# The Ultimate Guide to handling JWTs on frontend clients (GraphQL)

I JWT (token Web JSON, pronunciati 'jot') stanno diventando un modo popolare di gestire l'autenticazione. Questo post mira a demistificare ciò che è un JWT, discutere i suoi pro / contro e illustrare le migliori pratiche nell'implementazione del JWT sul lato client, tenendo presente la sicurezza. Tuttavia, abbiamo lavorato sugli esempi con un client GraphQL, ma i concetti si applicano a qualsiasi client frontend.

## Introduzione: che cos'è un JWT?

Ai fini di auth, un JWT è un token che viene emesso dal server. Il token ha un payload JSON che contiene informazioni specifiche per l'utente. Questo token può essere utilizzato dai client quando parlano con le API (inviandolo come intestazione HTTP) in modo che le API possano identificare l'utente rappresentato dal token e intraprendere azioni specifiche dell'utente.

Ma un client non può semplicemente creare un payload JSON casuale e impersonare un utente?

Buona domanda! Ecco perché un JWT contiene anche una firma. Questa firma viene creata dal server che ha emesso il token (diciamo il tuo endpoint di accesso) e qualsiasi altro server che riceve questo token può verificare in modo indipendente la firma per garantire che il payload JSON non sia stato manomesso e abbia informazioni che sono state emesse da un fonte legittima.

Ma se ho un JWT valido e firmato e qualcuno lo ruba al client, non possono usare il mio JWT per sempre?

Sì! Se viene rubato un JWT, il ladro può continuare a utilizzare il JWT. Un'API che accetta JWT esegue una verifica indipendente senza dipendere dalla fonte JWT, quindi il server API non ha modo di sapere se si è trattato di un token rubato! Questo è il motivo per cui i JWT hanno un valore di scadenza. E questi valori sono brevi. La pratica comune è di mantenerlo per circa 15 minuti, in modo che qualsiasi JWT trapelato cesserà di essere valido abbastanza rapidamente. Inoltre, assicurati che i JWT non perdano.

Questi 2 fatti portano a quasi tutte le peculiarità della gestione dei JWT! Il fatto che i JWT non debbano essere rubati e che debbano avere brevi tempi di scadenza nel caso vengano rubati.

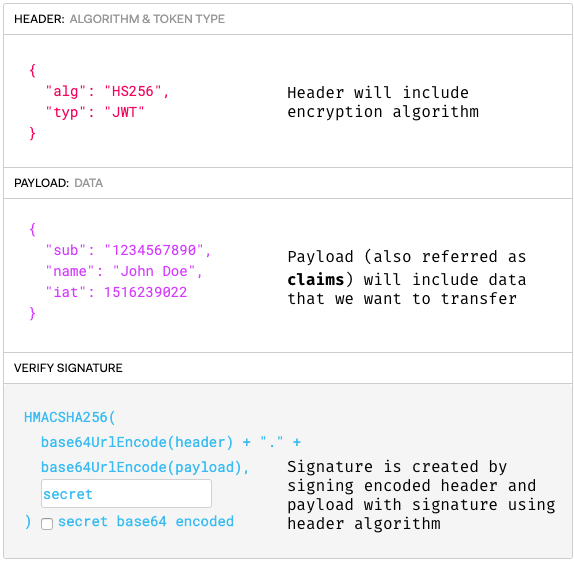
Ecco perché è anche molto importante non archiviare JWT sul client, ad esempio tramite cookie o localstorage. In questo modo rendi la tua app vulnerabile agli attacchi CSRF e XSS, tramite moduli o script dannosi per utilizzare o rubare il tuo token in giro nei cookie o nell'archivio locale.

