AUTENTICAZIONE DI SERVIZI REST CON JWT [SPRING]

24 Comments / Di Dario Frongillo / In Back End, Java, Spring / 16 Dicembre 2017

Di Dario Frongillo
16 Dicembre 2017

PREREQUISITI

Questo articolo è rivolto ai lettori che conoscono l'architettura RESTful e hanno le basi del framework Spring e del sotto progetto Spring Security.

INTRODUZIONE

Una delle esigenze più comuni del WEB è quella di restringere l'accessibilità di una o più pagine di un applicazione web ai soli utenti autenticati o addirittura ad una ristretta cerchia di utenti (ad es. agli utenti admin). Questa esigenza si presenta anche nel mondo delle API. Nella prima parte di questo articolo spiegherò il principio su cui si basa JWT mentre nella seconda parte scriveremo una semplice implementazione di autenticazione Spring su JWT.

AUTENTICAZIONE STATELESS

La gestione dello stato (se necessaria) avviene sul client, ottimizzando le prestazioni del server che non deve curare lo stato. L'approccio stateless di un architettura RESTful rende quindi inappropriato l'utilizzo dei cookie di sessione per il meccanismo di autenticazione.

La soluzione di autenticazione Stateless piu' semplice è l'autenticazione BASIC: ad ogni chiamata del client viene passato un token in un header http contenente le credenziali in modo che il server ad ogni chiamata possa autenticare il client. In questo modo mantengo stateless il mio backend RESTful ma ottengo un overhead dovuto al fatto che il server debba autenticare ad ogni richiesta il client. Questa soluzione non è quindi adatta a sistemi di produzione: immaginiamo di dover ri-eseguire ad ogni richiesta le query di autenticazione sul nostro db su un sistema con migliaia di utenti online nel singolo istante! Una soluzione completamente Stateless, scalabile e con nessuno overhead è quella oggetto di questo articolo: JWT (JSON WEB TOKEN)

JSON WEB TOKEN

Prima di analizzare il meccanismo di autenticazione su JWT

come un container per le claims degli user. I claims sono informazioni di un utente: chi è l'utente loggato (e altre info di contorno: ad es mail, nome, cognome), scadenza di un token (quindi scadenza dell'autenticazione dell'utente), privilegi dell'utente (è un admin o un normale utente) ed altre info customizzabili a seconda del contesto applicativo .
Tali informazioni sono firmate lato server secondo la specifica JSON Web Signature (JWS), quindi il server è in grado di riconoscere se sono state generate da lui o meno. Riportiamo un esempio di token JWT:

```
1.  eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbiIsImF1ZGllbmNlIjoid2ViIi
A
```

Come si evince dall'esempio identifichiamo 3 parti in un token JWT

Header : Se decodifichiamo la prima parte del token scopriamo che si riferisce ad un json che contiene le due voci "typ" e "alg". Il primo ha come valore sempre "JWT",

mentre il nodo “alg” contiene il nome dell’algoritmo usato per il

Di Dario Frongillo
16 Dicembre 2017

Payload : La seconda parte del token è il corpo vero e proprio che contiene dei dati “variabili” in base al contesto in formato json e codificati in base64. Nel nostro esempio il payload decodificato è

```
{
  "sub": "admin",
  "aud": "web",
  "roles": [
    "ROLE_ADMIN",
    "ROLE_USER"
  ],
  "isEnabled": true,
  "exp": 1512975450,
  "iat": 1512968250230
}
```

JWT propone dei campi “standard” da utilizzare nel payload (es: sub, audience, exp e iat) ma il programmatore può inserire proprietà aggiuntive (vedi isEnabled e roles da me definiti e non contemplati nei clams standard riportato nel seguente LINK). Andando ad analizzare il json del body possiamo interpretare dal token le seguenti info:

- utente admin (claim sub)
- ha come target il web (claim aud)
- ruoli utente ROLE_ADMIN, ROLE_USER (claim roles)

