

JWT & Spring

ROBERTO AMATO



JWT

1.0 Introduzione

Json Web Token (JWT) è uno standard, abbastanza recente, uscito all'inizio del 2015 che consente al server di scrivere in un messaggio (payload) chi è l'utente loggato (e altre info di contorno), incapsulare questo messaggio all'interno di un token e darlo al client che lo utilizzerà da ora in poi per le successive chiamate al server, informandolo così di chi sta effettuando la chiamata. Questo consentirà al server di avere già le informazioni di autenticazione direttamente nel token stesso, evitando così dover passare dal database o di usare le sessioni per memorizzare le informazioni sull'autenticazione.

JWT sta per Json-Web-Token ed è:

- ➔ uno standard (RFC-7519 <https://tools.ietf.org/html/rfc7519>) aperto che definisce in modo compatto per la trasmissione sicura di informazioni tra due parti

Queste informazioni che viaggiano tra una parte all'altra, sono considerate sicure perché sono firmate digitalmente, ovvero cifrate.

JWT viene firmato digitalmente (cifrato) utilizzando :

- 1 - una chiave segreta (generato con un algoritmo di Hashing)
oppure
- 2- una coppia di chiavi pubbliche/private usando RSA o ECDSA
 - [https://en.wikipedia.org/wiki/RSA_\(cryptosystem\)](https://en.wikipedia.org/wiki/RSA_(cryptosystem)) ,
 - https://it.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm



1.1 Scenari quando usare JWT

Due scenari tipici dove usar e JWT sono i seguenti:

1- **AUTORIZZAZIONE** di un utente (client) ad una particolare risorsa

Questo è lo scenario più comune per l'utilizzo di JWT.

Vediamo brevemente come funziona:

- Una volta che l'utente (client) ha effettuato l'accesso (login), se questo va a buon fine, allora il server genera un token che viene inviato nella risposta all'utente (client).
- L'utente (Client) ad ogni successiva richiesta includerà nell' header della richiesta tale token in modo che il server possa verificare l'autorizzazione a quella particolare risorsa richiesta.

NOTA: Vedi anche (SSO – Single Sign On)

2- **SCAMBIO DI INFORMAZIONI**

JWT oltre ad essere usati per l'autorizzazione di un utente (client) a determinate risorse, sono anche un buon modo per trasmettere in modo sicuro delle informazioni tra due parti.

Questo è possibile perché i JWT possono essere firmati (cifrati) e quindi assicurarsi che l'informazione provenga dal giusto mittente e non da malintenzionati.

Inoltre, poiché la firma viene calcolata usando:

- 1- l'intestazione (Header)
- 2- Payload (Body)

È anche possibile verificare (assicurarsi) che il contenuto del messaggio non sia stato alterato (manomesso).



1.2 Come è strutturato un JWT

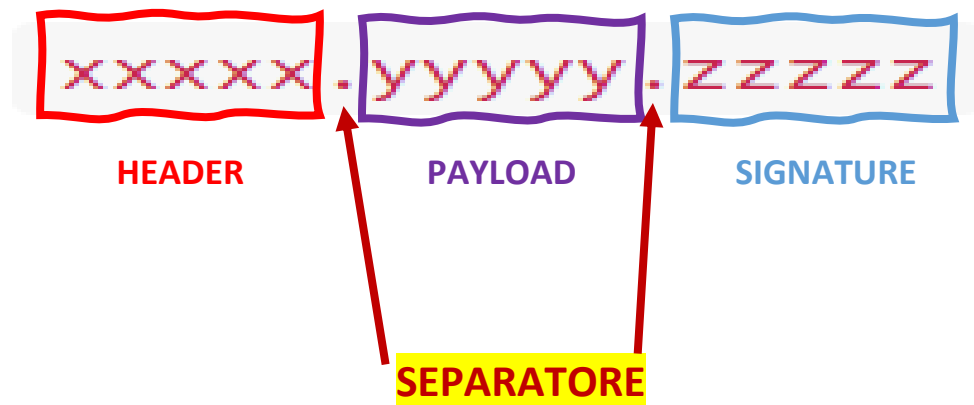
JWT ha un formato molto compatto. Questa caratteristica lo rende molto leggero nel trasferimento (meno byte da trasferire = meno occupazione di banda).

Json-Web-token è in sostanza composto da 3 parti :

- 1- Prima parte è chiamata HEADER
- 2- Secondo parte è chiamata PAYLOAD
- 3- Terza parte è chiamata SIGNATURE

NOTA BENE: Una parte viene separata dall'altra usando come delimitatore un punto '.'

Vediamo adesso un esempio astratto di JWT:



Ecco invece un esempio reale di JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwiaWF0IjoxNDc3MTM0NzQ4fQ.Ky3iKYcguIstYpDbMbIbDR5s7e_UF0PI1gal6VX5eyI
```



Adesso che abbiamo visto come è strutturato un JWT analizziamo nel dettaglio ognuna delle 3 parti di cui esso è formato.

- **HEADER**

Come abbiamo detto prima, la prima parte del JWT viene chiamata Header. Questa è tipicamente suddivisa in 2 parti:

1- **Type** (tipo di token come ad esempi JWT)

2- **SIGNATURE ALGORITHM** (tipo di algoritmo di cifratura come ad esempio HMAC SHA256, RSA, ECDSA, ecc...)

Esempio:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

L'immagine sopra illustra il JSON che rappresenta la prima parte (HEADER) del JWT prima di essere cifrato e diventare così →

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.)

- **PAYLOAD**

Come abbiamo detto prima, la seconda parte del JWT viene chiamata Payload.

Questa parte contiene le informazioni vere e proprie che vengono trasmesse tra le parti. Queste informazioni sono :



- **Claims** sono dichiarazioni su un entità (tipicamente l'utente)
- **Dati aggiionali**

Ci sono 3 tipi di claims:

- **Registered claims**
 - **Public claims**
 - **Private claims**
- **Registered-claims** sono un insieme predefinito di claims che non sono obbligatori ma solo consigliati.

Alcuni di questi sono :

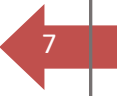
- **iss** (emittente)
- **exp** (tempo di scadenza)
- **sub** (soggetto)
- **aud** (pubblico)
- altri...

NOTA BENE: I nomi dei claims sono lunghi solo 3 caratteri appunto perché JWT deve essere compatto.

- **Public-claims** possono essere definiti a piacere da chi li usa, ma per evitare collisioni questi dovrebbero essere definiti in [IANA JSON Web Token Registry](https://www.iana.org/assignments/jwt/jwt.xhtml) (<https://www.iana.org/assignments/jwt/jwt.xhtml>).



- **Private-claims** sono claims custom (personali) creati per condividere delle informazioni tra le parti che concordano sull'utilizzo di questi claims e che non vogliono usare claim pubblici e privati



Esempio:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

L'immagine sopra illustra il JSON che rappresenta la seconda parte (PAYLOAD) del JWT contenente 3 claims (dichiarazioni): il primo (registered-claims e gli altri private-claims), prima di essere cifrato e quindi diventare così

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
→ qRG9lIiwiaXNTb2NpYWwiOnRydWV9.

NOTA BENE: Anche se i token sono firmati, queste informazioni, seppure protette (in quanto cifrate) da manomissioni, sono leggibili da chiunque. Quindi non bisogna inserire informazioni segrete nel payload o negli elementi di intestazione di un JWT a meno che queste informazioni non siano già cifrate di suo.

- **SIGNATURE**

Come abbiamo detto prima, la terza parte del JWT viene chiamata SIGNATURE (firma). Questa parte è quella che si occupa di rendere il JWT sicuro, ovvero di assicurarci l'idoneità del mittente.

Per creare questa parte del JWT abbiamo bisogno di 3 cose:



- L'intestazione (HEADER)
- Payload
- Chiave segreta (importante...parola segreta che solo il server che genera il JWT conosce)

E infine firmarlo utilizzando l'algoritmo di codifica specificata nell'header

Ad esempio se desideriamo usare l'algoritmo HMAC SHA256, allora la firma verrà creata nel seguente modo:

Cifratura usando algoritmo
HMAC SHA256

funzione che codifica i caratteri strani
Interpretati nell'URL

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  secret)
```

La firma viene usata per verificare (assicurarsi) che il messaggio non sia stato modificato durante il tragitto dal mittente al destinatario.

Nel caso di token firmati con una chiave privata, si può anche verificare che il mittente del JWT sia quello che dice di essere.

- **METTIAMO TUTTE LE 3 PARTI INSIEME**

L'output del JWT è costituito da 3 parti (3 pezzi di stringhe) separati da un punto '.'.



NOTA BENE: ciascuna delle prime 2 stringhe che costituiscono il JWT sono codificate in Base64 e anche codificate come URL (funzione `base64UrlEncode`) in modo da poter codificare tutti quei caratteri che si comportano in modo strano dentro l'URL come ad esempio gli spazi, la chiocciola, ecc...

Di seguito viene mostrato un JWT che ha codificato l'intestazione (HEADER) e il PAYLOAD visti in precedenza e firmato (cifrato) con una chiave segreta.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4  
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.  
4pcPyMD09o1PSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

Se vuoi esercitarti con JWT e metter in pratica questi concetti, possiamo usare il debugger [jwt.io](https://jwt.io/#debugger-io) (<https://jwt.io/#debugger-io>) per generare, decodificare e verificare JWT.



1.3 Come funziona in pratica JWT

Vediamo prima come funziona JWT nel suo uso più comune l'autenticazione.

Nell'autenticazione, quando un utente accede ad una applicazione web ad esempio, utilizzando le sue credenziali, Il server genera un JWT che restituirà all'utente nella risposta.

Successivamente, ogni volta che l'utente desidera accedere ad una risorsa protetta, deve inviare nella richiesta per quella risorsa anche il JWT, in genere viene inserito nell'header in un nuovo attributo chiamato: "Authorization" usando lo schema "Bearer".

Il contenuto dell'Header della richiesta per quella determinata risorsa protetta dovrebbe quindi essere come il seguente:

REQUEST-HEADER

GET /api/items/

...

```
Authorization: Bearer <token>
```

...

...

BODY (presente se metodo è POST)

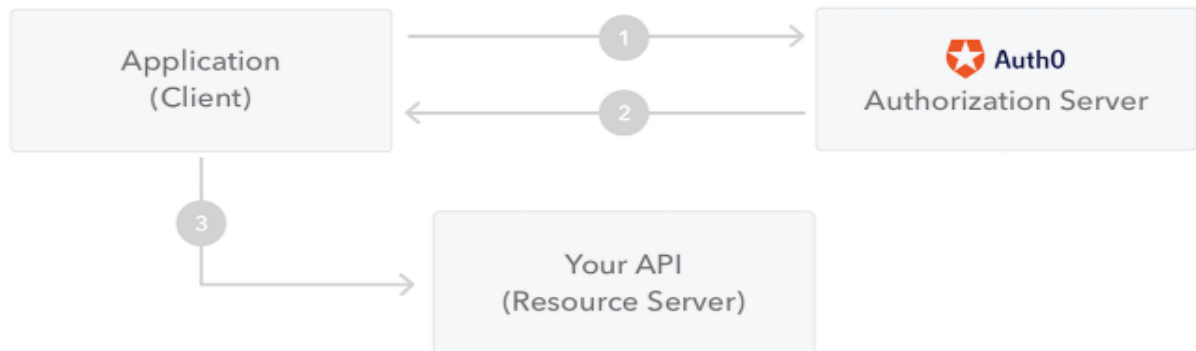
...

...

