Token Authentication

Rev 2.4 del 29/03/2022

Meccanismi di autenticazione

1. Server Based Authentication

Quando si fa un login, l'approccio tradizionale consisteva nel creare una sessione tra il server e il client e memorizzare lo user_id lato server all'interno della sessione, al fine di poter recuperare le informazioni di autenticazione in tutte le successive chiamate. Questa tecnica prende il nome di **server based authentication**

Con l'apertura delle reti ad un numero sempre maggiore di utenti con conseguente aumento del numero di richieste, **la tendenza attuale è quella di 'alleggerire' sempre di più i server spostando dati ed elaborazione sui client**, per cui la Server Web Authentication ha iniziato a perdere di significato.

Problemi legati alla Server Based Authentication

- **Overhead**: Ogni volta che un utente si è autenticato, il server deve creare un record relativo alla sessione. Questo di solito viene fatto in memoria e quando ci sono molti utenti che autenticano, aumenta l'overhead sul server.
- **Scalability**: Il fatto che le sessioni siano memorizzate in memoria, crea problemi di scalabilità. Se la session viene creata sul server di login, l'utente non potrà inviare richieste ad altri server. I cloud provider di solito replicano i server per gestire il carico applicativo; il fatto di avere informazioni vitali nella memoria di sessione limitano la capacità di scalare. Se abbiamo creato la sessione quando eravamo sul server A e successivamente il bilanciatore 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 possono essere memorizzarle all'interno di uno spazio condiviso su un **database no-sql**).

2. HTTP Based Authentication

Una prima alternativa alla Server Based Authentication è la cosiddetta **HTTP Based Authentication** che consiste nel trasmettere username e password in corrispondenza di ogni richiesta (all'interno di un'apposita intestazione HTTPs).

username e password dovrebbero pertanto essere salvati da qualche parte sul client (ad esempio all'interno del local storage). Salvare una password in local storage non è il meglio che si possa fare.

3. Token Authentication

Json Web Token (JWT) è uno standard abbastanza recente di Token Authentication, standardizzato all'inizio del 2015 in cui il server, in corrispondenza della validazione del login, provvede a creare un token 'cifrato' (*signature based*) contenente alcune informazioni dell'utente ed una **scadenza**.

Questo token viene trasmesso al client che lo utilizzerà come identificativo per tutte le successive richieste. Sostanzialmente invece di trasmettere username e password, in corrispondenza di ogni richiesta, al loro posto viene trasmesso il token.

Un esempio di token è rappresentato dalla API KEY che occorre scaricare per poter accedere alle Google Maps. In quel caso il token non viene rilasciato a seguito di un login, ma occorre **registrarsi** e **scaricare manualmente** il token che dovrà poi essere allegato ad ogni richiesta accodandolo alla url.

Vantaggi dell'autenticazione tramite Token

- Il meccanismo del token è completamente **stateless**, cioè le informazioni di autenticazione stanno direttamente nel token, evitando di dover passare dal database o di usare le sessioni per memorizzare le informazioni sull'autenticazione. Quindi non richiede la memorizzazione di alcuna informazione sul server. Inoltre se i server sono replicati è sufficiente che tutti utilizzino lo stesso tipo di token ed il problema è completamente risolto.
- Il token viene trasmesso nell'intestazione della richiesta. Se il token non è valido al dispatching della richiesta non si arriva nemmeno, per cui l'impegno del server è minimo.
- Lo stesso token può essere utilizzato anche per accedere a sistemi diversi rispetto a quello che lo ha generato. Meccanismo detto SSO **Single Sign On** utilizzato da google per uniformare gli accessi ai diversi servizi ma ormai utilizzato anche in grandi aziende che espongono molteplici portali relativamente a servizi differenti (spedizioni, logistica, etc).