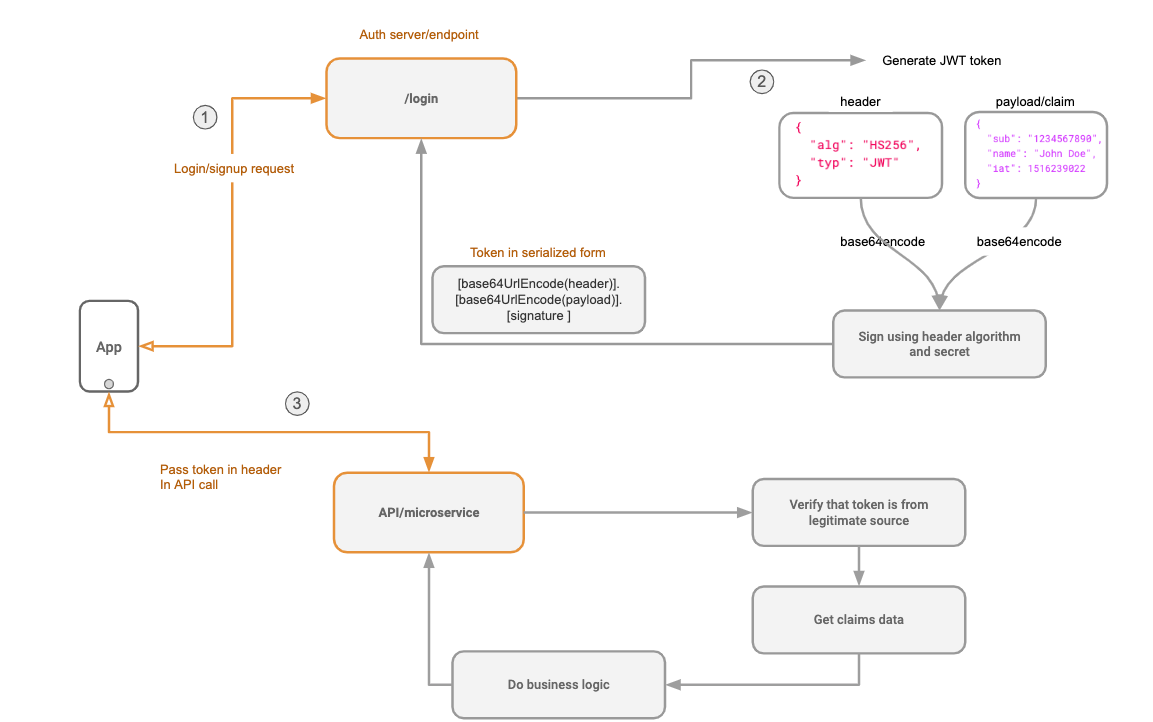
Quindi un JWT ha un tipo specifico di struttura? Che cosa sembra?

Un JWT assomiglia a questo, quando è serializzato:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.XbPfbIHMI6arZ3Y922BhjWgQzWXcXNrz0ogtVhfEd2o

Se decodifichi quella base64, otterrai JSON in 3 parti importanti: intestazione, payload e firma.





Ugh! Questo sembra complicato. Perché non dovrei attenermi ai buoni vecchi token di sessione?

Questa è una discussione dolorosa su Internet. La nostra breve (e supponente risposta) è che agli sviluppatori di backend piace usare JWT perché :

a) microservizi

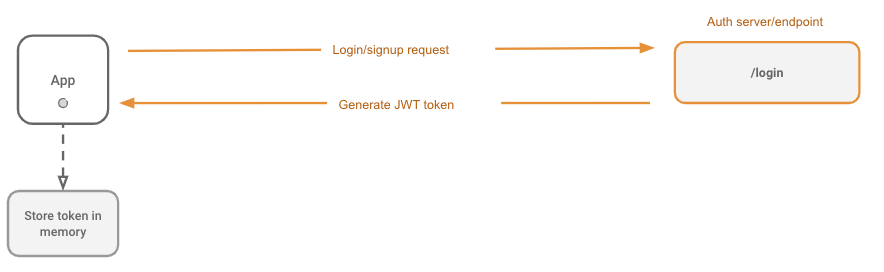
b) che non necessitano di un database di token centralizzato.

In una configurazione di microservizi, ciascun microservizio può verificare in modo indipendente che un token ricevuto da un client sia valido. Il microservizio può ulteriormente decodificare il token ed estrarre le informazioni pertinenti senza la necessità di avere accesso a un database di token centralizzato.

Questo è il motivo per cui agli sviluppatori API piacciono i JWT e noi (sul lato client) dobbiamo capire come usarli. Tuttavia, se riesci a cavartela con un token di sessione emesso dal tuo quadro monolitico preferito, sei assolutamente pronto e probabilmente non hai bisogno di JWT!

Nozioni di base: Login

Ora che abbiamo una conoscenza di base di cosa sia un JWT, creiamo un semplice flusso di login ed estraiamo il JWT. Questo è ciò che vogliamo ottenere:



Quindi, come possiamo iniziare?

Il processo di accesso non cambia davvero da quello che faresti di solito. Ad esempio, ecco un modulo di accesso che invia un nome utente / password a un endpoint di autenticazione e prende il token JWT dalla risposta. Questo potrebbe essere l'accesso con un provider esterno, un passaggio OAuth o OAuth2. Non importa, a condizione che il client ottenga finalmente un token JWT in risposta alla fase di successo dell'accesso finale.

Innanzitutto, creeremo un semplice modulo di accesso per inviare nome utente e password al nostro server di accesso. Il server emetterà il token JWT e lo memorizzeremo in memoria. In questo tutorial non ci concentreremo sul back-end del server di autenticazione, ma puoi dare un'occhiata nel repository di esempio per questo post sul blog.

