Iva
15
16 // 08
17 public class JwtAuthenticationFilter extends BasicAuthenticationFilter {
18     // Spring
19     private Logger logger = LoggerFactory.getLogger(SimpleAuthService.class);
20
21     public static final Key SECRET_KEY = Key.secretKeyFor(SignatureAlgorithm.HS256);
22     private JwtService jwtService;
23
24     public JwtAuthenticationFilter(AuthenticationManager authenticationManager, JwtService jwtService) {
25         super(authenticationManager);
26         this.jwtService = jwtService;
27     }
28
29     // 09
30
31     protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain)
32         throws ServletException, IOException {
33         // 10
34         String header = request.getHeader("Authorization");
35
36         // 11
37         if (request.getHeader(header) != null) {
38             // 12
39             chain.doFilter(request, response);
40             return;
41         }
42
43         // 13
44         // Cada vez que queremos acceder a un recurso tenemos que autenticarnos. Definimos al token de autenticación
45         UsernamePasswordAuthenticationToken authentication = null;
46         // Obtenemos las reglas
47         if (jwtService.validate(header) != null) {
48             // 14
49             // (Si el token es válido se asigna la autenticación
50             authentication = new UsernamePasswordAuthenticationToken(jwtService.getUserName(header), null, jwtService.getRoles(header));
51         }
52         // 15
53         // Almacenamos la autenticación dentro del contexto, para autentificar al usuario dentro del request.
54         SecurityContextHolder.getContext().setAuthentication(authentication);
55         chain.doFilter(request, response); // seguimos con el flujo de los filtros.
56     }
57
58     // 16
59     // Método para saber si requiere autenticación
60     protected boolean requiresAuthentication(String header) {
61         // 17
62         if (header == null || !header.startsWith("bearer ")) {
63             // 18
64             return false;
65         }
66         // 19
67         return true;
68     }
69 }

```

SpringSecurityConfig

```

1 package com.isapweb.springboot.app;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4
5 @EnableWebSecurity(securedEnabled = true, prePostEnabled = true)
6 @Configuration
7 public class SpringSecurityConfig extends WebSecurityConfigurerAdapter {
8
9     //Para hacer esto tiene que estar anotada con @Component
10     @Autowired
11     private LoginSuccessHandler successHandler;
12
13     //Inyectamos el passwordEncoder
14     @Autowired
15     private BCryptPasswordEncoder passwordEncoder;
16
17     //Inyectamos el passwordEncoder
18     @Autowired
19     private UserDetailsServiceImpl userDetailsService;
20
21     //Para el filtro, poner include el authentication en el constructor
22     @Autowired
23     private JwtService JWTService;
24
25
26     @Override
27     protected void configure(HttpSecurity http) throws Exception {
28         //Para asignar las rutas
29         http.authorizeRequests().antMatchers("/", "/css/**", "/js/**", "/images/**", "/listar", "/locala", "/listar-rest").permitAll()
30             .and()
31             .addFilter(new JwtAuthenticationFilter(authenticationManager(), JWTService))
32             .addFilter(new JwtAuthenticationFilter(authenticationManager(), JWTService))
33             .csrf().disable() //Desactivamos esta protección, porque vamos a usar rest, y no csrf.
34             .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS); //Queremos que el inicio de sesión sea sin estado.
35     }
36
37     //Método para configurar y registrar a los usuarios de nuestro sistema
38     @Autowired
39     public void configureGlobal(AuthenticationManagerBuilder builder) throws Exception {
40         //Configuramos para JPA
41         builder.userDetailsService(userDetailsService)
42             .passwordEncoder(passwordEncoder);
43     }
44 }

```

Constantes en el servicio JWT

En la clase service vamos a implementar una serie de constantes para optimizar el código:

```

@Component
public class JWTServiceImpl implements JWTService {
    //CONSTANTES:
    public static final Key SECRET_KEY = Keys.secretKeyFor(SignatureAlgorithm.HS512);
    public static final long EXPIRATION_DATE = 3600000L;
    public static final String TOKEN_PREFIX = "Bearer ";
    public static final String HEADER_STRING = "Authorization";

    //Authentication
    @Override
    public String create(Authentication auth) throws IOException {
        //Obtenemos el usuario:
        String username = ((User) auth.getPrincipal()).getUsername();
        //Obtenemos los roles:
        Collection<? extends GrantedAuthority> roles = auth.getAuthorities();
        Claims claims = Jwts.claims();
        claims.put("authorities", new ObjectMapper().writeValueAsString(roles)); //tenemos que pasar un json

        //Generamos la clave secreta de forma automática:
        SecretKey secretKey = Keys.secretKeyFor(SignatureAlgorithm.HS512);

        //Creamos el token para retornar al cliente:
        String token = Jwts.builder()
            .setClaims(claims)
            .setSubject(username) //Nombre de usuario
            .signWith(secretKey) //Clave secreta para firmar el token
            .setIssuedAt(new Date()) //Fecha de creación
            .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_DATE)) //Fecha de expiración para 1 hora
            .compact();

        return token;
    }

    //Authorization
    @Override
    public boolean validate(String token) {
        Claims claims = null;
    }
}

```

La forma de comprobarlo es mediante POSTMAN, se hace una petición de login, y se coge el token, luego se añade en la cabecera con el atributo authentication y se realiza la llamada al endpoint que queremos testear.