Autenticazione di servizi REST con JWT [SPRING]

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

Un servizio REST dovrebbe essere per definizione stateless: ovvero le interazioni tra client e server devono essere senza stato, rispettando la comunicazione stateless nativa del protocollo HTTP. 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 cerchiamo innanzitutto di capire cos' è un JSON WEB TOKEN: abbreviato più comunemente con l acronimo di JWT. JWT è uno standard che viene usato per "regolare" le richieste tra due parti. JSON Web Token è un security token (una stringa) che agisce 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:

eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhZG1pbiIsImF1ZGllbmNlIjoid2ViIiwicm9sZXMiOlsiUk9MRV9BRE1JTiIsIlJPTEVfVVNFUiJdLCJpc0VuYWJsZWQiOnRydWUsImV4cCI6MTUxMjk3NTQ1MCwiaWF0IjoxNTEyOTY4MjUwMjMwfQ.J49qG_yi0yMZP4K2PWddTNR7QyUloAR7qj_3QtLL_6A

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 token.
Nell’immagine sopra è stato usato l’algoritmo “HMAC-SHA256”.

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)
- utente abilitato (claim isEnabled)
- info relative al momento in cui è stato creato il token (iat) e sulla data di scadenza (exp): momento in cui tale token non sarà piu’ ritenuto valido dal server.

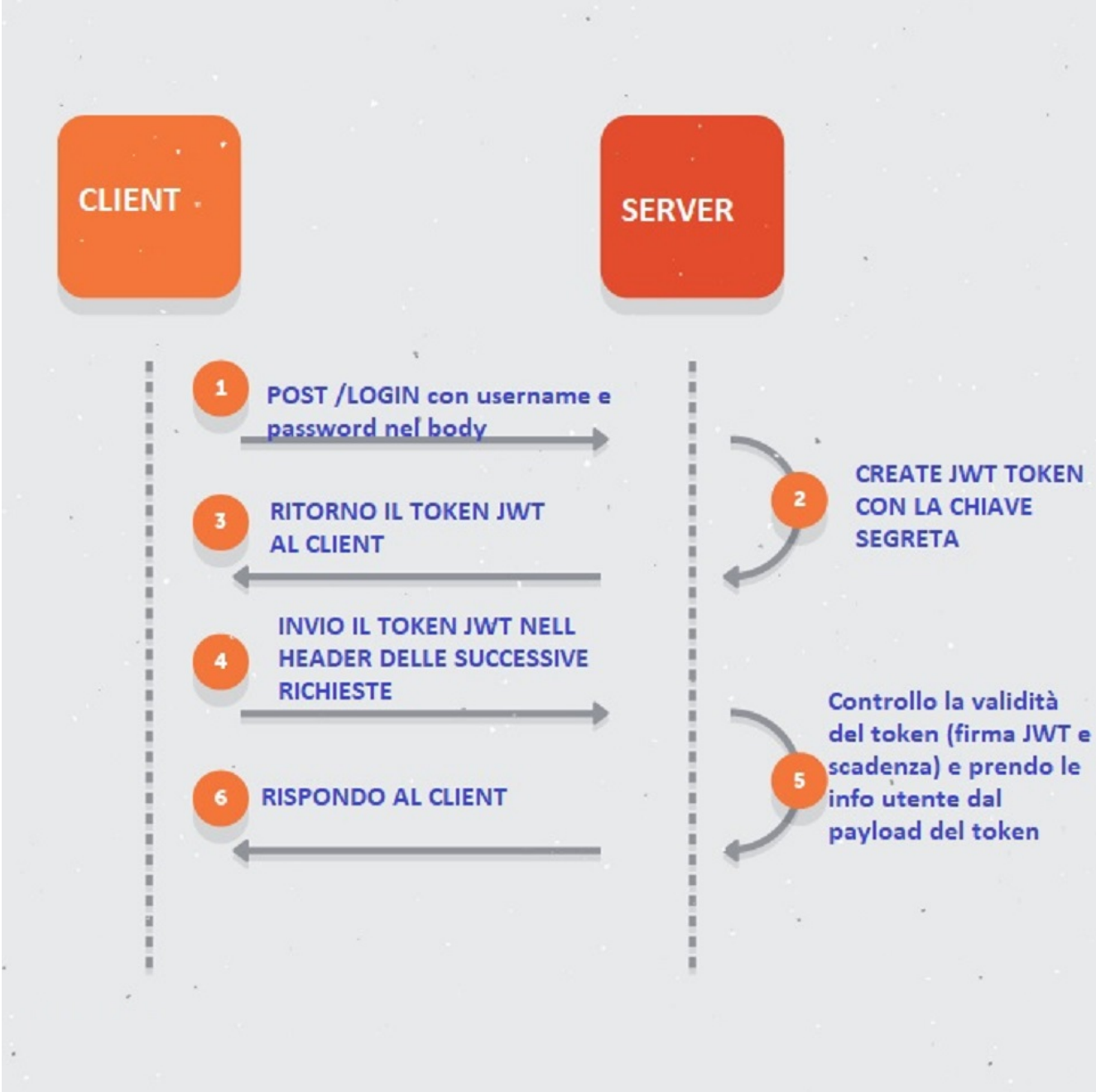
Signature: La terza parte di un token è la firma che non è altro che il risultato di una funzione hash 256 che prende in input la codifica base64 dell’header concatenandola con un punto alla codifica base64 del payload, il tutto codificato con la nostra “chiave segreta” che solo il server conosce.