Ecco come potrebbe apparire il gestore handleSubmit per un pulsante di accesso:

async function handleSubmit () {

//...

// Make the login API call

const response = await fetch(`/auth/login`, {

method: 'POST',

body: JSON.stringify({ username, password })

})

//...

// Extract the JWT from the response

const { jwt\_token } = await response.json()

//...

// Do something the token in the login method

await login({ jwt\_token })

}

L'API di accesso restituisce un token e quindi passiamo questo token a una funzione di accesso da / utils / auth dove possiamo decidere cosa fare del token una volta che lo abbiamo.

import { login } from '../utils/auth'

await login({ jwt\_token })

Quindi abbiamo il token, ora dove lo conserviamo?

Dobbiamo salvare il token JWT da qualche parte, in modo da poterlo inoltrare alla nostra API come intestazione. Potresti essere tentato di persistere nel deposito locale; non farlo! Questo è soggetto ad attacchi XSS.

Che ne dici di salvarlo in un cookie?

Anche la creazione di cookie sul client per salvare il JWT sarà soggetta a XSS. Se può essere letto sul client da Javascript al di fuori della tua app, può essere rubato. Potresti pensare che un cookie HttpOnly (creato dal server anziché dal client) possa essere d'aiuto, ma i cookie sono vulnerabili agli attacchi CSRF. È importante notare che le politiche CORS di HttpOnly e sensibili non possono impedire gli attacchi di invio di moduli CSRF e l'utilizzo dei cookie richiede un'adeguata strategia di mitigazione CSRF.

Si noti che le nuove specifiche sui cookie SameSite che stanno ottenendo un maggiore supporto nella maggior parte dei browser renderanno sicuri gli approcci basati sui cookie dagli attacchi CSRF. Potrebbe non essere una soluzione se i tuoi server Auth e API sono ospitati su domini diversi, ma altrimenti dovrebbe funzionare davvero bene!

Dove lo salviamo allora?

Bene, per ora, lo memorizzeremo in memoria (e arriveremo alle sessioni persistenti nella sezione seguente).

let inMemoryToken;

function login ({ jwt\_token, jwt\_token\_expiry }, noRedirect) {

inMemoryToken = {

token: jwt\_token,

expiry: jwt\_token\_expiry

};

if (!noRedirect) {

Router.push('/app')

}

}

Come puoi vedere qui memorizziamo il token in memoria. Sì, il token verrà annullato quando l'utente passa da una scheda all'altra, ma ci occuperemo in seguito. Spiegherò anche perché non ho la bandiera NoRedirect e jwt\_token\_expiry

Ok! Ora che abbiamo il token, cosa possiamo farci?

Usando nel nostro client API per passarlo come intestazione a ogni chiamata API

Verifica se un utente ha effettuato l'accesso verificando che la variabile JWT sia impostata.

Facoltativamente, possiamo persino decodificare il JWT sul client per accedere ai dati nel payload. Diciamo che abbiamo bisogno dell'ID utente o del nome utente sul client, che possiamo estrarre dal JWT.

Come controlliamo se il nostro utente ha effettuato l'accesso?

Controlliamo nel nostro utils / auth se la variabile token è impostata e se non lo è - reindirizzare alla pagina di accesso.

const jwt\_token = inMemoryToken;

if (!jwt\_token) {

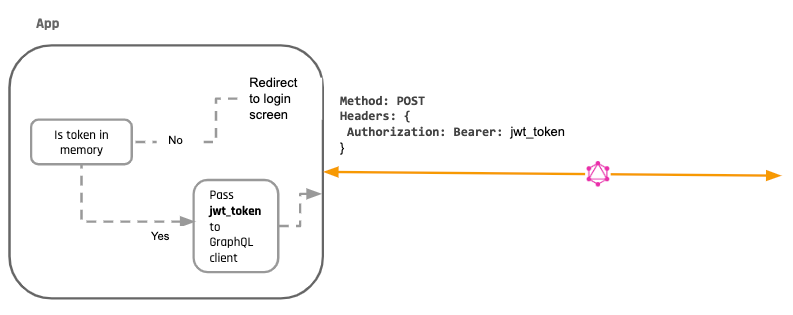
Router.push('/login')

}

return jwt\_token

Nozioni di base: configurazione del client

Ora è il momento di configurare il nostro client GraphQL. L'idea è di ottenere il token dalla variabile che abbiamo impostato e, se è presente, lo passiamo al nostro client GraphQL.



Supponendo che l'API GraphQL accetti un token di autenticazione JWT come intestazione di autorizzazione, è sufficiente configurare il client per impostare un'intestazione HTTP utilizzando il token JWT dalla variabile.