Sicurezza dell'autenticazione tramite Token

- La **pagina di login** che valida l'utente e distribuisce il token deve essere protetta in modo sistemistico facendo ad esempio in modo che, in caso di più richieste successive da parte dello stesso indirizzo IP, il sistema disattivi temporaneamente o definitivamente le richieste da parte di quell'indirizzo IP
- Durante le successive comunicazioni con il server occorre necessariamente 'proteggere' il token in modo che non possa essere intercettato. Questo è uno dei motivi che, nel **2015**, ha portato alla migrazione di quasi tutti i servizi da HTTP a **HTTPs**, anche quelli in apparenza meno sensibili. L'unica altra alternativa potrebbe essere quella di cifrare manualmente il token tramite un algoritmo noto sia al client che al server ma sconosciuto agli altri (che è poi esattamente ciò che fa HTTPs)
- Il token, anche se non contiene informazioni sensibili come ad esempio la password personale e non può in alcun modo essere modificato, in caso di intercettazione potrebbe comunque essere utilizzato per accedere indebitamente ai servizi. **Per questo ogni volta che ci si collega ad un servizio da un nuovo dispositivo viene immediatamente inviata una mail di 'NUOVO ACCESSO' in cui è possibile richiedere la disattivazione immediata del token**. Nel caso di servizi più sensibili come la home banking, per alcune operazioni viene attivato un secondo livello di sicurezza con l'inserimento di un PIN personale memorizzato su una scheda o su un dispositivo
- Spesso i token sono a rinnovo automatico, nel senso che se il token sta per scadere o è appena scaduto, il server ne crea ed invia uno nuovo con la scadenza aggiornata. Se non ci si collega per più giorni occorre poi rifare il login.

Gestione delle password tramite la libreria bcrypt

La libreria **`bcrypt`** distribuita tramite npm è scritta in C++ e non è supportata da heroku.
Sono invece supportate **`bcrypt-nodejs`** (più vecchia di bcrypt) e **`bcryptjs`** che è una libreria interamente scritta in javascript considerata del tutto equivalente a **`bcrypt`** (soltanto più lenta).

Funzionamento di bcrypt

Utilizza una salt-criptography. Il nome deriva dal fatto che il sale solitamente ostruisce le arterie ed in questo caso viene utilizzato per ostruire eventuali attacchi mirati alla scoperta delle password.
La ragione che sta alla base della salt-criptography è che di solito gli utenti tendono a scegliere password semplici e conosciute in modo da poterle facilmente ricordare.

Le password normalmente vengono salvate all'interno del database non in modo diretto ma tramite una impronta irreversibile come ad esempio una hash MD5. Se però la password è semplice esistono molte applicazioni che, tramite ricerca sequenziale, consentono di risalire alla password originaria.

Lo scopo della salt-criptography è quello di 'randomizzare' una password comune, in modo da creare una password risultante meno standard e difficilmente individuabile in un database di reverte.

La tecnica consiste nel generare un **saltValue** casuale (stringa lunga 16 caratteri) ed applicare un algoritmo al quale si passano come parametri (**saltValue,** password). Un apposito parametro detto **costo** (valore intero compreso tra 8 e 31 con default 10) definisce i turni di espansione della chiave, cioè 2^{costo} indica quante volte deve essere applicato l'algoritmo prima di arrivare alla password finale. Più alto è il costo maggiore è la sicurezza, ma l'algoritmo diventa più pesante.

Al termine dell'elaborazione bcrypt restituisce una stringa base64 così strutturata:

- 4 caratteri di **intestazione** fissa, di cui il primo e l'ultimo sono sempre un $, mentre i due caratteri centrali possono essere 2a, 2b, 2y
- 3 caratteri per il **costo** utilizzato per la generazione della chiave.
  Il costo è un numero su due caratteri compreso tra 08 e 31. Il terzo carattere è sempre un $.
- 22 caratteri di **saltValue** (128 bit (16 bytes) codificati su 22 caratteri base64)
- 31 caratteri di **hash** della password risultante (184 bit (23 bytes) codificati su 31 caratteri base64)

La stringa finale ha una lunghezza complessiva di 60 bytes. Esempio:

$2a$10$PIRaOciYNtt8GomIaXRHnOzsWosHmeaiZTPyjNGq/2IsGveut6i/q

Notare nella stringa precedente che bcrypt utilizza il set base della codifica base64 con + e /

Il fatto che il salt value venga memorizzato all'interno del database non costituisce un problema.
Il saltValue utilizza un alfabeto di 64 caratteri su una stringa lunga 22 caratteri, con un totale di **64^{22}** combinazioni cioè **$5\ 10^{39}$**. Per cui un sito di reverse per ogni password comune dovrebbe memorizzare **$5\ 10^{39}$** possibili combinazioni moltiplicato ancora per i turni di espansione della chiave

