Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC)

Article • 03/27/2024

This article describes how to secure a Blazor Web App with OpenID Connect (OIDC) using a sample app in the dotnet/blazor-samples GitHub repository (.NET 8 or later) (how to download).

This version of the article covers implementing OIDC with the Backend for Frontend (BFF) pattern. Change the article version selector to **OIDC without BFF pattern** if the app's specification doesn't call for adopting the BFF pattern.

The following specification is covered:

- The Blazor Web App uses the Auto render mode with global interactivity.
- Custom auth state provider services are used by the server and client apps to capture the user's authentication state and flow it between the server and client.
- This app is a starting point for any OIDC authentication flow. OIDC is configured manually in the app and doesn't rely upon Microsoft Entra ID or Microsoft Identity Web packages, nor does the sample app require Microsoft Azure hosting. However, the sample app can used with Entra, Microsoft Identity Web, and hosted in Azure.
- Automatic non-interactive token refresh.
- The Backend for Frontend (BFF) pattern is adopted using .NET Aspire for service discovery and YARP for proxying requests to a weather forecast endpoint on the backend app.
    - A backend web API uses JWT-bearer authentication to validate JWT tokens saved by the Blazor Web App in the sign-in cookie.
    - Aspire improves the experience of building .NET cloud-native apps. It provides a consistent, opinionated set of tools and patterns for building and running distributed apps.
    - YARP (Yet Another Reverse Proxy) is a library used to create a reverse proxy server.

Preview package warning

⚠ **Warning**

Technologies and packages used by the `BlazorWebAppOidcBff` sample app and described in this article are in ***preview*** release at this time. The article's content, the API, and the sample app aren't supported at this time and aren't currently recommended for production use. The sample app and guidance are subject to change without notice.

Prerequisite

.NET Aspire requires Visual Studio version 17.10 or later.

Sample app

The sample app consists of five projects:

- .NET Aspire:
  - `Aspire.AppHost`: Used to manage the high level orchestration concerns of the app.
  - `Aspire.ServiceDefaults`: Contains default .NET Aspire app configurations that can be extended and customized as needed.
- `MinimalApiJwt`: Backend web API, containing an example Minimal API endpoint for weather data.
- `BlazorWebAppOidc`: Server-side project of the Blazor Web App.
- `BlazorWebAppOidc.Client`: Client-side project of the Blazor Web App.

Access the sample apps through the latest version folder from the repository's root with the following link. The projects are in the `BlazorWebAppOidcBff` folder for .NET 8 or later.

View or download sample code (how to download)

.NET Aspire projects

For more information on using .NET Aspire and details on the `.AppHost` and `.ServiceDefaults` projects of the sample app, see the .NET Aspire documentation.

Confirm that you've met the prerequisites for .NET Aspire. For more information, see the *Prerequisites* section of Quickstart: Build your first .NET Aspire app.

Server-side Blazor Web App project (`BlazorWebAppOidc`)

The `BlazorWebAppOidc` project is the server-side project of the Blazor Web App. The project uses YARP to proxy requests to a weather forecast endpoint in the backend web API project (`MinimalApiJwt`) with the `access_token` stored in the authentication cookie.

The `BlazorWebAppOidc.http` file can be used for testing the weather data request. Note that the `BlazorWebAppOidc` project must be running to test the endpoint, and the endpoint is hardcoded into the file. For more information, see Use .http files in Visual Studio 2022.

> Note
>
> The server project uses IHttpContextAccessor/HttpContext, but never for interactively-rendered components. For more information, see Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering.

Configuration

This section explains how to configure the sample app.

> Note
>
> For Microsoft Entra ID and Azure AD B2C, you can use AddMicrosoftIdentityWebApp from Microsoft Identity Web (Microsoft.Identity.Web NuGet package, API documentation), which adds both the OIDC and Cookie authentication handlers with the appropriate defaults. The sample app and the guidance in this section doesn't use Microsoft Identity Web. The guidance demonstrates how to configure the OIDC handler *manually* for any OIDC provider. For more information on implementing Microsoft Identity Web, see the linked resources.

