

Use cookie authentication without ASP.NET Core Identity

02/11/2020 • 14 minutes to read • R  +10

In this article

[Configuration](#)

[Cookie Policy Middleware](#)

[Create an authentication cookie](#)

[Sign out](#)

[React to back-end changes](#)

[Persistent cookies](#)

[Absolute cookie expiration](#)

[Additional resources](#)

By [Rick Anderson](#)

ASP.NET Core Identity is a complete, full-featured authentication provider for creating and maintaining logins. However, a cookie-based authentication provider without ASP.NET Core Identity can be used. For more information, see [Introduction to Identity on ASP.NET Core](#).

[View or download sample code \(how to download\)](#)

For demonstration purposes in the sample app, the user account for the hypothetical user, Maria Rodriguez, is hardcoded into the app. Use the **Email** address

`maria.rodriquez@contoso.com` and any password to sign in the user. The user is authenticated in the `AuthenticateUser` method in the `Pages/Account/Login.cshtml.cs` file. In a real-world example, the user would be authenticated against a database.

Configuration

In the `Startup.ConfigureServices` method, create the Authentication Middleware services with the [AddAuthentication](#) and [AddCookie](#) methods:

C#

 Copy

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie();
```

[AuthenticationScheme](#) passed to `AddAuthentication` sets the default authentication scheme for the app. `AuthenticationScheme` is useful when there are multiple instances of cookie authentication and you want to [authorize with a specific scheme](#). Setting the `AuthenticationScheme` to `CookieAuthenticationDefaults.AuthenticationScheme` provides a value of "Cookies" for the scheme. You can supply any string value that distinguishes the scheme.

The app's authentication scheme is different from the app's cookie authentication scheme. When a cookie authentication scheme isn't provided to [AddCookie](#), it uses `CookieAuthenticationDefaults.AuthenticationScheme` ("Cookies").

The authentication cookie's [IsEssential](#) property is set to `true` by default. Authentication cookies are allowed when a site visitor hasn't consented to data collection. For more information, see [General Data Protection Regulation \(GDPR\) support in ASP.NET Core](#).

In `Startup.Configure`, call `UseAuthentication` and `UseAuthorization` to set the `HttpContext.User` property and run Authorization Middleware for requests. Call the `UseAuthentication` and `UseAuthorization` methods before calling `UseEndpoints`:

C#

 Copy

```
app.UseAuthentication();
app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapControllers();
    endpoints.MapRazorPages();
});
```

The [CookieAuthenticationOptions](#) class is used to configure the authentication provider options.

Set `CookieAuthenticationOptions` in the service configuration for authentication in the `Startup.ConfigureServices` method:

C#

 Copy

```
services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        ...
    });
```

Cookie Policy Middleware

[Cookie Policy Middleware](#) enables cookie policy capabilities. Adding the middleware to the app processing pipeline is order sensitive—it only affects downstream components registered in the pipeline.

C#

 Copy

```
app.UseCookiePolicy(cookiePolicyOptions);
```

Use [CookiePolicyOptions](#) provided to the Cookie Policy Middleware to control global characteristics of cookie processing and hook into cookie processing handlers when cookies are appended or deleted.

The default [MinimumSameSitePolicy](#) value is `SameSiteMode.Lax` to permit OAuth2 authentication. To strictly enforce a same-site policy of `SameSiteMode.Strict`, set the `MinimumSameSitePolicy`. Although this setting breaks OAuth2 and other cross-origin authentication schemes, it elevates the level of cookie security for other types of apps that don't rely on cross-origin request processing.

C#

 Copy

