## Refresh Tokens in ASP.NET Core Web Api

Una cosa che viene in mente quando si utilizzano i token di accesso per proteggere un'API Web è cosa si fa alla scadenza del token?

Chiedete di nuovo all'utente le credenziali? Questa non è davvero una buona opzione.

Questo post sul blog riguarda l'utilizzo dei token di aggiornamento per risolvere questo problema. In particolare in ASP.NET Core Web Apis con token JWT.

Innanzitutto, è davvero un grosso problema? Perché non abbiamo appena fissato una lunga data di scadenza nei token di accesso? Ad esempio, un mese o addirittura un anno?

Perché se lo facciamo e qualcuno riesce a ottenere quel token, può usarlo per un mese o un anno. Anche se cambi la password.

Questo perché un server si fiderà di un token se la sua firma è valida e l'unico modo per invalidarlo è cambiare la chiave utilizzata per firmarlo e ciò ha la conseguenza di invalidare i token di tutti gli altri.

Questo ci porta all'idea di utilizzare i token di aggiornamento.

Come funziona quindi un token di aggiornamento?

Immagina che quando ottieni un token di accesso ottieni anche un altro token monouso: il token di aggiornamento. L'app memorizza il token di aggiornamento e lo lascia da solo.

Ogni volta che l'app invia una richiesta al server, invia il token di accesso al suo interno (Autorizzazione: Bearer TokenGoesHere) in modo che il server sappia chi sei. Verrà un momento in cui il token scadrà e il server ti informerà in qualche modo.

In questo caso, l'app invia il token scaduto e il token di aggiornamento e ripristina un nuovo token e un token di aggiornamento.

Se succede qualcosa di sospetto, il token di aggiornamento può essere revocato, il che significa che quando l'app tenta di usarlo per ottenere un nuovo token di accesso, tale richiesta verrà rifiutata e l'utente dovrà inserire le credenziali per poter accedere nuovamente.

Per chiarire quest'ultimo punto, immagina che l'app memorizzi la posizione (ad esempio Dublino, Irlanda) della richiesta quando viene creato un token di aggiornamento. Se l'utente può accedere a queste informazioni e se esiste un accesso da un luogo che l'utente non riconosce, l'utente può revocare il token di aggiornamento in modo che quando il token di accesso scade chiunque lo stia utilizzando non sarà in grado di continuare a utilizzare l'applicazione. Questo è il motivo per cui è probabilmente una buona idea avere i token di accesso di breve durata (vale a dire essere validi per un paio di minuti).

Per utilizzare i token di aggiornamento, dobbiamo essere in grado di:

* Crea token di accesso (qui useremo JWT)
* Generare, salvare, recuperare e revocare i token di aggiornamento (lato server)
* Scambia un token JWT scaduto e aggiorna il token con un nuovo token JWT e aggiorna il token (ovvero aggiorna un token JWT)
* Utilizzare il middleware di autenticazione ASP.NET per autenticare un utente con token JWT
* Avere un modo per segnalare che il token di accesso è scaduto all'app (facoltativo)
* Quando il token scade, il client deve acquisire in modo trasparente un nuovo token

Se hai bisogno di informazioni su questi argomenti singolarmente, continua. Se vuoi vedere tutto questo lavorare insieme, puoi trovare [demo project here](https://github.com/ruidfigueiredo/RefreshTokensWebApiExample).

## Create JWT Access Tokens

Se si desidera una descrizione più completa di come utilizzare JWT con ASP.NET Core, lo consiglio [Secure a Web Api in ASP.NET Core](https://www.blinkingcaret.com/2017/09/06/secure-web-api-in-asp-net-core/).

Ecco un riassunto.

Per prima cosa devi aggiungere System.IdentityModel.Tokens.Jwt package:

$ dotnet add package System.IdentityModel.Tokens.Jwt

Per creare un nuovo JWT Token :

private string GenerateToken(IEnumerable<Claim> claims)

{

var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("the server key used to sign the JWT token is here, use more than 16 chars"));

var jwt = new JwtSecurityToken(issuer: "Blinkingcaret",

audience: "Everyone",

claims: claims, //the user's claims, for example new Claim[] { new Claim(ClaimTypes.Name, "The username"), //...

notBefore: DateTime.UtcNow,

expires: DateTime.UtcNow.AddMinutes(5),

signingCredentials: new SigningCredentials(key, SecurityAlgorithms.HmacSha256)

);

return new JwtSecurityTokenHandler().WriteToken(jwt); //the method is called WriteToken but returns a string

}

Qui stiamo creando un nuovo token jwt con una data di scadenza di 5 minuti firmata utilizzando HmacSha256.

## Generate, save, retrieve and revoke refresh tokens

I token di aggiornamento devono essere unici e non dovrebbe essere possibile (o deve essere molto difficile) indovinarli.