Metodi sincroni

```
var hash = bcrypt.hashSync(myPassword, 10);   // 10 è il costo
bcrypt.compareSync("my password", hash);      // true
bcrypt.compareSync("not my password", hash);  // false
```

Se la variabile `myPassword` contiene **stringa vuota**, viene codificata comunque in bcrypt
Se la variabile `myPassword` è **undefined**, hashSync va in errore

Metodi asincroni

Siccome il calcolo della hash richiede una fase di elaborazione da parte del server, è consigliato l'utilizzo della versione asincrona, in modo da non sprecare inutilmente tempo di CPU.
La versione asincrona inietta la hash come parametro della funzione di callback.

```
var hash = bcrypt.hash('myPassword', 10, function(err, hash) {
     console.log(hash)
}

bcrypt.compare("myPassword", hash, function(err, res) {
     // res === true/false
});
```

JWT Token Authentication

- L'utente si autentica mediante username e password
- In caso di credenziali valide il server restituisce un token 'firmato' digitalmente
- Il client (mediante java script) memorizza il token e lo invia in ogni richiesta successiva
- In corrispondenza di ogni richiesta il server controlla il token e, se valido, restituisce i dati richiesti

Un token JWT è costituito da tre parti:

1) header

E' un oggetto json contenente il tipo di token utilizzato (es JWT) ed il nome dell'algoritmo da utilizzare per la firma digitale (signature).
Esempio:

```
{
    "typ": "JWT"
    "alg": "HS256",  // SHA256
}
```

Questo oggetto viene serializzato mediante una codifica **encodebase64** che divide la sequenza binaria in gruppi di 6 bit creando in pratica quattro cifre ogni 3 caratteri. I caratteri utilizzati sono in tutto 64: le 26 lettere minuscole, le 26 lettere maiuscole, i 10 caratteri numerici, i caratteri + e / . In realtà JWT utilizza una variante che sostituisce + e / con - e _ Questa variante viene utilizzata quando la stringa codificata deve essere usata in una URL o in un filename. La codifica base64 relativa all'header precedente risulta essere la seguente:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
```

Questa stringa base 64 può essere facilmente decodificata utilizzando la funzione **decodebase64**

2) payload

Nel payload sono contenute le informazioni relative all'utente che si è appena autenticato (**claims** = dichiarazioni dell'utente). Esempio:

```
{
    "_id": "1234567890",
    "name": "John Doe",
    "admin": false
}
```

Il payload viene anch'esso codificato in encodebase64 ottenendo la seconda parte del token:

```
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9
```

options standard utilizzabili nel payload

Le specifiche JWT definiscono alcuni campi standard (**options**) da utilizzare all'interno di un payload JWT (wikipedia) :

code	*name*	*description*
iss	Issuer	Identifies principal that issued the JWT.
sub	Subject	Identifies the subject of the JWT.
aud	Audience	Identifies the recipients that the JWT is intended for. Each principal intended to process the JWT must identify itself with a value in the audience claim. If the principal processing the claim does not identify itself with a value in the aud claim when this claim is present, then the JWT must be rejected
exp	Expiration time	Identifies the expiration time on or after which the JWT must not be accepted for processing. The value should be in NumericDate format.
nbf	Not before	Identifies the time on which the JWT will start to be accepted for processing.
iat	Issued at	Identifies the time at which the JWT was issued.
jti	JWT ID	Case sensitive unique identifier of the token even among different issuers.

3) signature

In coda al token viene apposta una **firma digitale** (signature) realizzata utilizzando l'algoritmo indicato nell'header, ad esempio l'algoritmo **`HMAC SHA 256`** che crea impronte di lunghezza 256 bit, cioè 32 caratteri, che diventano 43 in encodebase64.

La firma digitale viene creata nel modo seguente:
Si costruisce una stringa s costituita da:

```
String s = base64UrlEncode(header) + "." + base64UrlEncode(payload)
```

Si applica a questa stringa l'algoritmo **SHA 256** utilizzando come chiave una **chiave privata del server**

```
String signature = HMACSHA256(s, key)
```

La signature encodebase64 ottenuta sarà la seguente:

```
TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

Token finale

Il token finale viene ottenuto con il concatenamento delle 3 stringhe precedenti suddivise da un puntino

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpv
aG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

Il server, in corrispondenza di ogni richiesta, dovrà verificare la presenza del token e la sua validità.
I passi da eseguire sono i seguenti:

- Sulla base dell'header e del payload **ricalcola la signature** utilizzando la propria chiave privata. Se la signature differisce da quella ricevuta significa che il token è corrotto (o modificato) e la richiesta viene rifiutata.
- Legge il payload verificando che la data di scadenza sia ancora valida
- Decide se eventualmente aggiornare il token

JWT.IO

Il sito **`jwt.io`** consente di visualizzare il token relativo ad un certo contenuto oppure, al contrario, partendo da un token, consente di visualizzarne il contenuto tramite una semplice decodifica base64.

Implementazione dell'algoritmo

Libreria: **jsonwebtoken** (non **jwt** che è invece una libreria per linux)

Si supponga di essere all'interno di una pagina di login in qualche modo protetta.
In corrispondenza di una richiesta di login valida il server crea e restituisce al client un token firmato con una propria chiave privata definita dal programmatore:

```
var privateKey = 'RESTFULAPIs';
var token = jwt.sign(
            {  _id: data._id,
               nome: data.nome,
               mail: data.mail
            },  privateKey);
res.setHeader("Set-Cookie", "token="+token+";max-age="+(60*60*24*7)+";Path=/")
```

Il metodo **jwt.sign()** crea un token completo (header, payload, signature) memorizzando nel payload i campi dichiarati nel primo parametro. Va bene qualsiasi campo informativo (esclusa ovviamente la password) con l'aggiunta di eventuali 'options' fra quelle indicate nella tabella precedente.
Il **secondo parametro** rappresenta la chiave privata da utilizzare per apporre la firma digitale.

Il risultato sarà il seguente:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJtYWlsIjoicGlwcG9AdmFsbGF1cmkuZWR1Iiwibm9tZSI6InBpcHBvIiwiX2lkIjoxLCJpYXQiOjE1MDk5ODk3NTF9.8UDcTOw0Eu9QRd1n_NfjaSXAKR5mXcND-gPI3-XMuwI

Copia / incollando questo token sul sito **jwt.io** (che esegue semplicemente la decodifica decodeBase64) si possono vedere le seguenti informazioni contenute nel token.

PAYLOAD:
Il campo **iat** (**issued at**) creato automaticamente da jwt.sign() indica la data di creazione del token
Il campo **exp** (expires) indica l'eventuale scadenza del token
Entrambe queste date sono espresse in **secondi** trascorsi al 1/1/1970 (il cosiddetto UNIX TIMESTAMP).
Occorre quindi dividere per 1000 il risultato restituito da **getTime**() :

```
var expire = new Date();   // Object di tipo Date contenente data corrente
expire =  Math.floor(expire.getTime()/1000) + 60*60;  // 1 ora
```

Per vedere il valore di queste date in formato ISODate è possibile utilizzare un qualsiasi `Unix timestamp converter` presente in rete

SIGNATURE contiene la firma digitale così ottenuta:

- si applica l'algoritmo HMAC SHA 256 al concatenamento di base64UrlEncode(header) + "." + base64UrlEncode(payload) sulla base della chiave segreta passata dall'applicazione a jwt.sign()
- L'impronta ottenuta viene a sua volta codificata in modalità base64UrlEncoded ed accodata ai due campi precedenti

Validazione del token in corrispondenza delle richieste successive

Su tutte le successive richieste il server dovrà provvedere alla validazione del token nel modo seguente:

```
if(req.headers && req.headers.cookie){
     let token = readCookie(req);
     jwt.verify(token, privateKey, function (err, data) {
          if(err)
              sendError(req, res, 401, "Unauthorized: invalid token");
          else {
              var payload = jwt.decode(token);
              var date = new Date();
              if(payload["exp"] < Math.floor(date.getTime()/1000))
                  sendError(res, 401, "Unauthoirized: token out of date");
              else   {
                    database access and response
              }
```

Il metodo **jwt.verify()** estrae le prime due componenti del token, ricalcola la firma digitale utilizzando la chiave privata ricevuta come parametro, e confronta la firma ricalcolata con quella ricevuta restituendo err = null in caso di esito positivo del confronto.

In realtà **jwt.verify()** :

- Verifica anche la data di scadenza dl token, generando automaticamente un errore in caso di token scaduto.
- Restituisce all'interno del campo **data** il payload del token, rendendo di fatto del tutto inutile l'utilizzo del metodo **jwt.decode**()

Il metodo **jwt.decode()** restituisce in modo diretto il payload di un token. Dispone di un secondo parametro che, se `true`, fa sì che venga restituito non solo il payload ma anche la header:

```
var decoded = jwt.decode(token, {complete: true});
console.log(decoded.header);
console.log(decoded.payload);
```

Come aggiornare soltanto il campo expires

Nel momento in cui il server riceve una nuova richiesta con token valido, dovrebbe riaggiornare la scadenza del token e rimandarlo indietro al client. Invece di rigenerare un nuovo token da zero, ricreando di conseguenza anche il campo IAT, si può utilizzare la seguente sintassi:

```
function rigeneraToken(payload, signature) {
    var exp = Math.floor(Date.now() / 1000) + 60;
    return jwt.sign({...payload, 'exp':exp}, signature);  //spread operator
};
```

In pratica vengono passati a jwt.sign tutti i campi del payload precedente, compreso IAT che dunque non viene ricreato, e poi un secondo campo exp che sovrascrive il precedente.
In questo modo IAT continua ad indicare la data di prima creazione del token

Considerazioni sulla Trasmissione del token

Una volta creato il token, il server lo può trasmettere al client attraverso due diverse modalità:

1. lo può trasmettere all'interno di un **cookie**. In questo caso, la gestione lato client è completamente trasparente. Il browser salverà il token nell'area dedicata ai cookies e lo ritrasmetterà in automatico in corrispondenza di ogni richiesta successiva.
2. lo può trasmettere il token al client all'interno di una intestazione HTTP preposta (tipicamente **authorization o x-auth-token**). In questo caso tipicamente il client andrà a salvare il token in localStorage e, in corrispondenza di ogni richiesta, andrà a leggerlo dal localStorage e ad aggiungerlo manualmente all'interno dell'http request.

Dal punto di vista della sicurezza la soluzione migliore sembrerebbe essere quella di salvare il token all'interno di un **cookie httponly=true e secure=true** non accessibile da java script.
il salvataggio del token all'interno del local storage è sconsigliato per motivi di vulnerabilità ad attacchi XSS e XSRF. Eventuali script malevoli inclusi nell'applicazione potrebbero accedere a tutti i nostri token.

jwt-authentication-best-practices: Don't store it in local storage (or session storage). If any of the third-party scripts you include in your page gets compromised, it can access all your users' tokens. The JWT needs to be stored inside an httpOnly cookie, a special kind of cookie that's only sent in HTTP requests to the server, and it's never accessible (both for reading or writing) from JavaScript running in the browser

Dal punto di vista della portabilità il discorso è ribaltato. Mentre nelle classiche **web app** tutto funziona correttamente, nel caso delle **APP** (sia Android che IOs) il cookie **non** viene salvato dalla APP su memoria persistente per cui, se si chiude e si riapre la APP, il cookie viene perso e l'utente deve ripetere il login. Per questo motivo qualcuno sconsiglia l'utilizzo dei cookies nelle APP mobile.
Viceversa non esiste il problema del local storage: la **APP**, una volta ricevuto il token all'interno di una http header, lo può salvare in local storage o in altra area privata in tutta sicurezza.

Soluzioni

1) Nel caso di java android esiste un oggetto **CookieManager()** che può rendere persistenti i cookie
   Nel caso di cordova si possono trovare diversi plug in mirati a rendere persistenti i cookies lato client
   https://github.com/DriveTimeInc/cordova-plugin-cookie-persistence
   https://github.com/surgeventures/cordova-plugin-cookie-manager
   In tutti i casi però i cookies devono essere accessibili da javascript (**httponly=false**) riducendo di fatto la sicurezza. Rimane inoltre il problema che i cookies ricevuti dal browser tramite **ajax** non sono disponibili all'interno della collezione document.cookie utilizzata dai precedenti plugin.
2) Volendo evitare di utilizzare i cookies, si può inviare il token all'interno di una intestazione http e salvarlo poi in local storage oppure, meglio, in un **native-storage** utilizzando ad esempio il seguente plugin ufficiale di cordova: `cordova-plugin-nativestorage`. In tal caso, rispetto al local storage, occorre differenziare il codice della **APP** dal codice della **web app**