Una volta arrivata la richiesta al server per quella specifica risorsa, il server verifica se quella risorsa che l'utente vorrebbe è protetta e se sì, controlla il JWT arrivato con la richiesta in modo da poter verificare se l'utente è autorizzato per accedere a tale risorsa.



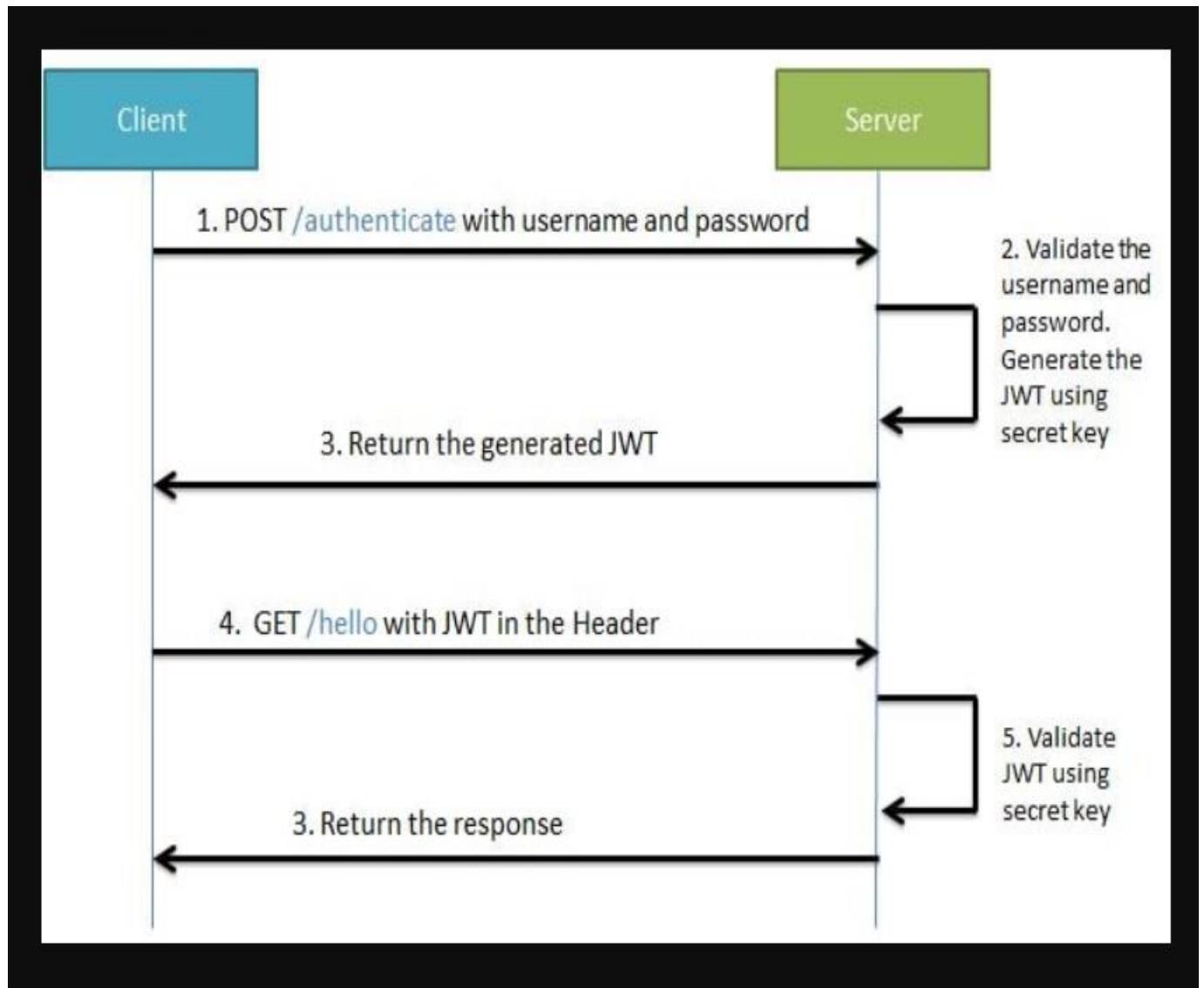
Il seguente diagramma mostra come ottenere e usare un JWT per accedere alle API o alle risorse:



- 1- L'applicazione client richiede l'autorizzazione al server per essere autorizzato.
- 2- Se il client viene autorizzato, allora il server restituisce il JWT, altrimenti risponde che il client non è autorizzato.
- 3- L'applicazione client utilizza il token per accedere ad una API o risorsa che è protetta dal server



Diagramma di sequenza delle operazioni coinvolte:



1.4 Vantaggi di JWT



1.5 Perché usare JWT?

Facciamo un passo Indietro

Quando facciamo una login, siamo abituati di solito a

- 1- Creare una sessione tra il server e il client e
- 2- Memorizzare user_id all'interno della sessione, al fine di recuperare le informazioni di autenticazione nelle successive chiamate.

Qui già sorge un piccolo problema:

➔ cosa succede se utilizziamo più server per bilanciare il carico del nostro sito?

Se abbiamo creato la sessione quando eravamo sul server A e successivamente il bilanciamento di risorse ci porta sul server B la sessione sul server A che aveva i nostri dati di autenticazione la perdiamo (in realtà per avere sessioni condivise su più server le sessioni non si memorizzano più localmente ma si tendono a memorizzarle all'interno di uno spazio condiviso...su un database no-sql ad esempio).

Se invece stiamo progettando un sistema api vorremmo utilizzare un token per far riferimento a una particolare sessione di autenticazione. Quindi quando viene effettuata una chiamata all'api login, effettuo lato server un'autenticazione, creo il token, lo memorizzo nel database associandolo ad utente (magari con un timestamp di scadenza) e lo restituisco al client. Il client passerà questo token nelle successive chiamate, il server accede al db e verifica a quale utente è associato quel token. OK già va un po' meglio...in realtà è peggio...stiamo creando un collo di bottiglia tra i vari server e il DB poiché ad ogni richiesta api partirà una query al database per capire a quale utente è associato quel token, oltre che a creare un problema di sicurezza (il database potrebbe essere attaccato da qualche utente malintenzionato e recuperare tutti i nostri token di sessione). Ma ecco così che entra in gioco **Json Web Token**.