Ecco quale configurazione con il client Apollo GraphQL che utilizza un middleware ApolloLink.

let appJWTToken

const httpLink = new HttpLink({uri: 'https://graphql-jwt-tutorial.herokuapp.com/v1/graphql'})

const authMiddleware = new ApolloLink((operation, forward)=> {

if (appJWTToken) {

operation.setContext({

headers: {

Authorization: `Bearer ${appJWTToken}`

}

});

}

return forward(operation);

})

const apolloClient = new ApolloClient({

link: concat(authMiddleware, httpLink),

cache: new InMemoryCache(),

});

Come puoi vedere dal codice, ogni volta che c'è un token, viene passato come intestazione a ogni richiesta.

Ma cosa succederà se non ci sono token?

Dipende dal flusso nell'applicazione. Supponiamo che tu reindirizzi l'utente alla pagina di accesso:

else {

Router.push('/login')

}

Cosa succede se un token scade mentre lo stiamo usando?

Supponiamo che il nostro token sia valido solo per 15 minuti. In questo caso probabilmente avremo un errore dalla nostra API che rifiuta la nostra richiesta (diciamo un errore 401: non autorizzato). Ricorda che ogni servizio che sa come utilizzare un JWT può verificarlo in modo indipendente e verificare se è scaduto o meno.

Aggiungiamo la gestione degli errori alla nostra app per gestire questo caso. Scriveremo il codice che verrà eseguito per ogni risposta API e controlleremo l'errore. Quando riceviamo l'errore token scaduto / non valido dalla nostra API, attiviamo il logout o il reindirizzamento al login.

Ecco come appare il codice se stiamo usando il client Apollo:

import { onError } from 'apollo-link-error';

const logoutLink = onError(({ networkError }) => {

if (networkError.statusCode === 401) logout();

})

const apolloClient = new ApolloClient({

link: logoutLink.concat(concat(authMiddleware, httpLink)),

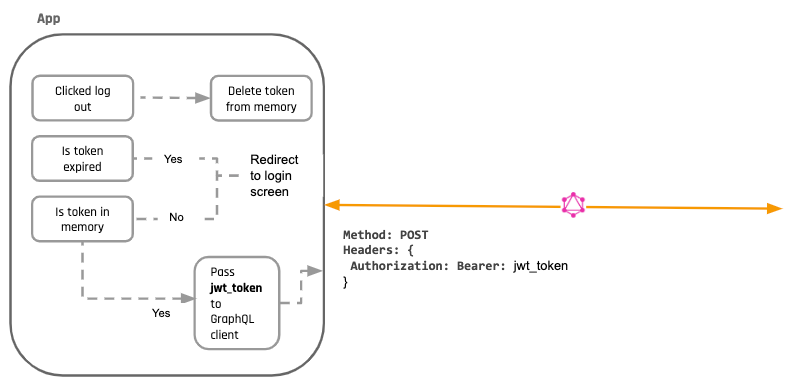
cache: new InMemoryCache(),

});

Si può notare che ciò si tradurrà in un'esperienza utente abbastanza schifosa. All'utente verrà chiesto di ripetere l'autenticazione ogni volta che il token scade. Questo è il motivo per cui le app implementano un flusso di lavoro di aggiornamento silenzioso che continua ad aggiornare il token JWT in background. Maggiori informazioni su questo nelle prossime sezioni seguenti!

Nozioni di base: disconnettersi

Con i JWT, un "logout" sta semplicemente eliminando il token sul lato client in modo che non possa essere utilizzato per le successive chiamate API.



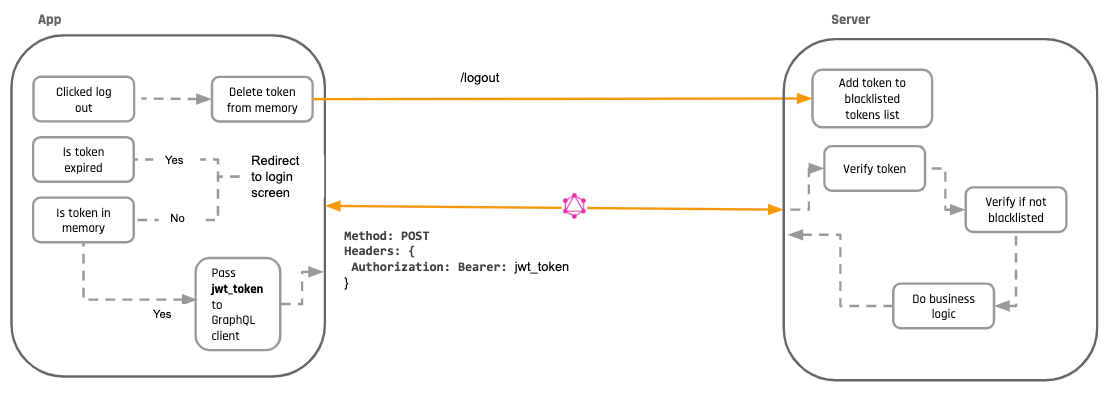
Quindi ... non c'è alcuna chiamata API / logout?

