Bloque 14. Spring Security

Conceptos Básicos de Autenticación y Autorización en API REST **1**
Seguridad en Spring Boot **2**
Creación de un login 3
Cómo funciona un token JWT 5
Creando un token JWT 6
Validación de un token JWT 7
Autorización **8**
Obtención del usuario logueado **8**
Autenticación con LDAP **9**

Conceptos Básicos de Autenticación y Autorización en API REST

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

- Servidor: Aplicación que contiene los recursos protegidos mediante una API REST.
- Cliente: Aplicación que realiza peticiones al servidor para interactuar con los recursos protegidos.
- Autenticación: Proceso mediante el cual un cliente garantiza su identidad. Un ejemplo sencillo sería el uso de un nombre de usuario y contraseña.
- Autorización: Proceso mediante el cual se determina si un cliente tiene la autoridad, o autorización, para acceder a ciertos recursos protegidos.

Ejemplos oficiales
Ejemplo usado en esta guía.

Para los siguiente ejemplos que se muestran a continuación, se usarán las siguientes dependencias:

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

<!-- LDAP dependencies-->
<dependency>
  <groupId>org.springframework.ldap</groupId>
  <artifactId>spring-ldap-core</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-ldap</artifactId>
</dependency>
<dependency>
  <groupId>com.unboundid</groupId>
  <artifactId>unboundid-ldapsdk</artifactId>
</dependency>

<!--JWT dependency-->
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt</artifactId>
  <version>0.12.5</version>
</dependency>
```

Seguridad en Spring Boot

Para este ejemplo, crearemos un login y diferentes endpoint securizados, usando una base de datos H2 y tokens JWT.

Para poder habilitar la seguridad en Spring, se requiere la notación **@EnableWebSecurity**, posteriormente para poder configurar a nivel global, usamos el siguiente bean:

```
protected HttpSecurity securityFilterChainPrivate(HttpSecurity http) throws Exception {
  http
        .csrf((AbstractHttpConfigurer::disable))//desabilito csrf
        .authorizeHttpRequests(authorize -> authorize
              .requestMatchers("/auth/login").permitAll())
        .authorizeHttpRequests(a->a.requestMatchers(HttpMethod.POST).hasRole("ADMIN"))
        .authorizeHttpRequests(authorize -> authorize
              .anyRequest().authenticated())
        .sessionManagement(session -> session
              .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class)
      .exceptionHandling(handler -> handler.authenticationEntryPoint(exceptionHandling));

  return http;
}
```

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
  super.securityFilterChainPrivate(http)
        .authenticationManager(databaseAuthenticationManager);

  return http.build();
}
```

Para este ejemplo, estamos usando la clase abstracta **WebSecurityConfigAbstract**, que luego se implementa con **DatabaseAuthenticationManager** y **LdapAuthenticationManager** esto depende del perfil que usemos, normalmente (y es lo que se recomienda) se podrá implementar todo el contenido del securityFilterChain en una misma clase .

En la clase abstracta definimos el método base que tendremos para configurar la seguridad:

- **CSRF**: esta funcionalidad sirve para añadir seguridad en los formularios, generando un token aleatorio que se deba recibir cuando se envíe cualquier

información de vuelta a Spring, para nuestro ejemplo lo deshabilitamos. Guide to CSRF Protection in Spring Security.

- **AuthorizeHttpRequests**: este método lo podremos usar tantas veces como queramos, pero filtrando del más restrictivo, al menos restrictivo, para este ejemplo, hemos buscado el endpoint del login, y directamente lo permitimos, para el siguiente, hemos filtrado todos los métodos POST, y requerimos que el usuario que entre, tenga el rol ADMIN, Y por último para el resto de endpoints, requerimos que el usuario esté autenticado.
- **SessionManagement**: Para autenticar al usuario a parte del login, se pueden usar tokens de sesión, esto es un token temporal que se le asigna a un usuario, una vez se haya autenticado, para este ejemplo usaremos tokens JWT, por lo que no requerimos de sesiones Guide to Spring Session.
- **AddFilterBefore**: Este método añade un filtro para cada llamada a cualquier endpoint al que se llame, servirá para poder autenticar a un usuario con un token JWT.
- **AuthenticationManager**: Este método sirve para definir quien atenderá a la llamada de authenticate, que suele realizarse en el endpoint "/login".
- **ExceptionHandling**: Aquí indicaremos la clase que manejará los errores de autenticación.