Esempio completo di validazione ed aggiornamento del token lato server

```
function controllaToken(req, res, next) {
   let token = readCookie(req);
   if (token == '') {
       inviaErroreToken(req, res, 403, "token mancante");
   }
   else {
       jwt.verify(token, privateKey, function(err, payload) {
           if (err)
               inviaErroreToken(req, res, 403, "token corrotto o scaduto");
           else {
               // ricreo il token in modo da aggiornare la data di scadenza
               let newToken = createToken(payload)
               writeCookie(res, newToken)
                 // Questo consente alle routes successive di poter accedere
                 // alle informazioni contenute nel token
               req.payload = payload;
               next();
           }
       });
   }
}

function readCookie(req) {
  var valoreCookie = "";
  if (req.headers.cookie) {
      var cookies = req.headers.cookie.split(';');
      for (let item of cookies) {
         item = item.split("=");
         // davanti alle chiavi ci sono in genere degli spazi di separazione
         // inseriti automaticamente in coda al punto_virgola di separazione
         if (item[0].trim() == "token") {
              valoreCookie = item[1];
              break;
         }
      }
  }
  return valoreCookie;
}

function createToken(data) {
    let token = jwt.sign({
            '_id': data._id,
            'username': data.username,
            'iat': data.iat || Math.floor(Date.now() / 1000),
            'exp': Math.floor(Date.now() / 1000 + DURATA_TOKEN)
        },   privateKey
    );
    return token;
}

function writeCookie(res, token) {
    res.set("Set-Cookie",
            `token=${token};max-age=${DURATA_TOKEN};Path=/;httponly=true`)
}
```

Appendici

Versione sincrona di jwt.verify()

Siccome la verifica del token è una operazione da eseguire molto frequentemente su richieste anche molto diverse fra loro, la versione asincrona di **jwt.verify** è abbastanza complicata da gestire.
Occorre ogni volta passare come parametro il nome della funzione da eseguire in caso di **`success`** ed il nome della funzione da eseguire in caso di **`fail`**.

Molto più semplice la gestione sincrona riportata di seguito:

- Se si specifica la funzione di callback, **`jwt.verify()`** agisce in modo **asincrono** e, dopo aver verificato il token, inietta il payload all'interno della funzione di callback.
- Se non si specifica la funzione di callback, **`jwt.verify()`** agisce in modo **sincrono** e, dopo aver verificato il token, restituisce il payload in caso di successo oppure solleva una eccezione in caso di insuccesso. Esempio di gestione sincrona:

```
var payload=""
try {
  payload = jwt.verify(token, key);
}
catch(err) {

}
```

Lettura del token lato client

Se il cookie **non** è **httpOnly**, il client può accedere al cookie attraverso la collezione **`document.cookie`** che restituisce una stringa contenente l'intera collezione dei cookies separati da punto e virgola + spazio.

Per recuperare le singole informazioni si può utilizzare la stessa procedura `readCookie()` utilizzata lato server applicata non a `req.headers.cookie` ma a `document.cookie`

```
function visualizzaCookie(){
   if (document.cookie) {
      var cookies = document.cookie.split(';');
      for (let item of cookies) {
         item = item.split("=");
         console.log(item[0], item[1])
      }
   }
}
```

Una volta ottenuto il token, questo è però memorizzato in formato base64. Inoltre JWT non utilizza la codifica base64 standard con i caratteri + e / ma utilizza una variante che sostituisce + e / con - e _ per cui, lato client, occorre eseguire anche la conversione inversa per ottenere il formato standard:

```
var base64Url = token.split('.')[1];
var base64 = base64Url.replace('-', '+').replace('_', '/');
var payload = JSON.parse(window.atob(base64));
```

In questo modo il client, prima di effettuare la richiesta di un servizio protetto, può controllare la data di scadenza del token, rimandando direttamente al login in caso di token scaduto.