Un endpoint di logout non è realmente necessario, poiché qualsiasi microservizio che accetta i tuoi JWT continuerà ad accettarlo. Se il tuo server di autenticazione elimina il JWT, non importa perché gli altri servizi continueranno ad accettarlo comunque (poiché l'intero punto dei JWT non doveva richiedere un coordinamento centralizzato).

Il token è ancora valido e può essere utilizzato. Cosa devo fare se devo garantire che il token non possa più essere usato?

Ecco perché è importante mantenere i valori di scadenza JWT su un valore basso. Ed è per questo che garantire che i tuoi JWT non vengano rubati è ancora più importante. Il token è valido (anche dopo averlo eliminato sul client), ma solo per un breve periodo per ridurre la probabilità che venga utilizzato in modo dannoso.

Inoltre, puoi aggiungere un flusso di lavoro nella lista nera ai tuoi JWT. In questo caso, puoi avere una chiamata API / logout e il tuo server di autenticazione inserisce i token in un "elenco non valido". Tuttavia, tutti i servizi API che consumano il JWT ora devono aggiungere un passaggio aggiuntivo alla loro verifica JWT per verificare con la "lista nera" centralizzata. Questo introduce di nuovo lo stato centrale e ci riporta a ciò che avevamo prima di usare i JWT.



La blacklist non nega i vantaggi di JWT che non necessita di alcuna memoria centrale?

In un certo senso. È una precauzione facoltativa che puoi prendere se sei preoccupato che il token possa essere rubato e utilizzato in modo improprio, ma aumenta anche la quantità di verifica che deve essere eseguita. Come puoi immaginare, questo ha portato a molti digrignamenti di denti su Internet.

Cosa succederà se accedo a diverse schede?

Un modo per risolverlo è l'introduzione di un listener di eventi globale su localstorage. Ogni volta che aggiorniamo questa chiave di logout in localstorage su una scheda, l'ascoltatore si attiverà sulle altre schede e attiverà anche un "logout" e reindirizzerà gli utenti alla schermata di login.

window.addEventListener('storage', this.syncLogout)

//....

syncLogout (event) {

if (event.key === 'logout') {

console.log('logged out from storage!')

Router.push('/login')

}

}

La blacklist non nega i vantaggi di JWT che non necessita di alcuna memoria centrale?

In un certo senso. È una precauzione facoltativa che puoi prendere se sei preoccupato che il token possa essere rubato e utilizzato in modo improprio, ma aumenta anche la quantità di verifica che deve essere eseguita. Come puoi immaginare, questo ha portato a molti digrignamenti di denti su Internet.

Cosa succederà se accedo a diverse schede?

Un modo per risolverlo è l'introduzione di un listener di eventi globale su localstorage. Ogni volta che aggiorniamo questa chiave di logout in localstorage su una scheda, l'ascoltatore si attiverà sulle altre schede e attiverà anche un "logout" e reindirizzerà gli utenti alla schermata di login.

Queste sono le 2 cose che ora dobbiamo fare al logout:

1. Annulla il token
2. Imposta la voce di logout nella memoria locale

async function logout () {

inMemoryToken = null;

const url = 'http://localhost:3010/auth/logout'

const response = await fetch(url, {

method: 'POST',

credentials: 'include',

})

// to support logging out from all windows

window.localStorage.setItem('logout', Date.now())

}

In tal caso, ogni volta che esci da una scheda, il listener di eventi si attiverà in tutte le altre schede e le reindirizzerà alla schermata di accesso.

Questo funziona attraverso le schede. Ma come posso "forzare il logout" di tutte le sessioni su dispositivi diversi ?!

Tratteremo questo argomento in modo più dettagliato in una sezione successiva: Forzare il logout.

Aggiornamento silenzioso

Ci sono 2 problemi principali che gli utenti della nostra app basata su JWT dovranno ancora affrontare:

Dati i nostri brevi tempi di scadenza sui JWT, l'utente verrà disconnesso ogni 15 minuti. Questa sarebbe un'esperienza abbastanza terribile. Idealmente, vorremmo probabilmente che il nostro utente avesse effettuato l'accesso per molto tempo.

Se un utente chiude la sua app e la riapre, dovrà accedere nuovamente. La loro sessione non è persistente perché non stiamo salvando il token JWT sul client da nessuna parte.

Per risolvere questo problema, la maggior parte dei provider JWT fornisce un token di aggiornamento. Un token di aggiornamento ha 2 proprietà:

Può essere utilizzato per effettuare una chiamata API (ad esempio / refresh\_token) per recuperare un nuovo token JWT prima della scadenza del precedente JWT.