En el bean final que se crea habremos definido un objeto de tipo **SecurityFilterChain**, conteniendo toda la configuración definida.

Creación de un login

Para poder autenticarnos, crearemos un endpoint login(), en el que por su body, se incluirán el usuario y contraseña.

```
@PostMapping("/login")
public LoginDto login(@RequestBody InputLoginDto inputLoginDto) {

  Authentication authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(inputLoginDto.getUsername(),
  inputLoginDto.getPassword()));

  if (authentication == null) {
     throw new BadCredentialsException("Invalid username or password.");
  }

  UserDetailsImpl userDetails = (UserDetailsImpl) authentication.getPrincipal();
  return LoginDto.builder()
        .type("Bearer")
        .token(jwtUtils.generateJwtToken(authentication))
        .refreshToken(jwtUtils.generateJwtToken(authentication, true))
```

```
  .authorities(authentication.getAuthorities().stream()
  .map(GrantedAuthority::getAuthority).toList()).username(userDetails.getUsername())
  .userType(userDetails.getUserType()).build();
}
```

Primero llamaremos al método authenticate de **authenticationManager**, que llamara internamente al manager gestionado por base de datos, que es el siguiente:

```
@Profile("databaseAuth")
@Component
@AllArgsConstructor
public class DatabaseAuthenticationManager implements AuthenticationManager {

  private final UserRepository userRepository;
  private final PasswordEncoder passwordEncoder;

  @Override
  public Authentication authenticate(Authentication authentication) throws
  AuthenticationException {

  UserJpa user = userRepository.findByName((String) authentication.getPrincipal())
          .orElseThrow(() -> new UsernameNotFoundException("User not found"));
  boolean matches = passwordEncoder.matches((CharSequence)
  authentication.getCredentials(), user.getPassword());

  if (!matches) {
      throw new BadCredentialsException("Wrong password");
   }

   //successful login
   user.setPassword(null);

  List<SimpleGrantedAuthority> authorities = List.of(new SimpleGrantedAuthority("ROLE_"
  + user.getRole().getName().toUpperCase()));

  return new UsernamePasswordAuthenticationToken(UserDetailsImpl.builder()
          .id(user.getId())
          .username(user.getName())
          .userType(UserDetailsImpl.UserType.DATABASE)
          .authorities(authorities)
          .build(), null, authorities);

  }
}
```