The following OpenIdConnectOptions configuration is found in the project's `Program` file on the call to AddOpenIdConnect:

- SignInScheme: Sets the authentication scheme corresponding to the middleware responsible of persisting user's identity after a successful authentication. The OIDC handler needs to use a sign-in scheme that's capable of persisting user credentials across requests. The following line is present merely for demonstration purposes. If omitted, DefaultSignInScheme is used as a fallback value.

```C#
oidcOptions.SignInScheme =
CookieAuthenticationDefaults.AuthenticationScheme;
```

- Scopes for `openid` and `profile` (Scope) (Optional): The `openid` and `profile` scopes are also configured by default because they're required for the OIDC handler to work, but these may need to be re-added if scopes are included in the `Authentication:Schemes:MicrosoftOidc:Scope` configuration. For general configuration guidance, see Configuration in ASP.NET Core and ASP.NET Core Blazor configuration.

```C#
oidcOptions.Scope.Add(OpenIdConnectScope.OpenIdProfile);
```

- SaveTokens: Defines whether access and refresh tokens should be stored in the AuthenticationProperties after a successful authorization. The value is set to `true` to authenticate requests for weather data from the backend web API project (`MinimalApiJwt`).

```C#
oidcOptions.SaveTokens = true;
```

- Scope for offline access (Scope): The `offline_access` scope is required for the refresh token.

```C#
oidcOptions.Scope.Add(OpenIdConnectScope.OfflineAccess);
```

- Scopes for obtaining weather data from the web API (Scope): The `Weather.Get` scope is configured in the Azure or Entra portal under **Expose an API**. This is necessary for backend web API project (`MinimalApiJwt`) to validate the access token with bearer JWT.

```C#
```

```
oidcOptions.Scope.Add("{APP ID URI}/{API NAME}");
```

Example:

- App ID URI (`{APP ID URI}`): `https://{DIRECTORY NAME}.onmicrosoft.com/{CLIENT ID}`
  - Directory Name (`{DIRECTORY NAME}`): `contoso`
  - Application (Client) Id (`{CLIENT ID}`): `4ba4de56-9cef-45d9-83fa-a4c18f9f5f0f`
- Scope configured for weather data from `MinimalApiJwt` (`{API NAME}`): `Weather.Get`

```C#
oidcOptions.Scope.Add("https://contoso.onmicrosoft.com/4ba4de56-
9cef-45d9-83fa-a4c18f9f5f0f/Weather.Get");
```

The preceding example pertains to an app registered in a tenant with an AAD B2C tenant type. If the app is registered in an ME-ID tenant, the App ID URI is different, thus the scope is different.

Example:

- App ID URI (`{APP ID URI}`): `api://{CLIENT ID}` with Application (Client) Id (`{CLIENT ID}`): `4ba4de56-9cef-45d9-83fa-a4c18f9f5f0f`
- Scope configured for weather data from `MinimalApiJwt` (`{API NAME}`): `Weather.Get`

```C#
oidcOptions.Scope.Add("api://4ba4de56-9cef-45d9-83fa-
a4c18f9f5f0f/Weather.Get");
```

- Authority and ClientId: Sets the Authority and Client ID for OIDC calls.

```C#
oidcOptions.Authority = "{AUTHORITY}";
oidcOptions.ClientId = "{CLIENT ID}";
```

Example:

- Authority (`{AUTHORITY}`): `https://login.microsoftonline.com/a3942615-d115-4eb7-bc84-9974abcf5064/v2.0/` (uses Tenant ID `a3942615-d115-4eb7-`

`bc84-9974abcf5064`)
    - Client Id (`{CLIENT ID}`): `4ba4de56-9cef-45d9-83fa-a4c18f9f5f0f`

  C#

  ```csharp
  oidcOptions.Authority =
  "https://login.microsoftonline.com/a3942615-d115-4eb7-bc84-
  9974abcf5064/v2.0/";
  oidcOptions.ClientId = "4ba4de56-9cef-45d9-83fa-a4c18f9f5f0f";
  ```

  Example for Microsoft Azure "common" authority:

  The "common" authority should be used for multi-tenant apps. You can also use the "common" authority for single-tenant apps, but a custom IssuerValidator is required, as shown later in this section.

