Introducción a Spring Security

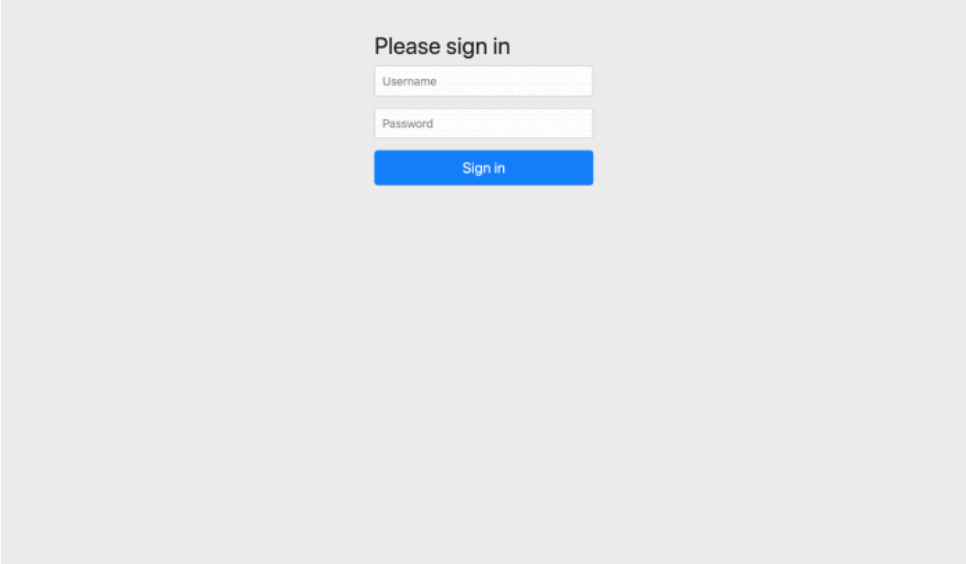
Por **Jonathan Faber** - 21 mayo, 2020

Índice de Contenidos

1. Autorización y Autenticación básica

Autenticación: verificamos la identidad del usuario.
Autorización: tipo de permisos que tiene ese usuario.

Comenzamos añadiendo la dependencia de Spring Boot Starter Security al pom.xml para habilitar la autenticación básica.
Si creamos un endpoint cualquiera e intentamos consumirlo, se mostrará un formulario de inicio de sesión proporcionado por Spring Security.



Comprueba la consola de la aplicación, se podrá ver una contraseña generada automáticamente. El nombre de usuario por defecto es «user».

```
Using generated security password: 30f20aec-fa73-4dee-a972-22f8ff77a1a5
```

Podemos incluso configurar nuestras propias credenciales en el fichero application.properties.

```
spring.security.user.name=jonathan
spring.security.user.password=1234
```

Para crear una clase de seguridad personalizada, necesitamos usar *@EnableWebSecurity* y extender la clase con *@WebSecurityConfigurerAdapter* para que podamos redefinir algunos de los métodos proporcionados. Spring Security te fuerza a hashear las contraseñas para que no se guarden en texto plano. Para los siguientes ejemplos, vamos a usar *PasswordEncoder*, aunque no debe ser una opción para proyectos reales, pero para este ejercicio es mas que suficiente. Una alternativa podría ser BCryptPasswordEncoder.

1.1 Autenticación

En intelliJ, podemos hacer cmd + N y comprobar los métodos que podemos sobrescribir(override methods). Vamos a utilizar el método *configure* que recibe como parámetro *AuthenticationManagerBuilder*. *auth* tiene diferentes métodos como *jdbcAuthentication*, *ldapAuthentication*, *userDetailsService*... pero usaremos *inMemoryAuthentication* para este ejemplo. Como su nombre lo indica, las credenciales del usuario se almacenan en memoria.

```
@EnableWebSecurity
public class WebSecurity extends WebSecurityConfigurerAdapter {
    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
                .withUser("user1").password("123").roles("APPRENTICE")
                .and()
                .withUser("user2").password("123").roles("SENSEI");
```

```
    }

    @Bean
    public PasswordEncoder getPasswordEncoder() {
        return NoOpPasswordEncoder.getInstance();
    }
}
```