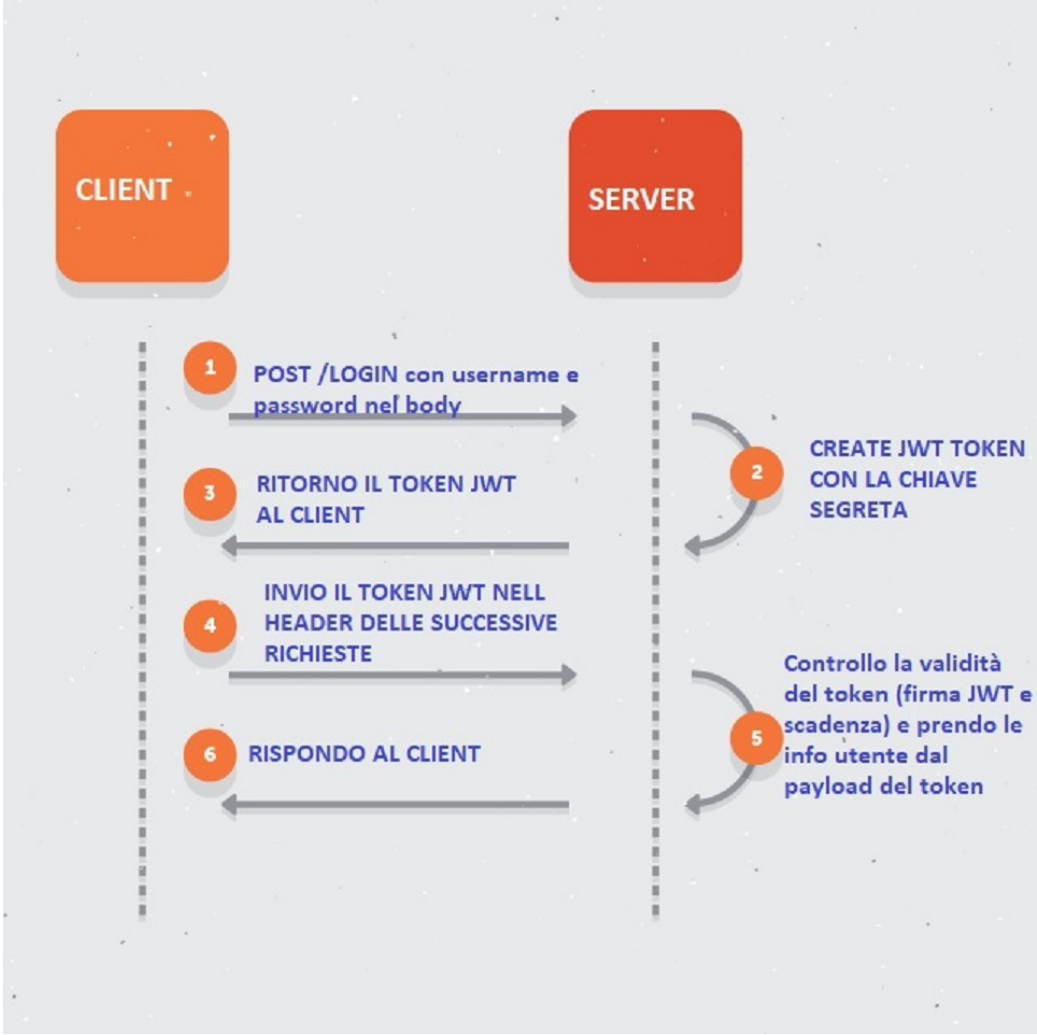
sarà più ritenuto valido dal server.

Signature: La terza parte di un token è la firma che non è altro che il risultato di una funzione hash 256 che prende in input la codifica base64 dell'header concatenandola con un punto alla codifica base64 del payload, il tutto codificato con la nostra "chiave segreta" che solo il server conosce.

Con questa piccola introduzione sui JSON WEB TOKEN capiamo subito che questi token possono esser un ottimo modo per trasportare all'interno di ogni richiesta HTTP info legate all'autenticazione di un utente in modo totalmente sicuro: il server grazie al campo signature è in grado di capire che tali info sono state generate da lui grazie alla firma e che quindi garantiscono che tale richiesta è da considerarsi relativa ad un utente autentificato.

FLUSSO LOGICO DI AUTENTICAZIONE JWT

autenticazione JWT può essere riassunto in questo schema.



1. Il client, utilizzando l'endpoint di login, invia le proprie credenziali al server per autenticarsi.
2. Il server verifica se le credenziali postate sono corrette. In

caso positivo costruisce un token jwt utilizzando la chiave

Di Dario Frongillo
16 Dicembre 2017

importante solo il server deve conoscere la chiave privata)

3. tale token viene inviato nella risposta all'interno di un header http (nella implementazione che proporremo sotto utilizzeremo l header http **X-auth**)
4. Il client salva il token jwt risultato dell'autenticazione, e nelle successive richieste richieste inoltra sempre nell'apposito header http **X-auth** il token restituito nel punto 3. Tale token funge da "passaporto" per garantire che tali successive richieste sono autenticate.
5. Il server riceve il token ad ogni successiva richiesta. Ogni volta si assicura che il token sia stato firmato e autentificato con la sua chiave privata ed infine che non sia scaduto. Poiché il token contiene al suo interno il payload con le informazioni necessarie all'autenticazione (es. username e ruoli utente), il server eviterà di fare query ogni volta per verificare a quale utente corrisponde quel token e di recuperare le info utente necessarie per ricostruire lo stato della sessione (ottimo

risultato per la scalabilità ed eliminando totalmente l'overhead della soluzione BASIC descritta ad inizio articolo).

6. Se passati i controlli descritti dal punto 5, il server tratterà

l'utente come autenticato e gestirà la richiesta. altrimenti

Di Dario Frongillo
16 Dicembre 2017

SVANTAGGI DI JWT

Come tutte le cose di questo mondo anche JWT ha qualche svantaggio o problematica che è opportuno conoscere se si vuole utilizzare.

1. **Pericolo di Compromette la Secret key utilizzata per firmare il token**: Utilizzando una sola chiave per firmare il token introduce un problema: se tale chiave viene scoperta l'intero sistema è compromesso. L'utente con cattive intenzioni che scopre la chiave può quindi accedere a dati sensibili! Se i vostri sistemi che utilizzano JWT sono sistemi con dati fortemente sensibili (es. banche) vi consiglio altamente di non utilizzare sempre la solita chiave segreta ma ad es. di utilizzare una chiave segreta diversa per giorno per firmare i

   vostri token; questo pero' implica che al cambio di chiave tutti token esistenti verranno invalidati.
2. **Impossibilità di gestire i client dal server**: Supponiamo che

un vostro cliente vi chiami per dirvi che uno degli utenti ha

tale funzionalità è delegata al client (ad es. una web app implementa il logout su JWT andando a pulire il token dal cookie). In tali situazioni uno dei possibili workaround è quello di generare il meccanismo di **blacklist:** ovvero predisporre ad es una tabella su DB contenente tutti gli utenti da mettere momentaneamente in blacklist; in questa soluzione il server ad ogni richiesta verificherà se l'utente riferito dal token è presente nella blacklist e in caso positivo rifiuterà tale richiesta. A causa dell'overhead introdotto dal controllo della blacklist, consiglio altamente di predisporre la blacklist su cache utilizzando ad esempio Redis. Dobbiamo però tenere presente che in questo modo il nostro sistema non è più completamente stateless a causa dell'introduzione dello "stato" della blacklist.

3. Impossibilità di essere a conoscenza degli user attualmente loggati: Essendo stateless il server non è a conoscenza degli utenti attualmente loggati.
4. Lunghezza di un token: Come descritto in questo articolo le info significative che servono al server vengono salvate nel payload del token; se le info di sessione sono molte il token può raggiungere dimensioni significative; supponiamo ad esempio che raggiunga le dimensioni di un 1 KB; questo

implica che viene introdotto un overhead di 1 KB per ogni

Di Dario Frongillo
16 Dicembre 2017

IMPLEMENTAZIONE JWT CON SPRING BOOT

Ora che sappiamo i principi dei JWT proviamo ad implementare il flusso logico di autenticazione basato su JWT descritto nel paragrafo precedente utilizzando il framework Spring Boot. Il codice sorgente lo trovare sul mio GITHUB .

CONFIGURAZIONI

Andiamo ad analizzare le configurazioni che ho predisposto nel file **application.properties**.

L'esempio utilizza come persistence layer Spring Data su DB mysql. Se volete quindi provare sulla vostra macchina il progetto

dovrete installarvi prima mysql e sostituire nel progetto scaricato il valore dei seguenti parametri.

```
1.  spring.datasource.username=root
```

Di Dario Frongillo
16 Dicembre 2017

Per la creazione e gestione del token JWT e delle info contenute in esso ho utilizzato la libreria **jjwt**

```
<dependency>
   <groupId>io.jsonwebtoken</groupId>
   <artifactId>jjwt</artifactId>
   <version>0.9.0</version>
</dependency>
```

Nel file **application.properties** trovate le configurazioni utilizzate dal server per l'autenticazione su jwt

```
jwt.header: X-Auth
jwt.secret: mySecret
jwt.expiration: 7200
```

Tali proprietà rappresentano:

1. il nome dell'header http in cui il server si aspetta di trovare il

token JWT

Di Dario Frongillo
16 Dicembre 2017

3. Il tempo di vita di un token in secondi.

SPRING SECURITY

In questa parte di articolo non mi dilungherò nel spiegare le basi di Spring Security ma mi concentrerò su come configurarlo per utilizzarlo con JWT.

Normalmente Spring Security, dopo un login avvenuto con successo, genera un cookie che viene passato automaticamente ad ogni richiesta client-server andando a memorizzare nell'elenco delle sessioni attive sul server le info relative a quell'utente. Grazie al cookie ad ogni richiesta client Spring security, tramite dei filtri che scattano prima di processare ogni richiesta, è in grado quindi di verificare se l'utente è autenticato ottenendo quello che Spring chiama **Principal:** ovvero le info utente dell'utente corrente. Nella

soluzione JWT dobbiamo quindi customizzare Spring Security per:

- non utilizzare Cookie

operazione dovrà essere processata prima delle operazioni di verifica autenticazione eseguite dai filtri standard di Spring Security.