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

Ora che abbiamo introdotto i JSON WEB TOKEN descriverò come possono essere utilizzati per implementare il meccanismo di autenticazione in un backend RESTful. Il meccanismo di 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 segreta e inserendo nel payload del token le informazioni utente e le data di creazione e scadenza del token; Tale token verrà firmato e autentificato con la sua chiave privata (**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 risponderà alla richiesta con un codice di errore HTTP 401 (utente non autorizzato).

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 perso il cellulare e che quindi dovete sloggare sul sistema tale utente per non permettere un uso improprio di tale utenza da altri. Purtroppo in JWT non esiste un vero e proprio logout; 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 richiesta.

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.

```
spring.datasource.username=root
spring.datasource.password=italiancoders
```

con l'username e password del vostro db.

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.
2. La chiave con cui il server firma il token (vedi

signature nel paragrafo precedente).
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
- validare il token prima di processare ogni richiesta e in caso di token valido, impostare nel contesto della request corrente l'autenticazione utente ricostruito dal token. Questa 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.

@Configuration

```
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter { @Autowired private JwtAuthenticationEntryPoint unauthorizedHandler; @Autowired private UserDetailsService userDetailsService; @Autowired public void configureAuthentication(AuthenticationManagerBuilder authenticationManagerBuilder) throws Exception { authenticationManagerBuilder .userDetailsService(this.userDetailsService) .passwordEncoder(passwordEncoder()); } @Bean public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); } @Bean public JwtAuthenticationTokenFilter authenticationTokenFilterBean() throws Exception { return new JwtAuthenticationTokenFilter(); } @Override protected void configure(HttpSecurity httpSecurity) throws Exception { httpSecurity .csrf().disable() .exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and() // non abbiamo bisogno di una sessione .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS).and() .cors().and() .authorizeRequests() .antMatchers( //HttpMethod.GET, "/", "/*.html", "/favicon.ico", "/**/*.html", "/**/*.css", "/**/*.js" ).permitAll() .antMatchers("/public/**").permitAll() .anyRequest().authenticated(); // Filtro Custom JWT httpSecurity.addFilterBefore(authenticationTokenFilterBean(), UsernamePasswordAuthenticationFilter.class); httpSecurity.headers().cacheControl(); }
}
```

Nello snippet di codice appena postato viene configurato Spring Security:

- 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 processate solo se l'utente è autenticato; questo vuol dire che tali richieste dovranno avere nell'header http un JSON WEB TOKEN valido.
  - 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.

public class JwtAuthenticationTokenFilter extends OncePerRequestFilter { @Autowired private UserDetailsService userDetailsService; @Autowired private JwtTokenUtil jwtTokenUtil; @Value("${jwt.header}") private String

```
tokenHeader;     @Override     protected     void
doFilterInternal(HttpServletRequest        request,
HttpServletResponse response, FilterChain chain) throws
ServletException, IOException { String authToken =
request.getHeader(this.tokenHeader); UserDetails userDetails =
null;   if(authToken   !=   null){   userDetails   =
jwtTokenUtil.getUserDetails(authToken); } if (userDetails !=
null && SecurityContextHolder.getContext().getAuthentication()
== null) { // Ricostruisco l userdetails con i dati contenuti
nel   token   //   controllo   integrita'   token   if
(jwtTokenUtil.validateToken(authToken,   userDetails))   {
UsernamePasswordAuthenticationToken authentication = new
UsernamePasswordAuthenticationToken(userDetails,   null,
userDetails.getAuthorities());  authentication.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));
SecurityContextHolder.getContext().setAuthentication(authentic
ation); } } chain.doFilter(request, response); }
}
```