Può essere tranquillamente persistito in tutte le sessioni sul client!

Come funziona un token di aggiornamento?

Questo token viene emesso come parte del processo di autenticazione insieme a JWT. Il server di autenticazione deve salvare questo token di aggiornamento e associarlo a un determinato utente nel proprio database, in modo da poter gestire la logica JWT rinnovabile.

Sul client, prima della scadenza del token JWT precedente, colleghiamo la nostra app per creare un endpoint / refresh\_token e acquisire un nuovo JWT.

Come può un token di aggiornamento persistere in modo sicuro sul client ?!

Il token di aggiornamento viene inviato dal server auth al client come cookie HttpOnly e viene automaticamente inviato dal browser in una chiamata API / refresh\_token.

Poiché Javascript lato client non è in grado di leggere o rubare un cookie HttpOnly, questo è un po 'meglio nel mitigare XSS che nel persisterlo come un normale cookie o nell'archivio locale.

Questo è sicuro dagli attacchi CSRF, perché anche se un attacco di invio di un modulo può effettuare una chiamata API / refresh\_token, l'attaccante non può ottenere il nuovo valore del token JWT che viene restituito.

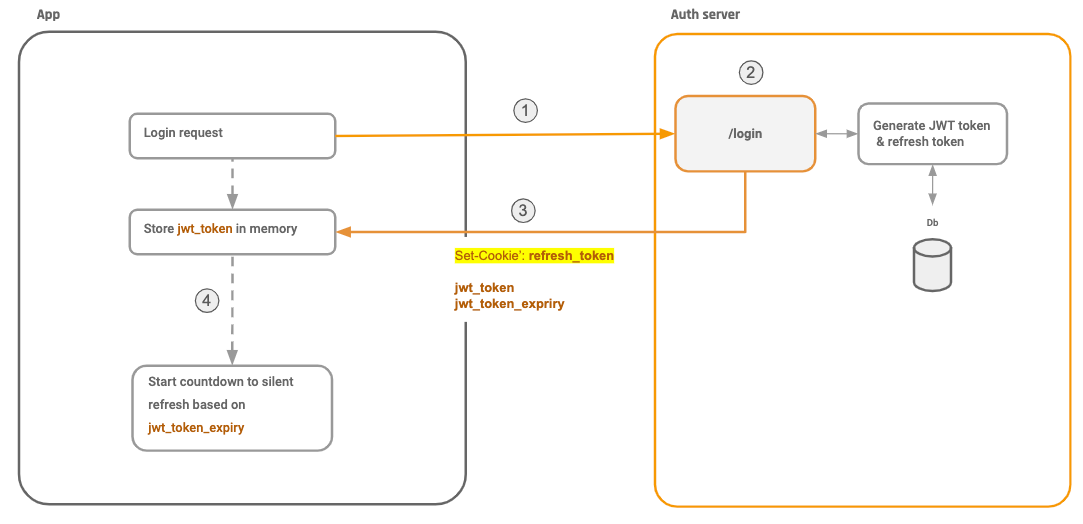
Per ricapitolare, è così che stiamo pensando a quale sarebbe il modo migliore di perseguire una sessione basata su JWT :?

Token JWT persistente nell'archivio locale (soggetto a XSS) <Token JWT persistente in un cookie HttpOnly (incline a CSRF, un po 'migliore per XSS) <Token di aggiornamento persistente in un cookie HttpOnly (sicuro da CSRF, un po' meglio per XSS) .

Si noti che mentre questo metodo non è resistente agli attacchi XSS gravi, in combinazione con le consuete tecniche di mitigazione XSS, un cookie HttpOnly è un modo consigliato di conservare le informazioni relative alla sessione. Ma persistendo la nostra sessione indirettamente tramite un token di aggiornamento, impediamo una vulnerabilità CSRF diretta che avremmo avuto con un token JWT.