  C#

  ```csharp
  oidcOptions.Authority =
  "https://login.microsoftonline.com/common/v2.0/";
  ```

- ClientSecret: The OIDC client secret.

  The following example is only for testing and demonstration purposes. Don't store the client secret in the app's assembly or check the secret into source control. Store the client secret in User Secrets, Azure Key Vault, or an environment variable.

  Authentication scheme configuration is automatically read from `builder.Configuration["Authentication:Schemes:{SCHEME NAME}:{PropertyName}"]`, where the `{SCHEME NAME}` placeholder is the scheme, which is `MicrosoftOidc` by default. Because configuration is preconfigured, a client secret can automatically be read via the `Authentication:Schemes:MicrosoftOidc:ClientSecret` configuration key. On the server using environment variables, name the environment variable `Authentication__Schemes__MicrosoftOidc__ClientSecret`:

  .NET CLI

  ```
  set Authentication__Schemes__MicrosoftOidc__ClientSecret={CLIENT
  SECRET}
  ```

For demonstration and testing only, the ClientSecret can be set directly. Don't set the value directly for deployed production apps. For slightly improved security, conditionally compile the line with the `DEBUG` symbol:

```C#
#if DEBUG
oidcOptions.ClientSecret = "{CLIENT SECRET}";
#endif
```

Example:

Client secret (`{CLIENT SECRET}`): `463471c8c4...f90d674bc9` (shortened for display)

```C#
#if DEBUG
oidcOptions.ClientSecret = "463471c8c4...137f90d674bc9";
#endif
```

- ResponseType: Configures the OIDC handler to only perform authorization code flow. Implicit grants and hybrid flows are unnecessary in this mode.

  In the Entra or Azure portal's **Implicit grant and hybrid flows** app registration configuration, do ***not** select either checkbox for the authorization endpoint to return **Access tokens** or **ID tokens**. The OIDC handler automatically requests the appropriate tokens using the code returned from the authorization endpoint.

  ```C#
  oidcOptions.ResponseType = OpenIdConnectResponseType.Code;
  ```

- MapInboundClaims and configuration of NameClaimType and RoleClaimType: Many OIDC servers use "`name`" and "`role`" rather than the SOAP/WS-Fed defaults in ClaimTypes. When MapInboundClaims is set to `false`, the handler doesn't perform claims mappings and the claim names from the JWT are used directly by the app. The following example manually maps the name and role claims:

ⓘ Note

MapInboundClaims must be set to `false` for most OIDC providers, which prevents renaming claims.

```C#
oidcOptions.MapInboundClaims = false;
oidcOptions.TokenValidationParameters.NameClaimType =
JwtRegisteredClaimNames.Name;
oidcOptions.TokenValidationParameters.RoleClaimType = "role";
```

- Path configuration: Paths must match the redirect URI (login callback path) and post logout redirect (signed-out callback path) paths configured when registering the application with the OIDC provider. In the Azure portal, paths are configured in the **Authentication** blade of the app's registration. Both the sign-in and sign-out paths must be registered as redirect URIs. The default values are `/signin-oidc` and `/signout-callback-oidc`.
  - CallbackPath: The request path within the app's base path where the user-agent is returned.

    In the Entra or Azure portal, set the path in the **Web** platform configuration's **Redirect URI**:

    > https://localhost/signin-oidc

    > Note
    >
    > A port isn't required for `localhost` addresses.

  - SignedOutCallbackPath: The request path within the app's base path where the user agent is returned after sign out from the identity provider.

    In the Entra or Azure portal, set the path in the **Web** platform configuration's **Redirect URI**:

    > https://localhost/signout-callback-oidc

    > Note
    >
    > A port isn't required for `localhost` addresses.

> ⓘ Note
>
> If using Microsoft Identity Web, the provider currently only redirects back to the SignedOutCallbackPath if the `microsoftonline.com` Authority (`https://login.microsoftonline.com/{TENANT ID}/v2.0/`) is used. This limitation doesn't exist if you can use the "common" Authority with Microsoft Identity Web. For more information, see postLogoutRedirectUri not working when authority url contains a tenant ID (AzureAD/microsoft-authentication-library-for-js #5783).

