Using Cookie Middleware without ASP.NET Core Identity

ASP.NET Core provides cookie middleware which serializes a user principal into an encrypted cookie and then, on subsequent requests, validates the cookie, recreates the principal and assigns it to the `User` property on `HttpContext`. If you want to provide your own login screens and user databases you can use the cookie middleware as a standalone feature.

Adding and configuring

The first step is adding the cookie middleware to your application. First use nuget to add the `Microsoft.AspNetCore.Authentication.Cookies` package. Then add the following lines to the `Configure` method in your *Startup.cs* file before the `app.UseMvc()` statement;

```
app.UseCookieAuthentication(new CookieAuthenticationOptions()
{
    AuthenticationScheme = "MyCookieMiddlewareInstance",
    LoginPath = new PathString("/Account/Unauthorized/"),
    AccessDeniedPath = new PathString("/Account/Forbidden/"),
    AutomaticAuthenticate = true,
    AutomaticChallenge = true
});
```

The code snippet above configures a few options;

- AuthenticationScheme - this is a value by which the middleware is known. This is useful when there are multiple instances of middleware and you want to limit authorization to one instance.
- LoginPath - this is the relative path requests will be redirected to when a user attempts to access a resource but has not been authenticated.
- AccessDeniedPath - this is the relative path requests will be redirected to when a user attempts to access a resource but does not pass any authorization policies for that resource.
- AutomaticAuthenticate - this flag indicates that the middleware should run on every request and attempt to validate and reconstruct any serialized principal it created.
- AutomaticChallenge - this flag indicates that the middleware should redirect the browser to the `LoginPath` or `AccessDeniedPath` when authorization fails.

Other options include the ability to set the issuer for any claims the middleware creates, the name of the cookie the middleware drops, the domain for the cookie and various security properties on the cookie. By default the cookie middleware will use appropriate security options for any cookies it creates, setting HTTPONLY to avoid the cookie being accessible in client side JavaScript and limiting the cookie to HTTPS if a request has come over HTTPS.

Creating an identity cookie

To create a cookie holding your user information you must construct a ClaimsPrincipal holding the information you wish to be serialized in the cookie. Once you have a suitable *ClaimsPrincipal* inside your controller method call

```
await HttpContext.Authentication.SignInAsync("MyCookieMiddlewareInstance", principal);
```

This will create an encrypted cookie and add it to the current response. The `AuthenticationScheme` specified during configuration must also be used when calling `SignInAsync`.

Under the covers the encryption used is ASP.NET's Data Protection system. If you are hosting on multiple machines, load balancing or using a web farm then you will need to configure data protection to use the same key ring and application identifier.

Signing out

To sign out the current user, and delete their cookie call the following inside your controller

```
await HttpContext.Authentication.SignOutAsync("MyCookieMiddlewareInstance");
```

Reacting to back-end changes

> **Warning**
>
> Once a principal cookie has been created it becomes the single source of identity - even if you disable a user in your back-end systems the cookie middleware has no knowledge of this and a user will continue to stay logged in as long as their cookie is valid.

The cookie authentication middleware provides a series of Events in its option class. The `ValidateAsync()` event can be used to intercept and override validation of the cookie identity.

Consider a back-end user database that may have a LastChanged column. In order to invalidate a cookie when the database changes you should first, when creating the cookie, add a LastChanged claim containing the current value. Then, when the database changes the LastChanged value should also be updated.