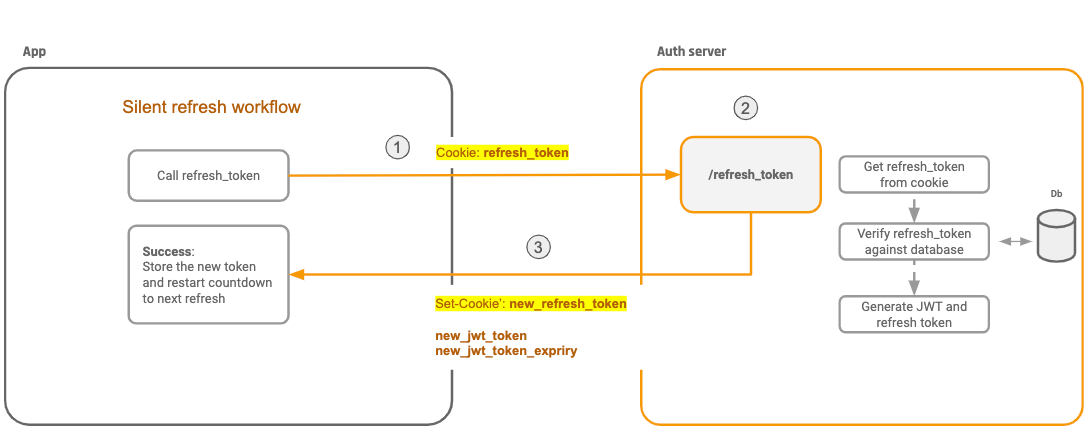
Come si presenta il nuovo processo di "login"?

Non cambia molto, tranne che un token di aggiornamento viene inviato insieme a JWT. Diamo un'occhiata di nuovo a un diagramma del processo di accesso, ma ora con la funzionalità refresh\_token:



1. L'utente accede con una chiamata API di accesso.
2. Il server genera token JWT e refresh\_token
3. Il server imposta un cookie HttpOnly con refresh\_token. jwt\_token e jwt\_token\_expiry vengono restituiti al client come payload JSON.
4. Jwt\_token è archiviato in memoria.
5. Viene avviato un conto alla rovescia per un futuro aggiornamento silenzioso basato su jwt\_token\_expiry

E ora, come appare il refresh silenzioso?



Ecco cosa succede:

1. Call / refresh\_token endpoint
2. Il server leggerà httpSolo cookie e se trova un refresh\_token valido, quindi ...
3. ... il server restituisce un nuovo jwt\_token e jwt\_token\_expiry al client e imposta anche un nuovo cookie token di aggiornamento tramite l'intestazione Set-Cookie.

Sessioni persistenti

Ora che possiamo assicurarci che i nostri utenti non continuino a disconnettersi, rivolgiamo la nostra attenzione al secondo problema delle sessioni persistenti.

Noterai che se gli utenti chiudono l'app e la riaprono (diciamo chiudendo la scheda del browser e riaprendola), gli verrà chiesto di accedere nuovamente.

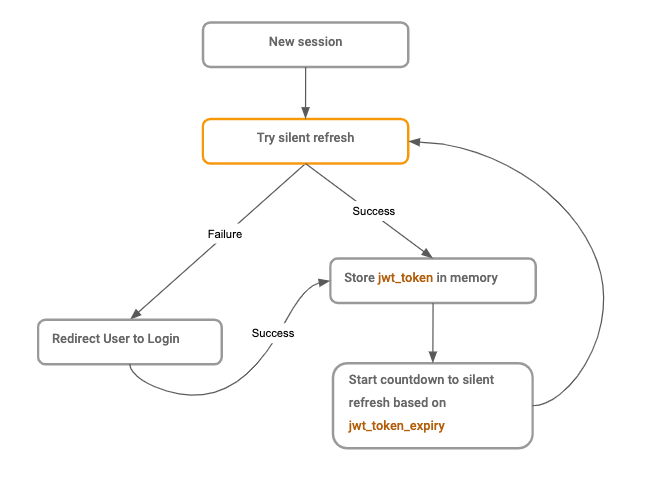
Le app di solito chiedono ai loro utenti se vogliono "rimanere connessi" tra le sessioni o, per impostazione predefinita, mantenere i loro utenti connessi. Questo è ciò che vorremmo implementare anche noi.

Attualmente, non possiamo farlo perché JWT è archiviato solo in memoria e non è persistente. Per ricapitolare, vai a questa sezione sopra per vedere perché non possiamo archiviare JWT nei cookie o nell'archivio locale direttamente.

Quindi, come possiamo continuare le sessioni in modo sicuro, allora?

Token di aggiornamento! Siamo stati in grado di mantenere i token di aggiornamento in modo sicuro e di utilizzarli per l'aggiornamento silenzioso (ovvero il rinnovo dei token JWT a scadenza breve senza chiedere agli utenti di accedere nuovamente). E possiamo anche usarli per recuperare un nuovo token JWT per una nuova sessione! Dai un'occhiata alla sezione precedente per discutere su come persistono i token di aggiornamento.

Diciamo che l'utente si è disconnesso dalla sessione corrente chiudendo la scheda del browser. Ora che l'utente visita di nuovo l'app, vediamo come appare il flusso:



Se vediamo che non abbiamo un JWT in memoria, allora attiviamo il flusso di lavoro di aggiornamento silenzioso

Se il token di aggiornamento è ancora valido (o non è stato revocato), otteniamo un nuovo JWT e siamo pronti per partire!