→ **L'idea che c'è alla base è che dopo l'autenticazione, il server prepara un token all'interno del quale racchiude un payload in cui viene dichiarato in modo esplicito chi è l'utente loggatosi.**

Dentro il token, oltre il payload viene inserita la firma dal server (che è costituita dal payload stesso criptato con la sua chiave segreta in codifica hash 256). Il client riceve il token e se vuole sarà libero di leggere il payload contenuto ma non potrà modificarlo poichè se lo facesse il token sarà invalidato dal server.

Il client dovrà comunicare al server il token ricevuto per tutte le successive chiamate in cui è richiesta l'autenticazione. Il server riceverà il token ed estrapolerà il payload ma prima si assicurerà che il token sia stato firmato e autenticato con la sua chiave privata.

Poichè il token contiene il payload con tutte le informazioni necessarie all'autenticazione (es. iduser), il server potrà evitare di passare ogni volta dal database per verificare a quale utente corrisponde quel token (ottimo per la scalabilità).



2.0 Implementazione sicurezza in Spring Tramite JWT

Vediamo adesso come utilizzare Spring security e lo standard JWT per la sicurezza delle API REST all'interno di un'applicazione Spring Boot.

Come abbiamo detto più volte prima un utente che effettua il login attraverso un client un riceve in cambio (se l'autenticazione va a buon fine) dal server un token, ovvero dalla nostra applicazione Spring Boot.

Vogliamo a questo punto

CREAZIONE API E GENERAZIONE TOKEN JWT

- ➔ Creare un'API di login per autenticare l'utente e come generare un token JWT codificato con chiave segreta e che contiene alcune informazioni/dichiarazioni (claims)
- ➔ Ad ogni richiesta di un client che giungerà alla nostra applicazione (server) in Spring Boot dovrà contenere nell'header il token di autenticazione.

DECODIFICA TOKEN JWT

- ➔ Vedremo dopo come decodificare il token ricevuto per autorizzare la richiesta del client.



2.1 Utilizzare Spring Security con JWT per la sicurezza delle API

Per usare Spring security all'interno del nostro progetto Spring Boot occorre eseguire i seguenti passi:

1. Aggiungere le dipendenze Maven per:

- Spring-Security
- Java-Jwt
- Joda-time

Il primo passo consiste nell'aggiungere *Spring Security* al progetto utilizzando la dipendenza *Maven* fornita dallo starter Spring.

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

Oltre a quella appena vista, aggiungiamo le dipendenze *java-jwt* e *joda-time*, ci serviranno nell'implementazione.

```
<dependency>
  <groupId>com.auth0</groupId>
  <artifactId>java-jwt</artifactId>
  <version>3.8.3</version>
</dependency>

<dependency>
  <groupId>joda-time</groupId>
  <artifactId>joda-time</artifactId>
</dependency>
```



Se conosci altre librerie che portano allo stesso risultato puoi sostituirle con quelle appena viste. In questo



2. Configurare Spring-Security

In questa fase andremo a creare 3 classi JAVA che ci serviranno per implementare lo strato di sicurezza per le nostre API:

- **Una classe custom “JwtProvider”** che marcheremo come componente di Spring e che servirà per la gestione del token JWT
- **Una classe custom “AuthorizationFilter”** che estende la classe di Spring-security BasicAuthenticationFilter.

Questa classe rappresenta il filtro che viene eseguito ad ogni chiamata http in ingresso.

Questo filtro “AuthorizationFilter” intercetterà ogni richiesta HTTP che giungerà al nostro server-web (Tomcat), verificando che la richiesta http pervenuta contenga il token e in caso positivo, lo decodifica estraendo i dati dell’utente che vuole accedere alla risorsa in modo da poter verificare se quell’utente è autorizzato per la risorsa che sta richiedendo

- **Una classe di configurazione custom “SecurityConfig”** che estende la classe di Spring-Security WebSecurityConfigurerAdapter.

3. Definire le API REST per l’autenticazione



2.3 Configurare Spring-Security

In questa fase andremo a creare 3 classi JAVA che ci serviranno per implementare lo strato di sicurezza per le nostre API:

- **Una classe custom “JwtProvider”** che marcheremo come componente di Spring e che servirà per la gestione del token JWT
- **Una classe custom “AuthorizationFilter”** che estende la classe di Spring-security BasicAuthenticationFilter.

Questa classe rappresenta il filtro che viene eseguito ad ogni chiamata http in ingresso.

- **Una classe di configurazione custom “SecurityConfig”** che estende la classe di Spring-Security WebSecurityConfigurerAdapter.

2.3.1 Aggiungere le proprietà

Aggiungiamo nel file “application.properties” sotto la cartella src/main/resources alcuni attributi che andremo ad utilizzare all’interno delle classi e che rappresentano rispettivamente:

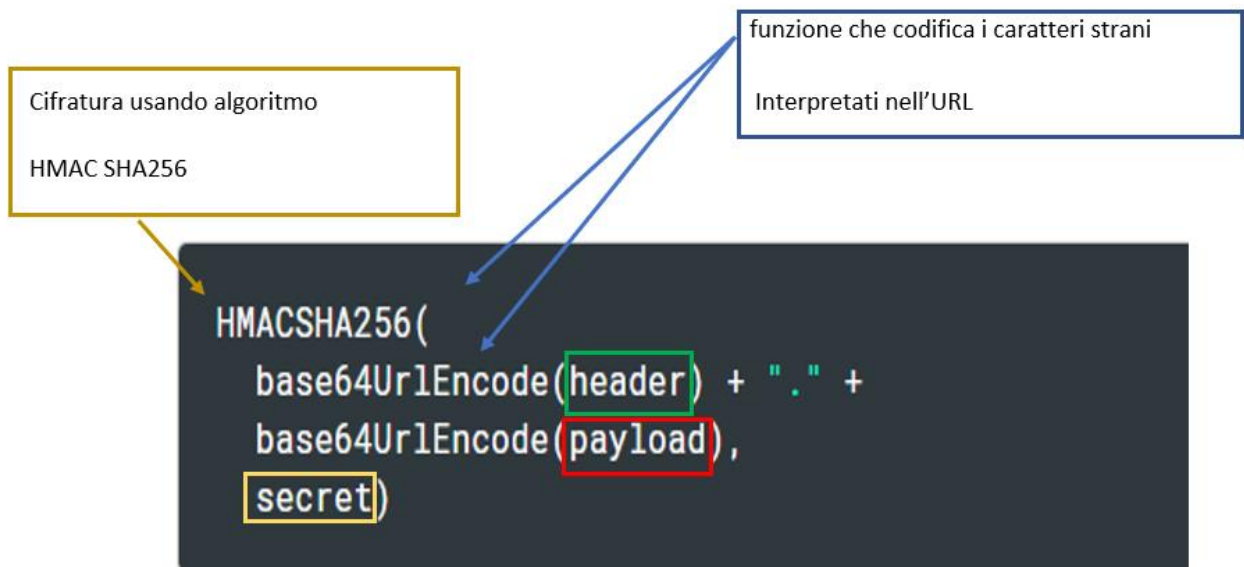
- La chiave segreta per la cifratura lato server (firma)
- Il prefisso per la stringa di autorizzazione
- Il nome dell’attributo/parametro presente nell’Header della richiesta HTTP

```
security.secret=chiavesupersegretissima  
security.prefix='Bearer '  
security.param=Authorization
```



NOTA_1:

Il valore di security.secret possiamo cambiarlo a piacimento, in quanto rappresenta la chiave segreta lato server con il quale verrà codificato (cifrato) il JWT costituendo l'ultima parte del JWT (quella denominato SIGNATURE che riassumiamo qui sotto nell'immagine)



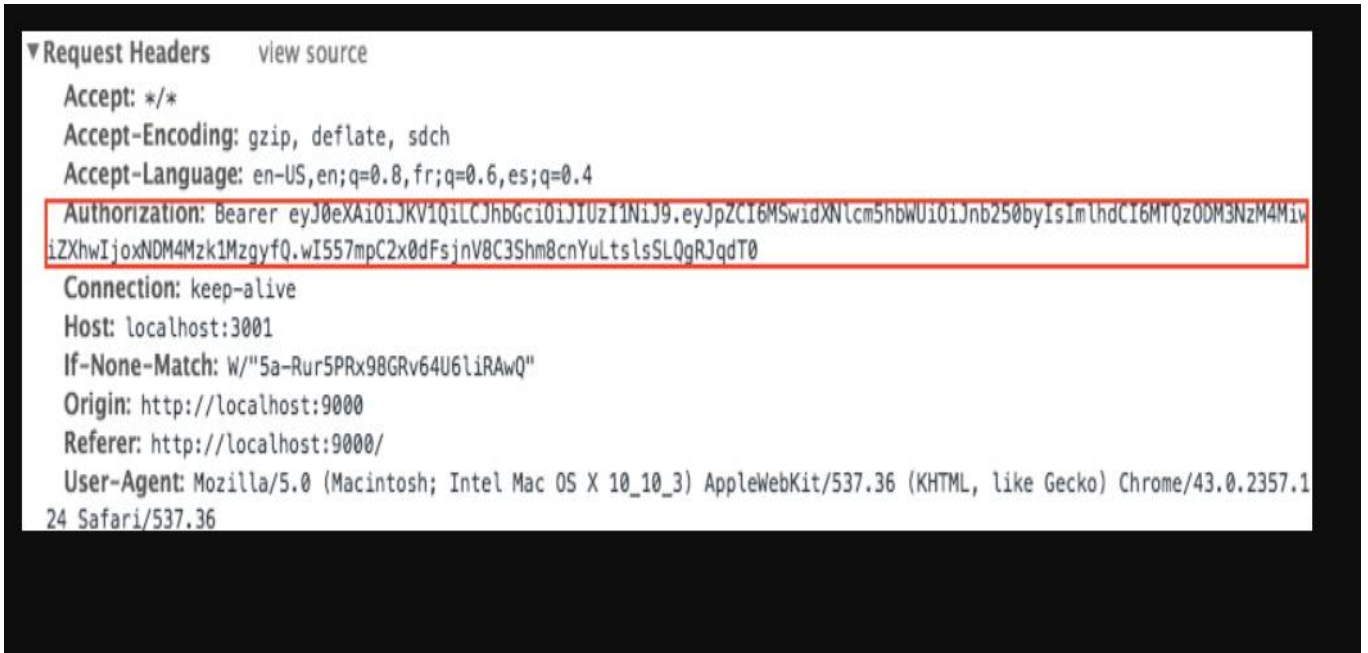
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09o1PSyXnrXCjTwXyr4Bsezdi1AVTmud2fU4



NOTA_2:

Invece il valore di security.prefix e security.param sono fissati.