- ricostruire il **Principal** di String a partire dal payload del token jwt e viceversa.

Riportiamo le classi principali che ho sviluppato per ottenere quanto appena riportato.

WebSecurityConfig.java

E' la classe fondamentale per configurare Spring Security.

```
1.  @Configuration
2.  @EnableWebSecurity
3.  @EnableGlobalMethodSecurity(prePostEnabled = true)
4.  public class WebSecurityConfig extends WebSecurityConfigurerAda
5.
6.      @Autowired
7.      private JwtAuthenticationEntryPoint unauthorizedHandler;
8.
9.      @Autowired
10.     private UserDetailsService userDetailsService;
11.
12.     @Autowired
13.     public void configureAuthentication(AuthenticationManagerBu
```

Di Dario Frongillo
16 Dicembre 2017


```
authenticationManagerBuilder) throws Exception {
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public JwtAuthenticationTokenFilter authenticationTokenFilt
Exception {
        return new JwtAuthenticationTokenFilter();
    }


    @Override
    protected void configure(HttpSecurity httpSecurity) throws
        httpSecurity
                .csrf().disable()

.exceptionHandling().authenticationEntryPoint(unauthorizedHandle
                // non abbiamo bisogno di una sessione

.sessionManagement().sessionCreationPolicy(SessionCreationPolicy
                .cors().and()
                .authorizeRequests()
                .antMatchers(
                        //HttpMethod.GET,
                        "/",
                        "/*.html",
                        "/favicon.ico",
                        "/**/*.html",
                        "/**/*.css",
                        "/**/*.js"
                ).permitAll()
                .antMatchers("/public/**").permitAll()
                .anyRequest().authenticated();

        // Filtro Custom JWT
        httpSecurity.addFilterBefore(authenticationTokenFilterB
UsernamePasswordAuthenticationFilter.class);

        httpSecurity.headers().cacheControl();
```

- Viene impostato come meccanismo di autenticazione il servizio che implementa l'interfaccia **UserDetailsService** utilizzando come password-encoder bCrypt, questo vuol dire che le password dovranno essere salvate sul db criptate con bCrypt. In breve dato un utente e una password(in chiaro) pervenute in una richiesta sull'url di login, Spring Security utilizzerà il servizio userDetailsService per verificare l'esistenza dell'utente e la correttezza della password.
- Attraverso il metodo configure vengono impostate le direttive di autenticazione:
    - Non tutti gli endpoint dovranno essere utilizzabili solo da utenti autenticati, alcuni potranno essere pubblici: vedi endpoint di login o di registrazione. Come distinguere gli endpoint protetti da quelli pubblici? Per semplicità ho configurato Spring Security per non richiedere autenticazione per gli URL che iniziano con

      public/** ; infatti vedremo successivamente che l'endpoint di login è stato configurato per questo motivo sotto l'url **public/login**. Tutte le richieste verso degli URL

che non iniziano per public/* dovranno essere quindi

- Viene creato e impostato un filtro di tipo **JwtAuthenticationTokenFilter** per scattare prima dei filtri di Spring Security che hanno il compito di verificare l'autenticazione.

Riportiamo quindi la classe appena citata: **JwtAuthenticationTokenFilter,**andando a descriverne lo scopo. Tale filtro ha l'importantissimo compito di intercettare ogni richiesta che arriva sul backend per verificare validità del token e impostare come autenticata la richiesta arrivata, ricostruendo l'userdetails a partire dei claims contenuti nel token.

```
1.  public class JwtAuthenticationTokenFilter extends
    OncePerRequestFilter {
2.
3.
4.      @Autowired
5.      private UserDetailsService userDetailsService;
6.
7.      @Autowired
8.      private JwtTokenUtil jwtTokenUtil;
9.
10.     @Value("${jwt.header}")
11.     private String tokenHeader;
12.
13.
14.     @Override
15.     protected void doFilterInternal(HttpServletRequest request,
    HttpServletResponse response, FilterChain chain) throws
    ServletException, IOException {
```

```
16.         String authToken = request.getHeader(this.tokenHeader);
22.         }
23.
24.         if (userDetails != null &&
SecurityContextHolder.getContext().getAuthentication() == null)
25.
26.             // Ricostruisco l userdetails con i dati contenuti
token
27.
28.
29.             // controllo integrita' token
30.             if (jwtTokenUtil.validateToken(authToken, userDetai
{
31.                 UsernamePasswordAuthenticationToken authenticat
new UsernamePasswordAuthenticationToken(userDetails, null,
userDetails.getAuthorities());
32.                 authentication.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
33.
SecurityContextHolder.getContext().setAuthentication(authenticat
34.             }
35.         }
36.
37.         chain.doFilter(request, response);
38.     }
39. }
```

Come si evince dal codice sopra, se il token viene giudicato valido viene impostato nel contesto della request l'autenticazione con relativo principal e viene invocata la chain.doFilter la quale farà scattare i filtri di Spring Security.