To implement an override for the `ValidateAsync()` event you must write a method with the following signature;

```
Task ValidateAsync(CookieValidatePrincipalContext context);
```

ASP.NET Core Identity implements this check as part of its SecurityStampValidator. A simple example would look something like as follows;

```
public static class LastChangedValidator
{
    public static async Task ValidateAsync(CookieValidatePrincipalContext context)
    {
        // Pull database from registered DI services.
        var userRepository =
context.HttpContext.RequestServices.GetRequiredService<IUserRepository>();
        var userPrincipal = context.Principal;

        // Look for the last changed claim.
        string lastChanged;
        lastChanged = (from c in userPrincipal.Claims
                       where c.Type == "LastUpdated"
                       select c.Value).FirstOrDefault();

        if (string.IsNullOrEmpty(lastChanged) ||
            !userRepository.ValidateLastChanged(userPrincipal, lastChanged))
        {
            context.RejectPrincipal();
            await
context.HttpContext.Authentication.SignOutAsync("MyCookieMiddlewareInstance");
        }
    }
}
```

This would then be wired up during cookie middleware configuration

```
app.UseCookieAuthentication(options =>
{
    options.Events = new CookieAuthenticationEvents
    {
        // Set other options
        OnValidatePrincipal = LastChangedValidator.ValidateAsync
    };
});
```

If you want to non-destructively update the user principal, for example, their name might have been updated, a decision which doesn't affect security in any way you can call `context.ReplacePrincipal()` and set the `context.ShouldRenew` flag to `true`.

Controlling cookie options

The `CookieAuthenticationOptions` class comes with various configuration options to enable you to fine tune the cookies created.

- **ClaimsIssuer** - the issuer to be used for the Issuer property on any claims created by the middleware.
- **CookieDomain** - the domain name the cookie will be served to. By default this is the host name the request was sent to. The browser will only serve the cookie to a matching host name. You may wish to adjust this to have cookies available to any host in your domain. For example setting the cookie domain to `.contoso.com` will make it available to `contoso.com`, `www.contoso.com`, `staging.www.contoso.com` etc.
- **CookieHttpOnly** - a flag indicating if the cookie should only be accessible to servers. This defaults to `true`. Changing this value may open your application to cookie theft should your application have a Cross Site Scripting bug.
- **CookiePath** - this can be used to isolate applications running on the same host name. If you have an app running in `/app1` and want to limit the cookies issued to just be sent to that application then you should set the `CookiePath` property to `/app1`. The cookie will now only be available to requests to `/app1` or anything underneath it.
- **CookieSecure** - a flag indicating if the cookie created should be limited to HTTPS, HTTP or HTTPS, or the same protocol as the request. This defaults to `SameAsRequest`.
- **ExpireTimeSpan** - the `TimeSpan` after which the cookie will expire. This is added to the current date and time to create the expiry date for the cookie.
- **SlidingExpiration** - a flag indicating if the cookie expiration date will be reset when the more than half of the `ExpireTimeSpan` interval has passed. The new expiry date will be moved forward to be the current date plus the `ExpireTimespan`. An absolute expiry time can be set by using the `AuthenticationProperties` class when calling `SignInAsync`. An absolute expiry can improve the security of your application by limiting the amount of time for which the authentication cookie is valid.

Persistent cookies and absolute expiry times

You may want to make the cookie expire be remembered over browser sessions. You may also want an absolute expiry to the identity and the cookie transporting it. You can do these things by using the `AuthenticationProperties` parameter on the

`HttpContext.Authentication.SignInAsync` method called when signing in an identity and creating the cookie. The `AuthenticationProperties` class is in the `Microsoft.AspNetCore.Http.Authentication` namespace.

For example;

```
await HttpContext.Authentication.SignInAsync(
    "MyCookieMiddlewareInstance",
    principal,
    new AuthenticationProperties
    {
        IsPersistent = true
    });
```

This code snippet will create an identity and corresponding cookie which will be survive through browser closures.Any sliding expiration settings previously configured via cookie options will still be honored, if the cookie expires whilst the browser is closed the browser will clear it once it is restarted.

```
await HttpContext.Authentication.SignInAsync(
    "MyCookieMiddlewareInstance",
    principal,
    new AuthenticationProperties
    {
        ExpiresUtc = DateTime.UtcNow.AddMinutes(20)
    });
```

This code snippet will create an identity and corresponding cookie which will be last for 20 minutes. This ignores any sliding expiration settings previously configured via cookie options.

The ExpiresUtc and IsPersistent properties are mutually exclusive.