```
var cookiePolicyOptions = new CookiePolicyOptions
{
    MinimumSameSitePolicy = SameSiteMode.Strict,
};
```

The Cookie Policy Middleware setting for `MinimumSameSitePolicy` can affect the setting of `Cookie.SameSite` in `CookieAuthenticationOptions` settings according to the matrix below.


MinimumSameSitePolicy	Cookie.SameSite	Resultant Cookie.SameSite setting

MinimumSameSitePolicy	Cookie.SameSite	Resultant Cookie.SameSite setting
SameSiteMode.None	SameSiteMode.None	SameSiteMode.None
	SameSiteMode.Lax	SameSiteMode.Lax
	SameSiteMode.Strict	SameSiteMode.Strict
SameSiteMode.Lax	SameSiteMode.None	SameSiteMode.Lax
	SameSiteMode.Lax	SameSiteMode.Lax
	SameSiteMode.Strict	SameSiteMode.Strict
SameSiteMode.Strict	SameSiteMode.None	SameSiteMode.Strict
	SameSiteMode.Lax	SameSiteMode.Strict
	SameSiteMode.Strict	SameSiteMode.Strict

Create an authentication cookie

To create a cookie holding user information, construct a [ClaimsPrincipal](#). The user information is serialized and stored in the cookie.

Create a [ClaimsIdentity](#) with any required [Claims](#) and call [SignInAsync](#) to sign in the user:

C# 

```

var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Email),
    new Claim("FullName", user.FullName),
    new Claim(ClaimTypes.Role, "Administrator"),
};

var claimsIdentity = new ClaimsIdentity(
    claims, CookieAuthenticationDefaults.AuthenticationScheme);

var authProperties = new AuthenticationProperties
{
    //AllowRefresh = <bool>,
    // Refreshing the authentication session should be allowed.

    //ExpiresUtc = DateTimeOffset.UtcNow.AddMinutes(10),
    // The time at which the authentication ticket expires. A
    // value set here overrides the ExpireTimeSpan option of
    // CookieAuthenticationOptions set with AddCookie.

    //IsPersistent = true,
    // Whether the authentication session is persisted across

```

```
// multiple requests. When used with cookies, controls
// whether the cookie's lifetime is absolute (matching the
// lifetime of the authentication ticket) or session-based.

//IssuedUtc = <DateTimeOffset>,
// The time at which the authentication ticket was issued.

//RedirectUri = <string>
// The full path or absolute URI to be used as an http
// redirect response value.
};

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    authProperties);
```

If you would like to see code comments translated to languages other than English, let us know in [this GitHub discussion issue](#).

`SignInAsync` creates an encrypted cookie and adds it to the current response. If `AuthenticationScheme` isn't specified, the default scheme is used.

`RedirectUri` is only used on a few specific paths by default, for example, the login path and logout paths. For more information see the [CookieAuthenticationHandler source](#).

ASP.NET Core's [Data Protection](#) system is used for encryption. For an app hosted on multiple machines, load balancing across apps, or using a web farm, [configure data protection](#) to use the same key ring and app identifier.

Sign out

To sign out the current user and delete their cookie, call [SignOutAsync](#):

C#	 Copy
<pre>await HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme);</pre>	

If `CookieAuthenticationDefaults.AuthenticationScheme` (or "Cookies") isn't used as the scheme (for example, "ContosoCookie"), supply the scheme used when configuring the authentication provider. Otherwise, the default scheme is used.

When the browser closes it automatically deletes session based cookies (non-persistent cookies), but no cookies are cleared when an individual tab is closed. The server is not notified of tab or browser close events.

React to back-end changes

Once a cookie is created, the cookie is the single source of identity. If a user account is disabled in back-end systems:

- The app's cookie authentication system continues to process requests based on the authentication cookie.
- The user remains signed into the app as long as the authentication cookie is valid.

The [ValidatePrincipal](#) event can be used to intercept and override validation of the cookie identity. Validating the cookie on every request mitigates the risk of revoked users accessing the app.

One approach to cookie validation is based on keeping track of when the user database changes. If the database hasn't been changed since the user's cookie was issued, there's no need to re-authenticate the user if their cookie is still valid. In the sample app, the database is implemented in `IUserRepository` and stores a `LastChanged` value. When a user is updated in the database, the `LastChanged` value is set to the current time.

In order to invalidate a cookie when the database changes based on the `LastChanged` value, create the cookie with a `LastChanged` claim containing the current `LastChanged` value from the database:

C#


 Copy