Potrebbe sembrare che un semplice GUID soddisfi questi criteri. Purtroppo il processo di generazione [GUIDs is not random](https://blogs.msdn.microsoft.com/oldnewthing/20120523-00/?p=7553).

Ciò significa che, date alcune guide, puoi facilmente indovinare il prossimo.

Per fortuna, c'è un generatore di numeri casuali sicuro in ASP.NET Core che possiamo usare per generare una stringa che è unica e anche se ne sono date alcune, è molto difficile prevedere come sarà la prossima:

using System.Security.Cryptography;

//...

public string GenerateRefreshToken()

{

var randomNumber = new byte[32];

using (var rng = RandomNumberGenerator.Create()){

rng.GetBytes(randomNumber);

return Convert.ToBase64String(randomNumber);

}

}

Qui stiamo generando un numero casuale lungo 32 byte e convertendolo in base64 in modo da poterlo utilizzare come stringa. Non ci sono linee guida per la lunghezza diversa da quella che dovrebbe portare a un token unico e difficile da indovinare. Ne ho scelti 32 ma anche 16 dovrebbero essere ok.

Dovremo generare token di aggiornamento quando generiamo per la prima volta un token JWT e quando "aggiorniamo" un token scaduto.

Ogni volta che generiamo un nuovo token di aggiornamento, dovremmo salvare in modo che sia collegato all'utente per il quale è stato emesso il token di accesso.

La versione più semplice di questo è solo avere una colonna aggiuntiva per il token di aggiornamento nella tabella dell'utente. Ciò ha la conseguenza di consentire all'utente di accedere solo in una posizione (esiste un solo token di aggiornamento valido per utente alla volta).

In alternativa, è possibile mantenere diversi token di aggiornamento per utente e salvare la posizione geografica, l'ora, ecc. Dalla richiesta che li ha originati in modo da poter fornire all'utente rapporti sull'attività.

Inoltre, è probabilmente una buona idea far scadere i token di aggiornamento, ad esempio dopo alcuni giorni (dovresti salvare una data di scadenza insieme al token di aggiornamento).

Una cosa che non devi dimenticare di fare è rimuovere un token di aggiornamento quando viene utilizzato in un'operazione di aggiornamento in modo che non possa essere utilizzato più di una volta.

## Exchange an expired JWT and refresh token for a new JWT token and refresh token (i.e. refresh a JWT token)

Per ottenere un nuovo token di accesso da uno scaduto, dobbiamo essere in grado di accedere ai claims all'interno del token anche se il token è scaduto.

Quando si utilizza il middleware di autenticazione ASP.NET Core per l'autenticazione dell'utente tramite JWT, verrà restituita una risposta 401 a un token scaduto.

Dobbiamo creare un'azione del controller che consenta agli utenti anonimi e che prenda il JWT e aggiorni i token.

Nell'azione del controller è necessario convalidare manualmente il token di accesso scaduto (esiste un'opzione per ignorare la durata del token) ed estrarre tutte le informazioni sull'utente in esso contenute.

Possiamo quindi utilizzare le informazioni dell'utente per recuperare il token di aggiornamento memorizzato. È quindi possibile confrontare il token di aggiornamento archiviato con quello inviato nella richiesta.

Se tutto va bene, creiamo un nuovo JWT e aggiorniamo i token, salviamo il nuovo token di aggiornamento, scartiamo il vecchio e inviamo il nuovo JWT e aggiorniamo i token al client.

Ecco come è possibile recuperare le informazioni dell'utente sotto forma di claims dal token JWT scaduto:

private ClaimsPrincipal GetPrincipalFromExpiredToken(string token)

{

var tokenValidationParameters = new TokenValidationParameters

{

ValidateAudience = false, //you might want to validate the audience and issuer depending on your use case

ValidateIssuer = false,

ValidateIssuerSigningKey = true,

IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("the server key used to sign the JWT token is here, use more than 16 chars")),

ValidateLifetime = false //here we are saying that we don't care about the token's expiration date

};

var tokenHandler = new JwtSecurityTokenHandler();

SecurityToken securityToken;

var principal = tokenHandler.ValidateToken(token, tokenValidationParameters, out securityToken);

var jwtSecurityToken = securityToken as JwtSecurityToken;

if (jwtSecurityToken == null || !jwtSecurityToken.Header.Alg.Equals(SecurityAlgorithms.HmacSha256, StringComparison.InvariantCultureIgnoreCase))

throw new SecurityTokenException("Invalid token");

return principal;

}

Le parti degne di nota nel frammento di cui sopra sono che stiamo usando ValidateLifeTime = false in TokenValidationParameters in modo che il token scaduto sia considerato valido. Inoltre, stiamo verificando che l'algoritmo utilizzato per firmare il token sia quello che ci aspettiamo (HmacSha256 in questo esempio).