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 userdetails e impostando una data di scadenza di un numero di secondi pari al valore della properties **jwt.expiration** (ho impostato sul mio ultimo commit la properties a 7200 sec ovvero 2h).

```
@Component
public class JwtTokenUtil implements Serializable { private
static final long serialVersionUID = -3301605591108950415L;
static final String CLAIM_KEY_USERNAME = "sub"; static final
String CLAIM_KEY_AUDIENCE = "audience"; static final String
CLAIM_KEY_CREATED   =   "iat";   static   final   String
CLAIM_KEY_AUTHORITIES = "roles"; static final String
```

```
CLAIM_KEY_IS_ENABLED = "isEnabled"; private static final String AUDIENCE_UNKNOWN = "unknown"; private static final String AUDIENCE_WEB = "web"; private static final String AUDIENCE_MOBILE = "mobile"; private static final String AUDIENCE_TABLET = "tablet"; @Value("${jwt.secret}") private String secret; @Autowired ObjectMapper objectMapper; @Value("${jwt.expiration}") private Long expiration; public String getUsernameFromToken(String token) { String username; try { final Claims claims = getClaimsFromToken(token); username = claims.getSubject(); } catch (Exception e) { username = null; } return username; } public JwtUser getUserDetails(String token) { if(token == null){ return null; } 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; } return expiration; } public String getAudienceFromToken(String token) { String audience; try { final Claims claims = getClaimsFromToken(token); audience = (String) claims.get(CLAIM_KEY_AUDIENCE); } catch (Exception e) { audience = null; } return audience; } private Claims getClaimsFromToken(String token) { Claims claims; try { claims = Jwts.parser() .setSigningKey(secret) .parseClaimsJws(token) .getBody(); } catch (Exception e) { claims = null; } return claims; } private Date generateExpirationDate() { return new Date(System.currentTimeMillis() + expiration * 1000); } private Boolean isTokenExpired(String token) { final Date expiration = getExpirationDateFromToken(token); return expiration.before(new Date()); } private String
```

```
generateAudience(Device device) { String audience = AUDIENCE_UNKNOWN; if (device.isNormal()) { audience = AUDIENCE_WEB; } else if (device.isTablet()) { audience = AUDIENCE_TABLET; } else if (device.isMobile()) { audience = AUDIENCE_MOBILE; } return audience; } private Boolean ignoreTokenExpiration(String token) { String audience = getAudienceFromToken(token); return (AUDIENCE_TABLET.equals(audience) || AUDIENCE_MOBILE.equals(audience)); } public String generateToken(UserDetails userDetails, Device device) throws JsonProcessingException { Map<String, Object> claims = new HashMap<>(); claims.put(CLAIM_KEY_USERNAME, userDetails.getUsername()); claims.put(CLAIM_KEY_AUDIENCE, generateAudience(device)); claims.put(CLAIM_KEY_CREATED, new Date()); List<String> auth =userDetails.getAuthorities().stream().map(role-> role.getAuthority()).collect(Collectors.toList()); claims.put(CLAIM_KEY_AUTHORITIES, auth); claims.put(CLAIM_KEY_IS_ENABLED,userDetails.isEnabled()); return generateToken(claims); } String generateToken(Map<String, Object> claims) { ObjectMapper mapper = new ObjectMapper(); return Jwts.builder() .setClaims(claims) .setExpiration(generateExpirationDate()) .signWith(SignatureAlgorithm.HS256, secret) .compact(); } public Boolean canTokenBeRefreshed(String token) { final Date created = getCreatedDateFromToken(token); return (!isTokenExpired(token) || ignoreTokenExpiration(token)); } public String refreshToken(String token) { String refreshedToken; try { final Claims claims = getClaimsFromToken(token); claims.put(CLAIM_KEY_CREATED, new Date()); refreshedToken = generateToken(claims); } catch (Exception e) { refreshedToken = null; } return refreshedToken; } public Boolean validateToken(String token, UserDetails userDetails) { JwtUser user = (JwtUser) userDetails; final String username = getUsernameFromToken(token); return ( username.equals(user.getUsername()) && !isTokenExpired(token)); }
}
```

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.

```
@RestController
public class AuthenticationRestController { @Value("${jwt.header}") private String tokenHeader; @Autowired private AuthenticationManager authenticationManager; @Autowired private JwtTokenUtil jwtTokenUtil; @Autowired private UserDetailsService userDetailsService; @RequestMapping(value = "public/login", method = RequestMethod.POST) public ResponseEntity<?> createAuthenticationToken(@RequestBody JwtAuthenticationRequest authenticationRequest, Device device, HttpServletResponse response) throws AuthenticationException, JsonProcessingException { // Effettuo l autenticazione final Authentication authentication = authenticationManager.authenticate( new UsernamePasswordAuthenticationToken( authenticationRequest.getUsername(), authenticationRequest.getPassword() ) ); SecurityContextHolder.getContext().setAuthentication(authentication); // Genero Token final UserDetails userDetails = userDetailsService.loadUserByUsername(authenticationRequest.getUsername()); final String token = jwtTokenUtil.generateToken(userDetails, device); response.setHeader(tokenHeader,token); // Ritorno il token return ResponseEntity.ok(new JwtAuthenticationResponse(userDetails.getUsername(),userDetails.getAuthorities())); } @RequestMapping(value = "protected/refresh-token", method = RequestMethod.GET) public ResponseEntity<?> refreshAndGetAuthenticationToken(HttpServletRequest request, HttpServletResponse response) { String token = request.getHeader(tokenHeader); UserDetails userDetails = (UserDetails)SecurityContextHolder.getContext().getAuthentication().getPrincipal(); if (jwtTokenUtil.canTokenBeRefreshed(token)) { String
```

```
refreshedToken = jwtTokenUtil.refreshToken(token); response.setHeader(tokenHeader,refreshedToken); return ResponseEntity.ok(new JwtAuthenticationResponse(userDetails.getUsername(),userDetails.getAuthorities())); } else { return ResponseEntity.badRequest().body(null); } } }
```

A questo punto vi invito a fare qualche prova andando a clonare il 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.

https://italiancoders.it/autenticazione-di-servizi-rest-con-jwt-spring/