Per validare il token e ricostruire l'userdetails, ho sviluppato una semplice classe wrapper della liberia jjwt : **JwtTokenUtil**. Tale classe costruisce il token andando a salvare nel payload del token

le informazioni utili contenute nella nostra implementazione di

Di Dario Frongillo
16 Dicembre 2017

2h).

```
1.  @Component
2.  public class JwtTokenUtil implements Serializable {
3.
4.      private static final long serialVersionUID =
-3301605591108950415L;
5.
6.      static final String CLAIM_KEY_USERNAME = "sub";
7.      static final String CLAIM_KEY_AUDIENCE = "audience";
8.      static final String CLAIM_KEY_CREATED = "iat";
9.      static final String CLAIM_KEY_AUTHORITIES = "roles";
10.     static final String CLAIM_KEY_IS_ENABLED = "isEnabled";
11.
12.     private static final String AUDIENCE_UNKNOWN = "unknown";
13.     private static final String AUDIENCE_WEB = "web";
14.     private static final String AUDIENCE_MOBILE = "mobile";
15.     private static final String AUDIENCE_TABLET = "tablet";
16.
17.     @Value("${jwt.secret}")
18.     private String secret;
19.
20.     @Autowired
21.     ObjectMapper objectMapper;
22.
23.     @Value("${jwt.expiration}")
24.     private Long expiration;
25.
26.     public String getUsernameFromToken(String token) {
27.         String username;
28.         try {
29.             final Claims claims = getClaimsFromToken(token);
30.             username = claims.getSubject();
31.         } catch (Exception e) {
32.             username = null;
33.         }
34.         return username;
35.     }
```

```
        try {
            final Claims claims = getClaimsFromToken(token);
            List<SimpleGrantedAuthority> authorities = null;
            if (claims.get(CLAIM_KEY_AUTHORITIES) != null) {
                authorities = ((List<String>)
claims.get(CLAIM_KEY_AUTHORITIES)).stream().map(role-> new
SimpleGrantedAuthority(role)).collect(Collectors.toList());
            }

            return new JwtUser(
                    claims.getSubject(),
                    "",
                    authorities,
                    (boolean) claims.get(CLAIM_KEY_IS_ENABLED)
            );
        } catch (Exception e) {
            return null;
        }

    }

    public Date getCreatedDateFromToken(String token) {
        Date created;
        try {
            final Claims claims = getClaimsFromToken(token);
            created = new Date((Long)
claims.get(CLAIM_KEY_CREATED));
        } catch (Exception e) {
            created = null;
        }
        return created;
    }

    public Date getExpirationDateFromToken(String token) {

        Date expiration;
        try {
            final Claims claims = getClaimsFromToken(token);
            expiration = claims.getExpiration();
        } catch (Exception e) {
            expiration = null;
```