La ragione di ciò è che in teoria qualcuno potrebbe creare un token JWT e [set the signing algorithm to “none”](https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/).

Il token JWT sarebbe valido (anche se non firmato). In questo modo usando un token di aggiornamento valido sarebbe possibile scambiare un token falso con un token JWT reale.

Ora abbiamo solo bisogno dell'azione del controller (dovrebbe essere un POST poiché ha effetti collaterali e anche i token sono troppo lunghi per i parametri della stringa di query):

[HttpPost]

public IActionResult Refresh(string token, string refreshToken)

{

var principal = GetPrincipalFromExpiredToken(token);

var username = principal.Identity.Name;

var savedRefreshToken = GetRefreshToken(username); //retrieve the refresh token from a data store

if (savedRefreshToken != refreshToken)

throw new SecurityTokenException("Invalid refresh token");

var newJwtToken = GenerateToken(principal.Claims);

var newRefreshToken = GenerateRefreshToken();

DeleteRefreshToken(username, refreshToken);

SaveRefreshToken(username, newRefreshToken);

return new ObjectResult(new {

token = newJwtToken,

refreshToken = newRefreshToken

});

}

Ci sono alcune ipotesi nel frammento sopra. C'è il recupero, il salvataggio e l'eliminazione che ho omesso, e anche il presupposto che ci sia un solo token di aggiornamento per utente che è lo scenario più semplice.

## Use the ASP.NET Core authentication middleware to authenticate a user using a JWT token

Dobbiamo configurare la pipeline del middleware ASP.NET Core in modo che se una richiesta arriva con un'intestazione Autorizzazione: Bearer JWT\_TOKEN valida l'utente è [“signed in”](https://www.blinkingcaret.com/2016/01/06/what-authenticated-asp-net-mvc/).

Se vuoi una discussione più approfondita su come impostare JWT, in particolare in ASP.NET Core, dai un'occhiata [Secure a Web Api in ASP.NET Core](https://www.blinkingcaret.com/2017/09/06/secure-web-api-in-asp-net-core/).

Dopo la versione 2.0 di ASP.NET Core aggiungiamo un singolo middleware di autenticazione alla pipeline e lo configuriamo in Startup.cs "ConfigureServices:

public void ConfigureServices(IServiceCollection services)

{

services.AddMvc();

//...

services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = "bearer";

options.DefaultChallengeScheme = "bearer";

}).AddJwtBearer("bearer", options =>

{

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateAudience = false,

ValidateIssuer = false,

ValidateIssuerSigningKey = true,

IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("the server key used to sign the JWT token is here, use more than 16 chars")),

ValidateLifetime = true,

ClockSkew = TimeSpan.Zero //the default for this setting is 5 minutes

};

options.Events = new JwtBearerEvents

{

OnAuthenticationFailed = context =>

{

if (context.Exception.GetType() == typeof(SecurityTokenExpiredException))

{

context.Response.Headers.Add("Token-Expired", "true");

}

return Task.CompletedTask;

}

};

});

}

Da notare nello snippet sopra è la gestione dell'evento OnAuthenticationFailed. Aggiungerà un'intestazione Token scaduto alla risposta quando arriva una richiesta con un token scaduto. Il client può utilizzare queste informazioni per decidere di utilizzare il token di aggiornamento. Tuttavia, possiamo solo fare in modo che il client tenti di utilizzare il token di aggiornamento quando ottiene una risposta 401. Faremo affidamento sull'intestazione Token scaduta nella risposta per il resto di questo post del blog.

Ora non ci resta che aggiungere il middleware di autenticazione alla pipeline:

public void Configure(IApplicationBuilder app, IHostingEnvironment env)

{

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseAuthentication();

//...

## Il client

L'obiettivo qui è quello di costruire un client API in grado di realizzare quando un token è scaduto e intraprendere le azioni appropriate per ottenere un nuovo token e fare tutto questo in modo trasparente.

Quando una richiesta non riesce a causa di un token di accesso scaduto, è necessario inviare una nuova richiesta a un endpoint di aggiornamento con i token di accesso e di aggiornamento. Dopo che la richiesta è stata completata e il client ottiene i nuovi token, è necessario ripetere la richiesta originale.