Il valore security.prefix sarà l'attributo (chiave) nell'Header di richiesta che conterrà il JWT come nell'esempio illustrato sotto:



```
▼ Request Headers  view source
Accept: */*
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8,fr;q=0.6,es;q=0.4
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpZCI6MSwidXNlcm5hbWUiOiJnb250byIsImhhdCI6MTQzODM3NzY4MiwiaXZhwIjozNDM4Mzk1MzgyfQ.wI557mpC2x0dFsjnV8C3Shm8cnYuLtslsSLQgRJqdT0
Connection: keep-alive
Host: localhost:3001
If-None-Match: W/"5a-Rur5PRx98GRv64U6liRAwQ"
Origin: http://localhost:9000
Referer: http://localhost:9000/
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_3) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/43.0.2357.124 Safari/537.36
```

2.3.2 La classe JwtProvider.java (classe di Utility)

Vediamo ora nel dettaglio quanto detto prima, partendo dalla creazione della classe JwtProvider.java che si occupa della gestione del JWT token.

@Component ← Fa in modo che questa classe diventi un bean di Spring, istanziando un oggetto di tipo JwtProvider e salvandolo nell'IOC Container in modo che chi ne ha bisogno Spring lo possa iniettarlo

@Slf4j

```
public class JwtProvider {
```

@Value("\${security.secret}") ← recupera dal file di properties il valore con chiave security.secret

private String secret;

@Value("\${security.prefix}") ← recupera dal file di properties il valore con chiave security.prefix

private String prefix;

metodo createJwt:

- input : void
- output: stringa che rappresenta il JWT

```
public String createJwt() {
    return JWT.create()
        .withSubject("subject")
        .withIssuer("issuer")
        .withIssuedAt(DateTime.now().toDate())
        .withClaim("someClaim", "someClaimDesc")
        .withExpiresAt(DateTime.now().plusMonths(1).toDate())
        .sign(Algorithm.HMAC256(secret));
}
```

← Classe della libreria auth0 di Spring che abbiamo messo tra le dipendenze del pom.xml.

← NOTA: Questa classe JWT implementa il pattern Builder.

metodo decodeJwt:

- input: String che rappresenta il JWT
- output: Oggetto di tipo DecodedJwt che rappresenta la decodifica del JWT, in modo da recuperare le info contenute in esso

```
public DecodedJWT decodeJwt(String jwt) {
    try {
        jwt = jwt.replace(prefix, "").trim();
        return JWT.require(Algorithm.HMAC256(secret)).build().verify(jwt);
    } catch (Exception e) {
        log.error("Invalid JWT", e);
    }
    return null;
}
```



La classe sopra implementa 2 metodi:

1. **String createJwt()**

Serve, come suggerisce il nome, per generare un nuovo token JWT

2. **DecodeJwt decodeJwt(String jwt)**

Serve per verificare la validità del JWT decodificandolo

NOTA BENE: Importante annotare la classe con Component di Spring, per essere iniettata in altri componenti.

Entrambi i metodi (creatJwt() e decodeJwt(String jwt)) usano la classe JWT della libreria auth0 inserita prima nel pom.xml.

Mentre viene fatto uso dell'oggetto DateTime fornito dalla libreria joda-time (anch'essa importata prima come dipendenza nel pom.xml) per la gestione delle date come la data di scadenza del token JWT.

Il token contiene nella sua rappresentazione alcune informazioni/dichiarazioni. L'oggetto JWT contiene alcuni metodi per l'inserimento di queste informazioni all'interno del token. Si distinguono dal prefisso "with" (withSubject, withClaim, ...), in particolare, si può aggiungere qualsiasi informazione/dichiarazione personalizzata (custom) all'interno di un Claim tramite il metodo withClaim/s dell'oggetto JWT della libreria auth0 aggiunta prima.

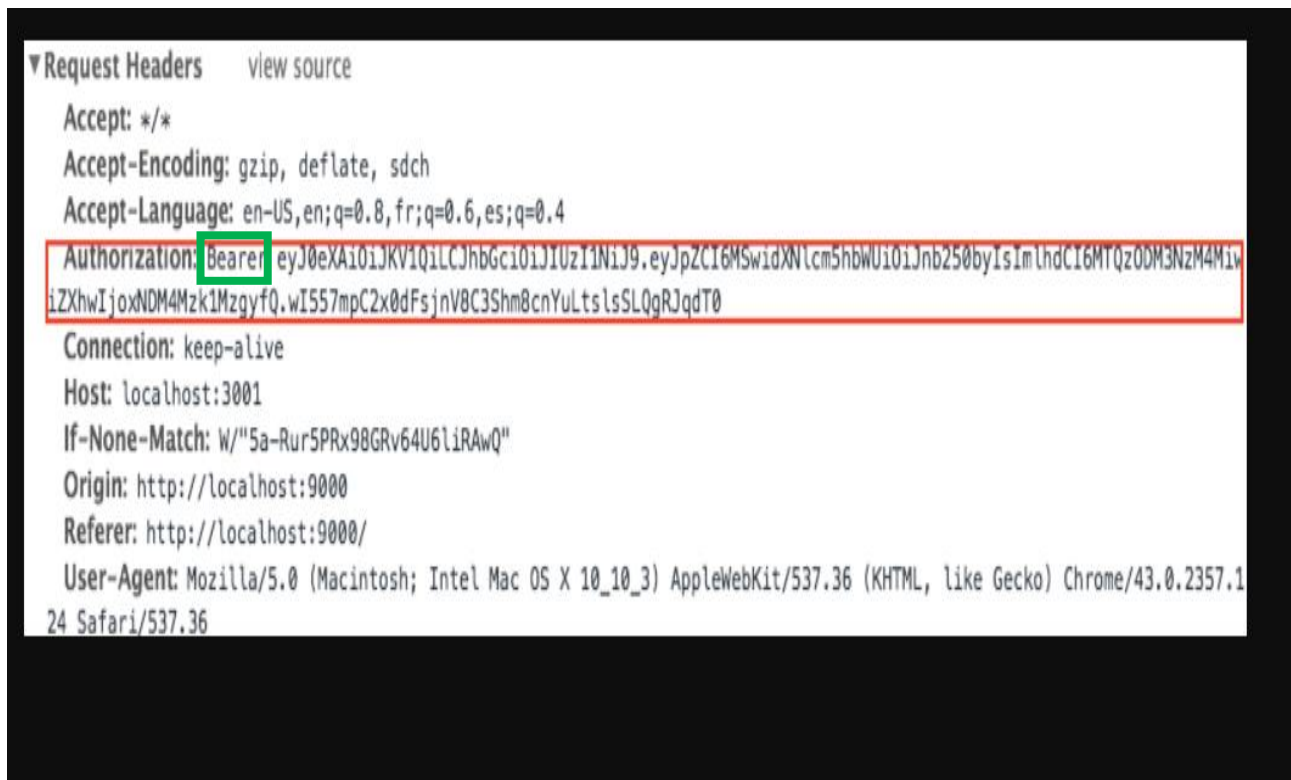
Infine il JWT viene firmato con l'algoritmo HMACSHA256 e con chiave segreta letta dal file di properties, usati anche nel metodo di decodifica.