1.2 Autorización

Definimos qué recursos deben estar securizados y cuales no. Ahora hacemos uso del método *configure* que recibe*HttpSecurity* como parámetro.

```
@Override
public void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/**").hasRole("SENSEI")
        .and().formLogin();
}
```

Si intentamos iniciar sesión con user2, accederemos correctamente ya que que hemos otorgado permisos a cualquier usuario que tenga el rol 'SENSEI'. Incluso podemos crear varios endpoints y otorgar distintas restricciones como vemos en el siguiente ejemplo. Debemos tener en cuenta que **las reglas más restrictivas deben estar en la parte superior**.

```
@Override
public void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
            .antMatchers("/sensei").hasRole("SENSEI")
            .antMatchers("/apprentice").hasRole("APPRENTICE")
            .antMatchers("/").permitAll()
            .and().formLogin();
}
```

Si quisiéramos añadir diferentes roles para un mismo recurso podríamos usar *hasAnyRole().* También podemos añadir filtros, cuyo objetivo es interceptar las peticiones y cada uno tiene su propia responsabilidad (añadiremos uno mas adelante)

```
.hasAnyRole("ROLE1", "ROLE2", "ROLE3")
.addFilterBefore()
.addFilterAfter()
```

1.3 UserDetailsService

Vamos a configurar Spring Security para que dependa de UserDetailsService, un servicio que nos permitirá cargar datos específicos del usuario.

```
@EnableWebSecurity
public class WebSecurity extends WebSecurityConfigurerAdapter {
    @Autowired
    private MyUserDetailService myUserDetailService;

    @Override
```

```
        protected void configure(AuthenticationManagerBuilder auth) throws Exception {
            auth.userDetailsService(myUserDetailService);
        }
    }
```

Procedemos a crear nuestra propia clase *UserDetailService* y *UserDetails.* En *MyUserDetailService* simplemente sobrescribimos el método *loadUserByUsername* que recibe el nombre de usuario por parámetro.

```
@Service
public class MyUserDetailService implements UserDetailsService {
    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundExceptio
        return new MyUserDetails(username);
    }
}
```

Cuando implementamos la interfaz UserDetails, podemos sobrescribir varios métodos. Voy a crear un campo 'username' para obtener el nombre de usuario. El resto de los métodos tendrá valores hardcoded. *getAuthorities* devuelve los permisos otorgados al usuario, en este caso añadiré solo el rol SENSEI. La contraseña tendrá el valor 'pass'.

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class MyUserDetails implements UserDetails {
    private String username;

    @Override
    public Collection getAuthorities() {
        return List.of(new SimpleGrantedAuthority("ROLE_SENSEI"));
    }

    @Override
    public String getPassword() {
        return "pass";
    }

    @Override
    public String getUsername() {
        return username;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }
```

```
    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}
```

El flujo cuando iniciemos la aplicación será el siguiente:

1. Se carga la configuración de *WebSecurity.*
2. Cuando el usuario introduce sus credenciales y éstas se envían, el filtro de autenticación de Spring Security intercepta la petición y se crea un objeto. *UsernamePasswordAuthenticationToken* con las credenciales.
3. *loadUserByUsername()* recibe el nombre de usuario.
4. Se crea un objeto *MyUserDetails* con el nombre de usuario enviado y todo lo demás con valores hardcoded (simulando ser un usuario que tengamos en base de datos) y se compara *MyUserDetails* con el objeto *UsernamePasswordAuthenticationToken*.
5. Si todo es correcto se accede al recurso, en caso contrario, permiso denegado.

Espero que se haya entendido, en caso de dudas te animo a que depures y pongas varios puntos de ruptura para comprobarlo.

En vez de directamente crear MyUserDetails, podríamos inyectar el repository y obtener la información de base de datos a partir de ese nombre de usuario. Evidentemente habría que cambiar la clase *MyUserDetails* para que en vez de recibir una string reciba un *User.*