```
var claims = new List<Claim>
{
    new Claim(ClaimTypes.Name, user.Email),
    new Claim("LastChanged", {Database Value})
};


var claimsIdentity = new ClaimsIdentity(
    claims,
    CookieAuthenticationDefaults.AuthenticationScheme);

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity));
```

To implement an override for the `ValidatePrincipal` event, write a method with the following signature in a class that derives from [CookieAuthenticationEvents](#):

C#	 Copy
<pre>ValidatePrincipal(CookieValidatePrincipalContext)</pre>	

The following is an example implementation of `CookieAuthenticationEvents`:

C#	 Copy
<pre>using System.Linq; using System.Threading.Tasks; using Microsoft.AspNetCore.Authentication; using Microsoft.AspNetCore.Authentication.Cookies; public class CustomCookieAuthenticationEvents : CookieAuthentication- Events { private readonly IUserRepository _userRepository; public CustomCookieAuthenticationEvents(IUserRepository userReposi- tory) { // Get the database from registered DI services. _userRepository = userRepository; } public override async Task ValidatePrincipal(CookieValidatePrinci- palContext context) { var userPrincipal = context.Principal; // Look for the LastChanged claim. var lastChanged = (from c in userPrincipal.Claims where c.Type == "LastChanged" select c.Value).FirstOrDefault(); if (string.IsNullOrEmpty(lastChanged) !_userRepository.ValidateLastChanged(lastChanged)) { context.RejectPrincipal(); await context.HttpContext.SignOutAsync(CookieAuthenticationDefaults.AuthenticationScheme); } } }</pre>	

Register the events instance during cookie service registration in the `Startup.ConfigureServices` method. Provide a [scoped service registration](#) for your `CustomCookieAuthenticationEvents` class:

C#	 Copy
<pre>services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme) .AddCookie(options => { options.EventsType = typeof(CustomCookieAuthenticationEvents); }); services.AddScoped<CustomCookieAuthenticationEvents>();</pre>	

Consider a situation in which the user's name is updated—a decision that doesn't affect security in any way. If you want to non-destructively update the user principal, call `context.ReplacePrincipal` and set the `context.ShouldRenew` property to `true`.

Warning


The approach described here is triggered on every request. Validating authentication cookies for all users on every request can result in a large performance penalty for the app.

Persistent cookies

You may want the cookie to persist across browser sessions. This persistence should only be enabled with explicit user consent with a "Remember Me" check box on sign in or a similar mechanism.

The following code snippet creates an identity and corresponding cookie that survives through browser closures. Any sliding expiration settings previously configured are honored. If the cookie expires while the browser is closed, the browser clears the cookie once it's restarted.

Set `IsPersistent` to `true` in [AuthenticationProperties](#):

C#	 Copy
----	--


```
// using Microsoft.AspNetCore.Authentication;

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true
    });
```

Absolute cookie expiration

An absolute expiration time can be set with [ExpiresUtc](#). To create a persistent cookie, `IsPersistent` must also be set. Otherwise, the cookie is created with a session-based lifetime and could expire either before or after the authentication ticket that it holds. When `ExpiresUtc` is set, it overrides the value of the [ExpireTimeSpan](#) option of [CookieAuthenticationOptions](#), if set.

The following code snippet creates an identity and corresponding cookie that lasts for 20 minutes. This ignores any sliding expiration settings previously configured.

C#

 Copy

```
// using Microsoft.AspNetCore.Authentication;

await HttpContext.SignInAsync(
    CookieAuthenticationDefaults.AuthenticationScheme,
    new ClaimsPrincipal(claimsIdentity),
    new AuthenticationProperties
    {
        IsPersistent = true,
        ExpiresUtc = DateTime.UtcNow.AddMinutes(20)
    });
```

Additional resources

- [Authorize with a specific scheme in ASP.NET Core](#)
- [Claims-based authorization in ASP.NET Core](#)
- [Policy-based role checks](#)
- [Host ASP.NET Core in a web farm](#)

Is this page helpful?

 Yes  No