NOTA BENE: Il parametro *prefix*, letto anch'esso dalle proprietà del file di properties, rappresenta il prefisso della stringa contenente il token ed è una caratteristica del meccanismo *Oauth2*.

Ad esempio, nell'header della richiesta possiamo trovare un'autenticazione di tipo

→ Bearer: Bearer eyJhbGciOiJIUzI1Ni....resto del jwt... .



2.3.3 La classe AuthorizationFilter.java

La seconda classe che andremo a creare rappresenta il filtro di autorizzazione che viene attivato in modo automatico (e che deve essere inserito nella configurazione SecurityConfig che vedremo successivamente a questo) quando viene ricevuta una richiesta HTTP da parte del client.

Questa classe, estende la classe BasicAuthorizationFilter e sovrascrive il comportamento del metodo ereditato doFilterInternal().

BasicAuthorizationFilter

```
...  
...  
...  
...  
+ doFilterInternal(HttpServletRequest req, HttpServletResponse res, FilterChain chain)  
...
```

(is a relation <extends>)

AuthorizationFilter

```
- JwtProvider jwtProvider  
- String prefix  
- String param  
...  
...  
@Override  
+ doFilterInternal(HttpServletRequest req, HttpServletResponse res, FilterChain chain)  
- UsernamePasswordAuthenticationToken getAuthentication(String header)
```



```
public class AuthorizationFilter extends BasicAuthenticationFilter {
```

```
    private JwtProvider jwtProvider;
```

```
    private String prefix;
```

```
    private String param;
```

Viene fatta una constructor Inject.

- Viene iniettata una istanza di JwtProvider
(Ecco perché serviva marcarlo come @Component prima)

```
@Autowired
```

```
public AuthorizationFilter(AuthenticationManager authenticationManager, JwtProvider jwtProvider, String prefix, String param) {
```

```
    super(authenticationManager);
```

```
    this.jwtProvider = jwtProvider;
```

```
    this.prefix = prefix;
```

```
    this.param = param;
```

Richiamiamo il costruttore del padre passando l'oggetto memorizzato nella variabile authenticationManager passato come primo parametro nel costruttore di questa classe

RIHIESTA http

RISPOSTA http

```
@Override
```

```
protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res, FilterChain chain) throws IOException, ServletException {
```

```
    String header = req.getHeader(param);
```

```
    if (header == null || !header.startsWith(prefix)) {
```

```
        chain.doFilter(req, res);
```

```
        return;
```

```
    }
```

```
    UsernamePasswordAuthenticationToken authentication = this.getAuthentication(header);
```

```
    SecurityContextHolder.getContext().setAuthentication(authentication);
```

```
    chain.doFilter(req, res);
```

```
}
```

Override del metodo ereditato doFilterInternal().

In questo modo ridefinisco il comportamento di questo metodo della classe padre

```
private UsernamePasswordAuthenticationToken getAuthentication(String header) {
```

```
    DecodedJWT decoded = jwtProvider.decodeJwt(header);
```

```
    return new UsernamePasswordAuthenticationToken(decoded.getSubject(), null, Collections.emptyList());
```

```
}
```



Nel costruttore della classe vengono passati come parametri :

- l'oggetto *JwtProvider* visto poco fa, lo utilizzeremo per decodificare il token,
- le proprietà del file *application.properties* che servono al metodo *doFilterInternal*.

Esaminiamo la funzione *doFilterInternal*, il cuore della classe.

Il metodo *doFilterInternal* viene appunto ereditato dalla classe *BasicAuthenticationFilter*, e ha come parametri :

1. la richiesta http proveniente dal client e diretta al server (**req**),
2. la risposta http del server al client(**res**)
3. la filter chain (**chain**).

Quello che faremo in questo metodo è

- ➔ verificare il token presente nella richiesta && decidere come proseguire nella catena dei filtri.

Per prima cosa si verifica la presenza del JWT all'interno dell'header della richiesta del parametro *Authorization*, e che questo JWT abbia come prefisso '*Bearer*'.

Nel caso di esito positivo proseguiremo con la decodifica del token, altrimenti, otterremo nella richiesta http fatta dal client un errore di autenticazione.

Se anche la decodifica del JWT ha esito positivo, allora il contesto di sicurezza di Spring sarà arricchito con un nuovo elemento di autenticazione oggetto di tipo *UsernamePasswordAuthenticationToken*, senza il quale la *FilterChain*, alla fine del suo percorso, restituirebbe comunque un errore di autenticazione.

La chiamata *doFilter* della *FilterChain* permette di proseguire nella catena e completare l'autorizzazione.

Bene, abbiamo creato il nostro filtro che permette di autorizzare una richiesta esterna tramite token JWT presente nell'header realizzando così la sicurezza delle API.

Ma, funziona?

Assolutamente No! Il filtro viene ignorato finché non lo aggiungiamo all'interno della catena dei filtri (*FilterChain*) di Spring Security e lo faremo creando una configurazione appropriata.