- RemoteSignOutPath: Requests received on this path cause the handler to invoke sign-out using the sign-out scheme.

  In the Entra or Azure portal, set the **Front-channel logout URL**:

  > https://localhost/signout-oidc

  > ⓘ Note
  >
  > A port isn't required for `localhost` addresses.

```C#
oidcOptions.CallbackPath = new PathString("{PATH}");
oidcOptions.SignedOutCallbackPath = new PathString("{PATH}");
oidcOptions.RemoteSignOutPath = new PathString("{PATH}");
```

Examples (default values):

```C#
oidcOptions.CallbackPath = new PathString("/signin-oidc");
oidcOptions.SignedOutCallbackPath = new PathString("/signout-
callback-oidc");
oidcOptions.RemoteSignOutPath = new PathString("/signout-oidc");
```

- (*Microsoft Azure only with the "common" endpoint*) TokenValidationParameters.IssuerValidator: Many OIDC providers work with the default issuer validator, but we need to account for the issuer parameterized with the Tenant ID (`{TENANT ID}`) returned by `https://login.microsoftonline.com/common/v2.0/.well-known/openid-`

`configuration`. For more information, see SecurityTokenInvalidIssuerException with OpenID Connect and the Azure AD "common" endpoint (AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet #1731).

Only for apps using Microsoft Entra ID or Azure AD B2C with the "common" endpoint:

C#

```
var microsoftIssuerValidator =
AadIssuerValidator.GetAadIssuerValidator(oidcOptions.Authority);
oidcOptions.TokenValidationParameters.IssuerValidator =
microsoftIssuerValidator.Validate;
```

Sample app code

Inspect the sample app for the following features:

- Automatic non-interactive token refresh with the help of a custom cookie refresher (`CookieOidcRefresher.cs`).
- The `PersistingAuthenticationStateProvider` class (`PersistingAuthenticationStateProvider.cs`) is a server-side AuthenticationStateProvider that uses PersistentComponentState to flow the authentication state to the client, which is then fixed for the lifetime of the WebAssembly application.
- Requests to the Blazor Web App are proxied to the backend web API project (`MinimalApiJwt`). `MapForwarder` in the `Program` file adds direct forwarding of HTTP requests that match the specified pattern to a specific destination using default configuration for the outgoing request, customized transforms, and default HTTP client:
  - When rendering the `Weather` component on the server, the component uses the `ServerWeatherForecaster` to proxy the request for weather data with the user's access token.
  - When the component is rendered on the client, the component uses the `ClientWeatherForecaster` service implementation, which uses a preconfigured HttpClient (in the client project's `Program` file) to make a web API call to the server project. A Minimal API endpoint (`/weather-forecast`) defined in the server project's `Program` file transforms the request with the user's access token to obtain the weather data.

For more information on (web) API calls using a service abstractions in Blazor Web Apps, see Call a web API from an ASP.NET Core Blazor app.

Client-side Blazor Web App project (`BlazorWebAppOidc.Client`)

The `BlazorWebAppOidc.Client` project is the client-side project of the Blazor Web App.

The `PersistentAuthenticationStateProvider` class (`PersistentAuthenticationStateProvider.cs`) is a client-side AuthenticationStateProvider that determines the user's authentication state by looking for data persisted in the page when it was rendered on the server. The authentication state is fixed for the lifetime of the WebAssembly application.

If the user needs to log in or out, a full page reload is required.

The sample app only provides a user name and email for display purposes. It doesn't include tokens that authenticate to the server when making subsequent requests, which works separately using a cookie that's included on HttpClient requests to the server.

Backend web API project (`MinimalApiJwt`)

The `MinimalApiJwt` project is a backend web API for multiple frontend projects. The project configures a Minimal API endpoint for weather data. Requests from the Blazor Web App server-side project (`BlazorWebAppOidc`) are proxied to the `MinimalApiJwt` project.

Configuration

Configure the project in the JwtBearerOptions of the AddJwtBearer call in the project's `Program` file:

- Audience: Sets the Audience for any received OpenID Connect token.