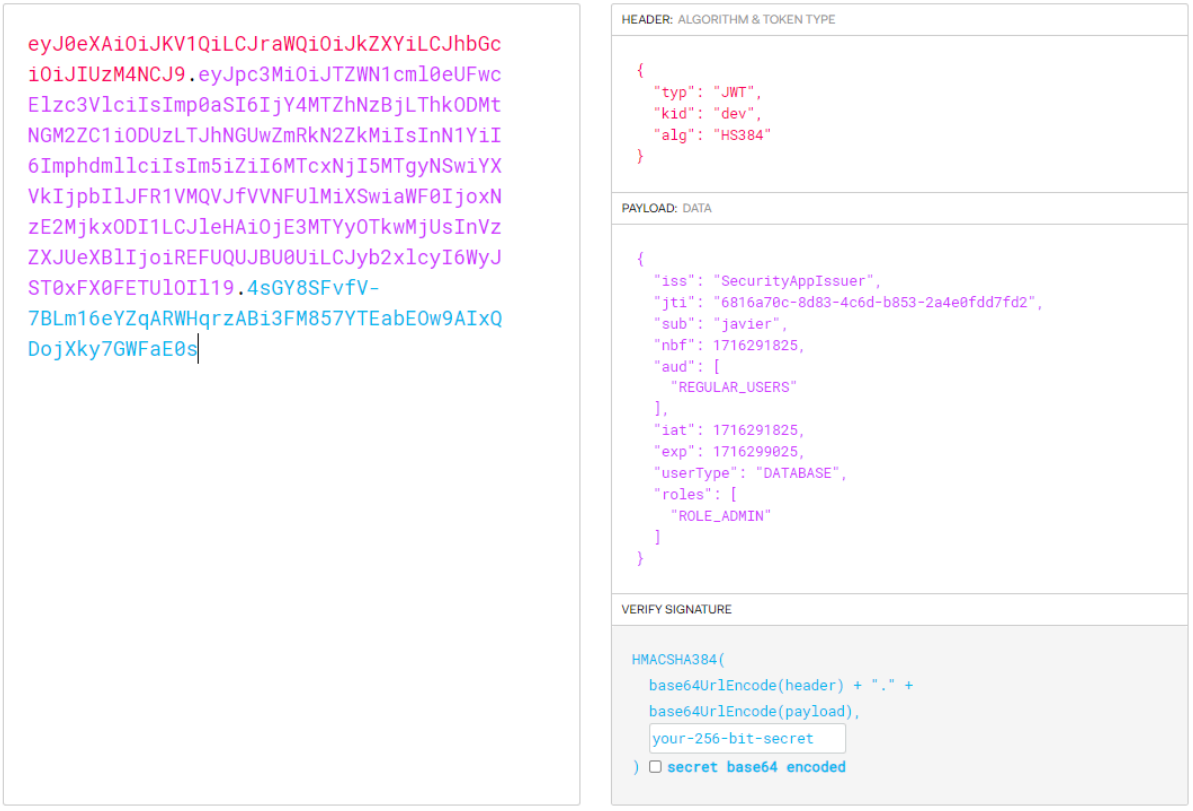
A través de los datos que se han proporcionado desde /login, se busca el usuario en la base de datos, y comprueba la contraseña, que se encuentra encriptada en bbdd.

Una vez se ha validado el usuario, se crea una lista de **SimpleGrantedAuthority**, que servirán para poder validar los roles que tenga un usuario, y posteriormente, se crea **UsernamePasswordAuthenticationToken**, que retornara una autenticación exitosa de nuestro usuario y devolverá los datos del usuario haciendo uso de la clase UserDetailsImpl, que es una implementación de la interfaz de Spring UserDetails.
En caso de error lanzará una excepción.

Volviendo al endpoint, vemos que se devuelve un objeto de tipo **LoginDto**, que contiene 2 tokens, y los datos del usuario.

Cómo funciona un token JWT

Un token JWT es un token firmado irevocable, esto implica que una vez expedido, siempre podremos identificar a un usuario, contiene una fecha de expiración, para que tras pasado un tiempo, el token ya no pueda ser usado.
Ejemplo de token:



Como vemos en la imagen, el token contiene 3 partes, en el header se informa el tipo de token, el tipo de encriptación y se puede incluir con que clave se ha firmado (KID).
La segunda parte es el Payload, que son los datos que podemos incluir en el token podemos incluir los siguientes datos:

- subject: el usuario al que queremos identificar.
- issued At: fecha de expedición.
- expiration: fecha de expiración.
- Not Valid Before: fecha desde que el token empieza a ser efectivo.
- JTI: Identificador único de un Token.
- issuer: quien expide el token.
- audience: Indica para quién va dirigido el token

Adicionalmente podemos incluir campos customizados, que nos puedan ayudar a identificar al usuario.
y por último, tenemos la firma, que sirve para poder verificar que el token lo hayamos firmado nosotros.
Referencia: JSON Web Token Introduction - jwt.io

Creando un token JWT

Para crear un token usaremos el siguiente método de la clase **JwtUtils.**

```
public String generateJwtToken(Authentication authentication, boolean refreshToken) {

  UserDetailsImpl userDetails = (UserDetailsImpl) authentication.getPrincipal();
  //if token generated is for refresh, give extra time
  Long extraExpirationTime = refreshToken ? 600000L : 0L;
  SecretKey key = Keys.hmacShaKeyFor(Decoders.BASE64URL.decode(jwtSecret));
  return Jwts.builder()
        .claims()
        .issuer(issuer)
        .subject((userDetails.getUsername()))
        .issuedAt(new Date())
        .expiration(new Date((new Date()).getTime() + jwtExpirationMs +
        extraExpirationTime))
        .add(CLAIM_USER_TYPE, userDetails.getUserType())
        .add(CLAIM_ROLES, userDetails.getAuthorities()
        .stream().map(GrantedAuthority::getAuthority).toList())
        .and()
        .signWith(key)
        .compact();
}
```

Usaremos los datos provenientes de la autenticación que previamente hemos realizado, para poder guardar los datos en el token.

Primero realizamos la comprobación de si el token a generar es de refresco, que añadirá 10 minutos más (600000 ms).

A continuación obtendremos la clave de firmado, o bien de un fichero de configuración o de un propio fichero a parte, en este caso usamos el fichero de configuración.

```
@Value("${jwt.secret}")
private String jwtSecret;
```

Y por último generamos el token, usando la clase Jwts, con .claims() introducimos el contenido del token, como quien lo expide (issuer), a quien autenticamos (subject), fechas de expedición y expiración y 2 campos customizados, uno para indicar el tipo de usuario y otro donde incluiremos los roles del usuario.

Validación de un token JWT

Para validar un token JWT usaremos el siguiente método, también alojado en **JwtUtils**

```
public Optional<UserDetailsImpl> validateJwtToken(String jwt) {
  try {
     SecretKey key = Keys.hmacShaKeyFor(Decoders.BASE64URL.decode(jwtSecret));
     Claims claimsJwe = Jwts.parser().verifyWith(key).build()
     .parse(jwt).accept(Jws.CLAIMS).getPayload();
     List<String> authorities = claimsJwe.get(CLAIM_ROLES, List.class);
     return Optional.of(UserDetailsImpl.builder()
           .username(claimsJwe.getSubject())
           .userType(UserDetailsImpl.UserType
           .valueOf(claimsJwe.get(CLAIM_USER_TYPE, String.class)))
           .authorities(authorities.stream().map(SimpleGrantedAuthority::new).toList())
           .build());
  } catch (MalformedJwtException e) {
     log.error("Invalid JWT token: {}", e.getMessage());
  } catch (ExpiredJwtException e) {
     log.error("JWT token is expired: {}", e.getMessage());
  } catch (UnsupportedJwtException e) {
     log.error("JWT token is unsupported: {}", e.getMessage());
  } catch (IllegalArgumentException e) {
     log.error("JWT claims string is empty: {}", e.getMessage());
  } catch (JwtException e) {
     log.error("Invalid JWT signature: {}", e.getMessage());
  }

  return Optional.empty();
}
```

Por parámetro se le pasará el token jwt, sin la parte de Bearer.

En primer lugar, recuperamos la clave privada y a través de la clase Jwts, usamos el método de verificación como se muestra en el ejemplo.
Junto con la verificación, recuperamos el payload que tiene el token.
En este paso de verificación se lanzará una excepción, esta la controlamos y en su lugar devolveremos un Optional vacío
Y si la verificación es exitosa, crearemos un objeto **UserDetailsImpl** a través de los datos que obtenemos del Payload del token.

Autorización

Como hemos visto anteriormente, la autorización consiste en verificar si un usuario puede acceder a ciertos recursos, esto es posible una vez realizada la autenticación.

Para controlar esto, se puede realizar mediante securityFilterChainPrivate como hemos visto anteriormente.
A través de este método, podemos restringir el acceso a ciertos usuarios, filtrando por roles, autoridades o si simplemente está autenticado.

A su vez, también podemos indicar que no es necesario que el usuario esté autenticado para acceder a los contenidos, como pasa con /auth/login.
Otra manera que tenemos para restringir el acceso a los endpoints, es en los propios controllers, para ello debemos agregar la notación: @EnableMethodSecurity en la clase de configuración, en nuestro caso **WebSecurityConfigDatabase** y posteriormente en los controladores, usaremos @PreAuthorize, como vemos en el siguiente ejemplo:

```
@GetMapping("/paged")
@PreAuthorize("hasRole('ADMIN') || hasRole('USER')")
public Page<CountryJpa> findAllPaged(@RequestParam(defaultValue = "0",
required = false) int page, @RequestParam(defaultValue = "10",
required = false) int pageSize) {
  return countryService.findAllCountriesPaged(PageRequest.of(page, pageSize));
}
```

Para este endpoint /paged, solo se podrá acceder si tenemos los roles de Admin o User.
Esta notación también se puede usar a nivel de clase, aplicando a todos los endpoints que contenga, pero si en cualquier método se usa de nuevo esta notación, prevalecerá la que tenga el método, sin validar la de nivel de clase.

Adicionalmente de esta notación hay más, que dejaremos en la siguiente guia:
[Introduction to Spring Method Security]

Obtención del usuario logueado

Dentro de cualquier lugar de la aplicación, siempre y cuando sea llamado desde un controller, podremos obtener el usuario que realiza la petición, para ello usaremos lo siguiente:

```
UserDetailsImpl user = (UserDetailsImpl)
SecurityContextHolder.getContext().getAuthentication().getPrincipal();
```

Desde SecurityContextHolder, podremos obtener la autenticación que previamente hemos generado desde **DatabaseAuthenticationManager** y poder realizar diferentes acciones dependiendo del usuario que esté logado.

Autenticación con LDAP

LDAP (Protocolo Ligero de Acceso a Directorios) es un protocolo estándar de la industria utilizado para acceder y mantener información de directorio en un servidor LDAP. Los servidores LDAP almacenan datos de directorio de manera jerárquica y son comúnmente utilizados para autenticación, autorización y almacenamiento de información de usuarios, grupos, dispositivos y otros recursos en una red.

Configuración basica en Springboot

```
@Configuration
@Profile("LDAPAuth")
public class SecurityLDAPConfig {

  @Value("${spring.ldap.urls}")
  private String url;

  @Value("${spring.ldap.username}")
  private String username;

  @Value("${spring.ldap.password}")
  private String password;

  @Autowired
  protected void configure(AuthenticationManagerBuilder auth) throws Exception {
     auth
          .ldapAuthentication()
             .userDnPatterns("uid={0},ou=people") // Patrón de búsqueda del usuario en el directorio
LDAP
          .contextSource()
          .url(url) // URL del servidor LDAP
          .managerDn(username) // Credenciales de administrador LDAP
          .managerPassword(password); // Contraseña del administrador LDAP
  }
}
```

En este ejemplo:

- userDnPatterns("uid={0},ou=people"): Especifica el patrón de búsqueda de usuarios en el servidor LDAP.
- contextSource(): Configura la conexión al servidor LDAP, especificando la URL del servidor, el DN del administrador y la contraseña del administrador

También es necesario, definir las siguientes properties.

```
#LDAP
spring.ldap.urls=ldap://localhost:389
spring.ldap.base=dc=nter,dc=es
spring.ldap.username=cn=admin,dc=nter,dc=es
spring.ldap.password=admin_password
```