2.3.4 La classe SecurityConfig.java

(Configurazione di Spring-Security)

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private JwtProvider jwtProvider;

    @Value("${security.prefix}")
    private String prefix;

    @Value("${security.param}")
    private String param;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .cors().and().csrf().disable();

        http
            .sessionManagement()
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and()
            .addFilter(
                new AuthorizationFilter(authenticationManager(), jwtProvider, prefix, param)
            )
            .authorizeRequests()
            .antMatchers("/public/**").permitAll()
            .anyRequest().authenticated();
    }
}
```



La classe *SecurityConfig* qui sopra rappresenta la nostra configurazione per Spring Security.

Cerchiamo di capire meglio quanto scritto.

SecurityConfig viene annotata con

- *@Configuration* per rappresentare una configurazione Spring,
- *@EnableWebSecurity*
- *@EnableGlobalMethodSecurity* riguardanti Spring Security.

NOTA BENE BENE:

Per abilitare Http Security in Spring occorre:

- 1- estendere la classe astratta *WebSecurityConfigurerAdapter***
- 2- implementare il metodo *configure*.**

Per fornire una configurazione di default dobbiamo agire sull'oggetto *HttpSecurity* passato come argomento nel metodo *configure*, come mostrato nell'esempio proposto.

Per essere sicuri che ogni richiesta sia autenticata basta aggiungere:

➔ `http.authorizeRequests().anyRequest().authenticated();`

NOTA:

La classe *HttpSecurity* implementa il pattern builder

Nella configurazione proposta, abbiamo permesso a tutte le richieste che iniziano con endpoint *'/public'* di essere escluse dalla catena di filtri di sicurezza. Riguardano le API che vogliamo esporre pubblicamente, come ad esempio la registrazione dell'utente e il login.

```
.antMatchers("/public/**").permitAll()
```



Per implementare Spring Security con JWT dobbiamo prendere in considerazione alcune osservazioni.

Di default, Spring Security utilizza un sistema di generazione cookie che vengono scambiati ad ogni richiesta client-server, e successivamente registra un utente autenticato tra le sessioni attive di Spring in un oggetto chiamato *Principal*.

Questo, insieme alla *filter chain* (catena dei filtri di sicurezza) di default, permette a Spring di verificare l'autenticazione dell'utente controllando le informazioni tra le sessioni attive.

Nella nostra implementazione JWT dobbiamo quindi:

- escludere i cookie
- non creare delle sessioni (tramite sessioni con politica *stateless*)
- modificare il comportamento della filter chain per validare il token

Abbiamo fatto questo aggiungendo alla configurazione (SecurityConfig.java) le righe:

```
.cors().and().csrf().disable()
```

```
.sessionManagement()  
    .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
```

Abbiamo anche aggiunto il filtro *AuthorizationFilter* estendendo il filtro di default *BasicAuthenticationFilter* visto poco fa.

```
.addFilter(new AuthorizationFilter(jwtProvider, prefix, param))
```

Così facendo, **abbiamo configurato correttamente Spring Security per gestire un'autenticazione con JWT**.



2.3.5 Definire un' API per l'autenticazione

Guardiamo ora un esempio di API di autenticazione per la generazione del JWT da parte del server.

2.3.6 Le classi DTO

```
public class LoginInputDto {  
    private String username;  
    private String password;  
  
    // getter and setter  
}
```

```
public class LoginOutputDTO {  
    private String token;  
  
    // getter e setter  
}
```



2.3.7 API di login

```
@RestController
@RequestMapping("public/authentication")
public class AuthenticationController {

    @Autowired
    private JwtProvider jwtProvider;

    @Autowired
    private UserService userService;

    @PostMapping
    public ResponseEntity authenticate(@RequestBody LoginInputDto body) {
        // verifica se l'utente è registrato su db
        User user = userService.getByUsernameAndPassword(body.getUsername(), body.getPassword());
        if (user == null) {
            return ResponseEntity.badRequest().build();
        }

        // se voluto si può inserire alcune informazioni dell'utente nel jwt modificando il metodo create
        String jwt = jwtProvider.createJwt();
        LoginOutputDto dto = new LoginOutputDto();
        dto.setToken(jwt);
        return ResponseEntity.ok(dto);
    }
}
```



Assumiamo di aver creato un'entità *User* e una classe service *UserService* per eseguire le CRUD sull'utente.

Per prima cosa verifichiamo che l'utente sia presente sul database (utente registrato).

Se così non fosse, restituiamo una risposta del tipo *BadRequest*, ovvero, i dati dell'utente non sono corretti.

L'API di registrazione non è altro che un endpoint in cui vengono passati i dati dell'utente (di solito tramite un form di registrazione) per essere salvati sul database.

In caso di successo, si restituisce un oggetto *LoginOutputDto* contenente il token generato da utilizzare nelle chiamate successive client-server.

Arrivato a questo punto, avrai configurato correttamente Spring Security per la gestione di un'autenticazione con JWT creando un'API per il login dell'utente, contribuendo così alla sicurezza delle API della tua applicazione Spring Boot.



RIFERIMENTI:

- 1- <https://giuseppetoto.it/json-web-token-un-nuovo-modo-per-autenticare-le-nostre-sessioni-in-modo-sicuro-e-scalabile-ee7b481d8b27>
- 2- <http://jwt.io/>
- 3- <https://tools.ietf.org/html/rfc7519>
- 4- <https://devnews.it/blog/posts/58ab32b157db07436c4c3569/cosa-sono-jwt-json-web-tokens>
- 5- <https://tools.ietf.org/html/rfc7519>
- 6- <https://lateralecloud.it/sicurezza-api-spring-security-jwt/>
- 7- <https://cdn.auth0.com/blog/aurelia/aurelia-auth-bearer.png?dl=1>

REPO GITHUB:

- <https://github.com/venflon1/SpringSecurityWithJWT.git>