  In the Azure or Entra portal: Match the value to just the path of the **Application ID URI** configured when adding the `Weather.Get` scope under **Expose an API**:

  ```C#
  jwtOptions.Audience = "{APP ID URI}";
  ```

Example:

App ID URI (`{APP ID URI}`): `https://{DIRECTORY NAME}.onmicrosoft.com/{CLIENT ID}`:

- Directory Name (`{DIRECTORY NAME}`): `contoso`
- Application (Client) Id (`{CLIENT ID}`): `4ba4de56-9cef-45d9-83fa-a4c18f9f5f0f`

C#

```csharp
jwtOptions.Audience = "https://contoso.onmicrosoft.com/4ba4de56-
9cef-45d9-83fa-a4c18f9f5f0f";
```

The preceding example pertains to an app registered in a tenant with an AAD B2C tenant type. If the app is registered in an ME-ID tenant, the App ID URI is different, thus the audience is different.

Example:

App ID URI (`{APP ID URI}`): `api://{CLIENT ID}` with Application (Client) Id (`{CLIENT ID}`): `4ba4de56-9cef-45d9-83fa-a4c18f9f5f0f`

C#

```csharp
jwtOptions.Audience = "api://4ba4de56-9cef-45d9-83fa-
a4c18f9f5f0f";
```

- Authority: Sets the Authority for making OpenID Connect calls. Match the value to the Authority configured for the OIDC handler in `BlazorWebAppOidc/Program.cs`:

C#

```csharp
jwtOptions.Authority = "{AUTHORITY}";
```

Example:

Authority (`{AUTHORITY}`): `https://login.microsoftonline.com/a3942615-d115-4eb7-bc84-9974abcf5064/v2.0/` (uses Tenant ID `a3942615-d115-4eb7-bc84-9974abcf5064`)

C#

```csharp
jwtOptions.Authority =
"https://login.microsoftonline.com/a3942615-d115-4eb7-bc84-
```

```
9974abcf5064/v2.0/";
```

The preceding example pertains to an app registered in a tenant with an AAD B2C tenant type. If the app is registered in an ME-ID tenant, the authority should match the issurer (`iss`) of the JWT returned by the identity provider:

```C#
jwtOptions.Authority = "https://sts.windows.net/a3942615-d115-
4eb7-bc84-9974abcf5064/";
```

Minimal API for weather data

Secure weather forecast data endpoint in the project's `Program` file:

```C#
app.MapGet("/weather-forecast", () =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)]
        ))
        .ToArray();
    return forecast;
}).RequireAuthorization();
```

The RequireAuthorization extension method requires authorization for the route definition. For any controllers that you add to the project, add the [Authorize] attribute to the controller or action.

Redirect to the home page on signout

When a user navigates around the app, the `LogInOrOut` component (`Layout/LogInOrOut.razor`) sets a hidden field for the return URL (`ReturnUrl`) to the value of the current URL (`currentURL`). When the user signs out of the app, the identity provider returns them to the page from which they signed out.

If the user signs out from a secure page, they're returned back to the same secure page after signing out only to be sent back through the authentication process. This behavior is fine when users need to switch accounts frequently. However, a alternative app specification may call for the user to be returned to the app's home page or some other page after signout. The following example shows how to set the app's home page as the return URL for signout operations.

The important changes to the `LogInOrOut` component are demonstrated in the following example. The `value` of the hidden field for the `ReturnUrl` is set to the home page at `/`. IDisposable is no longer implemented. The NavigationManager is no longer injected. The entire `@code` block is removed.

`Layout/LogInOrOut.razor`:

```razor
@using Microsoft.AspNetCore.Authorization

<div class="nav-item px-3">
    <AuthorizeView>
        <Authorized>
            <form action="authentication/logout" method="post">
                <AntiforgeryToken />
                <input type="hidden" name="ReturnUrl" value="/" />
                <button type="submit" class="nav-link">
                    <span class="bi bi-arrow-bar-left-nav-menu" aria-
hidden="true">
                    </span> Logout @context.User.Identity?.Name
                </button>
            </form>
        </Authorized>
        <NotAuthorized>
            <a class="nav-link" href="authentication/login">
                <span class="bi bi-person-badge-nav-menu" aria-hid-
den="true"></span>
                Login
            </a>
        </NotAuthorized>
    </AuthorizeView>
</div>
```

Cryptographic nonce

A *nonce* is a string value that associates a client's session with an ID token to mitigate replay attacks.

If you receive a nonce error during authentication development and testing, use a new InPrivate/incognito browser session for each test run, no matter how small the change made to the app or test user because stale cookie data can lead to a nonce error. For more information, see the Cookies and site data section.

A nonce isn't required or used when a refresh token is exchanged for a new access token. In the sample app, the `CookieOidcRefresher` (`CookieOidcRefresher.cs`) deliberately sets OpenIdConnectProtocolValidator.RequireNonce to `false`.

Troubleshoot

Logging

The server app is a standard ASP.NET Core app. See the ASP.NET Core logging guidance to enable a lower logging level in the server app.

To enable debug or trace logging for Blazor WebAssembly authentication, see the *Client-side authentication logging* section of ASP.NET Core Blazor logging with the article version selector set to ASP.NET Core 7.0 or later.

Common errors

- Misconfiguration of the app or Identity Provider (IP)

  The most common errors are caused by incorrect configuration. The following are a few examples:
  - Depending on the requirements of the scenario, a missing or incorrect Authority, Instance, Tenant ID, Tenant domain, Client ID, or Redirect URI prevents an app from authenticating clients.
  - Incorrect request scopes prevent clients from accessing server web API endpoints.
  - Incorrect or missing server API permissions prevent clients from accessing server web API endpoints.
  - Running the app at a different port than is configured in the Redirect URI of the IP's app registration. Note that a port isn't required for Microsoft Entra ID and an app running at a `localhost` development testing address, but the app's port configuration and the port where the app is running must match for non-`localhost` addresses.

  Configuration coverage in this article shows examples of the correct configuration. Carefully check the configuration looking for app and IP misconfiguration.

  If the configuration appears correct:

  - Analyze application logs.
  - Examine the network traffic between the client app and the IP or server app with the browser's developer tools. Often, an exact error message or a message with a clue to what's causing the problem is returned to the client by the IP or server app after making a request. Developer tools guidance is found in the following articles:
    - Google Chrome (Google documentation)
    - Microsoft Edge
    - Mozilla Firefox (Mozilla documentation)

  The documentation team responds to document feedback and bugs in articles (open an issue from the **This page** feedback section) but is unable to provide product support. Several public support forums are available to assist with troubleshooting an app. We recommend the following:
  - Stack Overflow (tag: blazor)
  - ASP.NET Core Slack Team
  - Blazor Gitter

  The preceding forums are not owned or controlled by Microsoft.

  For non-security, non-sensitive, and non-confidential reproducible framework bug reports, open an issue with the ASP.NET Core product unit. Don't open an issue with the product unit until you've thoroughly investigated the cause of a problem and can't resolve it on your own and with the help of the community on a public support forum. The product unit isn't able to troubleshoot individual apps that are broken due to simple misconfiguration or use cases involving third-party services. If a report is sensitive or confidential in nature or describes a potential security flaw in the product that attackers may exploit, see Reporting security issues and bugs (dotnet/aspnetcore GitHub repository).

- Unauthorized client for ME-ID

  > info: Microsoft.AspNetCore.Authorization.DefaultAuthorizationService[2] Authorization failed. These requirements were not met: DenyAnonymousAuthorizationRequirement: Requires an authenticated user.

