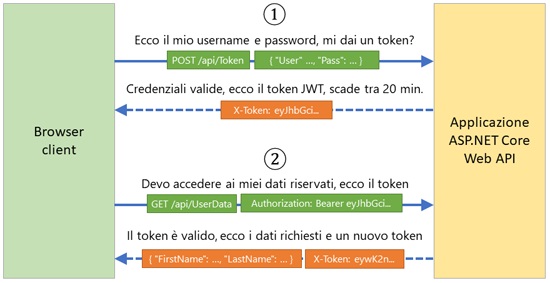
Autenticazione con JWT Token e ASP.NET Core Web API

Quando creiamo delle Single-Page Application o delle app mobile con Xamarin o Cordova, si pone sempre il problema di come far **autenticare gli utenti**. Usare *OAuth* è sicuramente fattibile ma ricorrere a questo protocollo potrebbe essere superfluo se il progetto è composto dalla sola applicazione client e dal backend Web API che mantiene i dati degli utenti in un proprio database locale. Se non necessitiamo di un identity provider esterno (es. Facebook, IdentityServer, ecc...), allora l'alternativa meno complessa consiste nel creare semplici **Token JWT** che verranno scambiati tra client e server.  
  
**JWT (JSON Web Token)** è uno standard aperto che definisce le regole per creare un token di accesso, ovvero una **stringa molto concisa**, di qualche centinaio di byte, che il server trasmette al client affinché possa provare la sua identità in ogni successiva richiesta HTTP. Ogni token è auto-contenuto, ossia **include i claim dell'utente** ed è **firmato digitalmente** con una chiave segreta, in modo che il client non possa alterarlo senza comprometterne l'integrità.  
  
Un token JWT, proprio come un cookie, serve ad evitare che il client debba fornire username e password ad ogni richiesta. A differenza dei cookie, però, sono ideali per essere usati con Web API: un token JWT, infatti, può essere facilmente inviato tale e quale via query string, in un'intestazione o nel corpo della richiesta HTTP, a discrezione dello sviluppatore.  
  
Una descrizione tecnica approfondita della struttura e delle peculiarità dei token JWT si trova nel sito di riferimento: [https://jwt.io](https://jwt.io/)  
  
L'interazione tra le parti si spiega facilmente: il client invia una prima richiesta fornendo le proprie credenziali per ottenere un token JWT. Il token viene creato dal server e consegnato al client per mezzo di un'intestazione della risposta. Il client quindi lo memorizza in una variabile o nello storage del dispositivo e lo restituisce al server come intestazione *Authorization* in tutte le successive richieste, specificando *Bearer* come *scheme* di autenticazione.



Il token JWT viene creato con una **data di scadenza** che può essere più o meno lunga, a seconda degli scenari di utilizzo. Se vogliamo garantire un buon livello di sicurezza possiamo creare **token short-lived**, ovvero con una scadenza molto breve (ad esempio di 20 minuti). Fintanto che il client invia richieste ad una qualsiasi action della Web API, il server potrà restituirgli nuovi token con scadenza rinnovata, permettendogli di continuare indefinitamente senza dover reinserire le credenziali. Per contro, username e password dovranno essere reinserite se l'utente resta inattivo e lascia scadere il token.  
  
Vediamo come realizzare quanto appena descritto con ASP.NET Core. Iniziamo creando un *TokenController* che permetterà al client di inviare le credenziali.

[Route("api/[controller]")]

public class TokenController : Controller

{

*// POST api/Token*

[HttpPost]

public IActionResult GetToken([FromBody] TokenRequest tokenRequest)

{

*//TokenRequest è una nostra classe contenente le proprietà Username e Password*

*//Avvisiamo il client se non ha fornito tali valori*

if(!ModelState.IsValid) {

return BadRequest();

}

*//Lo avvisiamo anche se non ha fornito credenziali valide*

if (!VerifyCredentials(tokenRequest.Username, tokenRequest.Password)) {

return Unauthorized();

}

*//Ok, l'utente ha fornito credenziali valide, creiamogli una ClaimsIdentity*

var identity = new ClaimsIdentity(JwtBearerDefaults.AuthenticationScheme);

