



[Home](#) > [.NET](#) > [.NET Development](#) > [Using Auth Cookies in ASP.NET Core](#)



[Camilo Reyes](#)

11 February 2019

88314 views

39

10

Using Auth Cookies in ASP.NET Core

Using cookie authorization in ASP.NET Core is seamless and flexible. In this article, Camilo Reyes explains why this might be a good choice for your next project and how to use the many options available.

Cookie-based authentication is the popular choice to secure customer facing web apps. For .NET programmers, ASP.NET Core has a good approach that is worth looking into. In this take, I will delve deep into the auth cookie using ASP.NET Core 2.1. Version 2.1 is the latest LTS version as of the time of this writing. So, feel free to follow along, I'll assume you're on Visual Studio or have enough C# chops to use a text editor. I will omit namespaces and using statements to keep code samples focused. If you get stuck, download the sample code found at the end.

With ASP.NET 2.1, you can use cookie-based authentication out of the box. There is no need for additional NuGet packages. New projects include a metapackage that has everything, which is *Microsoft.AspNetCore.App*. To follow along, type `dotnet new mvc` in a CLI or do *File > New Project* in Visual Studio.

For those of you who come from classic .NET, you may be aware of the *OWIN* auth cookie. You will be happy to know those same skills transfer over to ASP.NET Core quite well. The two implementations remain somewhat similar. With ASP.NET Core, you still configure the auth cookie, set up middleware, and set identity claims.

Setup

To begin, I'll assume you know enough about the ASP.NET MVC framework to gut the scaffolding into a skeleton web app. You need a *HomeController* with an `Index`, `Login`, `Logout`, and `Revoke` action methods. `Login` will redirect to `Index` after it signs you in, so it doesn't need a view. I'll omit showing view sample code since views are not the focus here. If you get lost, be sure to download the entire demo to play with it.

I'll use debug logs to show critical events inside the cookie authentication. Be sure to enable debug logs in *appsettings.json* and disable Microsoft and system logs.

My log setup looks like this:

```
"LogLevel": {  
  "Default": "Debug",  
  "System": "Warning",  
  "Microsoft": "Warning"  
}
```

Now you're ready to build a basic app with cookie authentication. I'll forgo HTML forms with a user name and password input fields. These front-end concerns only add clutter to what is more important which is the auth cookie. Starting with a skeleton app shows how effective it is to add an auth cookie from scratch. The app will sign you in automatically and land on an `Index` page with an auth cookie. Then, you can log out or revoke user access. I want you to pay attention to what happens to the auth cookie as I put authentication in place.

Cookie Options

Begin by configuring auth cookie options through middleware inside the **Startup** class. Cookie options tell the authentication middleware how the cookie behaves in the browser. There are many options, but I will only focus on those that affect cookie security the most.

- **HttpOnly:** A flag that says the cookie is only available to servers. The browser only sends the cookie but cannot access it through JavaScript.
- **SecurePolicy:** This limits the cookie to HTTPS. I recommend setting this to `Always` in prod. Leave it set to `None` in local.
- **SameSite:** Indicates whether the browser can use the cookie with cross-site requests. For OAuth authentication, set this to `Lax`. I am setting this to `Strict` because the auth cookie is only for a single site. Setting this to `None` does not set a cookie header value.

There are cookie options for both the auth cookie and a global cookie policy. Stay alert since the cookie policy can override auth cookie options and vice versa. Setting `HttpOnly` to `false` in the auth cookie will override cookie policy options. While setting `SameSite` in the cookie policy overrides auth cookie options. In my demo, I'll illustrate both scenarios, so it is crystal clear how this works.

In the `Startup` class, find the `ConfigureServices` method and type:

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.Cookie.HttpOnly = true;
        options.Cookie.SecurePolicy = _environment.IsDevelopment()
            ? CookieSecurePolicy.None : CookieSecurePolicy.Always;
        options.Cookie.SameSite = SameSiteMode.Lax;
    });
```

This creates the middleware service with the `AddAuthentication` and `AddCookie` methods. `AuthenticationScheme` is useful when there is more than one auth cookie. Many instances of the cookie authentication let you protect endpoints with many schemes. You supply any string value; the default is set to `Cookies`. Note that the **options** object is an instance of the `CookieAuthenticationOptions` class.

The `SecurePolicy` is set through a ternary operator that comes from `_environment`. This is a private property that gets set in the `Startup` constructor. Add `IHostingEnvironment` as a parameter and let dependency injection do the rest.

In this same `ConfigureServices` method, add the global cookie policy through middleware:

```
services.Configure<CookiePolicyOptions>(options =>
{
    options.MinimumSameSitePolicy = SameSiteMode.Strict;
    options.HttpOnly = HttpOnlyPolicy.None;
    options.Secure = _environment.IsDevelopment()
        ? CookieSecurePolicy.None : CookieSecurePolicy.Always;
});
```

Take a good look at `SameSite` and `HttpOnly` settings for both cookie options. When I set the auth cookie, you will see this set to `HttpOnly` and `Strict`. This illustrates how both options override each other.

Invoke this middleware inside the request pipeline in the `Configure` method:

```
app.UseCookiePolicy();  
app.UseAuthentication();
```

The cookie policy middleware is order sensitive. This means it only affects components after invocation. By invoking the authentication middleware, you will get a `HttpContext.User` property. Be sure to call this `UseAuthentication` method before calling `UseMvc`.

Login

In the `HomeController` add an `AllowAnonymous` filter to the Login and Logout methods. There are only two action methods available without an auth cookie.

Inside the `Startup` class, look for the `AddMvc` extension method and add a global auth filter:

```
services.AddMvc(options => options.Filters.Add(new AuthorizeFilter(
```

With the app secure, configure the cookie name and login / logout paths. Find where the rest of the `CookieAuthenticationOptions` are and do:

```
options.Cookie.Name = "SimpleTalk.AuthCookieAspNetCore";  
options.LoginPath = "/Home/Login";  
options.LogoutPath = "/Home/Logout";
```

This will cause the app to redirect to the login endpoint to sign in. However, before you can take this for a spin, you'll need to create the auth cookie.

Do this inside the `Login` action method in the `HomeController`:

```
var claims = new List<Claim>  
{  
    new Claim(ClaimTypes.Name, Guid.NewGuid().ToString())  
};  
  
var claimsIdentity = new ClaimsIdentity(  
    claims, CookieAuthenticationDefaults.AuthenticationScheme);  
var authProperties = new AuthenticationProperties();  
  
await HttpContext.SignInAsync(  
    CookieAuthenticationDefaults.AuthenticationScheme,
```

```
new ClaimsPrincipal(claimsIdentity),
authProperties);
```

`AuthenticationProperties` drive further auth cookie behavior in the browser. For example, the `IsPersistent` property persists the cookie across browser sessions. Be sure to get explicit user consent when you enable this property. `ExpiresUtc` sets an absolute expiration, be sure to enable `IsPersistent` and set it to true. The default values will give you a session cookie that goes away when you close the tab or browser window. I find the default values in this object enough for most use cases.

To take this for a spin load up the browser by going to the home or Index page. Note it redirects to Login which redirects back to Index with an auth cookie.

Once this loads it looks something like this. Be sure to take a good look at how the auth cookie is set:

The screenshot shows a web browser window with the address bar at `localhost:5000`. The page content includes a 'Welcome!' message, 'You are logged in.', links for 'Revoke' and 'Sign Out', and a copyright notice '© 2019 - Auth Cookie in ASP.NET Core'. Below the page content, a browser's developer tools 'Storage' tab is open, displaying a table of cookies. The table has columns for Name, Domain, Expires on, Value, HttpOnly, and sameSite. One cookie is listed: 'SimpleTalk.Auth...' with domain 'localhost', expires on 'Session', and a value starting with 'CfDJ8PGIiGy0m...'. The 'HttpOnly' property is set to 'true' and 'sameSite' is set to 'Strict'.

Name	Domain	Expires on	Value	HttpOnly	sameSite
SimpleTalk.Auth...	localhost	Session	CfDJ8PGIiGy0m...	true	Strict

JWT Identity Claim

Often, an auth cookie isn't enough to secure API endpoints or microservices. For the web app to call a service, it can use a JWT bearer token to authenticate. To make the access token accessible, place it inside the identity claims.

In the Login action method within `HomeController`, expand the list of claims with a JWT:

```
var userId = Guid.NewGuid().ToString();
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, userId),
    new Claim("access_token", GetAccessToken(userId))
};

private static string GetAccessToken(string userId)
{
    const string issuer = "localhost";
    const string audience = "localhost";

    var identity = new ClaimsIdentity(new List<Claim>
    {
        new Claim("sub", userId)
    });

    var bytes = Encoding.UTF8.GetBytes(userId);
    var key = new SymmetricSecurityKey(bytes);
    var signingCredentials = new SigningCredentials(
        key, SecurityAlgorithms.HmacSha256);

    var now = DateTime.UtcNow;
    var handler = new JwtSecurityTokenHandler();

    var token = handler.CreateJwtSecurityToken(
        issuer, audience, identity,
        now, now.Add(TimeSpan.FromHours(1)),
        now, signingCredentials);

    return handler.WriteToken(token);
}
```

I must caution, don't ever do this in production. Here, I use the user id as the signing key which is symmetric to keep it simple. In a prod environment use an asymmetric signing key with public and private keys. Client apps will then use a well-known configuration endpoint to validate the JWT.

Placing the JWT in `ClaimsIdentity` makes it accessible through the `HttpContext.User` property. For this app, say you want to put the JWT in a debug log to show off this fancy access token.

In the `Startup` class, create this middleware inside the `Configure` method:

```
app.Use(async (context, next) =>
{
    var principal = context.User as ClaimsPrincipal;
    var accessToken = principal?.Claims
        .FirstOrDefault(c => c.Type == "access_token");

    if (accessToken != null)
    {
        _logger.LogDebug(accessToken.Value);
    }

    await next();
});
```

The `_logger` is another private property you set through the constructor. Add `ILogger<Startup>` as a parameter and let dependency injection do the rest. Note the `ClaimsPrincipal` has a list of `Claims` you can iterate through. What I find useful is to look for a `Type` of claim like an `access_token` and get a `Value`. Because this is middleware always call `next()` so it doesn't block the request pipeline.

Logout

To log out of the web app and clear the auth cookie do:

```
await HttpContext.SignOutAsync(
    CookieAuthenticationDefaults.AuthenticationScheme);
```

This belongs inside the `Logout` action method in the `HomeController`. Note that you specify the authentication scheme. This tells the sign-out method which auth cookie it needs to blot out. Inspecting HTTP response headers reveals `Cache-Control` and `Pragma` headers set to `no-cache`. This shows the auth cookie disables browser caching when it wants to update the cookie. The `Login` action method responds with the same HTTP headers.

Revocation

There are use cases where the app needs to react to back-end user access changes. The auth cookie will secure the application, but, remains valid for the lifetime of the cookie. With a valid cookie, the end-user will not see any changes until they log out or the cookie expires. In ASP.NET Core 2.1, one way to validate changes is through cookie authentication events. The

validation event can do back-end lookups from identity claims in the auth cookie. Create the event by extending `CookieAuthenticationEvents`. Override the `ValidatePrincipal` method and set the event in the auth cookie options.

For example:

```
public class RevokeAuthenticationEvents : CookieAuthenticationEvents

    private readonly IMemoryCache _cache;
    private readonly ILogger _logger;

    public RevokeAuthenticationEvents(
        IMemoryCache cache,
        ILogger<RevokeAuthenticationEvents> logger)
    {
        _cache = cache;
        _logger = logger;
    }

    public override Task ValidatePrincipal(
        CookieValidatePrincipalContext context)
    {
        var userId = context.Principal.Claims
            .First(c => c.Type == ClaimTypes.Name);

        if (_cache.Get<bool>("revoke-" + userId.Value))
        {
            context.RejectPrincipal();

            _cache.Remove("revoke-" + userId.Value);
            _logger.LogDebug("Access has been revoked for: "
                + userId.Value + ".");
        }

        return Task.CompletedTask;
    }
}
```

To have `IMemoryCache` set by dependency injection, put `AddMemoryCache` inside the `ConfigureServices` method in the `Startup` class. Calling `RejectPrincipal` has an immediate effect and kicks you back out to Login to get a new auth cookie. Note this relies on in-memory persistence which gets set in the Revoke action method. Keep in mind that this event runs once per every request, so you want to use an efficient caching strategy. Doing an expensive lookup at every request will affect performance.

Revoke access by setting the cache inside the `Revoke` method in the `HomeController`:


```
var principal = HttpContext.User as ClaimsPrincipal;
var userId = principal?.Claims
    .First(c => c.Type == ClaimTypes.Name);

_cache.Set("revoke-" + userId.Value, true);
return View();
```

After visiting the `Revoke` endpoint, the change does not have an immediate effect. After revocation, navigating home will show the debug log and redirect to Login. Note that landing in the Index page again will have a brand new auth cookie.

To register this event, be sure to set the `EventsType` in the `CookieAuthenticationOptions`. You will need to provide a scoped service to register this `RevokeAuthenticationEvents`. Both are set inside the `ConfigureServices` method in the `Startup` class.

For example:

```
options.EventsType = typeof(RevokeAuthenticationEvents);
services.AddScoped<RevokeAuthenticationEvents>();
```

The `CookieValidatePrincipalContext` in `ValidatePrincipal` can do more than revocation if necessary. This context has `ReplacePrincipal` to update the principal, then renew the cookie by setting `ShouldRenew` to `true`.

Session Store

Setting a JWT in the claims to have a convenient way to access identity data works well. However, every identity claim you put in the principal ends up in the auth cookie. If you inspect the cookie, you will notice it doubles in size with an access token. As you add more claims in the principal the auth cookie gets bigger. You may hit HTTP header limits in a prod environment with many auth cookies. In IIS, the default max limit is set to 8KB-16KB depending on the version. You can increase the limit, but this means bigger payloads per request because of the cookies.

There are many ways to quell this problem, like a user session to keep all JWTs out of the auth cookie. If you have current code that accesses the identity through the principal, then this is not ideal. Moving identity data out of the principal is risky because it may lead to a complete rewrite.

One alternative is to use the `SessionStore` found in `CookieAuthenticationOptions`. OWIN, for example, has a similar property. Implement the `ITicketStore` interface and find a way to persist data in the back-end. Setting the `SessionStore` property defines a container to store the identity across requests. Only a session identifier gets sent to the browser in the auth cookie.

Say you want to use in-memory persistence instead of the auth cookie:

```
public class InMemoryTicketStore : ITicketStore
{
    private readonly IMemoryCache _cache;

    public InMemoryTicketStore(IMemoryCache cache)
    {
        _cache = cache;
    }

    public Task RemoveAsync(string key)
    {
        _cache.Remove(key);

        return Task.CompletedTask;
    }

    public Task<AuthenticationTicket> RetrieveAsync(string key)
    {
        var ticket = _cache.Get<AuthenticationTicket>(key);

        return Task.FromResult(ticket);
    }

    public Task RenewAsync(string key, AuthenticationTicket ticket)
    {
        _cache.Set(key, ticket);

        return Task.CompletedTask;
    }

    public Task<string> StoreAsync(AuthenticationTicket ticket)
    {
        var key = ticket.Principal.Claims
            .First(c => c.Type == ClaimTypes.Name).Value;

        _cache.Set(key, ticket);

        return Task.FromResult(key);
    }
}
```

Set an instance of this class in `SessionStore` inside `CookieAuthenticationOptions`, these options are set in the `ConfigureServices` method in the `Startup` class. One caveat is getting an instance since it needs a provider from `BuildServiceProvider`. A temporary IoC container here feels hacky and pines after a better solution.

A better approach is to use the options pattern in ASP.NET Core. Post-configuration scenarios set or change options at startup. With this solution, you leverage dependency injection without reinventing the wheel.

To put in place this options pattern, implement

`IPostConfigureOptions<CookieAuthenticationOptions>`:

```
public class ConfigureCookieAuthenticationOptions
    : IPostConfigureOptions<CookieAuthenticationOptions>
{
    private readonly ITicketStore _ticketStore;

    public ConfigureCookieAuthenticationOptions(ITicketStore ticketStore)
    {
        _ticketStore = ticketStore;
    }

    public void PostConfigure(string name,
        CookieAuthenticationOptions options)
    {
        options.SessionStore = _ticketStore;
    }
}
```

To register both `InMemoryTicketStore` and

`ConfigureCookieAuthenticationOptions`, place this in `ConfigureServices`:

```
services.AddTransient<ITicketStore, InMemoryTicketStore>();
services.AddSingleton<IPostConfigureOptions<CookieAuthenticationOptions>,
    ConfigureCookieAuthenticationOptions>();
```

Make sure you make this change in the `Startup` class. If you peek inside

`Configure<CookiePolicyOptions>`, for example, and crack open the code. Note the pattern; configuration runs through a singleton object and a lambda expression. ASP.NET Core uses this same options pattern under the hood. Now, firing this up in the browser and inspecting the auth cookie will have a much smaller footprint.

Conclusion

Implementing an auth cookie is seamless in ASP.NET Core 2.1. You configure cookie options, invoke middleware, and set identity claims. Sign in and sign out methods work based on an authentication scheme. Auth cookie options allow the app to react to back-end events and set a session store. The auth cookie is flexible enough to work well with any enterprise solution.