  Login callback error from ME-ID:
  - Error: `unauthorized_client`
  - Description: `AADB2C90058: The provided application is not configured to allow public clients.`

To resolve the error:

1. In the Azure portal, access the app's manifest.
2. Set the allowPublicClient attribute to `null` or `true`.

Cookies and site data

Cookies and site data can persist across app updates and interfere with testing and troubleshooting. Clear the following when making app code changes, user account changes with the provider, or provider app configuration changes:

- User sign-in cookies
- App cookies
- Cached and stored site data

One approach to prevent lingering cookies and site data from interfering with testing and troubleshooting is to:

- Configure a browser
  - Use a browser for testing that you can configure to delete all cookie and site data each time the browser is closed.
  - Make sure that the browser is closed manually or by the IDE for any change to the app, test user, or provider configuration.
- Use a custom command to open a browser in InPrivate or Incognito mode in Visual Studio:
  - Open **Browse With** dialog box from Visual Studio's **Run** button.
  - Select the **Add** button.
  - Provide the path to your browser in the **Program** field. The following executable paths are typical installation locations for Windows 10. If your browser is installed in a different location or you aren't using Windows 10, provide the path to the browser's executable.
    - Microsoft Edge: `C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe`
    - Google Chrome: `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`
    - Mozilla Firefox: `C:\Program Files\Mozilla Firefox\firefox.exe`
  - In the **Arguments** field, provide the command-line option that the browser uses to open in InPrivate or Incognito mode. Some browsers require the URL of the app.
    - Microsoft Edge: Use `-inprivate`.

    - Google Chrome: Use `--incognito --new-window {URL}`, where the placeholder `{URL}` is the URL to open (for example, `https://localhost:5001`).
    - Mozilla Firefox: Use `-private -url {URL}`, where the placeholder `{URL}` is the URL to open (for example, `https://localhost:5001`).
  - Provide a name in the **Friendly name** field. For example, `Firefox Auth Testing`.
  - Select the **OK** button.
  - To avoid having to select the browser profile for each iteration of testing with an app, set the profile as the default with the **Set as Default** button.
  - Make sure that the browser is closed by the IDE for any change to the app, test user, or provider configuration.

App upgrades

A functioning app may fail immediately after upgrading either the .NET Core SDK on the development machine or changing package versions within the app. In some cases, incoherent packages may break an app when performing major upgrades. Most of these issues can be fixed by following these instructions:

1. Clear the local system's NuGet package caches by executing dotnet nuget locals all --clear from a command shell.
2. Delete the project's `bin` and `obj` folders.
3. Restore and rebuild the project.
4. Delete all of the files in the deployment folder on the server prior to redeploying the app.

> ⓘ **Note**
>
> Use of package versions incompatible with the app's target framework isn't supported. For information on a package, use the NuGet Gallery or FuGet Package Explorer.

Run the server app

When testing and troubleshooting Blazor Web App, make sure that you're running the app from the server project.

Inspect the user

The following `UserClaims` component can be used directly in apps or serve as the basis for further customization.

`UserClaims.razor`:

razor

```razor
@page "/user-claims"
@using System.Security.Claims
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]

<PageTitle>User Claims</PageTitle>

<h1>User Claims</h1>

@if (claims.Count() > 0)
{
    <ul>
        @foreach (var claim in claims)
        {
            <li><b>@claim.Type:</b> @claim.Value</li>
        }
    </ul>
}

@code {
    private IEnumerable<Claim> claims = Enumerable.Empty<Claim>();

    [CascadingParameter]
    private Task<AuthenticationState>? AuthState { get; set; }

    protected override async Task OnInitializedAsync()
    {
        if (AuthState == null)
        {
            return;
        }

        var authState = await AuthState;
        claims = authState.User.Claims;
    }
}
```

Additional resources

- AzureAD/microsoft-identity-web GitHub repository: Helpful guidance on implementing Microsoft Identity Web for Microsoft Entra ID and Azure Active Directory B2C for ASP.NET Core apps, including links to sample apps and related

Azure documentation. Currently, Blazor Web Apps aren't explicitly addressed by the Azure documentation, but the setup and configuration of a Blazor Web App for ME-ID and Azure hosting is the same as it is for any ASP.NET Core web app.
- AuthenticationStateProvider service
- Manage authentication state in Blazor Web Apps

Collaborate with us on GitHub

The source for this content can be found on GitHub, where you can also create and review issues and pull requests. For more information, see our contributor guide.

.NET

ASP.NET Core feedback

ASP.NET Core is an open source project. Select a link to provide feedback:

Open a documentation issue

Provide product feedback