```
@Service
@RequiredArgsConstructor
public class MyUserDetailService implements UserDetailsService {
    private final UserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundExceptio
        User user = userRepository.findByUsername(username)
              .orElseThrow(() -> new UsernameNotFoundException("Username does not exist"))
        return new MyUserDetails(user);
    }
}
```

2. Añadiendo JWT

Json Web Token: estándar que define una forma auto contenida de transmitir información como JSON. Consta de tres partes separadas por puntos.

- **Header:** algoritmo (SHA256, HS512 ...) y el tipo de token.
- **Payload:** contiene las propiedades(**claims**) del token.
- **Signature:** header (codificado en base64), payload (codificado en base64), una clave, y todo firmado con el algoritmo del header.

Claim: porción de información en el cuerpo del token.

Comenzamos añadiendo la dependencia que nos permite crear jwt y validarlos.

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
```

2.1 Autenticamos al usuario y retornamos un token

Vamos a crear una clase que gestione todo lo relacionado con el token. En el siguiente ejemplo, al crear el token, estoy añadiendo el nombre de usuario como *subject*, los permisos de ese usuario(solo tenemos uno en la clase *MyUserDetails* y es *ROLE_SENSEI*) y la fecha en la que expira. Podemos agregar claims personalizados con *claim*(key, value) o pasar un mapa de claims, *setClaims()*. Para firmar el token simplemente voy a usar «key». En un proyecto real, podría recuperarse dicha key del archivo de configuración de la aplicación.

Para leer el token, necesitamos la clave secreta para validar la firma.

```
@Service
public class JwtService {
    private static final int EXPIRATION_TIME = 1000 * 60 * 60;
    private static final String AUTHORITIES = "authorities";
    private final String SECRET_KEY;

    public JwtService() {
        SECRET_KEY = Base64.getEncoder().encodeToString("key".getBytes());
    }

    public String createToken(UserDetails userDetails) {
        String username = userDetails.getUsername();
        Collection<? extends GrantedAuthority> authorities = userDetails.getAuthorities();
        return Jwts.builder()
                .setSubject(username)
                .claim(AUTHORITIES, authorities)
                .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
                .signWith(SignatureAlgorithm.HS512, SECRET_KEY)
                .compact();
    }

    public Boolean hasTokenExpired(String token) {
        return Jwts.parser()
                .setSigningKey(SECRET_KEY)
                .parseClaimsJws(token)
                .getBody()
                .getExpiration()
```

```
                .before(new Date());
    }

    public Boolean validateToken(String token, UserDetails userDetails) {
        String username = extractUsername(token);
        return (userDetails.getUsername().equals(username) && !hasTokenExpired(token));

    }

    public String extractUsername(String token) {
        return Jwts.parser()
                .setSigningKey(SECRET_KEY)
                .parseClaimsJws(token)
                .getBody()
                .getSubject();
    }

    public Collection<? extends GrantedAuthority> getAuthorities(String token) {
        Claims claims = Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getB
        return (Collection<? extends GrantedAuthority>) claims.get(AUTHORITIES);
    }
}
```

Cuando el usuario intenta iniciar sesión, esperamos un nombre de usuario y una contraseña (userAuthenticationRequest). Si la autenticación es correcta, enviamos el token (AuthenticationResponse).

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class AuthenticationRequest {
    private String username;
    private String password;
}

