Construcción de una API de generación de token

DigitalHouse>

Certified Tech Developer
The Ultimate Degree

Pasos para construir una API de generación de token

1- Agregar al POM xml las dependencias de JWT

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

2- Crear la clase JWTUtil

La misma va a estar compuesta por un generateToken que crea un token con expiración.

```
@Component
public class JwtUtil {

    private String SECRET_KEY = "secret";

    public String extractUserName(String token) {
        return extractClaimUsername(token);
    }
```

2- Crear la clase JWTUtil

La misma va a estar compuesta por un generateToken que crea un token con expiración.

```
    public Date extractExpiration(String token) {
        return extractClaimDate(token);
    }

    public Date extractClaimDate(String token){
        Claims claims = extractAllClaims(token);
        return claims.getExpiration();
}
```

```java
    public String extractClaimUsername(String token){
        Claims claims = extractAllClaims(token);
        return claims.getSubject();
    }

    private Claims extractAllClaims(String token) {
        return Jwts.parser().setSigningKey(SECRET_KEY).parseClaimsJws(token).getBody();
    }

    public String generateToken(UserDetails userDetails) {
        Map<String, Object> claims = new HashMap<>();
        return createToken(claims, userDetails.getUsername());
    }

    private String createToken(Map<String, Object> claims, String subject) {
        return Jwts.builder().setClaims(claims).setSubject(subject).setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 +60 * 10))
                .signWith(SignatureAlgorithm.HS256, SECRET_KEY).compact();
    }

    public Boolean validateToken(String token, UserDetails userDetails) {
        final String username = extractUserName(token);
        return (username.equals(userDetails.getUsername()) && !isTokenExpired(token));
    }

    private boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }
}
```

Como vemos, la librería Jwts tiene un método llamado builder que nos permite crear un token al que le pasaremos el usuario, le asignaremos una fecha de expiración y el algoritmo de encriptación.

3- Crear la clase SecurityConfiguration

Esta clase es la más importante de la configuración ya que podemos configurar la autenticación y darle acceso o no a las URLs que creamos necesarias.

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {

   @Autowired
   private MyUserDetailsService myUserDetailsService;

   @Autowired
   private JwtRequestFilter jwtRequestFilter;

   @Autowired
   private BCryptPasswordEncoder bCryptPasswordEncoder;

   @Override
   protected void configure(AuthenticationManagerBuilder auth) throws Exception {
       auth.userDetailsService(myUserDetailsService);
   }
```

```java
    @Override
    protected void configure(HttpSecurity http) throws Exception {

http.csrf().disable().authorizeRequests().antMatchers("/authenticate").permitAll().anyRequest().authenticated()
        .and().sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        http.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);
    }

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


    @Bean
    public DaoAuthenticationProvider daoAuthenticationProvider() {
        DaoAuthenticationProvider provider =
                new DaoAuthenticationProvider();
        provider.setPasswordEncoder(bCryptPasswordEncoder);
        provider.setUserDetailsService(myUserDetailsService);
        return provider;
    }
}
```

Nos ayuda a configurar un método de encriptación, podemos ver cómo son seteados nuestros bCryptPasswordEncoder y myUserDetailsService.

4- Agregar el JWTRequestFilter

El JWTRequestFilter hereda de OncePerRequestFilter. Acá es donde se va a corroborar si el token es correcto. ¡Atención! Antes de ejecutarse cualquier endpoint primero se ejecutará el método ***doFilterInternal***.

```
@Component
public class JwtRequestFilter extends OncePerRequestFilter {

   @Autowired
   private UserDetailsService userDetailsService;
   @Autowired
   private JwtUtil jwtUtil;
   @Override
   protected void doFilterInternal(HttpServletRequest httpServletRequest, HttpServletResponse
httpServletResponse, FilterChain filterChain) throws ServletException, IOException {
       final String authorizationHeader = httpServletRequest.getHeader("Authorization");
       String username= null;
       String jwt = null;
```

```
        if(authorizationHeader != null &&  authorizationHeader.startsWith("Bearer")) {
            jwt = authorizationHeader.substring(7);
            username = jwtUtil.extractUserName(jwt);
        }

        if(username != null && SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails  = this.userDetailsService.loadUserByUsername(username);
            if(jwtUtil.validateToken(jwt, userDetails)) {
                UsernamePasswordAuthenticationToken  usernamePasswordAuthenticationToken  = new
UsernamePasswordAuthenticationToken(userDetails,
                        null, userDetails.getAuthorities());
                usernamePasswordAuthenticationToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(httpServletRequest));

SecurityContextHolder.getContext().setAuthentication(usernamePasswordAuthenticationToken);
            }
        }
        filterChain.doFilter(httpServletRequest, httpServletResponse);
    }
}
```

5- Agregar la clase passwordEncoder

Esta clase va a crear un nuevo encoder de tipo BCryptPasswordEncoder.

```
@Configuration
public class PasswordEncoder {

    @Bean
    public BCryptPasswordEncoder bCryptPasswordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

6- Crear un controller

Este es el punto inicial para generar el token.

```
@RestController
public class JwtController {

   @Autowired
   private AuthenticationManager authenticationManager;
   @Autowired
   private UserDetailsService userDetailsService;
   @Autowired
   private JwtUtil jwtUtil;


   @RequestMapping(value = "/authenticate", method = RequestMethod.POST)
   public ResponseEntity<?> createAuthenticationToken(@RequestBody AuthenticationRequest authenticationRequest) throws Exception{
       try {
           authenticationManager.authenticate(new UsernamePasswordAuthenticationToken(authenticationRequest.getUsername(),
authenticationRequest.getPassword()));
       }catch (BadCredentialsException e) {
           throw new Exception("Incorrect", e);
       }
       final UserDetails userDetails = userDetailsService.loadUserByUsername(authenticationRequest.getUsername());
       final String jwt = jwtUtil.generateToken(userDetails);

       return ResponseEntity.ok(new AuthenticationResponse((jwt)));
   }
}
```

DigitalHouse>