L'implementazione per questo dipenderà dal tipo di client che si sta utilizzando. Qui descriveremo un possibile client JavaScript. Faremo affidamento sulla risposta a una richiesta con un token scaduto con un'intestazione denominata "Token-Scaduto". Utilizzeremo il recupero per eseguire richieste all'API Web.

async function fetchWithCredentials(url, options) {

var jwtToken = getJwtToken();

options = options || {};

options.headers = options.headers || {};

options.headers['Authorization'] = 'Bearer ' + jwtToken;

var response = await fetch(url, options);

if (response.ok) { //all is good, return the response

return response;

}

if (response.status === 401 && response.headers.has('Token-Expired')) {

var refreshToken = getRefreshToken();

var refreshResponse = await refresh(jwtToken, refreshToken);

if (!refreshResponse.ok) {

return response; //failed to refresh so return original 401 response

}

var jsonRefreshResponse = await refreshResponse.json(); //read the json with the new tokens

saveJwtToken(jsonRefreshResponse.token);

saveRefreshToken(jsonRefreshResponse.refreshToken);

return await fetchWithCredentials(url, options); //repeat the original request

} else { //status is not 401 and/or there's no Token-Expired header

return response; //return the original 401 response

}

}

Nello snippet sopra c'è getJwtToken, getRefreshToken, saveJwtToken e saveRefreshToken. In un browser, questi utilizzerebbero l'archivio locale del browser per salvare e recuperare i token, ad esempio:

function getJwtToken() {

return localStorage.getItem('token');

}

function getRefreshToken() {

return localStorage.getItem('refreshToken');

}

function saveJwtToken(token) {

localStorage.setItem('token', token);

}

function saveRefreshToken(refreshToken) {

localStorage.setItem('refreshToken', refreshToken);

}

C'è anche la funzione di aggiornamento. Questa funzione esegue una richiesta POST all'endpoint api per l'aggiornamento dei token, ad esempio se tale endpoint è in / token / refresh:

async function refresh(jwtToken, refreshToken) {

return fetch('token/refresh', {

method: 'POST',

body: `token=${encodeURIComponent(jwtToken)}&refreshToken=${encodeURIComponent(getRefreshToken())}`,

headers: {

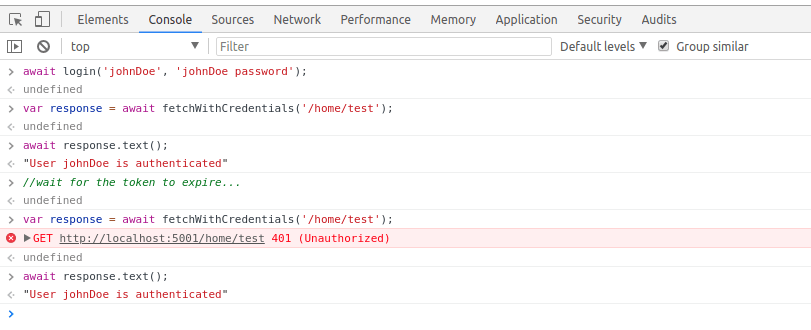
'Content-Type': 'application/x-www-form-urlencoded'

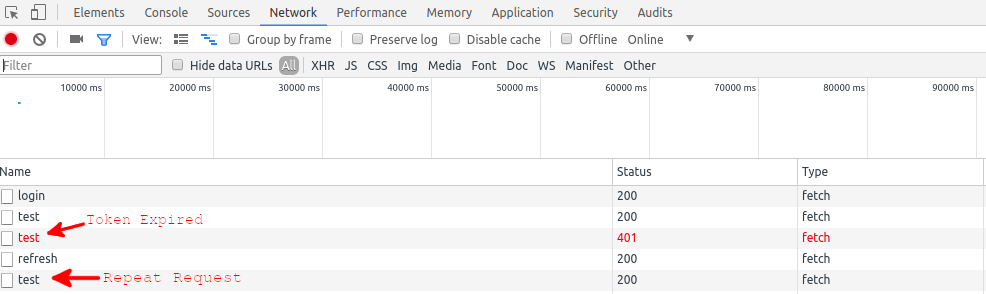
}

});

}

Ecco come apparirebbe questo client se lo provassi nella console degli strumenti di sviluppo di Chrome:





Una cosa da notare qui è il 401 che appare in rosso nella console. Ciò accade quando la richiesta non riesce a causa del token scaduto.

Se vedere "errore" (errore tra virgolette perché è un codice di stato valido e appropriato in questo caso) è qualcosa che vorresti evitare, puoi accedere alla data di scadenza del token in JavaScript e aggiornarlo prima che scada.

Un token JWT ha 3 parti separate da un ".". La seconda parte contiene i claims dell'utente e al suo interno c'è un claim chiamato exp che contiene il timestamp unix di quando il token scade. Ecco come ottenere un oggetto data JavaScript con la data di scadenza per un token JWT:

var claims = JSON.parse(atob(token.split('.')[1]));

var expirationDate = new Date(claims.exp\*1000); //unix timestamp is in seconds, javascript in milliseconds

Controllare la data di scadenza sembra complicato, quindi non consiglio di farlo (anche occuparsi di fusi orari è probabilmente un problema). Ho deciso di menzionarlo perché è qualcosa di interessante da sapere e ciò rende molto chiaro che ciò che si inserisce in un token JWT non è segreto, non può essere manomesso senza rendere invalido il token.