Como configuración, es necesario definir un fichero ldfi para definir los diferentes usuarios permitidos y sus diferentes roles.

```
dn: uid=bob,ou=people,dc=nter,dc=es
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Bob Hamilton
sn: Hamilton
uid: bob
userPassword: bobspassword
```

Para realizar la autenticación y compatibilidad con el proyecto se ha definido **WebSecurityConfigLDAP**, en el que hemos incluido lo siguiente:

```
@Bean
AuthenticationManager authenticationManager(BaseLdapPathContextSource
contextSource, LdapAuthoritiesPopulator authoritiesPopulator) {
  LdapBindAuthenticationManagerFactory factory = new
LdapBindAuthenticationManagerFactory(contextSource);
  factory.setUserDnPatterns("uid={0},ou=people");
  factory.setLdapAuthoritiesPopulator(authoritiesPopulator);
  return new LdapAuthenticationManager(factory.createAuthenticationManager());
}
```

Primero crear un Bean para instanciar el **AutenticationManager**, que se usará el que tiene por defecto el LDAP, generado por **LdapBindAuthenticationManagerFactory** y luego se customiza con la clase **LdapAuthenticationManager**.

Para el mapeo de roles, se usará el siguiente método:

```
@Bean
public LdapAuthoritiesPopulator ldapAuthoritiesPopulator(BaseLdapPathContextSource
contextSource) {
```

```
  DefaultLdapAuthoritiesPopulator authoritiesPopulator = new
DefaultLdapAuthoritiesPopulator(contextSource, "ou=groups");
  authoritiesPopulator.setGroupSearchFilter("uniqueMember={0}");
  authoritiesPopulator.setSearchSubtree(true);
  authoritiesPopulator.setIgnorePartialResultException(true);
  authoritiesPopulator.setGroupRoleAttribute("cn"); // Specify the attribute within the
group to be used as a role
  authoritiesPopulator.setRolePrefix("ROLE_"); // If your roles start with a prefix, specify it
here (e.g., "ROLE_")

  authoritiesPopulator.setSearchSubtree(true);
  return authoritiesPopulator;
}
```

Este método define cómo se deben buscar los grupos para los usuarios.

Y por último usaremos el FilterChain para LDAP de igual manera que se hace con el ejemplo anterior con base de datos:

```
@Bean
public SecurityFilterChain securityFilterChainLDAP(HttpSecurity http) throws Exception {
  securityFilterChainPrivate(http);

  return http.build();
}
```

A continuación, veremos el sistema para que se realice la autenticación de manera customizada, que se realiza con la clase **LdapAuthenticationManager**.

```
@Component
public class LdapAuthenticationManager implements AuthenticationManager {

  private final AuthenticationManager baseAuthManager;

  LdapAuthenticationManager(AuthenticationManager authenticationManager) {
     baseAuthManager = authenticationManager;
  }

  @Override
  public Authentication authenticate(Authentication authentication) throws
AuthenticationException {
     Authentication baseAuth = baseAuthManager.authenticate(authentication);
     LdapUserDetailsImpl user = (LdapUserDetailsImpl) baseAuth.getPrincipal();
```

```
    List<SimpleGrantedAuthority> authorities =
baseAuth.getAuthorities().stream().map(itm -> new
SimpleGrantedAuthority(itm.getAuthority())).toList();
    return new UsernamePasswordAuthenticationToken(UserDetailsImpl.builder()
            .username(user.getUsername())
            .userType(UserDetailsImpl.UserType.LDAP)
            .authorities(authorities)
            .build(), null, authorities);
  }
}
```

Esta clase recibe el AuthenticationManager que se ha generado previamente por LDAP, este se encargará de validar que el usuario exista, que su contraseña sea correcta, y de traer todos sus datos, esto se realiza en el método authenticate, que llama al mismo método de autenticar, de baseAuthManager, una vez realizado este primer paso, convertimos el objeto que nos trae LDAP, para que devuelva una autenticación con el objeto **UserDetailsImpl**.