@Data
@NoArgsConstructor
@AllArgsConstructor
public class AuthenticationResponse {
    private String token;
}
```

Vamos a crear un controlador que se encargue del login. authenticate() recibe *UsernamePasswordAuthenticationToken* para su validación y llama a *AuthenticationProvider* y delega dicha tarea. Si la validación falla se lanza una excepción, en caso contrario se crea el token y se devuelve al usuario.

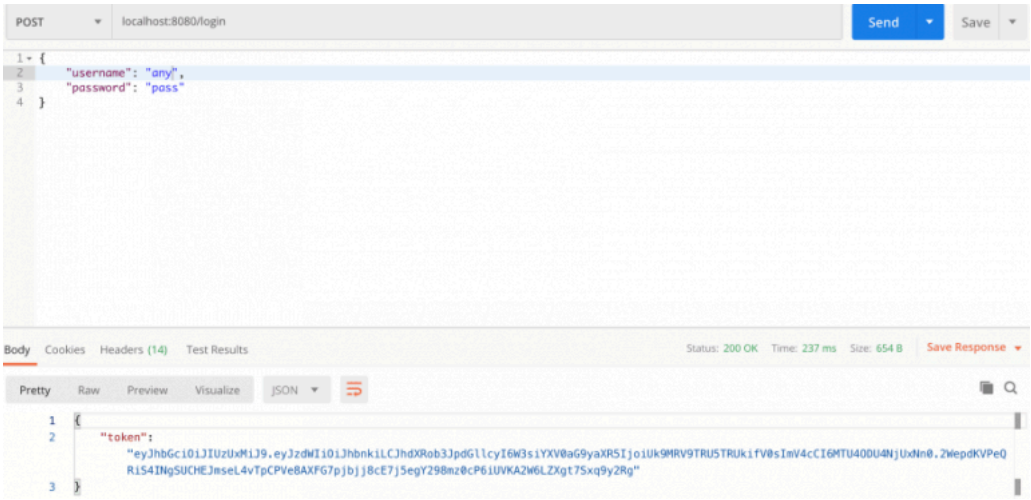
```
@RestController
@RequiredArgsConstructor
public class UserController {

    private final AuthenticationManager authenticationManager;
    private final MyUserDetailService myUserDetailService;
```

```
    private final JwtService jwtService;

    @PostMapping("/login")
    public AuthenticationResponse createToken(@RequestBody AuthenticationRequest authentic
        try {
            UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthe
            authenticationManager.authenticate(authentication);
        } catch (BadCredentialsException e) {
            throw new Exception("Invalid username or password", e);
        }
        UserDetails userDetails = myUserDetailService.loadUserByUsername(authenticationReq
        String token = jwtService.createToken(userDetails);
        return new AuthenticationResponse(token);
    }
}
```

En la clase WebScurity, debemos hacer el *override* de *authenticationManagerBean* si queremos inyectarlo(autowired) en *UserController*. Vemos cómo se han dado permisos para acceder al endpoint */login* pero cualquier otro recurso está protegido.

```
@EnableWebSecurity
public class WebSecurity extends WebSecurityConfigurerAdapter {

    @Autowired
    private MyUserDetailService myUserDetailService;

    @Override
    protected void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(myUserDetailService);
    }
    @Override
    public void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
                .authorizeRequests().antMatchers("/login").permitAll()
                .anyRequest().authenticated();
    }

    @Bean
    public PasswordEncoder getPasswordEncoder() {
        return NoOpPasswordEncoder.getInstance();
    }

    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }
}
```

Recordemos que la contraseña es 'pass' y el usuario podría ser cualquiera. Si intentamos hacer *login,* podremos ver el token devuelto.



2.2 Añadiendo un filtro

El siguiente post explica con más detalle los filtros.

SecurityContext: contiene la información del usuario autenticado.

Vamos a extraer el token de la cabecera *authorization* y validarlo. Para interceptar una solicitud, utilizamos filtros.
Primero, creamos *JwtAuthorizationFilter* que será responsable de la autorización del usuario. Obtenemos el token de la cabecera, extraemos el nombre de usuario y verificamos que sea válido. Si todo va bien, creamos el objeto de autenticación (UsernamePasswordAuthenticationToken), seteamos el usuario en *SecurityContext* y permitimos que la solicitud continúe con *filterChain.doFilter*.
He definido las constantes como campos de clase, pero se podrían extraer a una clase «JwtConstants».

```
@Component
public class JwtAuthorizationFilter extends OncePerRequestFilter {

    private static final String HEADER_TOKEN_PREFIX = "Bearer ";
    private static final String HEADER_AUTHORIZATION = "Authorization";

    @Autowired
    private MyUserDetailService myUserDetailService;

    @Autowired
    private JwtService jwtService;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpServletResponse respon
        String authorizationHeader = request.getHeader(HEADER_AUTHORIZATION);
        if (authorizationHeader != null && authorizationHeader.startsWith(HEADER_TOKEN_PRE
            String token = authorizationHeader.replace(HEADER_TOKEN_PREFIX, "");
            String username = jwtService.extractUsername(token);

            UserDetails userDetails = myUserDetailService.loadUserByUsername(username);
            if (jwtService.validateToken(token, userDetails)) {
                UsernamePasswordAuthenticationToken authentication = new UsernamePasswordA
                SecurityContextHolder.getContext().setAuthentication(authentication);
```

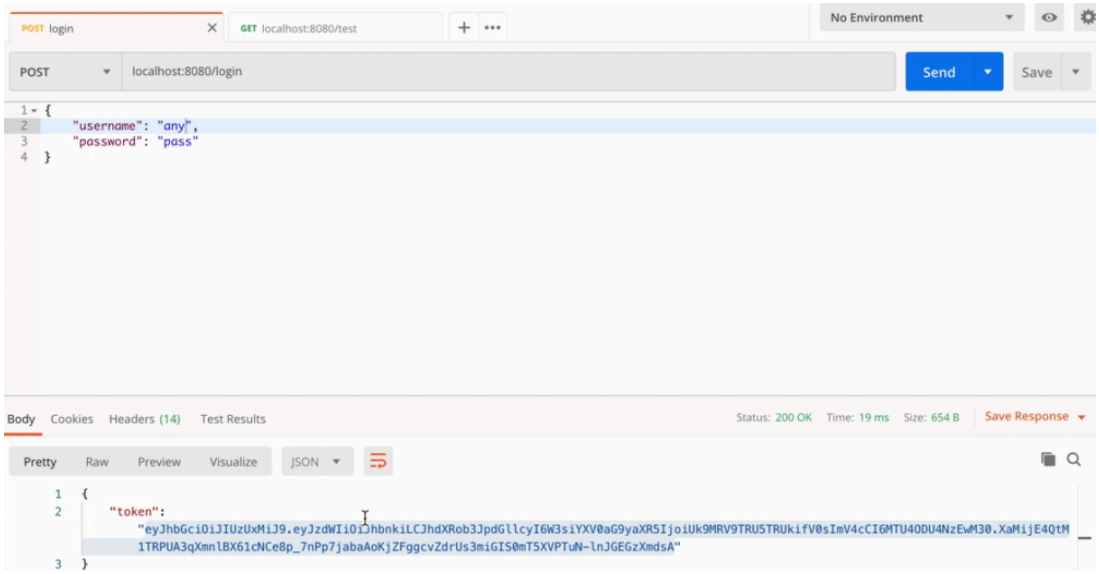
```
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

En la clase webSecurity, añadimos sessionManagement y creamos una política sin estado(stateless), ya que no queremos que spring cree ninguna sesión ni guarde el estado. En segundo lugar, añadimos el filtro creado antes del *UsernamePasswordFilter.*

```
   @Autowired
    private JwtAuthorizationFilter jwtAuthorizationFilter;

//...

    @Override
    public void configure(HttpSecurity http) throws Exception {
        http
                .csrf().disable()
                .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS
                .and()
                .authorizeRequests().antMatchers("/login").permitAll()
                .anyRequest().authenticated()
                .and().addFilterBefore(jwtAuthorizationFilter, UsernamePasswordAuthenticat
    }
//...
```

Vamos a hacer una prueba con Postman.

```
    @GetMapping("/test")
    public String getTest(){
        return "test";
    }
```



Para comprobar que el servidor no guarda el estado(**stateless**), intenta realizar una petición sin la cabecera *Authorization,* obtendrás un *403 Forbidden* ya que cada solicitud es independiente.

Jonathan Faber

Desarrollador de Software

Graduado en Ingeniería Informática, especialidad en Ingeniería del Software