Di Dario Frongillo
16 Dicembre 2017


```
79.             }
85.             try {
86.                 final Claims claims = getClaimsFromToken(token);
87.                 audience = (String)
    claims.get(CLAIM_KEY_AUDIENCE);
88.             } catch (Exception e) {
89.                 audience = null;
90.             }
91.             return audience;
92.         }
93.
94.         private Claims getClaimsFromToken(String token) {
95.             Claims claims;
96.             try {
97.                 claims = Jwts.parser()
98.                         .setSigningKey(secret)
99.                         .parseClaimsJws(token)
100.                        .getBody();
101.            } catch (Exception e) {
102.                claims = null;
103.            }
104.            return claims;
105.        }
106.
107.        private Date generateExpirationDate() {
108.            return new Date(System.currentTimeMillis() +
    expiration * 1000);
109.        }
110.
111.        private Boolean isTokenExpired(String token) {
112.            final Date expiration =
    getExpirationDateFromToken(token);
113.            return expiration.before(new Date());
114.        }
115.
116.        private String generateAudience(Device device) {
117.            String audience = AUDIENCE_UNKNOWN;
118.            if (device.isNormal()) {
119.                audience = AUDIENCE_WEB;
120.            } else if (device.isTablet()) {
121.                audience = AUDIENCE_TABLET;
```

```
122.             } else if (device.isMobile()) {
128.     private Boolean ignoreTokenExpiration(String token) {
129.         String audience = getAudienceFromToken(token);
130.         return (AUDIENCE_TABLET.equals(audience) ||
AUDIENCE_MOBILE.equals(audience));
131.     }
132.
133.     public String generateToken(UserDetails userDetails,
Device device) throws JsonProcessingException {
134.         Map<String, Object> claims = new HashMap<>();
135.         claims.put(CLAIM_KEY_USERNAME,
userDetails.getUsername());
136.         claims.put(CLAIM_KEY_AUDIENCE,
generateAudience(device));
137.         claims.put(CLAIM_KEY_CREATED, new Date());
138.         List<String> auth
=userDetails.getAuthorities().stream().map(role->
role.getAuthority()).collect(Collectors.toList());
139.         claims.put(CLAIM_KEY_AUTHORITIES, auth);
140.
claims.put(CLAIM_KEY_IS_ENABLED,userDetails.isEnabled());
141.
142.         return generateToken(claims);
143.     }
144.
145.     String generateToken(Map<String, Object> claims) {
146.         ObjectMapper mapper = new ObjectMapper();
147.
148.         return Jwts.builder()
149.                 .setClaims(claims)
150.                 .setExpiration(generateExpirationDate())
151.                 .signWith(SignatureAlgorithm.HS256, secret)
152.                 .compact();
153.     }
154.
155.     public Boolean canTokenBeRefreshed(String token) {
156.         final Date created = getCreatedDateFromToken(token);
157.         return  (!isTokenExpired(token) ||
ignoreTokenExpiration(token));
158.     }
159.
```

```
166.             } catch (Exception e) {
167.                 refreshedToken = null;
168.             }
169.             return refreshedToken;
170.         }
171.
172.         public Boolean validateToken(String token, UserDetails
    userDetails) {
173.             JwtUser user = (JwtUser) userDetails;
174.             final String username = getUsernameFromToken(token);
175.             return (
176.                     username.equals(user.getUsername())
177.                             && !isTokenExpired(token));
178.         }
179.  }
```

Abbiamo quindi adesso configurato Spring Security; ci rimane solo di andare a scrivere i controller degli endpoint di login e di refresh token. Quest'ultimo è un endpoint che, a partire da un token esistente, permette di restituire un nuovo token che avrà la data di scadenza calcolata a partire dall'istante in cui è stata generata la richiesta. Pertanto verrà utilizzato dai client per refreshare un token scaduto o prossimo alla scadenza.

```
1.  @RestController
2.  public class AuthenticationRestController {
3.
4.      @Value("${jwt.header}")
5.      private String tokenHeader;
6.
7.      @Autowired
8.      private AuthenticationManager authenticationManager;
9.
```

```
10.     @Autowired
16.     @RequestMapping(value = "public/login", method = RequestMet
17.     public ResponseEntity<?> createAuthenticationToken(@Request
JwtAuthenticationRequest authenticationRequest, Device device, H
response) throws AuthenticationException, JsonProcessingExceptic
18.
19.         // Effettuo l autenticazione
20.         final Authentication authentication = authenticationMan
21.                 new UsernamePasswordAuthenticationToken(
22.                         authenticationRequest.getUsername(),
23.                         authenticationRequest.getPassword()
24.                 )
25.         );
26.         SecurityContextHolder.getContext().setAuthentication(au
27.
28.         // Genero Token
29.         final UserDetails userDetails =
userDetailsService.loadUserByUsername(authenticationRequest.getU
30.         final String token = jwtTokenUtil.generateToken(userDet
31.         response.setHeader(tokenHeader,token);
32.         // Ritorno il token
33.         return ResponseEntity.ok(new
JwtAuthenticationResponse(userDetails.getUsername(),userDetails.
34.     }
35.
36.     @RequestMapping(value = "protected/refresh-token", method =
37.     public ResponseEntity<?> refreshAndGetAuthenticationToken(H
request, HttpServletResponse response) {
38.         String token = request.getHeader(tokenHeader);
39.         UserDetails userDetails =
40.
(UserDetails)SecurityContextHolder.getContext().getAuthenticatic
41.
42.         if (jwtTokenUtil.canTokenBeRefreshed(token)) {
43.             String refreshedToken = jwtTokenUtil.refreshToken(t
44.             response.setHeader(tokenHeader,refreshedToken);
45.
46.             return ResponseEntity.ok(new
JwtAuthenticationResponse(userDetails.getUsername(),userDetails.
47.         } else {
48.             return ResponseEntity.badRequest().body(null);
```

codice sorgente che trovate presso il repository github al seguente LINK.

CONCLUSIONI

In questo articolo abbiamo descritto in dettaglio JWT e come utilizzarlo per implementare un meccanismo di autenticazione Stateless. Nonostante gli svantaggi descritti reputo JWT un ottima soluzione su cui basare un sistema di autenticazione Stateless: semplice e altamente scalabile.

java jwt security spring

Design,progettazione e sviluppo di SPA e RIA Web application con framework Javascript. Attualmente mi occupo di sviluppo soluzioni software in ambito Banking e Finance: in particolare progettazione e sviluppo di applicativi web based per la realizzazione di sistemi di trading, interfacce con i mercati finanziari e di servizi real time.

ALTRO

24 Commenti ItalianCoders normativa sulla privacy Accedi
1
Consiglia Tweet Condividi Ordina dal più recente
Partecipa alla discussione...
ENTRA CON
O REGISTRATI SU DISQUS ?
Nome

• Rispondi • Condividi ›

Francesco Bianco • 9 mesi fa

Ciao Dario, sempre interessanti i tuoi articoli, complimenti, ti faccio una domanda:
Dove va conservato il TOKEN lato client per essere messo in sicurezza
e non essere intercettato da applicazioni malevole?

• Rispondi • Condividi ›

DieKaiser • un anno fa

Ciao Dario, intanto complimenti per questo articolo, molto chiaro ed esplicativo. Ti volevo chiedere una cosa che si allaccia molto all'ultimo post dei commenti: sto gestendo manualmente i token jwt piuttosto che quello generato da spring oauth2.0 per colpa della loadUserByUsername che avendo solo String username come parametro è troppo limitante per me. Mi sono quindi scontrato sul refresh_token, che come giustamente affermi non deve poter essere usato come un access_token, ma solo come strumento di refreshing dello stesso, e fai riferimento a un claim particolare per farlo riconoscere come tale. Mi sapresti dire quale? volendo potrei lasciare tutto con un unico token di risposta ma usare il refresh_token mi pare la soluzione più elegante.
Altra cosa: tramite Zuul (un proxy) ho gestito il refresh_token lato backend, secondo te è giusto o dovrei farlo fare al client tramite non so, un timer interno?

Grazie ancora per l'articolo, è stato per me fondamentale per ora.

• Rispondi • Condividi ›

francopentangeli • un anno fa

Ciao Dario,

complimenti per la guida. Davvero molto chiara!

mobile.
Al momento mi funziona tutto tranne che la verifica sull'utente loggato per le api (JwtAuthenticationEntryPoint).
Ecco le mie due configurazioni:

Web

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

@Autowired
```

vedi altro

1 • Rispondi • Condividi ›

Ewan Payne • 2 anni fa

Ciao Dario, complimenti per la guida. Mi sta aiutando molto in un mio progetto personale.
Tuttavia ho notato che hai associato all'oggetto user una mappa di Authorities. Si tratta di una tua scelta implementativa mettere un rapporto molti a molti tra utenti e ruoli?
Non prenderla come una critica ma non capisco perchè un utente debba avere + ruoli. Potresti spiegarmelo per favore ?

• Rispondi • Condividi ›

Dario Frongillo admin ➚ Ewan Payne • 2 anni fa

Ciao Ewan grazie dei complimenti. In sw medio-complessi un utente ha assolutamente piu ruoli. I ruoli vanno intesi come GRANT. Un utente puo avere associati piu ruoli : ad esempio in un ecommerce magari un utente puo avere ROLE_BUY e ROLE_SELL o solo uno dei due

• Rispondi • Condividi ›

per i token, in modo da realizzare una specie di sistema di logout?

• Rispondi • Condividi ›

Dario Frongillo admin ➚ Stefano De Angelis • 2 anni fa

ciao stefano grazie mille ":) jwt sviluppato in maniera rigorosa dovrebbe essere completamente stateless quindi le blacklist romperebbero questo vincolo. Puoi fare un compromesso: gestire una blacklist verificata dal filtro di spring (sia in fase di login che ad ogni richiesta). Una sorta di ban feature. Ovvero tieni traccia in un datastore (db o cache vedi tu) i token a cui spring deve rigettare le richieste. Normalmente si fa per problemi di sicurezza (bad user, perdita di un dispositivo ecc) ma assolutamente non per il logout. su jwt non esiste il concetto di logout. quando il token scade finisce la sessione utente. se hai bisogno di tenere traccia delle sessioni e di avere un vero logout lato server jwt non e' l approccio da usare

• Rispondi • Condividi ›

Stefano De Angelis ➚ Dario Frongillo
• 2 anni fa • edited

Ciao Dario, grazie per la risposta. Cosa consiglieresti tu allora per gestione login utenti per mobile application (e web app associata)? Il backend dovrebbe offrire API REST per servizi di autenticazione e accesso a risorse.

1 • Rispondi • Condividi ›

Salvatore Soldatini • 2 anni fa • edited

Dario, complimenti per il post e per il codice postato su Git!
Ho provato ad eseguire un nuovo metodo per aggiungere un nuovo cliente

UserDetails userDetails =
(UserDetails)SecurityContextHolder.getContext().getAuthentication().
if (jwtTokenUtil.validateToken(token, userDetails)) {
...
}

Il problema che ho riscontrato è il seguente:

-eseguo una POST: public/login che mi restituisce un token

vedi altro

• Rispondi • Condividi ›

xan • 2 anni fa

il sagro gral:
".sessionManagement().sessionCreationPolicy(SessionCreationPolic

2 ore per cercare online come rendere spring boot security stateless e alla fine sono arrivato qui

è incredibile come una cosa che dovrebbe essere il default per un backend restful sia cosi nascosta

grazie per il post

2 • Rispondi • Condividi ›

Giacomo Federici • 2 anni fa • edited

Ciao Dario, non so se leggerai mai questa mia richiesta, ma il mio problema è il seguente: Se invece di effettuare il login tramite servizio REST, lo effetuo direttamente sul client, come posso allo stesso tempo proteggere le risorse REST sul server con JWT? Non so, forse una volta loggato sul client e popolato l'oggetto UserDetails, dovrei passarlo in qualche modo al server che crea il

token? Eventualmente come si applicherebbe questa soluzione

ciao giacomo, ce" un problema di fondo nello scenario che mi stai descrivendo (se non ho capito male): l autenticazione la effettua sempre il server e non il client! Anche se non trovo il senso puoi fare il tuo login lato client e poi autenticarti sulla rest in modo che il be ti fornisca un token che puoi usare per le successive richieste autenticate

Rispondi • Condividi ›

italianCoders admin ➚ Giacomo Federici • 2 anni fa

“Approve”

2018-07-07 11:16 GMT+02:00 dario frongillo <italiancoders@gmail.com>:

Rispondi • Condividi ›

Federico Barone • 2 anni fa

Ciao, scusa se ti disturbo, potresti spiegarmi come effettuare una richiesta di login mediante Postman?

Rispondi • Condividi ›

Alessandro Imbrenda • 2 anni fa

Ciao complimenti per il tutorial! Ho cercato molto sul web ma non riesco a spiegarmi perché bisogna fare questo: SecurityContextHolder.getContext().setAuthentication(authentication)

Anche in altri tutorial viene utilizzato ma senza spiegazioni. A me sembra superfluo.

Rispondi • Condividi ›

Dario Frongillo admin ➚ Alessandro Imbrenda • 2 anni fa

Ciao Alessandro grazie mille :)
1) Ricordati che JWT e' stateless .. quindi ogni richiesta e'

come se fosse la prima :)

Di Dario Frongillo
16 Dicembre 2017

24 Comments

setta nel contesto di spring security che sei autenticato e l'autentication bean authentication. Tale bean sono le info di sessione.. in qualsiasi controller autenticato con SecurityContextHolder.getContext().getAuthentication(); potrai prenderti le info di sessione dell utente e verificare ad esempio se un utente puo fare una determinata operazione con i dati recuperati da SecurityContextHolder.getContext().getAuthentication() che non sono altro che i dati che hai deciso di mettere nel token

Ciao Dario :)

• Rispondi • Condividi ›

Alessandro Imbrenda ➚ Dario Frongillo
• 2 anni fa • edited

Ah ok capito. Quindi in realtá é utile se vuoi passare i dati salvati nel token tramite il context e se ne hai bisogno.

• Rispondi • Condividi ›

Dario Frongillo admin ➚ Alessandro Imbrenda
• 2 anni fa

si ma nel filtro devi farlo altrimenti non risulti autenticato per spring security.. ricordati che jwt e' stateless. con quel filtro e' come se facessi un login automatico ad ogni richiesta sulla base dei dati del token

• Rispondi • Condividi ›

Alessandro Imbrenda ➚ Dario Frongillo
• 2 anni fa

Ok perfetto.. Peró nel controller di login a cosa

serve?

Di Dario Frongillo
16 Dicembre 2017

24 Comments

un refresh automatico del token quando quest'ultimo è scaduto?

• Rispondi • Condividi ›

Dario Frongillo admin ➚ Diego Squillaci • 2 anni fa

grazie mille Diego.. in jwt essendo la durata parte del token

I NOSTRI PARTNER

NEEDFORNERD

Privacy Policy

AUTENTICAZIONE DI SERVIZI REST CON JWT [SPRING]

24 Comments / Di Dario Frongillo / In Back End, Java, Spring / 16 Dicembre 2017