*//Aggiungiamo uno o più claim relativi all'utente loggato*

identity.AddClaim(new Claim(ClaimTypes.Name, tokenRequest.Username));

*//Incapsuliamo l'identità in una ClaimsPrincipal l'associamo alla richiesta corrente*

HttpContext.User = new ClaimsPrincipal(identity);

*//Non è necessario creare il token qui, lo possiamo creare da un middleware*

return NoContent();

}

private bool VerifyCredentials(string username, string password) {

*//TODO: Modificare questa implementazione, che è puramente dimostrativa*

return username == "Admin" && password == "Password";

}

}

Anziché far creare il token JWT dal *TokenController*, scriviamo un middleware che si occuperà specificatamente di questo compito. Infatti, dal momento che i middleware di ASP.NET Core vengono eseguiti a ogni richiesta, esso potrà generare nuovi token JWT indipendentemente dall'action di ASP.NET Core Web API invocata dal client.

public class JwtTokenMiddleware {

private readonly RequestDelegate next;

public JwtTokenMiddleware(RequestDelegate next)

{

this.next = next;

}

public async Task Invoke(HttpContext context) {

context.Response.OnStarting(() => {

var identity = context.User.Identity as ClaimsIdentity;

*//Se la richiesta era autenticata, allora creiamo un nuovo token JWT*

if (identity.IsAuthenticated) {

*//Il client potrà usare questo nuovo token nella sua prossima richiesta*

var token = CreateTokenForIdentity(identity);

*//Usiamo l'intestazione X-Token, ma non è obbligatorio che si chiami così*

context.Response.Headers.Add("X-Token", token);

}

return Task.CompletedTask;

});

await next.Invoke(context);

}

*//In questo metodo creiamo il token a partire dai claim della ClaimsIdentity*

private StringValues CreateTokenForIdentity(ClaimsIdentity identity)

{

var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes("MiaChiaveSegreta"));

var credentials = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

var token = new JwtSecurityToken(

issuer: "Issuer",

audience: "Audience",

claims: identity.Claims,

expires: DateTime.Now.AddMinutes(20),

signingCredentials: credentials

);

var tokenHandler = new JwtSecurityTokenHandler();

var serializedToken = tokenHandler.WriteToken(token);

return serializedToken;

}

}

Come si nota, la creazione del token ha richiesto solo poche righe di codice grazie alla classe **JwtSecurityTokenHandler** che prepara il contenuto in base ai claim trovati nella *ClaimsIdentity* e genera la firma digitale inclusa nel token.  
  
Ora registriamo il middleware dal metodo *Configure* della classe *Startup*. Inoltre, registriamo anche il middleware di autenticazione di ASP.NET Core che avrà il compito di esaminare la richiesta per determinare se il token fornito è integro e valido.

app.UseMiddleware<JwtTokenMiddleware>();

app.UseAuthentication();

//Qui registriamo altri middleware (es. Mvc)

Non resta che configurare il middleware di autenticazione di ASP.NET Core in modo che supporti i token JWT. Nel metodo *ConfigureServices* della classe *Startup*, aggiungiamo il seguente codice.

services.AddAuthentication(options => {

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

}).AddJwtBearer(options => {

options.TokenValidationParameters = new TokenValidationParameters{

ValidateIssuer = true,

ValidateAudience = true,

ValidateIssuerSigningKey = true,

*//Importante: indicare lo stesso Issuer, Audience e chiave segreta*

*//usati anche nel JwtTokenMiddleware*

ValidIssuer = "Issuer",

ValidAudience = "Audience",

IssuerSigningKey = new SymmetricSecurityKey(

Encoding.UTF8.GetBytes("MiaChiaveSegreta")

),

*//Tolleranza sulla data di scadenza del token*

ClockSkew = TimeSpan.Zero

};

});

Un'applicazione dimostrativa è disponibile su GitHub al seguente indirizzo: <https://github.com/BrightSoul/AspNetCoreWebApiJwtTokenDemo>