Possibile caso di errore:

Nel caso in cui il nostro token di aggiornamento scada (supponiamo che l'utente ritorni all'app dopo molto tempo) o venga revocato (a causa di un "logout forzato", ad esempio) il client riceverà un errore 401 per un refresh\_token non autorizzato. Un altro caso potrebbe essere quello di non avere alcun refresh\_token in primo luogo, nel qual caso avremo anche un errore dal nostro endpoint / refresh\_token e reindirizzeremo l'utente alla schermata di accesso.

Ecco un codice di esempio che mostra come gestiremo questa gestione degli errori utilizzando un logoutLink

const logoutLink = onError(({ networkError }) => {

if (networkError.statusCode === 401) logout();

})

Forza il logout, noto anche come logout di tutte le sessioni / dispositivi

Ora che gli utenti sono connessi per sempre e rimangono connessi tra le sessioni, c'è un nuovo problema di cui dobbiamo preoccuparci: forzare il logout o il logout da tutte le sessioni e tutti i dispositivi.

Le implementazioni dei token di aggiornamento delle sezioni precedenti ci mostrano che possiamo persistere nelle sessioni e rimanere connessi.

In questo caso, una semplice implementazione di "force logout" richiede al server di autenticazione di invalidare tutti i token di aggiornamento associati a un determinato utente.

Questa è principalmente un'implementazione sul backend del server di autenticazione e non richiede alcuna gestione speciale sul client. A parte un pulsante "Force Logout" sulla tua app forse :)

Rendering lato server (SSR)

Nel rendering lato server vi sono ulteriori complessità coinvolte nella gestione dei token JWT.

Questo è ciò che vogliamo:

Il browser invia una richiesta a un URL dell'app

Il server SSR esegue il rendering della pagina in base all'identità dell'utente

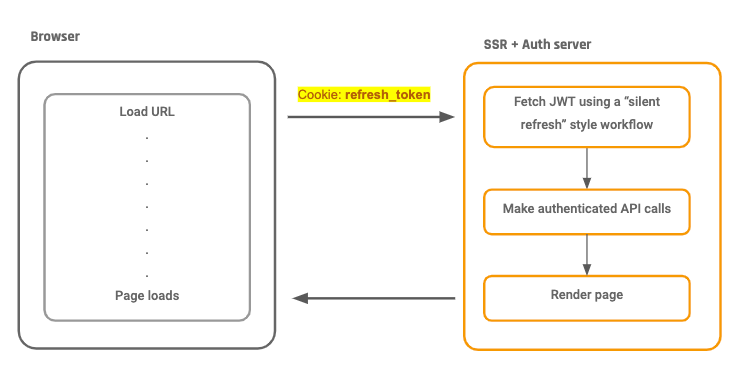
L'utente ottiene la pagina di rendering e quindi continua a utilizzare l'app come SPA (app a pagina singola)

Come fa il server SSR a sapere se l'utente ha effettuato l'accesso?

Il browser deve inviare alcune informazioni sull'identità dell'utente corrente al server SSR. L'unico modo per farlo è tramite un cookie.

Poiché abbiamo già implementato flussi di lavoro di token di aggiornamento tramite cookie, quando facciamo una richiesta al server SSR, dobbiamo assicurarci che anche il token di aggiornamento sia inviato insieme.

Nota: per SSR su pagine autenticate, è fondamentale che il dominio dell'API di autenticazione (e quindi il dominio del cookie refresh\_token) sia lo stesso dominio del server SSR. Altrimenti, i nostri cookie non verranno inviati al server SSR!



Questo è ciò che fa il server SSR:

1. Alla ricezione di una richiesta di rendering di una determinata pagina, il server SSR acquisisce il cookie refresh\_token.
2. Il server SSR utilizza il cookie refresh\_token per ottenere un nuovo JWT per l'utente
3. Il server SSR utilizza il nuovo token JWT ed effettua tutte le richieste GraphQL autenticate per recuperare i dati corretti

L'utente può continuare a fare richieste API autenticate una volta caricata la pagina SSR?

No, non senza un po 'di confusione in giro purtroppo!

Una volta che il server SSR restituisce il codice HTML renderizzato, l'unica identificazione rimasta nel browser sull'identità dell'utente è il vecchio cookie token di aggiornamento che è già stato utilizzato dal server SSR!

Se il nostro codice app tenta di utilizzare questo cookie token di aggiornamento per recuperare un nuovo JWT, questa richiesta fallirà e l'utente verrà disconnesso.

Per risolvere questo, il server SSR dopo aver eseguito il rendering della pagina deve inviare l'ultimo cookie token di aggiornamento, in modo che il browser possa utilizzarlo!

L'intero flusso SSR, end to end:

