



Configuration sections of this article's guidance show examples of the correct configuration. Carefully check each section of the article 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)
- For releases of Blazor where a JSON Web Token (JWT) is used, decode the contents of the token used for authenticating a client or accessing a server web API, depending on where the problem is occurring. For more information, see [Inspect the content of a JSON Web Token \(JWT\)](#).

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 cyberattackers 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 `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
 - Mozilla Firefox: Use `-private -url {URL}`, where the `{URL}` placeholder 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](#).

Inspect the user

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

`User.razor`:

razor

```
@page "/user"
@attribute [Authorize]
@using System.Text.Json
@using System.Security.Claims
@inject IAccessTokenProvider AuthorizationService

<h1>@AuthenticatedUser?.Identity?.Name</h1>

<h2>Claims</h2>

@foreach (var claim in AuthenticatedUser?.Claims ?? Array.Empty<Claim>())
{
    <p class="claim">@(claim.Type): @claim.Value</p>
}

<h2>Access token</h2>

<p id="access-token">@AccessToken?.Value</p>

<h2>Access token claims</h2>

@foreach (var claim in GetAccessTokenClaims())
{
    <p>@(claim.Key): @claim.Value.ToString()</p>
}

@if (AccessToken != null)
{
    <h2>Access token expires</h2>

    <p>Current time: <span id="current-time">@DateTimeOffset.Now</span></p>
    <p id="access-token-expires">@AccessToken.Expires</p>

    <h2>Access token granted scopes (as reported by the API)</h2>

    @foreach (var scope in AccessToken.GrantedScopes)
    {
        <p>Scope: @scope</p>
    }
}

@code {
    [CascadingParameter]
    private Task<AuthenticationState> AuthenticationState { get; set; }

    public ClaimsPrincipal AuthenticatedUser { get; set; }
```

```

public AccessToken AccessToken { get; set; }

protected override async Task OnInitializedAsync()
{
    await base.OnInitializedAsync();
    var state = await AuthenticationState;
    var accessTokenResult = await
AuthorizationService.RequestAccessToken();

    if (!accessTokenResult.TryGetToken(out var token))
    {
        throw new InvalidOperationException(
            "Failed to provision the access token.");
    }

    AccessToken = token;

    AuthenticatedUser = state.User;
}

protected IDictionary<string, object> GetAccessTokenClaims()
{
    if (AccessToken == null)
    {
        return new Dictionary<string, object>();
    }

    // header.payload.signature
    var payload = AccessToken.Value.Split(".")[1];
    var base64Payload = payload.Replace('-', '+').Replace('_', '/')
        .PadRight(payload.Length + (4 - payload.Length % 4) % 4, '=');

    return JsonSerializer.Deserialize<IDictionary<string, object>>(
        Convert.FromBase64String(base64Payload));
}
}

```

Inspect the content of a JSON Web Token (JWT)

To decode a JSON Web Token (JWT), use Microsoft's jwt.ms tool. Values in the UI never leave your browser.

Example encoded JWT (shortened for display):

```

eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ilg1ZVhrNHh5b2pORnVtMWtsMlI0
djhkbE5QNC1j...
bQdHBHGcQQRbW7Wmo6SWYG4V_bU55Ug_PW4pLP20tTS8Ct7_uwy9DWrzCMzp
D-EiwT5ljXwlGX3lXVjHlIX50IVlydBoPQtadvT7saKo1G5Jmutgq41o-dmz6-
yBMKV2_nXA25Q

```

Example JWT decoded by the tool for an app that authenticates against Azure AAD B2C:

JSON

```
{
  "typ": "JWT",
  "alg": "RS256",
  "kid": "X5eXk4xyojNFum1kl2Ytv8dlNP4-c57d06QGTVBwaNk"
}.{
  "exp": 1610059429,
  "nbf": 1610055829,
  "ver": "1.0",
  "iss": "https://mysiteb2c.b2clogin.com/11112222-bbbb-3333-cccc-4444dddd5555/v2.0/",
  "sub": "aaaaaaaa-0000-1111-2222-bbbbbbbbbbbb",
  "aud": "00001111-aaaa-2222-bbbb-3333cccc4444",
  "nonce": "bbbb0000-cccc-1111-dddd-2222eeee3333",
  "iat": 1610055829,
  "auth_time": 1610055822,
  "idp": "idp.com",
  "tfp": "B2C_1_signupsigin"
}.[Signature]
```

Additional resources

- [Identity and account types for single- and multitenant apps](#)
- [ASP.NET Core Blazor WebAssembly additional security scenarios](#)
- [Build a custom version of the Authentication.MSAL JavaScript library](#)
- [Unauthenticated or unauthorized web API requests in an app with a secure default client](#)
- [Cloud authentication with Azure Active Directory B2C in ASP.NET Core](#)
- [Tutorial: Create an Azure Active Directory B2C tenant](#)
- [Tutorial: Register an application in Azure Active Directory B2C](#)
- [Microsoft identity platform documentation](#)

ASP.NET Core Blazor WebAssembly additional security scenarios

Article • 10/08/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article describes additional security scenarios for Blazor WebAssembly apps.

Attach tokens to outgoing requests

[AuthorizationMessageHandler](#) is a [DelegatingHandler](#) used to process access tokens. Tokens are acquired using the [IAccessTokenProvider](#) service, which is registered by the framework. If a token can't be acquired, an [AccessTokenNotAvailableException](#) is thrown. [AccessTokenNotAvailableException](#) has a [Redirect](#) method that navigates to [AccessTokenResult.InteractiveRequestUrl](#) using the given [AccessTokenResult.InteractionOptions](#) to allow refreshing the access token.

For convenience, the framework provides the [BaseAddressAuthorizationMessageHandler](#) preconfigured with the app's base address as an authorized URL. **Access tokens are only added when the request URI is within the app's base URI.** When outgoing request URIs aren't within the app's base URI, use a custom [AuthorizationMessageHandler class \(recommended\)](#) or [configure the AuthorizationMessageHandler](#).

Note

In addition to the client app configuration for server API access, the server API must also allow cross-origin requests (CORS) when the client and the server don't reside at the same base address. For more information on server-side CORS configuration, see the [Cross-Origin Resource Sharing \(CORS\)](#) section later in this article.

In the following example:

- `AddHttpClient` adds `HttpClientFactory` and related services to the service collection and configures a named `HttpClient` (WebAPI). `HttpClient.BaseAddress` is the base address of the resource URI when sending requests. `HttpClientFactory` is provided by the `Microsoft.Extensions.Http` [NuGet package](#).
- `BaseAddressAuthorizationMessageHandler` is the `DelegatingHandler` used to process access tokens. Access tokens are only added when the request URI is within the app's base URI.
- `HttpClientFactory.CreateClient` creates and configures an `HttpClient` instance for outgoing requests using the configuration that corresponds to the named `HttpClient` (WebAPI).

In the following example, `HttpClientFactoryServiceCollectionExtensions.AddHttpClient` is an extension in `Microsoft.Extensions.Http`. Add the package to an app that doesn't already reference it.

ⓘ Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at [Package consumption workflow \(NuGet documentation\)](#). Confirm correct package versions at [NuGet.org](#) [↗](#).

C#

```
using System.Net.Http;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddHttpClient("WebAPI",
    client => client.BaseAddress = new
Uri("https://api.contoso.com/v1.0"))
    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();

builder.Services.AddScoped(sp => sp.GetRequiredService<IHttpClientFactory>()
    .CreateClient("WebAPI"));
```

The configured `HttpClient` is used to make authorized requests using the `try-catch` pattern:

razor

```
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject HttpClient Http

...
```



```
protected override async Task OnInitializedAsync()
{
    try
    {
        var examples =
            await Http.GetFromJsonAsync<ExampleType[]>("ExampleAPIMethod");

        ...
    }
    catch (AccessTokenNotAvailableException exception)
    {
        exception.Redirect();
    }
}
```

Custom authentication request scenarios

The following scenarios demonstrate how to customize authentication requests and how to obtain the login path from authentication options.

Customize the login process

Manage additional parameters to a login request with the following methods one or more times on a new instance of [InteractiveRequestOptions](#):

- [TryAddAdditionalParameter](#)
- [TryRemoveAdditionalParameter](#)
- [TryGetAdditionalParameter](#)

In the following `LoginDisplay` component example, additional parameters are added to the login request:

- `prompt` is set to `login`: Forces the user to enter their credentials on that request, negating single sign on.
- `loginHint` is set to `peter@contoso.com`: Pre-fills the username/email address field of the sign-in page for the user to `peter@contoso.com`. Apps often use this parameter during re-authentication, having already extracted the username from a previous sign in using the `preferred_username` claim.

Shared/LoginDisplay.razor:

C#

```

@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject NavigationManager Navigation

<AuthorizeView>
    <Authorized>
        Hello, @context.User.Identity?.Name!
        <button @onclick="BeginLogout">Log out</button>
    </Authorized>
    <NotAuthorized>
        <button @onclick="BeginLogin">Log in</button>
    </NotAuthorized>
</AuthorizeView>

@code{
    public void BeginLogout()
    {
        Navigation.NavigateToLogout("authentication/logout");
    }

    public void BeginLogin()
    {
        InteractiveRequestOptions requestOptions =
            new()
            {
                Interaction = InteractionType.SignIn,
                ReturnUrl = Navigation.Uri,
            };

        requestOptions.TryAddAdditionalParameter("prompt", "login");
        requestOptions.TryAddAdditionalParameter("loginHint",
            "peter@contoso.com");

        Navigation.NavigateToLogin("authentication/login", requestOptions);
    }
}

```

For more information, see the following resources:

- [InteractiveRequestOptions](#)
- [Popup request parameter list](#)

Customize options before obtaining a token interactively

If an [AccessTokenNotAvailableException](#) occurs, manage additional parameters for a new identity provider access token request with the following methods one or more times on a new instance of [InteractiveRequestOptions](#):

- [TryAddAdditionalParameter](#)

- [TryRemoveAdditionalParameter](#)
- [TryGetAdditionalParameter](#)

In the following example that obtains JSON data via web API, additional parameters are added to the redirect request if an access token isn't available ([AccessTokenNotAvailableException](#) is thrown):

- `prompt` is set to `login`: Forces the user to enter their credentials on that request, negating single sign on.
- `loginHint` is set to `peter@contoso.com`: Pre-fills the username/email address field of the sign-in page for the user to `peter@contoso.com`. Apps often use this parameter during re-authentication, having already extracted the username from a previous sign in using the `preferred_username` claim.

C#

```
try
{
    var examples = await Http.GetFromJsonAsync<ExampleType[]>
("ExampleAPIMethod");

    ...
}
catch (AccessTokenNotAvailableException ex)
{
    ex.Redirect(requestOptions => {
        requestOptions.TryAddAdditionalParameter("prompt", "login");
        requestOptions.TryAddAdditionalParameter("loginHint",
"peter@contoso.com");
    });
}
```

The preceding example assumes that:

- The presence of an `@using/using` statement for API in the [Microsoft.AspNetCore.Components.WebAssembly.Authentication](#) namespace.
- `HttpClient` injected as `Http`.

For more information, see the following resources:

- [InteractiveRequestOptions](#)
- [Redirect request parameter list](#) [↗](#)

Customize options when using an `IAccessTokenProvider`

If obtaining a token fails when using an [IAccessTokenProvider](#), manage additional parameters for a new identity provider access token request with the following methods one or more times on a new instance of [InteractiveRequestOptions](#):

- [TryAddAdditionalParameter](#)
- [TryRemoveAdditionalParameter](#)
- [TryGetAdditionalParameter](#)

In the following example that attempts to obtain an access token for the user, additional parameters are added to the login request if the attempt to obtain a token fails when [TryGetToken](#) is called:

- `prompt` is set to `login`: Forces the user to enter their credentials on that request, negating single sign on.
- `loginHint` is set to `peter@contoso.com`: Pre-fills the username/email address field of the sign-in page for the user to `peter@contoso.com`. Apps often use this parameter during re-authentication, having already extracted the username from a previous sign in using the `preferred_username` claim.

C#

```
var tokenResult = await TokenProvider.RequestAccessToken(  
    new AccessTokenRequestOptions  
    {  
        Scopes = new[] { ... }  
    });  
  
if (!tokenResult.TryGetToken(out var token))  
{  
    tokenResult.InteractionOptions.TryAddAdditionalParameter("prompt",  
"login");  
    tokenResult.InteractionOptions.TryAddAdditionalParameter("loginHint",  
"peter@contoso.com");  
  
    Navigation.NavigateToLogin(accessTokenResult.InteractiveRequestUrl,  
        accessTokenResult.InteractionOptions);  
}
```

The preceding example assumes:

- The presence of an `@using/using` statement for API in the [Microsoft.AspNetCore.Components.WebAssembly.Authentication](#) namespace.
- [IAccessTokenProvider](#) injected as `TokenProvider`.

For more information, see the following resources:

- [InteractiveRequestOptions](#)

- [Popup request parameter list](#)

Logout with a custom return URL

The following example logs out the user and returns the user to the `/goodbye` endpoint:

C#

```
Navigation.NavigateToLogout("authentication/logout", "goodbye");
```

Obtain the login path from authentication options

Obtain the configured login path from [RemoteAuthenticationOptions](#):

C#

```
var loginPath =  
  
RemoteAuthOptions.Get(Options.DefaultName).AuthenticationPaths.LogInPath;
```

The preceding example assumes:

- The presence of an `@using/using` statement for API in the following namespaces:
 - [Microsoft.AspNetCore.Components.WebAssembly.Authentication](#)
 - [Microsoft.Extensions.Options](#)
- `IOptionsSnapshot<RemoteAuthenticationOptions<ApiAuthorizationProviderOptions>` injected as `RemoteAuthOptions`.

Custom `AuthorizationMessageHandler` class

This guidance in this section is recommended for client apps that make outgoing requests to URIs that aren't within the app's base URI.

In the following example, a custom class extends [AuthorizationMessageHandler](#) for use as the [DelegatingHandler](#) for an [HttpClient](#). [ConfigureHandler](#) configures this handler to authorize outbound HTTP requests using an access token. The access token is only attached if at least one of the authorized URLs is a base of the request URI ([HttpRequestMessage.RequestUri](#)).

C#

```
using Microsoft.AspNetCore.Components;  
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
```

```

public class CustomAuthorizationMessageHandler : AuthorizationMessageHandler
{
    public CustomAuthorizationMessageHandler(IAccessTokenProvider provider,
        NavigationManager navigation)
        : base(provider, navigation)
    {
        ConfigureHandler(
            authorizedUrls: [ "https://api.contoso.com/v1.0" ],
            scopes: [ "example.read", "example.write" ]);
    }
}

```

In the preceding code, the scopes `example.read` and `example.write` are generic examples not meant to reflect valid scopes for any particular provider.

In the `Program` file, `CustomAuthorizationMessageHandler` is registered as a transient service and is configured as the `DelegatingHandler` for outgoing `HttpResponseMessage` instances made by a named `HttpClient`.

In the following example, `HttpClientFactoryServiceCollectionExtensions.AddHttpClient` is an extension in `Microsoft.Extensions.Http`. Add the package to an app that doesn't already reference it.

❗ Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at [Package consumption workflow \(NuGet documentation\)](#). Confirm correct package versions at [NuGet.org](#).

C#

```

builder.Services.AddTransient<CustomAuthorizationMessageHandler>();

builder.Services.AddHttpClient("WebAPI",
    client => client.BaseAddress = new
Uri("https://api.contoso.com/v1.0"))
    .AddHttpMessageHandler<CustomAuthorizationMessageHandler>();

```

❗ Note

In the preceding example, the `CustomAuthorizationMessageHandler` `DelegatingHandler` is registered as a transient service for `AddHttpMessageHandler`. Transient registration is recommended for

[IHttpClientFactory](#), which manages its own DI scopes. For more information, see the following resources:

- [Utility base component classes to manage a DI scope](#)
- [Detect client-side transient disposables](#)

The configured [HttpClient](#) is used to make authorized requests using the [try-catch](#) pattern. Where the client is created with [CreateClient](#) ([Microsoft.Extensions.Http](#) package), the [HttpClient](#) is supplied instances that include access tokens when making requests to the server API. If the request URI is a relative URI, as it is in the following example ([ExampleAPIMethod](#)), it's combined with the [BaseAddress](#) when the client app makes the request:

```
razor

@Inject IHttpClientFactory ClientFactory

...

@code {
    protected override async Task OnInitializedAsync()
    {
        try
        {
            var client = ClientFactory.CreateClient("WebAPI");

            var examples =
                await client.GetFromJsonAsync<ExampleType[]>
                ("ExampleAPIMethod");

            ...
        }
        catch (AccessTokenNotAvailableException exception)
        {
            exception.Redirect();
        }
    }
}
```

Configure [AuthorizationMessageHandler](#)

[AuthorizationMessageHandler](#) can be configured with authorized URLs, scopes, and a return URL using the [ConfigureHandler](#) method. [ConfigureHandler](#) configures the handler to authorize outbound HTTP requests using an access token. The access token is only attached if at least one of the authorized URLs is a base of the request URI

([HttpRequestMessage.RequestUri](#)). If the request URI is a relative URI, it's combined with the [BaseAddress](#).

In the following example, [AuthorizationMessageHandler](#) configures an [HttpClient](#) in the `Program` file:

C#

```
using System.Net.Http;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddScoped(sp => new HttpClient(
    sp.GetRequiredService<AuthorizationMessageHandler>()
    .ConfigureHandler(
        authorizedUrls: [ "https://api.contoso.com/v1.0" ],
        scopes: [ "example.read", "example.write" ])
    .InnerHandler = new HttpClientHandler())
{
    BaseAddress = new Uri("https://api.contoso.com/v1.0")
});
```

In the preceding code, the scopes `example.read` and `example.write` are generic examples not meant to reflect valid scopes for any particular provider.

Typed `HttpClient`

A typed client can be defined that handles all of the HTTP and token acquisition concerns within a single class.

`WeatherForecastClient.cs`:

C#

```
using System.Net.Http.Json;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using static {ASSEMBLY NAME}.Data;

public class WeatherForecastClient(HttpClient http)
{
    private WeatherForecast[]? forecasts;

    public async Task<WeatherForecast[]> GetForecastAsync()
    {
        try
        {
            forecasts = await http.GetFromJsonAsync<WeatherForecast[]>(
                "WeatherForecast");
        }
    }
}
```



```

    }
    catch (AccessTokenNotAvailableException exception)
    {
        exception.Redirect();
    }

    return forecasts ?? Array.Empty<WeatherForecast>();
}
}

```

In the preceding example, the `WeatherForecast` type is a static class that holds weather forecast data. The `{ASSEMBLY NAME}` placeholder is the app's assembly name (for example, `using static BlazorSample.Data;`).

In the following example, `HttpClientFactoryServiceCollectionExtensions.AddHttpClient` is an extension in `Microsoft.Extensions.Http`. Add the package to an app that doesn't already reference it.

ⓘ Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at [Package consumption workflow \(NuGet documentation\)](#). Confirm correct package versions at [NuGet.org](#) [↗].

In the `Program` file:

C#

```

using System.Net.Http;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddHttpClient<WeatherForecastClient>(
    client => client.BaseAddress = new
Uri("https://api.contoso.com/v1.0"))
    .AddHttpMessageHandler<BaseAddressAuthorizationMessageHandler>();

```

In a component that fetches weather data:

razor

```

@inject WeatherForecastClient Client

...

protected override async Task OnInitializedAsync()

```

```
{  
    forecasts = await Client.GetForecastAsync();  
}
```

Configure the `HttpClient` handler

The handler can be further configured with [ConfigureHandler](#) for outbound HTTP requests.

In the following example, [HttpClientFactoryServiceCollectionExtensions.AddHttpClient](#) is an extension in [Microsoft.Extensions.Http](#). Add the package to an app that doesn't already reference it.

ⓘ Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at [Package consumption workflow \(NuGet documentation\)](#). Confirm correct package versions at [NuGet.org](#) [↗].

In the `Program` file:

C#

```
builder.Services.AddHttpClient<WeatherForecastClient>(
    client => client.BaseAddress = new
Uri("https://api.contoso.com/v1.0"))
    .AddHttpMessageHandler(sp =>
sp.GetRequiredService<AuthorizationMessageHandler>())
    .ConfigureHandler(
    authorizedUrls: new [] { "https://api.contoso.com/v1.0" },
    scopes: new[] { "example.read", "example.write" }));
```

In the preceding code, the scopes `example.read` and `example.write` are generic examples not meant to reflect valid scopes for any particular provider.

Unauthenticated or unauthorized web API requests in an app with a secure default client

An app that ordinarily uses a secure default [HttpClient](#) can also make unauthenticated or unauthorized web API requests by configuring a named [HttpClient](#).

In the following example, [HttpClientFactoryServiceCollectionExtensions.AddHttpClient](#) is an extension in [Microsoft.Extensions.Http](#). Add the package to an app that doesn't already reference it.

ⓘ Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at [Package consumption workflow \(NuGet documentation\)](#). Confirm correct package versions at [NuGet.org](#) [↗].

In the `Program` file:

C#

```
builder.Services.AddHttpClient("WebAPI.NoAuthenticationClient",  
    client => client.BaseAddress = new Uri("https://api.contoso.com/v1.0"));
```

The preceding registration is in addition to the existing secure default [HttpClient](#) registration.

A component creates the [HttpClient](#) from the [IHttpClientFactory](#) ([Microsoft.Extensions.Http](#) [↗] package) to make unauthenticated or unauthorized requests:

razor

```
@inject IHttpClientFactory ClientFactory  
  
...  
  
@code {  
    protected override async Task OnInitializedAsync()  
    {  
        var client =  
        ClientFactory.CreateClient("WebAPI.NoAuthenticationClient");  
  
        var examples = await client.GetFromJsonAsync<ExampleType[]>(  
            "ExampleNoAuthentication");  
  
        ...  
    }  
}
```

ⓘ Note

The controller in the server API, `ExampleNoAuthenticationController` for the preceding example, isn't marked with the `[Authorize]` attribute.

The decision whether to use a secure client or an insecure client as the default `HttpClient` instance is up to the developer. One way to make this decision is to consider the number of authenticated versus unauthenticated endpoints that the app contacts. If the majority of the app's requests are to secure API endpoints, use the authenticated `HttpClient` instance as the default. Otherwise, register the unauthenticated `HttpClient` instance as the default.

An alternative approach to using the `IHttpClientFactory` is to create a `typed client` for unauthenticated access to anonymous endpoints.

Request additional access tokens

Access tokens can be manually obtained by calling `IAccessTokenProvider.RequestAccessToken`. In the following example, an additional scope is required by an app for the default `HttpClient`. The Microsoft Authentication Library (MSAL) example configures the scope with `MsalProviderOptions`:

In the `Program` file:

```
C#

builder.Services.AddMsalAuthentication(options =>
{
    ...

    options.ProviderOptions.AdditionalScopesToConsent.Add("{CUSTOM SCOPE 1}");
    options.ProviderOptions.AdditionalScopesToConsent.Add("{CUSTOM SCOPE 2}");
})
```

The `{CUSTOM SCOPE 1}` and `{CUSTOM SCOPE 2}` placeholders in the preceding example are custom scopes.

⚠ Note

`AdditionalScopesToConsent` isn't able to provision delegated user permissions for Microsoft Graph via the Microsoft Entra ID consent UI when a user first uses an app

registered in Microsoft Azure. For more information, see [Use Graph API with ASP.NET Core Blazor WebAssembly](#).

The `IAccessTokenProvider.RequestAccessToken` method provides an overload that allows an app to provision an access token with a given set of scopes.

In a Razor component:

razor

```
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject IAccessTokenProvider TokenProvider

...

var tokenResult = await TokenProvider.RequestAccessToken(
    new AccessTokenRequestOptions
    {
        Scopes = new[] { "{CUSTOM SCOPE 1}", "{CUSTOM SCOPE 2}" }
    });

if (tokenResult.TryGetToken(out var token))
{
    ...
}
```

The `{CUSTOM SCOPE 1}` and `{CUSTOM SCOPE 2}` placeholders in the preceding example are custom scopes.

`AccessTokenResult.TryGetToken` returns:

- `true` with the `token` for use.
- `false` if the token isn't retrieved.

Cross-Origin Resource Sharing (CORS)

When sending credentials (authorization cookies/headers) on CORS requests, the `Authorization` header must be allowed by the CORS policy.

The following policy includes configuration for:

- Request origins (`http://localhost:5000`, `https://localhost:5001`).
- Any method (verb).
- `Content-Type` and `Authorization` headers. To allow a custom header (for example, `x-custom-header`), list the header when calling `WithHeaders`.

- Credentials set by client-side JavaScript code (`credentials` property set to `include`).

C#

```
app.UseCors(policy =>
    policy.WithOrigins("http://localhost:5000", "https://localhost:5001")
        .AllowAnyMethod()
        .WithHeaders(HeaderNames.ContentType, HeaderNames.Authorization,
            "x-custom-header")
        .AllowCredentials());
```

For more information, see [Enable Cross-Origin Requests \(CORS\) in ASP.NET Core](#) and the sample app's HTTP Request Tester component (`Components/HttpRequestTester.razor`).

Handle token request errors

When a single-page application (SPA) authenticates a user using OpenID Connect (OIDC), the authentication state is maintained locally within the SPA and in the Identity Provider (IP) in the form of a session cookie that's set as a result of the user providing their credentials.

The tokens that the IP emits for the user typically are valid for short periods of time, about one hour normally, so the client app must regularly fetch new tokens. Otherwise, the user would be logged-out after the granted tokens expire. In most cases, OIDC clients are able to provision new tokens without requiring the user to authenticate again thanks to the authentication state or "session" that is kept within the IP.

There are some cases in which the client can't get a token without user interaction, for example, when for some reason the user explicitly logs out from the IP. This scenario occurs if a user visits `https://login.microsoftonline.com` and logs out. In these scenarios, the app doesn't know immediately that the user has logged out. Any token that the client holds might no longer be valid. Also, the client isn't able to provision a new token without user interaction after the current token expires.

These scenarios aren't specific to token-based authentication. They are part of the nature of SPAs. An SPA using cookies also fails to call a server API if the authentication cookie is removed.

When an app performs API calls to protected resources, you must be aware of the following:

- To provision a new access token to call the API, the user might be required to authenticate again.
- Even if the client has a token that seems to be valid, the call to the server might fail because the token was revoked by the user.

When the app requests a token, there are two possible outcomes:

- The request succeeds, and the app has a valid token.
- The request fails, and the app must authenticate the user again to obtain a new token.

When a token request fails, you need to decide whether you want to save any current state before you perform a redirection. Several approaches exist to store state with increasing levels of complexity:

- Store the current page state in session storage. During the [OnInitializedAsync lifecycle method](#) ([OnInitializedAsync](#)), check if state can be restored before continuing.
- Add a query string parameter and use that as a way to signal the app that it needs to re-hydrate the previously saved state.
- Add a query string parameter with a unique identifier to store data in session storage without risking collisions with other items.

Save app state before an authentication operation with session storage

The following example shows how to:

- Preserve state before redirecting to the login page.
- Recover the previous state after authentication using a query string parameter.

razor

```
...
@using System.Text.Json
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject IAccessTokenProvider TokenProvider
@inject IJSRuntime JS
@inject NavigationManager Navigation

<EditForm Model="User" OnSubmit="OnSaveAsync">
    <label>
        First Name:
        <InputText @bind-Value="User!.Name" />
    </label>
```

```

<label>
    Last Name:
    <InputText @bind-Value="User!.LastName" />
</label>
<button type="submit">Save User</button>
</EditForm>

@code {
    public Profile User { get; set; } = new Profile();

    protected override async Task OnInitializedAsync()
    {
        var currentQuery = new Uri(Navigation.Uri).Query;

        if (currentQuery.Contains("state=resumeSavingProfile"))
        {
            var user = await JS.InvokeAsync<string>
("sessionStorage.getItem",
    "resumeSavingProfile");

            if (!string.IsNullOrEmpty(user))
            {
                User = JsonSerializer.Deserialize<Profile>(user);
            }
        }
    }

    public async Task OnSaveAsync()
    {
        var http = new HttpClient();
        http.BaseAddress = new Uri(Navigation.BaseUri);

        var resumeUri = Navigation.Uri + $"?state=resumeSavingProfile";

        var tokenResult = await TokenProvider.RequestAccessToken(
            new AccessTokenRequestOptions
            {
                ReturnUrl = resumeUri
            });

        if (tokenResult.TryGetToken(out var token))
        {
            http.DefaultRequestHeaders.Add("Authorization",
                $"Bearer {token.Value}");
            await http.PostAsJsonAsync("Save", User);
        }
        else
        {
            await JS.InvokeVoidAsync("sessionStorage.setItem",
                "resumeSavingProfile", JsonSerializer.Serialize(User));
            Navigation.NavigateTo(tokenResult.InteractiveRequestUrl);
        }
    }

    public class Profile

```



```
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
}
}
```

Save app state before an authentication operation with session storage and a state container

During an authentication operation, there are cases where you want to save the app state before the browser is redirected to the IP. This can be the case when you're using a state container and want to restore the state after the authentication succeeds. You can use a custom authentication state object to preserve app-specific state or a reference to it and restore that state after the authentication operation successfully completes. The following example demonstrates the approach.

A state container class is created in the app with properties to hold the app's state values. In the following example, the container is used to maintain the counter value of the default [Blazor project template's](#) `Counter` component (`Counter.razor`). Methods for serializing and deserializing the container are based on [System.Text.Json](#).

C#

```
using System.Text.Json;

public class StateContainer
{
    public int CounterValue { get; set; }

    public string GetStateForLocalStorage() =>
        JsonSerializer.Serialize(this);

    public void SetStateFromLocalStorage(string locallyStoredState)
    {
        var deserializedState =
            JsonSerializer.Deserialize<StateContainer>(locallyStoredState);

        CounterValue = deserializedState.CounterValue;
    }
}
```

The `Counter` component uses the state container to maintain the `currentCount` value outside of the component:

razor

```
@page "/counter"
@inject StateContainer State

<h1>Counter</h1>

<p>Current count: @currentCount</p>

<button class="btn btn-primary" @onclick="IncrementCount">Click me</button>

@code {
    private int currentCount = 0;

    protected override void OnInitialized()
    {
        if (State.CounterValue > 0)
        {
            currentCount = State.CounterValue;
        }
    }

    private void IncrementCount()
    {
        currentCount++;
        State.CounterValue = currentCount;
    }
}
```

Create an `ApplicationAuthenticationState` from `RemoteAuthenticationState`. Provide an `Id` property, which serves as an identifier for the locally-stored state.

`ApplicationAuthenticationState.cs`:

C#

```
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

public class ApplicationAuthenticationState : RemoteAuthenticationState
{
    public string? Id { get; set; }
}
```

The `Authentication` component (`Authentication.razor`) saves and restores the app's state using local session storage with the `StateContainer` serialization and deserialization methods, `GetStateForLocalStorage` and `SetStateFromLocalStorage`:

razor

```

@page "/authentication/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject IJSRuntime JS
@inject StateContainer State

<RemoteAuthenticatorViewCore Action="Action"

TAuthenticationState="ApplicationAuthenticationState"
    AuthenticationState="AuthenticationState"
    OnLogInSucceeded="RestoreState"
    OnLogOutSucceeded="RestoreState" />

@code {
    [Parameter]
    public string? Action { get; set; }

    public ApplicationAuthenticationState AuthenticationState { get; set; }
    =
        new ApplicationAuthenticationState();

    protected override async Task OnInitializedAsync()
    {
        if
(RemoteAuthenticationActions.IsAction(RemoteAuthenticationActions.LogIn,
        Action) ||

RemoteAuthenticationActions.IsAction(RemoteAuthenticationActions.LogOut,
        Action))
        {
            AuthenticationState.Id = Guid.NewGuid().ToString();

            await JS.InvokeVoidAsync("sessionStorage.setItem",
                AuthenticationState.Id, State.GetStateForLocalStorage());
        }

        private async Task RestoreState(ApplicationAuthenticationState state)
        {
            if (state.Id != null)
            {
                var locallyStoredState = await JS.InvokeAsync<string>(
                    "sessionStorage.getItem", state.Id);

                if (locallyStoredState != null)
                {
                    State.SetStateFromLocalStorage(locallyStoredState);
                    await JS.InvokeVoidAsync("sessionStorage.removeItem",
state.Id);
                }
            }
        }
    }
}

```

This example uses Microsoft Entra (ME-ID) for authentication. In the `Program` file:

- The `ApplicationAuthenticationState` is configured as the Microsoft Authentication Library (MSAL) `RemoteAuthenticationState` type.
- The state container is registered in the service container.

C#

```
builder.Services.AddMsalAuthentication<ApplicationAuthenticationState>
(options =>
{
    builder.Configuration.Bind("AzureAd",
options.ProviderOptions.Authentication);
});

builder.Services.AddSingleton<StateContainer>();
```

Customize app routes

The [Microsoft.AspNetCore.Components.WebAssembly.Authentication](#) library uses the routes shown in the following table for representing different authentication states.

 Expand table

Route	Purpose
<code>authentication/login</code>	Triggers a sign-in operation.
<code>authentication/login-callback</code>	Handles the result of any sign-in operation.
<code>authentication/login-failed</code>	Displays error messages when the sign-in operation fails for some reason.
<code>authentication/logout</code>	Triggers a sign-out operation.
<code>authentication/logout-callback</code>	Handles the result of a sign-out operation.
<code>authentication/logout-failed</code>	Displays error messages when the sign-out operation fails for some reason.
<code>authentication/logged-out</code>	Indicates that the user has successfully logout.
<code>authentication/profile</code>	Triggers an operation to edit the user profile.
<code>authentication/register</code>	Triggers an operation to register a new user.

The routes shown in the preceding table are configurable via `RemoteAuthenticationOptions<TRemoteAuthenticationProviderOptions>.AuthenticationPaths`. When setting options to provide custom routes, confirm that the app has a route that handles each path.

In the following example, all of the paths are prefixed with `/security`.

`Authentication` component (`Authentication.razor`):

razor

```
@page "/security/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action" />

@code{
    [Parameter]
    public string? Action { get; set; }
}
```

In the `Program` file:

C#

```
builder.Services.AddApiAuthorization(options => {
    options.AuthenticationPaths.LogInPath = "security/login";
    options.AuthenticationPaths.LogInCallbackPath = "security/login-
callback";
    options.AuthenticationPaths.LogInFailedPath = "security/login-failed";
    options.AuthenticationPaths.LogOutPath = "security/logout";
    options.AuthenticationPaths.LogOutCallbackPath = "security/logout-
callback";
    options.AuthenticationPaths.LogOutFailedPath = "security/logout-failed";
    options.AuthenticationPaths.LogOutSucceededPath = "security/logged-out";
    options.AuthenticationPaths.ProfilePath = "security/profile";
    options.AuthenticationPaths.RegisterPath = "security/register";
});
```

If the requirement calls for completely different paths, set the routes as described previously and render the `RemoteAuthenticatorView` with an explicit action parameter:

razor

```
@page "/register"

<RemoteAuthenticatorView Action="RemoteAuthenticationActions.Register" />
```

You're allowed to break the UI into different pages if you choose to do so.

Customize the authentication user interface

`RemoteAuthenticatorView` includes a default set of UI fragments for each authentication state. Each state can be customized by passing in a custom `RenderFragment`. To customize the displayed text during the initial login process, can change the `RemoteAuthenticatorView` as follows.

`Authentication` component (`Authentication.razor`):


```
razor

@page "/security/{action}"
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication

<RemoteAuthenticatorView Action="@Action">
    <LoggingIn>
        You are about to be redirected to https://login.microsoftonline.com.
    </LoggingIn>
</RemoteAuthenticatorView>

@code{
    [Parameter]
    public string? Action { get; set; }
}
```

The `RemoteAuthenticatorView` has one fragment that can be used per authentication route shown in the following table.

 Expand table

Route	Fragment
authentication/login	<LoggingIn>
authentication/login-callback	<CompletingLoggingIn>
authentication/login-failed	<LogInFailed>
authentication/logout	<LogOut>
authentication/logout-callback	<CompletingLogOut>
authentication/logout-failed	<LogOutFailed>
authentication/logged-out	<LogOutSucceeded>

Route	Fragment
authentication/profile	<UserProfile>
authentication/register	<Registering>

Customize the user

Users bound to the app can be customized.

Customize the user with a payload claim

In the following example, the app's authenticated users receive an `amr` claim for each of the user's authentication methods. The `amr` claim identifies how the subject of the token was authenticated in Microsoft identity platform v1.0 [payload claims](#). The example uses a custom user account class based on [RemoteUserAccount](#).

Create a class that extends the [RemoteUserAccount](#) class. The following example sets the `AuthenticationMethod` property to the user's array of `amr` JSON property values. `AuthenticationMethod` is populated automatically by the framework when the user is authenticated.

C#

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

public class CustomUserAccount : RemoteUserAccount
{
    [JsonPropertyName("amr")]
    public string[]? AuthenticationMethod { get; set; }
}
```

Create a factory that extends [AccountClaimsPrincipalFactory<TAccount>](#) to create claims from the user's authentication methods stored in

`CustomUserAccount.AuthenticationMethod`:

C#

```
using System.Security.Claims;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication.Internal;

public class CustomAccountFactory(NavigationManager navigation,
```

```

IAccessTokenProviderAccessor accessor)
: AccountClaimsPrincipalFactory<CustomUserAccount>(accessor)
{
    public override async ValueTask<ClaimsPrincipal> CreateUserAsync(
        CustomUserAccount account, RemoteAuthenticationUserOptions options)
    {
        var initialUser = await base.CreateUserAsync(account, options);

        if (initialUser.Identity != null &&
initialUser.Identity.IsAuthenticated)
        {
            var userIdentity = (ClaimsIdentity)initialUser.Identity;

            if (account.AuthenticationMethod is not null)
            {
                foreach (var value in account.AuthenticationMethod)
                {
                    userIdentity.AddClaim(new Claim("amr", value));
                }
            }

            return initialUser;
        }
    }
}

```

Register the `CustomAccountFactory` for the authentication provider in use. Any of the following registrations are valid:

- [AddOidcAuthentication](#):

```

C#

using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddOidcAuthentication<RemoteAuthenticationState,
    CustomUserAccount>(options =>
{
    ...
}))
.AddAccountClaimsPrincipalFactory<RemoteAuthenticationState,
    CustomUserAccount, CustomAccountFactory>();

```

- [AddMsalAuthentication](#):

```

C#

using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

```



```
...

builder.Services.AddMsalAuthentication<RemoteAuthenticationState,
    CustomUserAccount>(options =>
{
    ...
})
.AddAccountClaimsPrincipalFactory<RemoteAuthenticationState,
    CustomUserAccount, CustomAccountFactory>();
```

- [AddApiAuthorization](#):

```
C#

using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

...

builder.Services.AddApiAuthorization<RemoteAuthenticationState,
    CustomUserAccount>(options =>
{
    ...
})
.AddAccountClaimsPrincipalFactory<RemoteAuthenticationState,
    CustomUserAccount, CustomAccountFactory>();
```

ME-ID security groups and roles with a custom user account class

For an additional example that works with ME-ID security groups and ME-ID Administrator Roles and a custom user account class, see [ASP.NET Core Blazor WebAssembly with Microsoft Entra ID groups and roles](#).

Authenticate users to only call protected third party APIs

Authenticate the user with a client-side OAuth flow against the third-party API provider:

```
C#

builder.services.AddOidcAuthentication(options => { ... });
```

In this scenario:

- The server hosting the app doesn't play a role.
- APIs on the server can't be protected.
- The app can only call protected third-party APIs.

Authenticate users with a third-party provider and call protected APIs on the host server and the third party

Configure Identity with a third-party login provider. Obtain the tokens required for third-party API access and store them.

When a user logs in, Identity collects access and refresh tokens as part of the authentication process. At that point, there are a couple of approaches available for making API calls to third-party APIs.

Use a server access token to retrieve the third-party access token

Use the access token generated on the server to retrieve the third-party access token from a server API endpoint. From there, use the third-party access token to call third-party API resources directly from Identity on the client.

We don't recommend this approach. This approach requires treating the third-party access token as if it were generated for a public client. In OAuth terms, the public app doesn't have a client secret because it can't be trusted to store secrets safely, and the access token is produced for a confidential client. A confidential client is a client that has a client secret and is assumed to be able to safely store secrets.

- The third-party access token might be granted additional scopes to perform sensitive operations based on the fact that the third-party emitted the token for a more trusted client.
- Similarly, refresh tokens shouldn't be issued to a client that isn't trusted, as doing so gives the client unlimited access unless other restrictions are put into place.

Make API calls from the client to the server API in order to call third-party APIs

Make an API call from the client to the server API. From the server, retrieve the access token for the third-party API resource and issue whatever call is necessary.

We recommend this approach. While this approach requires an extra network hop through the server to call a third-party API, it ultimately results in a safer experience:

- The server can store refresh tokens and ensure that the app doesn't lose access to third-party resources.
- The app can't leak access tokens from the server that might contain more sensitive permissions.

Use OpenID Connect (OIDC) v2.0 endpoints

The authentication library and [Blazor project templates](#) use OpenID Connect (OIDC) v1.0 endpoints. To use a v2.0 endpoint, configure the JWT Bearer [JwtBearerOptions.Authority](#) option. In the following example, ME-ID is configured for v2.0 by appending a `v2.0` segment to the [Authority](#) property:

C#

```
using Microsoft.AspNetCore.Authentication.JwtBearer;

...

builder.Services.Configure<JwtBearerOptions>(
    JwtBearerDefaults.AuthenticationScheme,
    options =>
    {
        options.Authority += "/v2.0";
    });
```

Alternatively, the setting can be made in the app settings (`appsettings.json`) file:

JSON

```
{
  "Local": {
    "Authority": "https://login.microsoftonline.com/common/oauth2/v2.0/",
    ...
  }
}
```

If tacking on a segment to the authority isn't appropriate for the app's OIDC provider, such as with non-ME-ID providers, set the [Authority](#) property directly. Either set the property in [JwtBearerOptions](#) or in the app settings file (`appsettings.json`) with the `Authority` key.

The list of claims in the ID token changes for v2.0 endpoints. Microsoft documentation on the changes has been retired, but guidance on the claims in an ID token is available in the [ID token claims reference](#).

Configure and use gRPC in components

To configure a Blazor WebAssembly app to use the [ASP.NET Core gRPC framework](#):

- Enable gRPC-Web on the server. For more information, see [gRPC-Web in ASP.NET Core gRPC apps](#).
- Register gRPC services for the app's message handler. The following example configures the app's authorization message handler to use the [GreeterClient service from the gRPC tutorial](#) (the `Program` file).

ⓘ Note

Prerendering is enabled by default in Blazor Web Apps, so you must account for the component rendering first from the server and then from the client. Any prerendered state should flow to the client so that it can be reused. For more information, see [Prerender ASP.NET Core Razor components](#).

C#

```
using System.Net.Http;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Grpc.Net.Client;
using Grpc.Net.Client.Web;

...

builder.Services.AddScoped(sp =>
{
    var baseAddressMessageHandler =
        sp.GetRequiredService<BaseAddressAuthorizationMessageHandler>();
    baseAddressMessageHandler.InnerHandler = new HttpClientHandler();
    var grpcWebHandler =
        new GrpcWebHandler(GrpcWebMode.GrpcWeb, baseAddressMessageHandler);
    var channel =
        GrpcChannel.ForAddress(builder.HostEnvironment.BaseAddress,
            new GrpcChannelOptions { HttpClient = grpcWebHandler });

    return new Greeter.GreeterClient(channel);
});
```

A component in the client app can make gRPC calls using the gRPC client (`Grpc.razor`):

razor

```
@page "/grpc"
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]
@inject Greeter.GreeterClient GreeterClient

<h1>Invoke gRPC service</h1>

<p>
```

```
<input @bind="name" placeholder="Type your name" />
<button @onclick="GetGreeting" class="btn btn-primary">Call gRPC
service</button>
</p>
```

Server response: @serverResponse

```
@code {
    private string name = "Bert";
    private string? serverResponse;

    private async Task GetGreeting()
    {
        try
        {
            var request = new HelloRequest { Name = name };
            var reply = await GreeterClient.SayHelloAsync(request);
            serverResponse = reply.Message;
        }
        catch (Grpc.Core.RpcException ex)
            when (ex.Status.DebugException is
                AccessTokenNotAvailableException tokenEx)
        {
            tokenEx.Redirect();
        }
    }
}
```

To use the `Status.DebugException` property, use [Grpc.Net.Client](#) version 2.30.0 or later.

For more information, see [gRPC-Web in ASP.NET Core gRPC apps](#).

Replace the `AuthenticationService` implementation

The following subsections explain how to replace:

- Any JavaScript `AuthenticationService` implementation.
- The Microsoft Authentication Library for JavaScript (`MSAL.js`).

Replace any JavaScript `AuthenticationService` implementation

Create a JavaScript library to handle your custom authentication details.

 **Warning**

The guidance in this section is an implementation detail of the default [RemoteAuthenticationService<TRemoteAuthenticationState, TAccount, TProviderOptions>](#). The TypeScript code in this section applies specifically to ASP.NET Core in .NET 7 and is subject to change without notice in upcoming releases of ASP.NET Core.

TypeScript

```
// .NET makes calls to an AuthenticationService object in the Window.
declare global {
    interface Window { AuthenticationService: AuthenticationService }
}

export interface AuthenticationService {
    // Init is called to initialize the AuthenticationService.
    public static init(settings: UserManagerSettings &
AuthorizeServiceSettings, logger: any) : Promise<void>;

    // Gets the currently authenticated user.
    public static getUser() : Promise<{[key: string] : string }>;

    // Tries to get an access token silently.
    public static getAccessToken(options: AccessTokenRequestOptions) :
Promise<AccessTokenResult>;

    // Tries to sign in the user or get an access token interactively.
    public static signIn(context: AuthenticationContext) :
Promise<AuthenticationResult>;

    // Handles the sign-in process when a redirect is used.
    public static async completeSignIn(url: string) :
Promise<AuthenticationResult>;

    // Signs the user out.
    public static signOut(context: AuthenticationContext) :
Promise<AuthenticationResult>;

    // Handles the signout callback when a redirect is used.
    public static async completeSignOut(url: string) :
Promise<AuthenticationResult>;
}

// The rest of these interfaces match their C# definitions.

export interface AccessTokenRequestOptions {
    scopes: string[];
    returnUrl: string;
}

export interface AccessTokenResult {
    status: AccessTokenResultStatus;
    token?: AccessToken;
```

```

}

export interface AccessToken {
  value: string;
  expires: Date;
  grantedScopes: string[];
}

export enum AccessTokenResultStatus {
  Success = 'Success',
  RequiresRedirect = 'RequiresRedirect'
}

export enum AuthenticationResultStatus {
  Redirect = 'Redirect',
  Success = 'Success',
  Failure = 'Failure',
  OperationCompleted = 'OperationCompleted'
};

export interface AuthenticationResult {
  status: AuthenticationResultStatus;
  state?: unknown;
  message?: string;
}

export interface AuthenticationContext {
  state?: unknown;
  interactiveRequest: InteractiveAuthenticationRequest;
}

export interface InteractiveAuthenticationRequest {
  scopes?: string[];
  additionalRequestParameters?: { [key: string]: any };
};

```

You can import the library by removing the original `<script>` tag and adding a `<script>` tag that loads the custom library. The following example demonstrates replacing the default `<script>` tag with one that loads a library named `CustomAuthenticationService.js` from the `wwwroot/js` folder.

In `wwwroot/index.html` before the Blazor script (`_framework/blazor.webassembly.js`) inside the closing `</body>` tag:

diff

```

- <script
src="_content/Microsoft.Authentication.WebAssembly.Msal/AuthenticationService.js"></script>
+ <script src="js/CustomAuthenticationService.js"></script>

```

For more information, see [AuthenticationService.ts](#) in the [dotnet/aspnetcore GitHub repository](#).

ⓘ Note

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see [How to select a version tag of ASP.NET Core source code \(dotnet/AspNetCore.Docs #26205\)](#).

Replace the Microsoft Authentication Library for JavaScript (MSAL.js)

If an app requires a custom version of the [Microsoft Authentication Library for JavaScript \(MSAL.js\)](#), perform the following steps:

1. Confirm the system has the latest developer .NET SDK or obtain and install the latest developer SDK from [.NET Core SDK: Installers and Binaries](#). Configuration of internal NuGet feeds isn't required for this scenario.
2. Set up the `dotnet/aspnetcore` GitHub repository for development following the documentation at [Build ASP.NET Core from Source](#). Fork and clone or download a ZIP archive of the [dotnet/aspnetcore GitHub repository](#).
3. Open the `src/Components/WebAssembly/Authentication.Msal/src/Interop/package.json` file and set the desired version of `@azure/msal-browser`. For a list of released versions, visit the [@azure/msal-browser npm website](#) and select the **Versions** tab.
4. Build the `Authentication.Msal` project in the `src/Components/WebAssembly/Authentication.Msal/src` folder with the `yarn build` command in a command shell.
5. If the app uses [compressed assets \(Brotli/Gzip\)](#), compress the `Interop/dist/Release/AuthenticationService.js` file.
6. Copy the `AuthenticationService.js` file and compressed versions (`.br` / `.gz`) of the file, if produced, from the `Interop/dist/Release` folder into the app's `publish/wwwroot/_content/Microsoft.Authentication.WebAssembly.Msal` folder in the app's published assets.

Pass custom provider options

Define a class for passing the data to the underlying JavaScript library.

Important

The class's structure must match what the library expects when the JSON is serialized with [System.Text.Json](#).

The following example demonstrates a `ProviderOptions` class with [JsonPropertyName](#) attributes matching a hypothetical custom provider library's expectations:

C#

```
public class ProviderOptions
{
    public string? Authority { get; set; }
    public string? MetadataUrl { get; set; }

    [JsonPropertyName("client_id")]
    public string? ClientId { get; set; }

    public IList<string> DefaultScopes { get; set; } = [ "openid", "profile"
];

    [JsonPropertyName("redirect_uri")]
    public string? RedirectUri { get; set; }

    [JsonPropertyName("post_logout_redirect_uri")]
    public string? PostLogoutRedirectUri { get; set; }

    [JsonPropertyName("response_type")]
    public string? ResponseType { get; set; }

    [JsonPropertyName("response_mode")]
    public string? ResponseMode { get; set; }
}
```

Register the provider options within the DI system and configure the appropriate values:

C#

```
builder.Services.AddRemoteAuthentication<RemoteAuthenticationState,
RemoteUserAccount,
    ProviderOptions>(options => {
    options.ProviderOptions.Authority = "...";
    options.ProviderOptions.MetadataUrl = "...";
    options.ProviderOptions.ClientId = "...";
    options.ProviderOptions.DefaultScopes = [ "openid", "profile",
"myApi" ];
    options.ProviderOptions.RedirectUri =
```

```
"https://localhost:5001/authentication/login-callback";
    options.ProviderOptions.PostLogoutRedirectUri =
"https://localhost:5001/authentication/logout-callback";
    options.ProviderOptions.ResponseType = "...";
    options.ProviderOptions.ResponseMode = "...";
});
```

The preceding example sets redirect URIs with regular string literals. The following alternatives are available:

- [TryCreate](#) using `IWebAssemblyHostEnvironment.BaseAddress`:

C#

```
Uri.TryCreate(
    $"{builder.HostEnvironment.BaseAddress}authentication/login-
callback",
    UriKind.Absolute, out var redirectUri);
options.RedirectUri = redirectUri;
```

- [Host builder configuration](#):

C#

```
options.RedirectUri = builder.Configuration["RedirectUri"];
```

wwwroot/appsettings.json:

JSON

```
{
  "RedirectUri": "https://localhost:5001/authentication/login-callback"
}
```

Additional resources

- [Use Graph API with ASP.NET Core Blazor WebAssembly](#)
- [Cookie-based request credentials \(*Call web API* article\)](#)
- [HttpClient and HttpRequestMessage with Fetch API request options \(*Call web API* article\)](#)

Microsoft Entra (ME-ID) groups, Administrator Roles, and App Roles

Article • 10/21/2024

This article explains how to configure Blazor WebAssembly to use Microsoft Entra ID (ME-ID) groups and roles.

ME-ID provides several authorization approaches that can be combined with ASP.NET Core Identity:

- Groups
 - Security
 - Microsoft 365
 - Distribution
- Roles
 - ME-ID built-in Administrator Roles
 - App Roles

The guidance in this article applies to the Blazor WebAssembly ME-ID deployment scenarios described in the following articles:

- [Standalone with Microsoft Accounts](#)
- [Standalone with ME-ID](#)

The examples in this article take advantage of new .NET/C# features. When using the examples with .NET 7 or earlier, minor modifications are required. However, the text and code examples that pertain to interacting with ME-ID and Microsoft Graph are the same for all versions of ASP.NET Core.

Sample app

Access the sample app, named `BlazorWebAssemblyEntraGroupsAndRoles`, through the latest version folder from the repository's root with the following link. The sample is provided for .NET 8 or later. See the sample app's `README` file for steps on how to run the app.

The sample app includes a `UserClaims` component for displaying a user's claims. The `UserData` component displays the user's basic account properties.

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

Prerequisite

The guidance in this article implements the Microsoft Graph API per the *Graph SDK* guidance in [Use Graph API with ASP.NET Core Blazor WebAssembly](#). Follow the *Graph SDK* implementation guidance to configure the app and test it to confirm that the app can obtain Graph API data for a test user account. Additionally, see the [Graph API article's security article cross-links](#) to review Microsoft Graph security concepts.

When testing with the Graph SDK locally, we recommend using a new in-private/incognito browser session for each test to prevent lingering cookies from interfering with tests. For more information, see [Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID](#).

ME-ID app registration online tools

This article refers to the [Azure portal](#) throughout when prompting you to configure the app's ME-ID app registration, but the [Microsoft Entra Admin Center](#) is also a viable option for managing ME-ID app registrations. Either interface can be used, but the guidance in this article specifically covers gestures for the Azure portal.

Scopes

Permissions and *scopes* mean the same thing and are used interchangeably in security documentation and the Azure portal. Unless the text is referring to the Azure portal, this article uses *scope/scopes* when referring to Graph permissions.

Scopes are case insensitive, so `User.Read` is the same as `user.read`. Feel free to use either format, but we recommend a consistent choice across application code.

To permit [Microsoft Graph API](#) calls for user profile, role assignment, and group membership data, the app is configured with the ***delegated*** `User.Read` scope (`https://graph.microsoft.com/User.Read`) in the Azure portal because access to read user data is determined by the scopes granted (delegated) to individual users. This scope is required in addition to the scopes required in ME-ID deployment scenarios described by the articles listed earlier (*Standalone with Microsoft Accounts* or *Standalone with ME-ID*).

Additional required scopes include:

- ***Delegated*** `RoleManagement.Read.Directory` scope (`https://graph.microsoft.com/RoleManagement.Read.Directory`): Allows the app to

read the role-based access control (RBAC) settings for your company's directory, on behalf of the signed-in user. This includes reading directory role templates, directory roles, and memberships. Directory role memberships are used to create `directoryRole` claims in the app for ME-ID built-in Administrator Roles. Admin consent is required.

- **Delegated** `AdministrativeUnit.Read.All` scope (<https://graph.microsoft.com/AdministrativeUnit.Read.All>): Allows the app to read administrative units and administrative unit membership on behalf of the signed-in user. These memberships are used to create `administrativeUnit` claims in the app. Admin consent is required.

For more information, see [Overview of permissions and consent in the Microsoft identity platform](#) and [Overview of Microsoft Graph permissions](#).

Custom user account

Assign users to ME-ID security groups and ME-ID Administrator Roles in the Azure portal.

The examples in this article:

- Assume that a user is assigned to the ME-ID *Billing Administrator* role in the Azure portal ME-ID tenant for authorization to access server API data.
- Use [authorization policies](#) to control access within the app.

Extend `RemoteUserAccount` to include properties for:

- `Roles`: ME-ID App Roles array (covered in the [App Roles](#) section)
- `oid`: Immutable [object identifier claim \(oid\)](#) (uniquely identifies a user within and across tenants)

`CustomUserAccount.cs`:

C#

```
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;

namespace BlazorWebAssemblyEntraGroupsAndRoles;

public class CustomUserAccount : RemoteUserAccount
{
    [JsonPropertyName("roles")]
    public List<string>? Roles { get; set; }
}
```

```
[JsonPropertyName("oid")]
public string? Oid { get; set; }
}
```

Add a package reference to app for [Microsoft.Graph](#).

ⓘ Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at [Package consumption workflow \(NuGet documentation\)](#). Confirm correct package versions at [NuGet.org](#).

Add the Graph SDK utility classes and configuration in the *Graph SDK* guidance of the [Use Graph API with ASP.NET Core Blazor WebAssembly](#) article. Specify the `User.Read`, `RoleManagement.Read.Directory`, and `AdministrativeUnit.Read.All` scopes for the access token as the article shows in its example `wwwroot/appsettings.json` file.

Add the following custom user account factory to the app. The custom user factory is used to establish:

- App Role claims (`role`) (covered in the [App Roles](#) section).
- Example user profile data claims for the user's mobile phone number (`mobilePhone`) and office location (`officeLocation`).
- ME-ID Administrator Role claims (`directoryRole`).
- ME-ID Administrative Unit claims (`administrativeUnit`).
- ME-ID Group claims (`directoryGroup`).
- An [ILogger](#) (`logger`) for convenience in case you wish to log information or errors.

`CustomAccountFactory.cs`:

C#

```
using System.Security.Claims;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication.Internal;
using Microsoft.Graph;
using Microsoft.Kiota.Abstractions.Authentication;

namespace BlazorWebAssemblyEntraGroupsAndRoles;

public class CustomAccountFactory(IAccessTokenProviderAccessor accessor,
```

```

        IServiceProvider serviceProvider, ILogger<CustomAccountFactory>
logger,
        IConfiguration config)
    : AccountClaimsPrincipalFactory<CustomUserAccount>(accessor)
{
    private readonly ILogger<CustomAccountFactory> logger = logger;
    private readonly IServiceProvider serviceProvider = serviceProvider;
    private readonly string? baseUrl = string.Join("/",
        config.GetSection("MicrosoftGraph")["BaseUrl"],
        config.GetSection("MicrosoftGraph")["Version"]);

    public override async ValueTask<ClaimsPrincipal> CreateUserAsync(
        CustomUserAccount account,
        RemoteAuthenticationUserOptions options)
    {
        var initialUser = await base.CreateUserAsync(account, options);

        if (initialUser.Identity is not null &&
            initialUser.Identity.IsAuthenticated)
        {
            var userIdentity = initialUser.Identity as ClaimsIdentity;

            if (userIdentity is not null && !string.IsNullOrEmpty(baseUrl)
&&
                account.Oid is not null)
            {
                {
                    account?.Roles?.ForEach((role) =>
                    {
                        userIdentity.AddClaim(new Claim("role", role));
                    });

                    try
                    {
                        var client = new GraphServiceClient(
                            new HttpClient(),
                            serviceProvider
                                .GetRequiredService<IAuthenticationProvider>(),
                            baseUrl);

                        var user = await client.Me.GetAsync();

                        if (user is not null)
                        {
                            userIdentity.AddClaim(new Claim("mobilephone",
                                user.MobilePhone ?? "(000) 000-0000"));
                            userIdentity.AddClaim(new Claim("officelocation",
                                user.OfficeLocation ?? "Not set"));
                        }

                        var memberOf = client.Users[account?.Oid].MemberOf;

                        var graphDirectoryRoles = await
memberOf.GraphDirectoryRole.GetAsync();

                        if (graphDirectoryRoles?.Value is not null)

```

```

        {
            foreach (var entry in graphDirectoryRoles.Value)
            {
                if (entry.RoleTemplateId is not null)
                {
                    userIdentity.AddClaim(
                        new Claim("directoryRole",
entry.RoleTemplateId));
                }
            }
        }

        var graphAdministrativeUnits = await
memberOf.GraphAdministrativeUnit.GetAsync();

        if (graphAdministrativeUnits?.Value is not null)
        {
            foreach (var entry in
graphAdministrativeUnits.Value)
            {
                if (entry.Id is not null)
                {
                    userIdentity.AddClaim(
                        new Claim("administrativeUnit",
entry.Id));
                }
            }
        }

        var graphGroups = await memberOf.GraphGroup.GetAsync();

        if (graphGroups?.Value is not null)
        {
            foreach (var entry in graphGroups.Value)
            {
                if (entry.Id is not null)
                {
                    userIdentity.AddClaim(
                        new Claim("directoryGroup", entry.Id));
                }
            }
        }
    }
    catch (AccessTokenNotAvailableException exception)
    {
        exception.Redirect();
    }
}

return initialUser;
}
}

```


The preceding code:

- Doesn't include transitive memberships. If the app requires direct and transitive group membership claims, replace the `MemberOf` property (`IUserMemberOfCollectionWithReferencesRequestBuilder`) with `TransitiveMemberOf` (`IUserTransitiveMemberOfCollectionWithReferencesRequestBuilder`).
- Sets GUID values in `directoryRole` claims are ME-ID Administrator Role [Template IDs](#) (`Microsoft.Graph.Models.DirectoryRole.RoleTemplateId`). Template IDs are stable identifiers for creating user authorization policies in apps, which is covered later in this article. Don't use `entry.Id` for directory role claim values, as they aren't stable across tenants.

Next, configure the MSAL authentication to use the custom user account factory.

Confirm that the `Program` file uses the

[Microsoft.AspNetCore.Components.WebAssembly.Authentication](#) namespace:

C#

```
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
```

Update the [AddMsalAuthentication](#) call to the following. Note that the Blazor framework's [RemoteUserAccount](#) is replaced by the app's `CustomUserAccount` for the MSAL authentication and account claims principal factory:

C#

```
builder.Services.AddMsalAuthentication<RemoteAuthenticationState,  
    CustomUserAccount>(options =>  
    {  
        builder.Configuration.Bind("AzureAd",  
            options.ProviderOptions.Authentication);  
        options.UserOptions.RoleClaim = "role";  
    })  
    .AddAccountClaimsPrincipalFactory<RemoteAuthenticationState,  
    CustomUserAccount,  
        CustomAccountFactory>();
```

Confirm the presence of the *Graph SDK* code in the `Program` file described by the [Use Graph API with ASP.NET Core Blazor WebAssembly](#) article:

C#

```
var baseUrl =  
    string.Join("/",
```

```

        builder.Configuration.GetSection("MicrosoftGraph")["BaseUrl"] ??
            "https://graph.microsoft.com",
        builder.Configuration.GetSection("MicrosoftGraph")["Version"] ??
            "v1.0");
var scopes = builder.Configuration.GetSection("MicrosoftGraph:Scopes")
    .Get<List<string>>() ?? [ "user.read" ];

builder.Services.AddGraphClient(baseUrl, scopes);

```

❗ Important

Confirm in the app's registration in the Azure portal that the following permissions are granted:

- User.Read
- RoleManagement.Read.Directory (Requires admin consent)
- AdministrativeUnit.Read.All (Requires admin consent)

Confirm that the `wwwroot/appsettings.json` configuration is correct per the *Graph SDK* guidance.

`wwwroot/appsettings.json`:

JSON

```

{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/{TENANT ID}",
    "ClientId": "{CLIENT ID}",
    "ValidateAuthority": true
  },
  "MicrosoftGraph": {
    "BaseUrl": "https://graph.microsoft.com",
    "Version": "v1.0",
    "Scopes": [
      "User.Read",
      "RoleManagement.Read.Directory",
      "AdministrativeUnit.Read.All"
    ]
  }
}

```

Provide values for the following placeholders from the app's ME-ID registration in the Azure portal:

- `{TENANT ID}`: The Directory (Tenant) Id GUID value.

- `{CLIENT ID}`: The Application (Client) Id GUID value.

Authorization configuration

Create a [policy](#) for each [App Role](#) (by role name), ME-ID built-in Administrator Role (by Role Template Id/GUID), or security group (by Object Id/GUID) in the `Program` file. The following example creates a policy for the ME-ID built-in *Billing Administrator* role:

C#

```
builder.Services.AddAuthorizationCore(options =>
{
    options.AddPolicy("BillingAdministrator", policy =>
        policy.RequireClaim("directoryRole",
            "b0f54661-2d74-4c50-afa3-1ec803f12efe"));
});
```

For the complete list of IDs (GUIDs) for ME-ID Administrator Roles, see [Role template IDs](#) in the ME-ID documentation. For an Azure security or O365 group ID (GUID), see the **Object Id** for the group in the Azure portal **Groups** pane of the app's registration. For more information on authorization policies, see [Policy-based authorization in ASP.NET Core](#).

In the following examples, the app uses the preceding policy to authorize the user.

The [AuthorizeView component](#) works with the policy:

razor

```
<AuthorizeView Policy="BillingAdministrator">
    <Authorized>
        <p>
            The user is in the 'Billing Administrator' ME-ID Administrator
            Role
            and can see this content.
        </p>
    </Authorized>
    <NotAuthorized>
        <p>
            The user is NOT in the 'Billing Administrator' role and sees
            this
            content.
        </p>
    </NotAuthorized>
</AuthorizeView>
```

Access to an entire component can be based on the policy using an [\[Authorize\] attribute directive](#) ([AuthorizeAttribute](#)):

razor

```
@page "/"
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize(Policy = "BillingAdministrator")]
```

If the user isn't authorized, they're redirected to the ME-ID sign-in page.

A policy check can also be [performed in code with procedural logic](#).

CheckPolicy.razor:

razor

```
@page "/checkpolicy"
@using Microsoft.AspNetCore.Authorization
@inject IAuthorizationService AuthorizationService

<h1>Check Policy</h1>

<p>This component checks a policy in code.</p>

<button @onclick="CheckPolicy">Check 'BillingAdministrator' policy</button>

<p>Policy Message: @policyMessage</p>

@code {
    private string policyMessage = "Check hasn't been made yet.";

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

    private async Task CheckPolicy()
    {
        var user = (await authenticationStateTask).User;

        if ((await AuthorizationService.AuthorizeAsync(user,
            "BillingAdministrator")).Succeeded)
        {
            policyMessage = "Yes! The 'BillingAdministrator' policy is met.";
        }
        else
        {
            policyMessage = "No! 'BillingAdministrator' policy is NOT met.";
        }
    }
}
```

Using the preceding approaches, you can also create policy-based access for security groups, where the GUID used for the policy matches the

App Roles

To configure the app in the Azure portal to provide App Roles membership claims, see [Add app roles to your application and receive them in the token](#) in the ME-ID documentation.

The following example assumes that the app is configured with two roles, and the roles are assigned to a test user:

- Admin
- Developer

Although you can't [assign roles to groups](#) without an ME-ID Premium account, you can assign roles to users and receive role claims for users with a standard Azure account. The guidance in this section doesn't require an ME-ID Premium account.

Take either of the following approaches add app roles in ME-ID:

- When working with the **default directory**, follow the guidance in [Add app roles to your application and receive them in the token](#) to create ME-ID roles.
- If you **aren't working with the default directory**, edit the app's manifest in the Azure portal to establish the app's roles manually in the `appRoles` entry of the manifest file. The following is an example `appRoles` entry that creates `Admin` and `Developer` roles. These example roles are used later at the component level to implement access restrictions:

Important

The following approach is only recommended for apps that aren't registered in the Azure account's default directory. For apps registered in the default directory, see the preceding bullet of this list.

JSON

```
"appRoles": [  
  {  
    "allowedMemberTypes": [  
      "User"  
    ],  
  },  
]
```

```

    "description": "Administrators manage developers.",
    "displayName": "Admin",
    "id": "{ADMIN GUID}",
    "isEnabled": true,
    "lang": null,
    "origin": "Application",
    "value": "Admin"
  },
  {
    "allowedMemberTypes": [
      "User"
    ],
    "description": "Developers write code.",
    "displayName": "Developer",
    "id": "{DEVELOPER GUID}",
    "isEnabled": true,
    "lang": null,
    "origin": "Application",
    "value": "Developer"
  }
],

```

For the `{ADMIN GUID}` and `{DEVELOPER GUID}` placeholders in the preceding example, you can generate GUIDs with an [online GUID generator](#) (Google search result for "guid generator") [↗](#).

To assign a role to a user (or group if you have a Premium tier Azure account):

1. Navigate to **Enterprise applications** in the ME-ID area of the [Azure portal](#) [↗](#).
2. Select the app. Select **Manage > Users and groups** from the sidebar.
3. Select the checkbox for one or more user accounts.
4. From the menu above the list of users, select **Edit assignment**.
5. For the **Select a role** entry, select **None selected**.
6. Choose a role from the list and use the **Select** button to select it.
7. Use the **Assign** button at the bottom of the screen to assign the role.

Multiple roles are assigned in the Azure portal by *re-adding a user* for each additional role assignment. Use the **Add user/group** button at the top of the list of users to re-add a user. Use the preceding steps to assign another role to the user. You can repeat this process as many times as needed to add additional roles to a user (or group).

The `CustomAccountFactory` shown in the [Custom user account](#) section is set up to act on a `role` claim with a JSON array value. Add and register the `CustomAccountFactory` in the app as shown in the [Custom user account](#) section. There's no need to provide code to remove the original `role` claim because it's automatically removed by the framework.

In the `Program` file, add or confirm the claim named `"role"` as the role claim for `ClaimsPrincipal.IsInRole` checks:

C#

```
builder.Services.AddMsalAuthentication(options =>
{
    ...

    options.UserOptions.RoleClaim = "role";
});
```

❗ Note

If you prefer to use the `directoryRoles` claim (ME-ID Administrator Roles), assign `"directoryRoles"` to the `RemoteAuthenticationUserOptions.RoleClaim`.

After you've completed the preceding steps to create and assign roles to users (or groups if you have a Premium tier Azure account) and implemented the `CustomAccountFactory` with the Graph SDK, as explained earlier in this article and in [Use Graph API with ASP.NET Core Blazor WebAssembly](#), you should see an `role` claim for each assigned role that a signed-in user is assigned (or roles assigned to groups that they are members of). Run the app with a test user to confirm the claims are present as expected. When testing with the Graph SDK locally, we recommend using a new in-private/incognito browser session for each test to prevent lingering cookies from interfering with tests. For more information, see [Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID](#).

Component authorization approaches are functional at this point. Any of the authorization mechanisms in components of the app can use the `Admin` role to authorize the user:

- [AuthorizeView component](#)

razor

```
<AuthorizeView Roles="Admin">
```

- [\[Authorize\] attribute directive \(AuthorizeAttribute\)](#)

razor

```
@attribute [Authorize(Roles = "Admin")]
```

- Procedural logic

```
C#  
  
var authState = await  
AuthenticationStateProvider.GetAuthenticationStateAsync();  
var user = authState.User;  
  
if (user.IsInRole("Admin")) { ... }
```

Multiple role tests are supported:

- Require that the user be in **either** the `Admin` or `Developer` role with the `AuthorizeView` component:

```
razor  
  
<AuthorizeView Roles="Admin, Developer">  
    ...  
</AuthorizeView>
```

- Require that the user be in **both** the `Admin` and `Developer` roles with the `AuthorizeView` component:

```
razor  
  
<AuthorizeView Roles="Admin">  
    <AuthorizeView Roles="Developer" Context="innerContext">  
        ...  
    </AuthorizeView>  
</AuthorizeView>
```

For more information on the `Context` for the inner `AuthorizeView`, see [ASP.NET Core Blazor authentication and authorization](#).

- Require that the user be in **either** the `Admin` or `Developer` role with the `[Authorize]` attribute:

```
razor  
  
@attribute [Authorize(Roles = "Admin, Developer")]
```


- Require that the user be in **both** the `Admin` and `Developer` roles with the `[Authorize]` attribute:

razor

```
@attribute [Authorize(Roles = "Admin")]
@attribute [Authorize(Roles = "Developer")]
```

- Require that the user be in **either** the `Admin` or `Developer` role with procedural code:

razor

```
@code {
    private async Task DoSomething()
    {
        var authState = await AuthenticationStateProvider
            .GetAuthenticationStateAsync();
        var user = authState.User;

        if (user.IsInRole("Admin") || user.IsInRole("Developer"))
        {
            ...
        }
        else
        {
            ...
        }
    }
}
```

- Require that the user be in **both** the `Admin` and `Developer` roles with procedural code by changing the **conditional OR** (`||`) to a **conditional AND** (`&&`) in the preceding example:

C#

```
if (user.IsInRole("Admin") && user.IsInRole("Developer"))
```

Multiple role tests are supported:

- Require that the user be in **either** the `Admin` or `Developer` role with the `[Authorize]` attribute:

C#

```
[Authorize(Roles = "Admin, Developer")]
```

- Require that the user be in **both** the `Admin` and `Developer` roles with the `[Authorize]` attribute:

C#

```
[Authorize(Roles = "Admin")]
[Authorize(Roles = "Developer")]
```

- Require that the user be in **either** the `Admin` or `Developer` role with procedural code:

C#

```
static readonly string[] scopeRequiredByApi = new string[] {
    "API.Access" };

...

[HttpGet]
public IEnumerable<ReturnType> Get()
{
    HttpContext.VerifyUserHasAnyAcceptedScope(scopeRequiredByApi);

    if (User.IsInRole("Admin") || User.IsInRole("Developer"))
    {
        ...
    }
    else
    {
        ...
    }

    return ...
}
```

- Require that the user be in **both** the `Admin` and `Developer` roles with procedural code by changing the **conditional OR (||)** to a **conditional AND (&&)** in the preceding example:

C#

```
if (User.IsInRole("Admin") && User.IsInRole("Developer"))
```

Because .NET string comparisons are case-sensitive, matching role names is also case-sensitive. For example, `Admin` (uppercase `A`) is not treated as the same role as `admin` (lowercase `a`).

Pascal case is typically used for role names (for example, `BillingAdministrator`), but the use of Pascal case isn't a strict requirement. Different casing schemes, such as camel case, kebab case, and snake case, are permitted. Using spaces in role names is also unusual but permitted. For example, `billing administrator` is an unusual role name format in .NET apps but valid.

Additional resources

- [Role template IDs \(ME-ID documentation\)](#)
- [groupMembershipClaims attribute \(ME-ID documentation\)](#)
- [Add app roles to your application and receive them in the token \(ME-ID documentation\)](#)
- [Application roles \(Azure documentation\)](#)
- [Claims-based authorization in ASP.NET Core](#)
- [Role-based authorization in ASP.NET Core](#)
- [ASP.NET Core Blazor authentication and authorization](#)

Use Graph API with ASP.NET Core Blazor WebAssembly

Article • 11/06/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).


This article explains how to use [Microsoft Graph](#) in Blazor WebAssembly apps, which enables apps to access Microsoft Cloud resources.

Two approaches are covered:

- **Graph SDK:** The [Microsoft Graph SDK](#) simplifies building high-quality, efficient, and resilient apps that access Microsoft Graph. Select the **Graph SDK** button at the top of this article to adopt this approach.
- **Named HttpClient with Graph API:** A [named HttpClient](#) can issue [Microsoft Graph API](#) requests directly to Microsoft Graph. Select the **Named HttpClient with Graph API** button at the top of this article to adopt this approach.

The guidance in this article isn't meant to replace the [Microsoft Graph documentation](#) and Azure security guidance in other Microsoft documentation sets. Assess the security guidance in the [Additional resources](#) section of this article before implementing Microsoft Graph in a production environment. Follow Microsoft's best practices to limit the vulnerabilities of your apps.

Additional approaches for working with Microsoft Graph and Blazor WebAssembly are provided by the following Microsoft Graph and Azure samples:

- [Microsoft Graph sample Blazor WebAssembly app](#) : The sample adopts a factory-based approach with the [Microsoft Graph SDK](#) to obtain Office 365 data. The sample uses a default [HttpClient](#) to make Graph client requests. If you need to use the default [HttpClient](#) with the base address of the app (`BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)`) for other purposes, for example to load data from the web root of the app, consider refactoring the Graph client factory to use a [named HttpClient](#) dedicated to Graph requests.

- [ASP.NET Core 8.0 Blazor WebAssembly | standalone app | user sign-in, protected web API access \(Microsoft Graph\) | Microsoft identity platform](#) [↗](#): The sample is a [Progressive Web Application \(PWA\)](#) that uses an authorization message handler and [HttpClient](#) to obtain Graph data. As with the preceding sample, this sample also uses a default [HttpClient](#) to make Graph client requests. Instead of using the Microsoft Graph SDK, the sample retrieves user account data as JSON via a [Microsoft Graph API](#) request in a component. This is a similar technique to the **Named HttpClient with Graph API** approach in this article (use the button at the top of this article to see the guidance), except that this article's guidance uses a [named HttpClient](#) dedicated to Graph requests.

To provide feedback on either of the preceding two samples, open an issue on the sample's GitHub repository. If you're opening an issue for the Azure sample, provide a link to the sample in your opening comment because the Azure sample repository ([Azure-Samples](#)) contains many samples. Describe the problem in detail and include sample code as needed. Place a minimal app into GitHub that reproduces the problem or error. Be sure to remove Azure account configuration data from the sample before you commit it to the public repository.

To provide feedback or seek assistance with this article or ASP.NET Core, see [ASP.NET Core Blazor fundamentals](#).

Important

The scenarios described in this article apply to using Microsoft Entra (ME-ID) as the identity provider, not AAD B2C. Using Microsoft Graph with a client-side Blazor WebAssembly app and the AAD B2C identity provider isn't supported at this time because the app would require a client secret, which can't be secured in the client-side Blazor app. For an AAD B2C standalone Blazor WebAssembly app use Graph API, create a backend server (web) API to access Graph API on behalf of users. The client-side app authenticates and authorizes users to [call the web API](#) to securely access Microsoft Graph and return data to the client-side Blazor app from your server-based web API. The client secret is safely maintained in the server-based web API, not in the Blazor app on the client. **Never store a client secret in a client-side Blazor app.**

The examples in this article take advantage of new .NET/C# features. When using the examples with .NET 7 or earlier, minor modifications are required. However, the text and code examples that pertain to interacting with Microsoft Graph are the same for all versions of ASP.NET Core.

The following guidance applies to Microsoft Graph v4. If you're upgrading an app from SDK v4 to v5, see the [Microsoft Graph .NET SDK v5 changelog and upgrade guide](#).

The Microsoft Graph SDK for use in Blazor apps is called the *Microsoft Graph .NET Client Library*.

The Graph SDK examples require the following package references in the standalone Blazor WebAssembly app. The first two packages are already referenced if the app has been enabled for MSAL authentication, for example when creating the app by following the guidance in [Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID](#).

- [Microsoft.AspNetCore.Components.WebAssembly.Authentication](#)
- [Microsoft.Authentication.WebAssembly.Msal](#)
- [Microsoft.Extensions.Http](#)
- [Microsoft.Graph](#)

ⓘ Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at [Package consumption workflow \(NuGet documentation\)](#). Confirm correct package versions at [NuGet.org](#).

In the Azure portal, grant delegated permissions (scopes)[†] for Microsoft Graph data that the app should be able to access on behalf of a user. For the example in this article, the app's registration should include delegated permission to read user data (Microsoft.Graph > User.Read scope in **API permissions**, Type: Delegated). The User.Read scope allows users to sign in to the app and allows the app to read the profile and company information of signed-in users. For more information, see [Overview of permissions and consent in the Microsoft identity platform](#) and [Overview of Microsoft Graph permissions](#).

[†]*Permissions* and *scopes* mean the same thing and are used interchangeably in security documentation and the Azure portal. Unless the text is referring to the Azure portal, this article uses *scope/scopes* when referring to Graph permissions.

Scopes are case insensitive, so User.Read is the same as user.read. Feel free to use either format, but we recommend a consistent choice across application code.

After adding the Microsoft Graph API scopes to the app's registration in the Azure portal, add the following app settings configuration to the `wwwroot/appsettings.json` file in the app, which includes the Graph base URL with the Microsoft Graph version and

scopes. In the following example, the `User.Read` scope is specified for the examples in later sections of this article. Scopes aren't case sensitive.

JSON

```
"MicrosoftGraph": {
  "BaseUrl": "https://graph.microsoft.com",
  "Version": "{VERSION}",
  "Scopes": [
    "user.read"
  ]
}
```

In the preceding example, the `{VERSION}` placeholder is the version of the Microsoft Graph API (for example: `v1.0`).

The following is an example of a complete `wwwroot/appsettings.json` configuration file for an app that uses ME-ID as its identity provider, where reading user data (`user.read` scope) is specified for Microsoft Graph:

JSON

```
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/{TENANT ID}",
    "ClientId": "{CLIENT ID}",
    "ValidateAuthority": true
  },
  "MicrosoftGraph": {
    "BaseUrl": "https://graph.microsoft.com",
    "Version": "v1.0",
    "Scopes": [
      "user.read"
    ]
  }
}
```

In the preceding example, the `{TENANT ID}` placeholder is the Directory (tenant) ID, and the `{CLIENT ID}` placeholder is the Application (client) ID. For more information, see [Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID](#).

Add the following `GraphClientExtensions` class to the standalone app. The scopes are provided to the `Scopes` property of the `AccessTokenRequestOptions` in the `AuthenticateRequestAsync` method. The `IHttpProvider.OverallTimeout` is extended from the default value of 100 seconds to 300 seconds to give the `HttpClient` more time to receive a response from Microsoft Graph.

When an access token isn't obtained, the following code doesn't set a Bearer authorization header for Graph requests.

GraphClientExtensions.cs:

C#

```
using System.Net.Http.Headers;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Microsoft.Authentication.WebAssembly.Msal.Models;
using Microsoft.Graph;

namespace BlazorSample;

internal static class GraphClientExtensions
{
    public static IServiceCollection AddGraphClient(
        this IServiceCollection services, string? baseUrl, List<string>?
scopes)
    {
        if (string.IsNullOrEmpty(baseUrl) || scopes?.Count == 0)
        {
            return services;
        }

        services.Configure<RemoteAuthenticationOptions<MsalProviderOptions>>
(
            options =>
            {
                {
                    scopes?.ForEach((scope) =>
                    {
options.ProviderOptions.DefaultAccessTokenScopes.Add(scope);
                    });
                });

                services.AddScoped<IAuthenticationProvider,
GraphAuthenticationProvider>();

                services.AddScoped<IHttpProvider, HttpClientHttpProvider>(sp =>
                    new HttpClientHttpProvider(new HttpClient()));

                services.AddScoped(sp =>
                {
                    return new GraphServiceClient(
                        baseUrl,
                        sp.GetRequiredService<IAuthenticationProvider>(),
                        sp.GetRequiredService<IHttpProvider>());
                });

                return services;
            }
        }
```



```

private class GraphAuthenticationProvider(IAccessTokenProvider
tokenProvider,
    IConfiguration config) : IAuthenticationProvider
{
    private readonly IConfiguration config = config;

    public IAccessTokenProvider TokenProvider { get; } = tokenProvider;

    public async Task AuthenticateRequestAsync(HttpRequestMessage
request)
    {
        var result = await TokenProvider.RequestAccessToken(
            new AccessTokenRequestOptions()
            {
                Scopes =
config.GetSection("MicrosoftGraph:Scopes").Get<string[]>()
            });

        if (result.TryGetToken(out var token))
        {
            request.Headers.Authorization ??= new
AuthenticationHeaderValue(
                "Bearer", token.Value);
        }
    }
}

private class HttpClientHttpProvider(HttpClient client) : IHttpProvider
{
    private readonly HttpClient client = client;

    public ISerializer Serializer { get; } = new Serializer();

    public TimeSpan OverallTimeout { get; set; } =
TimeSpan.FromSeconds(300);

    public Task<HttpResponseMessage> SendAsync(HttpRequestMessage
request)
    {
        return client.SendAsync(request);
    }

    public Task<HttpResponseMessage> SendAsync(HttpRequestMessage
request,
        HttpCompletionOption completionOption,
        CancellationToken cancellationToken)
    {
        return client.SendAsync(request, completionOption,
cancellationToken);
    }

    public void Dispose()
    {
    }
}

```

```
}  
}
```

❗ Important

See the [DefaultAccessTokenScopes versus AdditionalScopesToConsent](#) section for an explanation on why the preceding code uses [DefaultAccessTokenScopes](#) to add the scopes rather than [AdditionalScopesToConsent](#).

In the `Program` file, add the Graph client services and configuration with the `AddGraphClient` extension method:

C#

```
var baseUrl = string.Join("/",  
    builder.Configuration.GetSection("MicrosoftGraph")["BaseUrl"] ??  
        "https://graph.microsoft.com",  
    builder.Configuration.GetSection("MicrosoftGraph")["Version"] ??  
        "v1.0");  
var scopes = builder.Configuration.GetSection("MicrosoftGraph:Scopes")  
    .Get<List<string>>() ?? [ "user.read" ];  
  
builder.Services.AddGraphClient(baseUrl, scopes);
```

Call Graph API from a component using the Graph SDK

The following `UserData` component uses an injected `GraphServiceClient` to obtain the user's ME-ID profile data and display their mobile phone number. For any test user that you create in ME-ID, make sure that you give the user's ME-ID profile a mobile phone number in the Azure portal.

`UserData.razor`:

razor

```
@page "/user-data"  
@using Microsoft.AspNetCore.Authorization  
@using Microsoft.Graph  
@attribute [Authorize]  
@inject GraphServiceClient Client  
  
<PageTitle>User Data</PageTitle>
```

```

<h1>Microsoft Graph User Data</h1>

@if (!string.IsNullOrEmpty(user?.MobilePhone))
{
    <p>Mobile Phone: @user.MobilePhone</p>
}

@code {
    private Microsoft.Graph.User? user;

    protected override async Task OnInitializedAsync()
    {
        var request = Client.Me.Request();
        user = await request.GetAsync();
    }
}

```

Add a link to the component's page in the `NavMenu` component (`Layout/NavMenu.razor`):

```

razor

<div class="nav-item px-3">
    <NavLink class="nav-link" href="user-data">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
        User Data
    </NavLink>
</div>

```

Tip

To add users to an app, see the [Assign users to an app registration with or without app roles](#) section.

When testing with the Graph SDK locally, we recommend using a new InPrivate/incognito browser session for each test to prevent lingering cookies from interfering with tests. For more information, see [Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID](#).

Customize user claims using the Graph SDK

In the following example, the app creates mobile phone number and office location claims for a user from their ME-ID user profile's data. The app must have the `User.Read` Graph API scope configured in ME-ID. Any test users for this scenario must have a mobile phone number and office location in their ME-ID profile, which can be added via the Azure portal.

In the following custom user account factory:

- An [ILogger](#) (`logger`) is included for convenience in case you wish to log information or errors in the `CreateUserAsync` method.
- In the event that an [AccessTokenNotAvailableException](#) is thrown, the user is redirected to the identity provider to sign into their account. Additional or different actions can be taken when requesting an access token fails. For example, the app can log the [AccessTokenNotAvailableException](#) and create a support ticket for further investigation.
- The framework's [RemoteUserAccount](#) represents the user's account. If the app requires a custom user account class that extends [RemoteUserAccount](#), swap your custom user account class for [RemoteUserAccount](#) in the following code.

`CustomAccountFactory.cs`:

C#

```
using System.Security.Claims;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using Microsoft.AspNetCore.Components.WebAssembly.Authentication.Internal;
using Microsoft.Graph;

public class CustomAccountFactory(IAccessTokenProviderAccessor accessor,
    IServiceProvider serviceProvider, ILogger<CustomAccountFactory>
logger)
    : AccountClaimsPrincipalFactory<RemoteUserAccount>(accessor)
{
    private readonly ILogger<CustomAccountFactory> logger = logger;
    private readonly IServiceProvider serviceProvider = serviceProvider;

    public override async ValueTask<ClaimsPrincipal> CreateUserAsync(
        RemoteUserAccount account,
        RemoteAuthenticationUserOptions options)
    {
        var initialUser = await base.CreateUserAsync(account, options);

        if (initialUser.Identity is not null &&
            initialUser.Identity.IsAuthenticated)
        {
            var userIdentity = initialUser.Identity as ClaimsIdentity;

            if (userIdentity is not null)
            {
                try
                {
                    var client = ActivatorUtilities
                        .CreateInstance<GraphServiceClient>
                        (serviceProvider);
                    var request = client.Me.Request();
                    var user = await request.GetAsync();
                }
            }
        }
    }
}
```

```

        if (user is not null)
        {
            userIdentity.AddClaim(new Claim("mobilephone",
                user.MobilePhone ?? "(000) 000-0000"));
            userIdentity.AddClaim(new Claim("officelocation",
                user.OfficeLocation ?? "Not set"));
        }
    }
    catch (AccessTokenNotAvailableException exception)
    {
        exception.Redirect();
    }
}

return initialUser;
}
}

```

Configure the MSAL authentication to use the custom user account factory.

Confirm that the `Program` file uses the

[Microsoft.AspNetCore.Components.WebAssembly.Authentication](#) namespace:

C#

```
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
```

The example in this section builds on the approach of reading the base URL with version and scopes from app configuration via the `MicrosoftGraph` section in `wwwroot/appsettings.json` file. The following lines should already be present in the `Program` file from following the guidance earlier in this article:

C#

```

var baseUrl = string.Join("/",
    builder.Configuration.GetSection("MicrosoftGraph")["BaseUrl"] ??
        "https://graph.microsoft.com",
    builder.Configuration.GetSection("MicrosoftGraph")["Version"] ??
        "v1.0");
var scopes = builder.Configuration.GetSection("MicrosoftGraph:Scopes")
    .Get<List<string>>() ?? [ "user.read" ];

builder.Services.AddGraphClient(baseUrl, scopes);

```

In the `Program` file, find the call to the [AddMsalAuthentication](#) extension method. Update the code to the following, which includes a call to

[AddAccountClaimsPrincipalFactory](#) that adds an account claims principal factory with the `CustomAccountFactory`.

If the app uses a custom user account class that extends [RemoteUserAccount](#), swap the custom user account class for [RemoteUserAccount](#) in the following code.

C#

```
builder.Services.AddMsalAuthentication<RemoteAuthenticationState,  
    RemoteUserAccount>(options =>  
    {  
        builder.Configuration.Bind("AzureAd",  
            options.ProviderOptions.Authentication);  
    })  
    .AddAccountClaimsPrincipalFactory<RemoteAuthenticationState,  
    RemoteUserAccount,  
        CustomAccountFactory>();
```

You can use the following `UserClaims` component to study the user's claims after the user authenticates with ME-ID:

`UserClaims.razor`:

razor

```
@page "/user-claims"  
@using System.Security.Claims  
@using Microsoft.AspNetCore.Authorization  
@attribute [Authorize]  
@inject AuthenticationStateProvider AuthenticationStateProvider  
  
<h1>User Claims</h1>  
  
@if (claims.Any())  
{  
    <ul>  
        @foreach (var claim in claims)  
        {  
            <li>@claim.Type: @claim.Value</li>  
        }  
    </ul>  
}  
else  
{  
    <p>No claims found.</p>  
}  
  
@code {  
    private IEnumerable<Claim> claims = Enumerable.Empty<Claim>();  
  
    protected override async Task OnInitializedAsync()
```

```

{
    var authState = await AuthenticationStateProvider
        .GetAuthenticationStateAsync();
    var user = authState.User;

    claims = user.Claims;
}
}

```

Add a link to the component's page in the `NavMenu` component (`Layout/NavMenu.razor`):

```

razor

<div class="nav-item px-3">
    <NavLink class="nav-link" href="user-claims">
        <span class="bi bi-list-nested-nav-menu" aria-hidden="true"></span>
        User Claims
    </NavLink>
</div>

```

When testing with the Graph SDK locally, we recommend using a new InPrivate/incognito browser session for each test to prevent lingering cookies from interfering with tests. For more information, see [Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID](#).

Assign users to an app registration with or without app roles

You can add users to an app registration and assign roles to users with the following steps in the Azure portal.

To add a user, select **Users** from the ME-ID area of the Azure portal:

1. Select **New user** > **Create new user**.
2. Use the **Create user** template.
3. Supply the user's information in the **Identity** area.
4. You can generate an initial password or assign an initial password that the user changes when they sign in for the first time. If you use the password generated by the portal, make a note of it now.
5. Select **Create** to create the user. When **Create new user** interface closes, select **Refresh** to update the user list and show the new user.
6. For the examples in this article, assign a mobile phone number to the new user by selecting their name from the users list, selecting **Properties**, and editing the contact information to provide a mobile phone number.

To assign users to the app *without app roles*:

1. In the ME-ID area of the Azure portal, open **Enterprise applications**.
2. Select the app from the list.
3. Select **Users and groups**.
4. Select **Add user/group**.
5. Select a user.
6. Select the **Assign** button.

To assign users to the app *with app roles*:

1. Add roles to the app's registration in the Azure portal following the guidance in [ASP.NET Core Blazor WebAssembly with Microsoft Entra ID groups and roles](#).
2. In the ME-ID area of the Azure portal, open **Enterprise applications**.
3. Select the app from the list.
4. Select **Users and groups**.
5. Select **Add user/group**.
6. Select a user and select their role for accessing the app. Multiple roles are assigned to a user by repeating the process of adding the user to the app until all of the roles for a user are assigned. Users with multiple roles are listed once for each assigned role in the **Users and groups** list of users for the app.
7. Select the **Assign** button.

DefaultAccessTokenScopes versus **AdditionalScopesToConsent**

The examples in this article provision Graph API scopes with [DefaultAccessTokenScopes](#), not [AdditionalScopesToConsent](#).

[AdditionalScopesToConsent](#) isn't used because it's unable to provision Graph API scopes for users when they sign in to the app for the first time with MSAL via the Azure consent UI. When the user attempts to access Graph API for the first time with the Graph SDK, they're confronted with an exception:

```
Microsoft.Graph.Models.ODataErrors.ODataError: Access token is empty.
```

After a user provisions Graph API scopes provided via [DefaultAccessTokenScopes](#), the app can use [AdditionalScopesToConsent](#) for a subsequent user sign in. However, changing app code makes no sense for a production app that requires the periodic addition of new users with delegated Graph scopes or adding new delegated Graph API scopes to the app.

The preceding discussion of how to provision scopes for Graph API access when the user first signs into the app only applies to:

- Apps that adopt the Graph SDK.
- Apps that use a named [HttpClient](#) for Graph API access that asks users to consent to Graph scopes on their first sign in to the app.

When using a named [HttpClient](#) that doesn't ask users to consent to Graph scopes on their first sign in, users are redirected to the Azure consent UI for Graph API scopes consent *when they first request access to Graph API* via the [DelegatingHandler](#) of the preconfigured, named [HttpClient](#). When Graph scopes aren't consented initially with the named [HttpClient](#) approach, neither [DefaultAccessTokenScopes](#) nor [AdditionalScopesToConsent](#) are called by the app. For more information, see the [named HttpClient coverage in this article](#).

Additional resources

General guidance

- [Microsoft Graph documentation](#)
- [Microsoft Graph sample Blazor WebAssembly app](#) [↗](#): This sample demonstrates how to use the Microsoft Graph .NET SDK to access data in Office 365 from Blazor WebAssembly apps.
- [Build .NET apps with Microsoft Graph tutorial](#) and [Microsoft Graph sample ASP.NET Core app](#) [↗](#): Although these resources don't directly apply to calling Graph from *client-side* Blazor WebAssembly apps, the ME-ID app configuration and Microsoft Graph coding practices in the linked resources are relevant for standalone Blazor WebAssembly apps and should be consulted for general best practices.

Security guidance

- [Microsoft Graph auth overview](#)
- [Overview of Microsoft Graph permissions](#)
- [Microsoft Graph permissions reference](#)
- [Overview of permissions and consent in the Microsoft identity platform](#)
- [Enhance security with the principle of least privilege](#)
- [Azure privilege escalation articles on the Internet \(Google search result\)](#) [↗](#)
- [Microsoft Security Best Practices: Securing privileged access](#)

Secure an ASP.NET Core Blazor Web App with Microsoft Entra ID

Article • 11/19/2024

This article describes how to secure a Blazor Web App with [Microsoft identity platform/Microsoft Identity Web packages](#) for [Microsoft Entra ID](#) [↗] using a sample app.

The following specification is covered:

- The Blazor Web App uses the [Auto render mode with global interactivity \(InteractiveAuto\)](#).
- The server project calls [AddAuthenticationStateSerialization](#) to add a server-side authentication state provider that uses [PersistentComponentState](#) to flow the authentication state to the client. The client calls [AddAuthenticationStateDeserialization](#) to deserialize and use the authentication state passed by the server. The authentication state is fixed for the lifetime of the WebAssembly application.
- The app uses [Microsoft Entra ID](#) [↗], based on [Microsoft Identity Web](#) packages.
- Automatic non-interactive token refresh is managed by the framework.
- The app uses server-side and client-side service abstractions to display generated weather data:
 - When rendering the `Weather` component on the server to display weather data, the component uses the `ServerWeatherForecaster` on the server to directly obtain weather data (not via a web API call).
 - When the `Weather` 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's Minimal API (`/weather-forecast`) for weather data. The Minimal API endpoint obtains the weather data from the `ServerWeatherForecaster` class and returns it to the client for rendering by the component.

Sample app

The sample app consists of two projects:

- `BlazorWebAppEntra`: Server-side project of the Blazor Web App, containing an example [Minimal API](#) endpoint for weather data.
- `BlazorWebAppEntra.Client`: Client-side project of the Blazor Web App.

Access sample apps through the latest version folder from the repository's root with the following link. The projects are in the `BlazorWebAppEntra` folder for .NET 9 or later.

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

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

The `BlazorWebAppEntra` project is the server-side project of the Blazor Web App.

The `BlazorWebAppEntra.http` file can be used for testing the weather data request. Note that the `BlazorWebAppEntra` 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](#).

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

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

If the user needs to log in or out during client-side rendering, a full page reload is initiated.

Configuration

This section explains how to configure the sample app.

`AddMicrosoftIdentityWebApp` from [Microsoft Identity Web](#) ([Microsoft.Identity.Web NuGet package](#) [API documentation](#)) is configured by the `AzureAd` section of the server project's `appsettings.json` file.

In the app's registration in the Entra or Azure portal, use a **Web** platform configuration with a **Redirect URI** of `https://localhost/signin-oidc` (a port isn't required). Confirm that **ID tokens** and access tokens under **Implicit grant and hybrid flows** are **not** selected. The OpenID Connect handler automatically requests the appropriate tokens using the code returned from the authorization endpoint.

Configure the app

In the server project's app settings file (`appsettings.json`), provide the app's `AzureAd` section configuration. Obtain the application (client) ID, tenant (publisher) domain, and directory (tenant) ID from the app's registration in the Entra or Azure portal:

JSON

```
"AzureAd": {  
  "CallbackPath": "/signin-oidc",  
  "ClientId": "{CLIENT ID}",  
  "Domain": "{DOMAIN}",  
  "Instance": "https://login.microsoftonline.com/",  
  "ResponseType": "code",  
  "TenantId": "{TENANT ID}"  
},
```

Placeholders in the preceding example:

- `{CLIENT ID}`: The application (client) ID.
- `{DOMAIN}`: The tenant (publisher) domain.
- `{TENANT ID}`: The directory (tenant) ID.

Example:

JSON

```
"AzureAd": {  
  "CallbackPath": "/signin-oidc",  
  "ClientId": "00001111-aaaa-2222-bbbb-3333cccc4444",  
  "Domain": "contoso.onmicrosoft.com",  
  "Instance": "https://login.microsoftonline.com/",  
  "ResponseType": "code",  
  "TenantId": "aaaabbbb-0000-cccc-1111-dddd2222eeee"  
},
```

The callback path (`CallbackPath`) must match the redirect URI (login callback path) configured when registering the application in the Entra or Azure portal. Paths are configured in the **Authentication** blade of the app's registration. The default value of `CallbackPath` is `/signin-oidc` for a registered redirect URI of `https://localhost/signin-oidc` (a port isn't required).

Warning

Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is *always insecure*. In test/staging and production

environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the [Secret Manager tool](#) is recommended for securing sensitive data. For more information, see [Securely maintain sensitive data and credentials](#).

Establish the client secret

Create a client secret in the app's Entra ID registration in the Entra or Azure portal (**Manage** > **Certificates & secrets** > **New client secret**). Use the **Value** of the new secret in the following guidance.

Use either or both of the following approaches to supply the client secret to the app:

- [Secret Manager tool](#): The Secret Manager tool stores private data on the local machine and is only used during local development.
- [Azure Key Vault](#): You can store the client secret in a key vault for use in any environment, including for the Development environment when working locally. Some developers prefer to use key vaults for staging and production deployments and use the [Secret Manager tool](#) for local development.

We strongly recommend that you avoid storing client secrets in project code or configuration files. Use secure authentication flows, such as either or both of the approaches in this section.

Secret Manager tool

The [Secret Manager tool](#) can store the server app's client secret under the configuration key `AzureAd:ClientSecret`.

The [sample app](#) hasn't been initialized for the Secret Manager tool. Use a command shell, such as the Developer PowerShell command shell in Visual Studio, to execute the following command. Before executing the command, change the directory with the `cd` command to the server project's directory. The command establishes a user secrets identifier (`<UserSecretsId>`) in the server app's project file, which is used internally by the tooling to track secrets for the app:

```
.NET CLI
```

```
dotnet user-secrets init
```

Execute the following command to set the client secret. The `{SECRET}` placeholder is the client secret obtained from the app's Entra registration:

```
.NET CLI
```

```
dotnet user-secrets set "AzureAd:ClientSecret" "{SECRET}"
```

If using Visual Studio, you can confirm that the secret is set by right-clicking the server project in **Solution Explorer** and selecting **Manage User Secrets**.

Azure Key Vault

[Azure Key Vault](#) provides a safe approach for providing the app's client secret to the app.

To create a key vault and set a client secret, see [About Azure Key Vault secrets \(Azure documentation\)](#), which cross-links resources to get started with Azure Key Vault. To implement the code in this section, record the key vault URI and the secret name from Azure when you create the key vault and secret. When you set the access policy for the secret in the **Access policies** panel:

- Only the **Get** secret permission is required.
- Select the application as the **Principal** for the secret.

Important

A key vault secret is created with an expiration date. Be sure to track when a key vault secret is going to expire and create a new secret for the app prior to that date passing.

The following `GetKeyVaultSecret` method retrieves a secret from a key vault. Add this method to the server project. Adjust the namespace (`BlazorSample.Helpers`) to match your project namespace scheme.

Helpers/AzureHelper.cs:

```
C#
```

```
using Azure;  
using Azure.Identity;
```

```

using Azure.Security.KeyVault.Secrets;

namespace BlazorSample.Helpers;

public static class AzureHelper
{
    public static string GetKeyVaultSecret(string tenantId, string vaultUri,
string secretName)
    {
        DefaultAzureCredentialOptions options = new()
        {
            // Specify the tenant ID to use the dev credentials when running
the app locally
            // in Visual Studio.
            VisualStudioTenantId = tenantId,
            SharedTokenCacheTenantId = tenantId
        };

        var client = new SecretClient(new Uri(vaultUri), new
DefaultAzureCredential(options));
        var secret = client.GetSecretAsync(secretName).Result;

        return secret.Value.Value;
    }
}

```

Where services are registered in the server project's `Program` file, obtain and apply the client secret using the following code:

C#

```

var tenantId = builder.Configuration.GetValue<string>("AzureAd:TenantId")!;
var vaultUri = builder.Configuration.GetValue<string>("AzureAd:VaultUri")!;
var secretName = builder.Configuration.GetValue<string>
("AzureAd:SecretName")!;

builder.Services.Configure<MicrosoftIdentityOptions>(
    OpenIdConnectDefaults.AuthenticationScheme,
    options =>
    {
        options.ClientSecret =
            AzureHelper.GetKeyVaultSecret(tenantId, vaultUri, secretName);
    });

```

If you wish to control the environment where the preceding code operates, for example to avoid running the code locally because you've opted to use the [Secret Manager tool](#) for local development, you can wrap the preceding code in a conditional statement that checks the environment:

C#

```
if (!context.HostingEnvironment.IsDevelopment())
{
    ...
}
```

In the `AzureAd` section of `appsettings.json`, add the following `VaultUri` and `SecretName` configuration keys and values:

JSON

```
"VaultUri": "{VAULT_URI}",
"SecretName": "{SECRET_NAME}"
```

In the preceding example:

- The `{VAULT_URI}` placeholder is the key vault URI. Include the trailing slash on the URI.
- The `{SECRET_NAME}` placeholder is the secret name.

Example:

JSON

```
"VaultUri": "https://contoso.vault.azure.net/",
"SecretName": "BlazorWebAppEntra"
```

Configuration is used to facilitate supplying dedicated key vaults and secret names based on the app's environmental configuration files. For example, you can supply different configuration values for `appsettings.Development.json` in development, `appsettings.Staging.json` when staging, and `appsettings.Production.json` for the production deployment. For more information, see [ASP.NET Core Blazor configuration](#).

Redirect to the home page on sign out

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 signing out. The following example shows how to set the app's home page as the return URL for sign-out operations.

The important changes to the `LogInOrOut` component are demonstrated in the following example. There's no need to provide a hidden field for the `ReturnUrl` set to the home page at `/` because that's the default path. `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 />
                <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-
hidden="true"></span>
                        Login
                    </a>
                </NotAuthorized>
            </AuthorizeView>
        </div>
```

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 cyberattackers 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 `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
 - Mozilla Firefox: Use `-private -url {URL}`, where the `{URL}` placeholder 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

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

<PageTitle>User Claims</PageTitle>

<h1>User Claims</h1>

@if (claims.Any())
{
    <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

- [Microsoft identity platform documentation](#)
- [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](#)
- [Service abstractions in Blazor Web Apps](#)

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

Article • 10/28/2024

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 without adopting the [Backend for Frontend \(BFF\) pattern](#). The BFF pattern is useful for making authenticated requests to external services. Change the article version selector to **OIDC with BFF pattern** if the app's specification calls 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 be used with Entra, Microsoft Identity Web, and hosted in Azure.
- Automatic non-interactive token refresh.
- Securely calls a (web) API in the server project for data.

Sample app

The sample app consists of two projects:

- `BlazorWebAppOidc`: Server-side project of the Blazor Web App, containing an example [Minimal API](#) endpoint for weather data.
- `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 `BlazorWebAppOidc` folder for .NET 8 or later.

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

Server-side Blazor Web App project

(BlazorWebApp0idc)

The BlazorWebApp0idc project is the server-side project of the Blazor Web App.

The BlazorWebApp0idc.http file can be used for testing the weather data request. Note that the BlazorWebApp0idc 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 or 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.

Establish the client secret

⚠ Warning

Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is *always insecure*. In test/staging and production

environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the [Secret Manager tool](#) is recommended for securing sensitive data. For more information, see [Securely maintain sensitive data and credentials](#).

For local development testing, use the [Secret Manager tool](#) to store the server app's client secret under the configuration key

```
Authentication:Schemes:MicrosoftOidc:ClientSecret.
```

ⓘ Note

If the app uses Microsoft Entra ID or Azure AD B2C, create a client secret in the app's registration in the Entra or Azure portal (**Manage > Certificates & secrets > New client secret**). Use the **Value** of the new secret in the following guidance.

The [sample app](#) hasn't been initialized for the Secret Manager tool. Use a command shell, such as the Developer PowerShell command shell in Visual Studio, to execute the following command. Before executing the command, change the directory with the `cd` command to the server project's directory. The command establishes a user secrets identifier (`<UserSecretsId>` in the server app's project file):

```
.NET CLI
```

```
dotnet user-secrets init
```

Execute the following command to set the client secret. The `{SECRET}` placeholder is the client secret obtained from the app's registration:

```
.NET CLI
```

```
dotnet user-secrets set "Authentication:Schemes:MicrosoftOidc:ClientSecret"
"{SECRET}"
```

If using Visual Studio, you can confirm the secret is set by right-clicking the server project in **Solution Explorer** and selecting **Manage User Secrets**.

Configure the app

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. This property is set to `false` to reduce the size of the final authentication cookie.

C#

```
oidcOptions.SaveTokens = false;
```

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

C#

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

- [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/aaaabbbb-0000-cccc-1111-dddd2222eeee/v2.0/` (uses Tenant ID `aaaabbbb-0000-cccc-1111-dddd2222eeee`)
- Client Id ({CLIENT ID}): `00001111-aaaa-2222-bbbb-3333cccc4444`

C#

```
oidcOptions.Authority = "https://login.microsoftonline.com/aaaabbbb-0000-cccc-1111-dddd2222eeee/v2.0/";  
oidcOptions.ClientId = "00001111-aaaa-2222-bbbb-3333cccc4444";
```

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#

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

- [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 sets the role claim type to "roles," which is appropriate for [Microsoft Entra ID \(ME-ID\)](#). Consult your identity provider's documentation for more information.

ⓘ **Note**

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

C#

```
oidcOptions.MapInboundClaims = false;  
oidcOptions.TokenValidationParameters.NameClaimType = "name";  
oidcOptions.TokenValidationParameters.RoleClaimType = "roles";
```

- 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 when using Microsoft Entra ID. Most other OIDC providers require a correct port.

- **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 when using Microsoft Entra ID. Most other OIDC providers require a correct port.

ⓘ 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 when using Microsoft Entra ID. Most other OIDC providers require a correct port.

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 server project calls [AddAuthenticationStateSerialization](#) to add a server-side authentication state provider that uses [PersistentComponentState](#) to flow the authentication state to the client. The client calls [AddAuthenticationStateDeserialization](#) to deserialize and use the authentication state passed by the server. The authentication state is fixed for the lifetime of the WebAssembly application.
- An example requests to the Blazor Web App for weather data is handled by a Minimal API endpoint (`/weather-forecast`) in the `Program` file (`Program.cs`). The endpoint requires authorization by calling [RequireAuthorization](#). For any controllers that you add to the project, add the [\[Authorize\]](#) attribute to the controller or action.
- The app securely calls a (web) API in the server project for weather data:

- When rendering the `Weather` component on the server, the component uses the `ServerWeatherForecaster` on the server to obtain weather data directly (not via a web API call).
- 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 obtains the weather data from the `ServerWeatherForecaster` and returns the data to the client.

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 (`BlazorWebApp0idc.Client`)

The `BlazorWebApp0idc.Client` project is the client-side project of the Blazor Web App.

The client calls [AddAuthenticationStateDeserialization](#) to deserialize and use the authentication state passed by 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.

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. There's no need to provide a hidden field for the `ReturnUrl` set to the home page at `/` because that's the default path. `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 />
                <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-
hidden="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 Incognito/private 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`.

Application roles for apps not registered with Microsoft Entra (ME-ID)

This section pertains to apps that don't use [Microsoft Entra ID \(ME-ID\)](#) as the identity provider. For apps registered with ME-ID, see the [Application roles for apps registered with Microsoft Entra \(ME-ID\)](#) section.

Configure the role claim type (`TokenValidationParameters.RoleClaimType`) in the `OpenIdConnectOptions` of `Program.cs`:

C#

```
oidcOptions.TokenValidationParameters.RoleClaimType = "{ROLE CLAIM TYPE}";
```

For many OIDC identity providers, the role claim type is `role`. Check your identity provider's documentation for the correct value.

Replace the `UserInfo` class in the `BlazorWebAppOidc.Client` project with the following class.

`UserInfo.cs`:

C#

```
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;
using System.Security.Claims;

namespace BlazorWebAppOidc.Client;

// Add properties to this class and update the server and client
// AuthenticationStateProviders to expose more information about
// the authenticated user to the client.
public sealed class UserInfo
{
    public required string UserId { get; init; }
    public required string Name { get; init; }
    public required string[] Roles { get; init; }

    public const string UserIdClaimType = "sub";
    public const string NameClaimType = "name";
    private const string RoleClaimType = "role";
}
```

```

public static UserInfo FromClaimsPrincipal(ClaimsPrincipal principal) =>
    new()
    {
        UserId = GetRequiredClaim(principal, UserIdClaimType),
        Name = GetRequiredClaim(principal, NameClaimType),
        Roles = principal.FindAll(RoleClaimType).Select(c => c.Value)
            .ToArray(),
    };

public ClaimsPrincipal ToClaimsPrincipal() =>
    new(new ClaimsIdentity(
        Roles.Select(role => new Claim(RoleClaimType, role))
            .Concat([
                new Claim(UserIdClaimType, UserId),
                new Claim(NameClaimType, Name),
            ]),
        authenticationType: nameof(UserInfo),
        nameType: NameClaimType,
        roleType: RoleClaimType));

private static string GetRequiredClaim(ClaimsPrincipal principal,
    string claimType) =>
    principal.FindFirst(claimType)?.Value ??
    throw new InvalidOperationException(
        $"Could not find required '{claimType}' claim.");
}

```

At this point, Razor components can adopt [role-based and policy-based authorization](#). Application roles appear in `role` claims, one claim per role.

Application roles for apps registered with Microsoft Entra (ME-ID)

Use the guidance in this section to implement application roles, ME-ID security groups, and ME-ID built-in administrator roles for apps using [Microsoft Entra ID \(ME-ID\)](#).

The approach described in this section configures ME-ID to send groups and roles in the authentication cookie header. When users are only a member of a few security groups and roles, the following approach should work for most hosting platforms without running into a problem where headers are too long, for example with IIS hosting that has a default header length limit of 16 KB (`MaxRequestBytes`). If header length is a problem due to high group or role membership, we recommend not following the guidance in this section in favor of implementing [Microsoft Graph](#) to obtain a user's groups and roles from ME-ID separately, an approach that doesn't inflate the size of the authentication cookie. For more information, see [Bad Request - Request Too Long - IIS Server \(dotnet/aspnetcore #57545\)](#).

Configure the role claim type ([TokenValidationParameters.RoleClaimType](#)) in [OpenIdConnectOptions](#) of `Program.cs`. Set the value to `roles`:

C#

```
oidcOptions.TokenValidationParameters.RoleClaimType = "roles";
```

Although you can't [assign roles to groups](#) without an ME-ID Premium account, you can assign roles to users and receive role claims for users with a standard Azure account. The guidance in this section doesn't require an ME-ID Premium account.

When working with the default directory, follow the guidance in [Add app roles to your application and receive them in the token \(ME-ID documentation\)](#) to configure and assign roles. If you aren't working with the default directory, edit the app's manifest in the Azure portal to establish the app's roles manually in the `appRoles` entry of the manifest file. For more information, see [Configure the role claim \(ME-ID documentation\)](#).

A user's Azure security groups arrive in `groups` claims, and a user's built-in ME-ID administrator role assignments arrive in [well-known IDs \(wids\) claims](#). Values for both claim types are GUIDs. When received by the app, these claims can be used to establish [role and policy authorization in Razor components](#).

In the app's manifest in the Azure portal, set the [groupMembershipClaims attribute](#) to `All`. A value of `All` results in ME-ID sending all of the security/distribution groups (`groups` claims) and roles (`wids` claims) of the signed-in user. To set the `groupMembershipClaims` attribute:

1. Open the app's registration in the Azure portal.
2. Select **Manage** > **Manifest** in the sidebar.
3. Find the `groupMembershipClaims` attribute.
4. Set the value to `All` (`"groupMembershipClaims": "All"`).
5. Select the **Save** button.

Replace the `UserInfo` class in the `BlazorWebApp0Idc.Client` project with the following class.

`UserInfo.cs`:

C#

```
using Microsoft.AspNetCore.Components.WebAssembly.Authentication;  
using System.Security.Claims;
```

```

namespace BlazorWebAppOidc.Client;

// Add properties to this class and update the server and client
// AuthenticationStateProviders to expose more information about
// the authenticated user to the client.
public sealed class UserInfo
{
    public required string UserId { get; init; }
    public required string Name { get; init; }
    public required string[] Roles { get; init; }
    public required string[] Groups { get; init; }
    public required string[] Wids { get; init; }

    public const string UserIdClaimType = "sub";
    public const string NameClaimType = "name";
    private const string RoleClaimType = "roles";
    private const string GroupsClaimType = "groups";
    private const string WidsClaimType = "wids";

    public static UserInfo FromClaimsPrincipal(ClaimsPrincipal principal) =>
        new()
        {
            UserId = GetRequiredClaim(principal, UserIdClaimType),
            Name = GetRequiredClaim(principal, NameClaimType),
            Roles = principal.FindAll(RoleClaimType).Select(c => c.Value)
                .ToArray(),
            Groups = principal.FindAll(GroupsClaimType).Select(c => c.Value)
                .ToArray(),
            Wids = principal.FindAll(WidsClaimType).Select(c => c.Value)
                .ToArray(),
        };

    public ClaimsPrincipal ToClaimsPrincipal() =>
        new(new ClaimsIdentity(
            Roles.Select(role => new Claim(RoleClaimType, role))
                .Concat(Groups.Select(role => new Claim(GroupsClaimType,
role)))
                .Concat(Wids.Select(role => new Claim(WidsClaimType, role)))
                .Concat([
                    new Claim(UserIdClaimType, UserId),
                    new Claim(NameClaimType, Name),
                ]),
            authenticationType: nameof(UserInfo),
            nameType: NameClaimType,
            roleType: RoleClaimType));

    private static string GetRequiredClaim(ClaimsPrincipal principal,
        string claimType) =>
        principal.FindFirst(claimType)?.Value ??
        throw new InvalidOperationException(
            $"Could not find required '{claimType}' claim.");
}

```

At this point, Razor components can adopt [role-based and policy-based authorization](#):

- Application roles appear in `roles` claims, one claim per role.
- Security groups appear in `groups` claims, one claim per group. The security group GUIDs appear in the Azure portal when you create a security group and are listed when selecting **Identity > Overview > Groups > View**.
- Built-in ME-ID administrator roles appear in `wids` claims, one claim per role. The `wids` claim with a value of `b79fbf4d-3ef9-4689-8143-76b194e85509` is always sent by ME-ID for non-guest accounts of the tenant and doesn't refer to an administrator role. Administrator role GUIDs (*role template IDs*) appear in the Azure portal when selecting **Roles & admins**, followed by the ellipsis (...) > **Description** for the listed role. The role template IDs are also listed in [Microsoft Entra built-in roles \(Entra documentation\)](#).

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 cyberattackers 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 `allowPublicClientAttribute` 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 `{URL}` placeholder is the URL to open (for example, `https://localhost:5001`).
 - Mozilla Firefox: Use `-private -url {URL}`, where the `{URL}` placeholder 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

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

<PageTitle>User Claims</PageTitle>

<h1>User Claims</h1>

@if (claims.Any())
{
    <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](#)
- [Refresh token during http request in Blazor Interactive Server with OIDC \(dotnet/aspnetcore #55213\) !\[\]\(467d80e979964f7f8c752fb22248b5b7_img.jpg\)](#)

Threat mitigation guidance for ASP.NET Core Blazor static server-side rendering

Article • 11/19/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article explains the security considerations that developers should take into account when developing Blazor Web Apps with static server-side rendering.

Blazor combines three different models in one for writing interactive web apps. Traditional server-side rendering, which is a request/response model based on HTTP. Interactive server-side rendering, which is a rendering model based on SignalR. Finally, client-side rendering, which is a rendering model based on WebAssembly.

All of the general security considerations defined for the interactive rendering modes apply to Blazor Web Apps when there are interactive components rendering in one of the supported render modes. The following sections explain the security considerations specific to non-interactive server-side rendering in Blazor Web Apps and the specific aspects that apply when render modes interact with each other.

General considerations for server-side rendering

The server-side rendering (SSR) model is based on the traditional request/response model of HTTP. As such, there are common areas of concern between SSR and request/response HTTP. General security considerations and specific threats must be successfully mitigated. The framework provides built-in mechanisms for managing some of these threats, but other threats are specific to app code and must be handled by the app. These threats can be categorized as follows:

- **Authentication and authorization:** The app must ensure that the user is authenticated and authorized to access the app and the resources it exposes. The framework provides built-in mechanisms for authentication and authorization, but

the app must ensure that the mechanisms are properly configured and used. The built-in mechanisms for authentication and authorization are covered in the [Blazor documentation's Server security node](#) and in the [ASP.NET Core documentation's Security and Identity node](#), so they won't be covered here.

- **Input validation and sanitization:** All input arriving from a client must be validated and sanitized before use. Otherwise, the app might be exposed to attacks, such as SQL injection, cross-site scripting, cross-site request forgery, open redirection, and other forms of attacks. The input might come from anywhere in the request.
- **Session management:** Properly managing user sessions is critical to ensure that the app isn't exposed to attacks, such as session fixation, session hijacking, and other attacks. Information stored in the session must be properly protected and encrypted, and the app's code must prevent a malicious user from guessing or manipulating sessions.
- **Error handling and logging:** The app must ensure that errors are properly handled and logged. Otherwise, the app might be exposed to attacks, such as information disclosure. This can happen when the app returns sensitive information in the response or when the app returns detailed error messages with data that can be used to attack the app.
- **Data protection:** Sensitive data must be properly protected, which includes app logic when running on WebAssembly, since it can be easily reverse-engineered.
- **Denial of service:** The app must ensure that it isn't exposed to attacks, such as denial of service. This happens for example, when the app isn't properly protected against brute force attacks or when an action can cause the app to consume too many resources.

Input validation and sanitization

All input arriving from the client must be considered untrusted unless its information was generated and protected on the server, such as a CSRF token, an authentication cookie, a session identifier, or any other payload that's protected with authenticated encryption.

Input is normally available to the app through a binding process, for example via the [\[SupplyParameterFromQuery\] attribute](#) or [\[SupplyParameterFromForm\] attribute](#). Before processing this input, the app must make sure that the data is valid. For example, the app must confirm that there were no binding errors when mapping the form data to a component property. Otherwise, the app might process invalid data.

If the input is used to perform a redirect, the app must make sure that the input is valid and that it isn't pointing to a domain considered invalid or to an invalid subpath within the app base path. Otherwise, the app may be exposed to open redirection attacks, where a cyberattacker can craft a link that redirects the user to a malicious site.

If the input is used to perform a database query, app must confirm that the input is valid and that it isn't exposing the app to SQL injection attacks. Otherwise, a cyberattacker might be able to craft a malicious query that can be used to extract information from the database or to modify the database.

Data that might have come from user input also must be sanitized before included in a response. For example, the input might contain HTML or JavaScript that can be used to perform cross-site scripting attacks, which can be used to extract information from the user or to perform actions on behalf of the user.

The framework provides the following mechanisms to help with input validation and sanitization:

- All bound form data is validated for basic correctness. If an input can't be parsed, the binding process reports an error that the app can discover before taking any action with the data. The built-in [EditForm](#) component takes this into account before invoking the [OnValidSubmit](#) form callback. Blazor avoids executing the callback if there are one or more binding errors.
- The framework uses an antiforgery token to protect against cross-site request forgery attacks. For more information, see [ASP.NET Core Blazor authentication and authorization](#) and [ASP.NET Core Blazor forms overview](#).

All input and permissions must be validated on the server at the time of performing a given action to ensure that the data is valid and accurate at that time and that the user is allowed to perform the action. This approach is consistent with the [security guidance provided for interactive server-side rendering](#).

Session management

Session management is handled by the framework. The framework uses a session cookie to identify the user session. The session cookie is protected using the ASP.NET Core Data Protection APIs. The session cookie isn't accessible to JavaScript code running on the browser and it can't be easily guessed or manipulated by a user.

With regard to other session data, such as data stored within services, the session data should be stored within scoped services, as scoped services are unique per a given user

session, as opposed to singleton services which are shared across all user sessions in a given process instance.

When it comes to SSR, there's not much difference between scoped and transient services in most cases, as the lifetime of the service is limited to a single request. There's a difference in two scenarios:

- If the service is injected in more than one location or at different times during the request.
- If the service might be used in an interactive server context, where it survives multiple renders and its fundamental that the service is scoped to the user session.

Error handling and logging

The framework provides built-in logging for the app at the framework level. The framework logs important events, such as when the antiforgery token for a form fails to validate, when a root component starts to render, and when an action is dispatched. The app is responsible for logging any other events that might be important to record.

The framework provides built-in error handling for the app at the framework level. The framework handles errors that happen during the rendering of a component and either uses the error boundary mechanism to display a friendly error message or allows the error to bubble up to the exception handling middleware, which is configured to render the error page.

Errors that occur during streaming rendering after the response has started to be sent to the client are displayed in the final response as a generic error message. Details about the cause of the error are only included during development.

ASP.NET Core Data Protection

The framework offers mechanisms for protecting sensitive information for a given user session and ensures that the built-in components use these mechanisms to protect sensitive information, such as protecting user identity when using cookie authentication. Outside of scenarios handled by the framework, developer code is responsible for protecting other app-specific information. The most common way of doing this is via the ASP.NET Core Data Protection (DP) APIs or any other form of encryption. As a general rule, the app is responsible for:

- Making sure that a user can't inspect or modify the private information of another user.

- Making sure that a user can't modify user data of another user, such as an internal identifier.

With regard to DP, you must clearly understand where the code is executing. For the static server-side rendering (static SSR) and interactive server-side rendering (interactive SSR), code is stored on the server and never reaches the client. For the Interactive WebAssembly render mode, the app code *always reaches the client*, which means that any sensitive information stored in the app code is available to anyone with access to the app. Obfuscation and other similar technique to "protect" the code isn't effective. Once the code reaches the client, it can be reverse-engineered to extract the sensitive information.

Denial of service

At the server level, the framework provides limits on request/response parameters, such as the maximum size of the request and the header size. In regard to app code, Blazor's form mapping system defines limits similar to those defined by the MVC model binding system:

- Limit on the maximum number of errors.
- Limit on the maximum recursion depth for the binder.
- Limit on the maximum number of elements bound in a collection.

In addition, there are limits defined for the form, such as the maximum form key size and value size and the maximum number of entries.

In general, the app must evaluate when there's a chance that a request triggers an asymmetric amount of work by the server. Examples of this include when the user sends a request parameterized by N and the server performs an operation in response that is N times as expensive, where N is a parameter that a user controls and can grow indefinitely. Normally, the app must either impose a limit on the maximum N that it's willing to process or ensure that any operation is either less, equal, or more expensive than the request by a constant factor.

This aspect has more to do with the difference in growth between the work the client performs and the work the server performs than with a specific $1 \rightarrow N$ comparison. For example, a client might submit a work item (inserting elements into a list) that takes N units of time to perform, but the server needs N^2 to process (because it might be doing something very naive). It's the difference between N and N^2 that matters.

As such, there's a limit on how much work the server must be willing to do, which is specific to the app. This aspect applies to server-side workloads, since the resources are

on the server, but doesn't necessarily apply to WebAssembly workloads on the client in most cases.

The other important aspect is that this isn't only reserved to CPU time. It also applies to any resources, such as memory, network, and space on disk.

For WebAssembly workloads, there's usually little concern over the amount of work the client performs, since the client is normally limited by the resources available on the client. However, there are some scenarios where the client might be impacted, if for example, an app displays data from other users and one user is capable of adding data to the system that forces the clients that display the data to perform an amount of work that isn't proportional to the amount of data added by the user.

Recommended (non-exhaustive) check list

- Ensure that the user is authenticated and authorized to access the app and the resources it exposes.
- Validate and sanitize all input coming from a client before using it.
- Properly manage user sessions to ensure that state isn't mistakenly shared across users.
- Handle and log errors properly to avoid exposing sensitive information.
- Log important events in the app to identify potential issues and audit actions performed by users.
- Protect sensitive information using the ASP.NET Core Data Protection APIs or one of the available components ([Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage](#), [PersistentComponentState](#)).
- Ensure that the app understands the resources that can be consumed by a given request and has limits in place to avoid denial of service attacks.

Threat mitigation guidance for ASP.NET Core Blazor interactive server-side rendering

Article • 11/19/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article explains how to mitigate security threats in interactive server-side Blazor.

Apps adopt a *stateful* data processing model, where the server and client maintain a long-lived relationship. The persistent state is maintained by a [circuit](#), which can span connections that are also potentially long-lived.

When a user visits a site, the server creates a circuit in the server's memory. The circuit indicates to the browser what content to render and responds to events, such as when the user selects a button in the UI. To perform these actions, a circuit invokes JavaScript functions in the user's browser and .NET methods on the server. This two-way JavaScript-based interaction is referred to as [JavaScript interop \(JS interop\)](#).

Because JS interop occurs over the Internet and the client uses a remote browser, apps share most web app security concerns. This topic describes common threats to server-side Blazor apps and provides threat mitigation guidance focused on Internet-facing apps.

In constrained environments, such as inside corporate networks or intranets, some of the mitigation guidance either:

- Doesn't apply in the constrained environment.
- Isn't worth the cost to implement because the security risk is low in a constrained environment.

Interactive Server Components with WebSocket compression enabled

[Compression](#) can expose the app to side-channel attacks against the TLS encryption of the connection, such as [CRIME](#) and [BREACH](#) attacks. These types of attacks require that the cyberattacker:

- Force a browser to issue requests with a payload the cyberattacker controls to a vulnerable site via cross-site form posting or by embedding the site inside an iframe of another site.
- Observe the length of the compressed and encrypted response over the network.

For the app to be vulnerable, it must reflect the payload from the cyberattacker in the response, for example, by writing out the path or the query string into the response. Using the length of the response, the cyberattacker can "guess" any information on the response, bypassing the encryption of the connection.

Generally speaking, Blazor apps can enable compression over the WebSocket connection with appropriate security measures:

- The app can be vulnerable when it takes content from the request (for example, the path or query string) that can be influenced by a cyberattacker and reproduces it into the HTML of the page or otherwise makes it part of the response.
- Blazor applies the following security measures automatically:
 - When compression is configured, Blazor automatically blocks embedding the app into an iframe, which blocks the initial (uncompressed) response from the server from rendering and precludes the WebSocket connection from ever starting.
 - The restriction on embedding the app into an iframe can be relaxed. However, relaxing the restriction exposes the app to attack if the embedding document becomes compromised via a cross-site scripting vulnerability, as that gives the cyberattacker a way to execute the attack.
- Normally for this type of attack to take place, the app must repeatedly reproduce the content in the responses so that the cyberattacker can guess the response. Given how Blazor renders (it renders once and then produces diffs of the content only for the elements that changed) this is hard for a cyberattacker to accomplish. However, it isn't impossible for a cyberattacker, so care must be taken to avoid rendering sensitive information alongside external information that can be manipulated by a cyberattacker. Some examples of this are:
 - Render [Personally Identifiable Information \(PII\)](#) on the page at the same time as rendering database data that was added by another user.

- Rendering PII information on to the page at the same time as data coming from another user via JS interop or a local singleton service on the server.

In general, we recommend that you avoid rendering components that contain sensitive information alongside components that can render data from untrusted sources as part of the same render batch. Untrusted sources include route parameters, query strings, data from JS interop, and any other source of data that a third-party user can control (databases, external services).

Shared state

Server-side Blazor apps live in server memory, and multiple app sessions are hosted within the same process. For each app session, Blazor starts a circuit with its own dependency injection container scope, thus scoped services are unique per Blazor session.

Warning

We don't recommend apps on the same server share state using singleton services unless extreme care is taken, as this can introduce security vulnerabilities, such as leaking user state across circuits.

You can use stateful singleton services in Blazor apps if they're specifically designed for it. For example, use of a singleton memory cache is acceptable because a memory cache requires a key to access a given entry. Assuming users don't have control over the cache keys that are used with the cache, state stored in the cache doesn't leak across circuits.

For general guidance on state management, see [ASP.NET Core Blazor state management](#).

`IHttpContextAccessor` / `HttpContext` in Razor components

`IHttpContextAccessor` must be avoided with interactive rendering because there isn't a valid `HttpContext` available.

`IHttpContextAccessor` can be used for components that are statically rendered on the server. However, we recommend avoiding it if possible.

`HttpContext` can be used as a [cascading parameter](#) only in *statically-rendered root components* for general tasks, such as inspecting and modifying headers or other

properties in the `App` component (`Components/App.razor`). The value is always `null` for interactive rendering.

C#

```
[CascadingParameter]  
public HttpContext? HttpContext { get; set; }
```

For scenarios where the [HttpContext](#) is required in interactive components, we recommend flowing the data via persistent component state from the server. For more information, see [ASP.NET Core server-side and Blazor Web App additional security scenarios](#).

Resource exhaustion

Resource exhaustion can occur when a client interacts with the server and causes the server to consume excessive resources. Excessive resource consumption primarily affects:

- [CPU](#)
- [Memory](#)
- [Client connections](#)

Denial of Service (DoS) attacks usually seek to exhaust an app or server's resources. However, resource exhaustion isn't necessarily the result of an attack on the system. For example, finite resources can be exhausted due to high user demand. DoS is covered further in the [DoS section](#).

Resources external to the Blazor framework, such as databases and file handles (used to read and write files), may also experience resource exhaustion. For more information, see [ASP.NET Core Best Practices](#).

CPU

CPU exhaustion can occur when one or more clients force the server to perform intensive CPU work.

For example, consider an app that calculates a *Fibonacci number*. A Fibonacci number is produced from a Fibonacci sequence, where each number in the sequence is the sum of the two preceding numbers. The amount of work required to reach the answer depends on the length of the sequence and the size of the initial value. If the app doesn't place limits on a client's request, the CPU-intensive calculations may dominate

the CPU's time and diminish the performance of other tasks. Excessive resource consumption is a security concern impacting availability.

CPU exhaustion is a concern for all public-facing apps. In regular web apps, requests and connections time out as a safeguard, but Blazor apps don't provide the same safeguards. Blazor apps must include appropriate checks and limits before performing potentially CPU-intensive work.

Memory

Memory exhaustion can occur when one or more clients force the server to consume a large amount of memory.

For example, consider an app with a component that accepts and displays a list of items. If the Blazor app doesn't place limits on the number of items allowed or the number of items rendered back to the client, the memory-intensive processing and rendering may dominate the server's memory to the point where performance of the server suffers. The server may crash or slow to the point that it appears to have crashed.

Consider the following scenario for maintaining and displaying a list of items that pertain to a potential memory exhaustion scenario on the server:

- The items in a `List<T>` property or field use the server's memory. If the app allows the list of items to grow unbounded, there's a risk of the server running out of memory. Running out of memory causes the current session to end (crash) and all of the concurrent sessions in that server instance receive an out-of-memory exception. To prevent this scenario from occurring, the app must use a data structure that imposes an item limit on concurrent users.
- If a paging scheme isn't used for rendering, the server uses additional memory for objects that aren't visible in the UI. Without a limit on the number of items, memory demands may exhaust the available server memory. To prevent this scenario, use one of the following approaches:
 - Use paginated lists when rendering.
 - Only display the first 100 to 1,000 items and require the user to enter search criteria to find items beyond the items displayed.
 - For a more advanced rendering scenario, implement lists or grids that support *virtualization*. Using virtualization, lists only render a subset of items currently visible to the user. When the user interacts with the scrollbar in the UI, the component renders only those items required for display. The items that aren't currently required for display can be held in secondary storage, which is the ideal approach. Undisplayed items can also be held in memory, which is less ideal.

ⓘ Note

Blazor has built-in support for virtualization. For more information, see [ASP.NET Core Razor component virtualization](#).

Blazor apps offer a similar programming model to other UI frameworks for stateful apps, such as WPF, Windows Forms, or Blazor WebAssembly. The main difference is that in several of the UI frameworks the memory consumed by the app belongs to the client and only affects that individual client. For example, a Blazor WebAssembly app runs entirely on the client and only uses client memory resources. For a server-side Blazor app, the memory consumed by the app belongs to the server and is shared among clients on the server instance.

Server-side memory demands are a consideration for all server-side Blazor apps. However, most web apps are stateless, and the memory used while processing a request is released when the response is returned. As a general recommendation, don't permit clients to allocate an unbound amount of memory as in any other server-side app that persists client connections. The memory consumed by a server-side Blazor app persists for a longer time than a single request.

ⓘ Note

During development, a profiler can be used or a trace captured to assess memory demands of clients. A profiler or trace won't capture the memory allocated to a specific client. To capture the memory use of a specific client during development, capture a dump and examine the memory demand of all the objects rooted at a user's circuit.

Client connections

Connection exhaustion can occur when one or more clients open too many concurrent connections to the server, preventing other clients from establishing new connections.

Blazor clients establish a single connection per session and keep the connection open for as long as the browser window is open. Given the persistent nature of the connections and the stateful nature of server-side Blazor apps, connection exhaustion is a greater risk to availability of the app.

There's no limit on the number of connections per user for an app. If the app requires a connection limit, take one or more of the following approaches:

- Require authentication, which naturally limits the ability of unauthorized users to connect to the app. For this scenario to be effective, users must be prevented from provisioning new users on demand.
- Limit the number of connections per user. Limiting connections can be accomplished via the following approaches. Exercise care to allow legitimate users to access the app (for example, when a connection limit is established based on the client's IP address).
 - Application level
 - Endpoint routing extensibility.
 - Require authentication to connect to the app and keep track of the active sessions per user.
 - Reject new sessions upon reaching a limit.
 - Proxy WebSocket connections to an app through the use of a proxy, such as the [Azure SignalR Service](#) that multiplexes connections from clients to an app. This provides an app with greater connection capacity than a single client can establish, preventing a client from exhausting the connections to the server.
 - Server level
 - Use a proxy/gateway in front of the app. For example, [Azure Application Gateway](#) is a web traffic (OSI layer 7) load balancer that enables you to manage traffic to your web applications. For more information, see [Overview of WebSocket support in Application Gateway](#).
 - Although Long Polling is supported for Blazor apps, which would permit the adoption of [Azure Front Door](#), [WebSockets is the recommended transport protocol](#). As of September, 2024, [Azure Front Door](#) doesn't support WebSockets, but support for WebSockets is under consideration. For more information, see [Support WebSocket connections on Azure Front Door](#).

Denial of Service (DoS) attacks

[Denial of Service \(DoS\) attacks](#) involve a client causing the server to exhaust one or more of its resources making the app unavailable. Blazor apps include default limits and rely on other ASP.NET Core and SignalR limits that are set on [CircuitOptions](#) to protect against DoS attacks:

- [CircuitOptions.DisconnectedCircuitMaxRetained](#)
- [CircuitOptions.DisconnectedCircuitRetentionPeriod](#)
- [CircuitOptions.JSInteropDefaultCallTimeout](#)
- [CircuitOptions.MaxBufferedUnacknowledgedRenderBatches](#)
- [HubConnectionContextOptions.MaximumReceiveMessageSize](#)

For more information and configuration coding examples, see the following articles:

- [ASP.NET Core Blazor SignalR guidance](#)
- [ASP.NET Core SignalR configuration](#)

Interactions with the browser (client)

A client interacts with the server through JS interop event dispatching and render completion. JS interop communication goes both ways between JavaScript and .NET:

- Browser events are dispatched from the client to the server in an asynchronous fashion.
- The server responds asynchronously rerendering the UI as necessary.

JavaScript functions invoked from .NET

For calls from .NET methods to JavaScript:

- All invocations have a configurable timeout after which they fail, returning a [OperationCanceledException](#) to the caller.
 - There's a default timeout for the calls ([CircuitOptions.JSInteropDefaultCallTimeout](#)) of one minute. To configure this limit, see [Call JavaScript functions from .NET methods in ASP.NET Core Blazor](#).
 - A cancellation token can be provided to control the cancellation on a per-call basis. Rely on the default call timeout where possible and time-bound any call to the client if a cancellation token is provided.
- The result of a JavaScript call can't be trusted. The Blazor app client running in the browser searches for the JavaScript function to invoke. The function is invoked, and either the result or an error is produced. A malicious client can attempt to:
 - Cause an issue in the app by returning an error from the JavaScript function.
 - Induce an unintended behavior on the server by returning an unexpected result from the JavaScript function.

Take the following precautions to protect against the preceding scenarios:

- Wrap JS interop calls within [try-catch](#) statements to account for errors that might occur during the invocations. For more information, see [Handle errors in ASP.NET Core Blazor apps](#).
- Validate data returned from JS interop invocations, including error messages, before taking any action.

.NET methods invoked from the browser

Don't trust calls from JavaScript to .NET methods. When a .NET method is exposed to JavaScript, consider how the .NET method is invoked:

- Treat any .NET method exposed to JavaScript as you would a public endpoint to the app.
 - Validate input.
 - Ensure that values are within expected ranges.
 - Ensure that the user has permission to perform the action requested.
 - Don't allocate an excessive quantity of resources as part of the .NET method invocation. For example, perform checks and place limits on CPU and memory use.
 - Take into account that static and instance methods can be exposed to JavaScript clients. Avoid sharing state across sessions unless the design calls for sharing state with appropriate constraints.
 - For instance methods exposed through [DotNetObjectReference](#) objects that are originally created through dependency injection (DI), the objects should be registered as scoped objects. This applies to any DI service that the app uses.
 - For static methods, avoid establishing state that can't be scoped to the client unless the app is explicitly sharing state by-design across all users on a server instance.
 - Avoid passing user-supplied data in parameters to JavaScript calls. If passing data in parameters is absolutely required, ensure that the JavaScript code handles passing the data without introducing [Cross-site scripting \(XSS\)](#) vulnerabilities. For example, don't write user-supplied data to the DOM by setting the `innerHTML` property of an element. Consider using [Content Security Policy \(CSP\)](#) to disable `eval` and other unsafe JavaScript primitives. For more information, see [Enforce a Content Security Policy for ASP.NET Core Blazor](#).
- Avoid implementing custom dispatching of .NET invocations on top of the framework's dispatching implementation. Exposing .NET methods to the browser is an advanced scenario, not recommended for general Blazor development.

Events

Events provide an entry point to an app. The same rules for safeguarding endpoints in web apps apply to event handling in Blazor apps. A malicious client can send any data it wishes to send as the payload for an event.

For example:

- A change event for a `<select>` could send a value that isn't within the options that the app presented to the client.
- An `<input>` could send any text data to the server, bypassing client-side validation.

The app must validate the data for any event that the app handles. The Blazor framework [forms components](#) perform basic validations. If the app uses custom forms components, custom code must be written to validate event data as appropriate.

Events are asynchronous, so multiple events can be dispatched to the server before the app has time to react by producing a new render. This has some security implications to consider. Limiting client actions in the app must be performed inside event handlers and not depend on the current rendered view state.

Consider a counter component that should allow a user to increment a counter a maximum of three times. The button to increment the counter is conditionally based on the value of `count`:

razor

```
<p>Count: @count</p>

@if (count < 3)
{
    <button @onclick="IncrementCount" value="Increment count" />
}

@code
{
    private int count = 0;

    private void IncrementCount()
    {
        count++;
    }
}
```

A client can dispatch one or more increment events before the framework produces a new render of this component. The result is that the `count` can be incremented *over three times* by the user because the button isn't removed by the UI quickly enough. The correct way to achieve the limit of three `count` increments is shown in the following example:

razor

```
<p>Count: @count</p>

@if (count < 3)
```

```

{
    <button @onclick="IncrementCount" value="Increment count" />
}

@code
{
    private int count = 0;

    private void IncrementCount()
    {
        if (count < 3)
        {
            count++;
        }
    }
}

```

By adding the `if (count < 3) { ... }` check inside the handler, the decision to increment `count` is based on the current app state. The decision isn't based on the state of the UI as it was in the previous example, which might be temporarily stale.

Protect against multiple dispatches

If an event callback invokes a long running operation asynchronously, such as fetching data from an external service or database, consider using a safeguard. The safeguard can prevent the user from enqueueing multiple operations while the operation is in progress with visual feedback. The following component code sets `isLoading` to `true` while `DataService.GetDataAsync` obtains data from the server. While `isLoading` is `true`, the button is disabled in the UI:

razor

```

<button disabled="@isLoading" @onclick="UpdateData">Update</button>

@code {
    private bool isLoading;
    private Data[] data = Array.Empty<Data>();

    private async Task UpdateData()
    {
        if (!isLoading)
        {
            isLoading = true;
            data = await DataService.GetDataAsync(DateTime.Now);
            isLoading = false;
        }
    }
}

```

The safeguard pattern demonstrated in the preceding example works if the background operation is executed asynchronously with the `async - await` pattern.

Cancel early and avoid use-after-dispose

In addition to using a safeguard as described in the [Protect against multiple dispatches](#) section, consider using a [CancellationToken](#) to cancel long-running operations when the component is disposed. This approach has the added benefit of avoiding *use-after-dispose* in components:

razor

```
@implements IDisposable

...

@code {
    private readonly CancellationTokensource TokenSource =
        new CancellationTokensource();

    private async Task UpdateData()
    {
        ...

        data = await DataService.GetDataAsync(DateTime.Now,
        TokenSource.Token);

        if (TokenSource.Token.IsCancellationRequested)
        {
            return;
        }

        ...
    }

    public void Dispose()
    {
        TokenSource.Cancel();
    }
}
```

Avoid events that produce large amounts of data

Some DOM events, such as `oninput` or `onscroll`, can produce a large amount of data. Avoid using these events in server-side Blazor server.

Additional security guidance

The guidance for securing ASP.NET Core apps apply to server-side Blazor apps and are covered in the following sections of this article:

- [Logging and sensitive data](#)
- [Protect information in transit with HTTPS](#)
- [Cross-site scripting \(XSS\)](#)
- [Cross-origin protection](#)
- [Click-jacking](#)
- [Open redirects](#)

Logging and sensitive data

JS interop interactions between the client and server are recorded in the server's logs with [ILogger](#) instances. Blazor avoids logging sensitive information, such as actual events or JS interop inputs and outputs.

When an error occurs on the server, the framework notifies the client and tears down the session. The client receives a generic error message that can be seen in the browser's developer tools.

The client-side error doesn't include the call stack and doesn't provide detail on the cause of the error, but server logs do contain such information. For development purposes, sensitive error information can be made available to the client by [enabling detailed errors](#).

Warning

Exposing error information to clients on the Internet is a security risk that should always be avoided.

Protect information in transit with HTTPS

Blazor uses SignalR for communication between the client and the server. Blazor normally uses the transport that SignalR negotiates, which is typically WebSockets.

Blazor doesn't ensure the integrity and confidentiality of the data sent between the server and the client. Always use HTTPS.

Cross-site scripting (XSS)

Cross-site scripting (XSS) allows an unauthorized party to execute arbitrary logic in the context of the browser. A compromised app could potentially run arbitrary code on the client. The vulnerability could be used to potentially perform a number of malicious actions against the server:

- Dispatch fake/invalid events to the server.
- Dispatch fail/invalid render completions.
- Avoid dispatching render completions.
- Dispatch interop calls from JavaScript to .NET.
- Modify the response of interop calls from .NET to JavaScript.
- Avoid dispatching .NET to JS interop results.

The Blazor framework takes steps to protect against some of the preceding threats:

- Stops producing new UI updates if the client isn't acknowledging render batches. Configured with [CircuitOptions.MaxBufferedUnacknowledgedRenderBatches](#).
- Times out any .NET to JavaScript call after one minute without receiving a response from the client. Configured with [CircuitOptions.JSInteropDefaultCallTimeout](#).
- Performs basic validation on all input coming from the browser during JS interop:
 - .NET references are valid and of the type expected by the .NET method.
 - The data isn't malformed.
 - The correct number of arguments for the method are present in the payload.
 - The arguments or result can be deserialized correctly before invoking the method.
- Performs basic validation in all input coming from the browser from dispatched events:
 - The event has a valid type.
 - The data for the event can be deserialized.
 - There's an event handler associated with the event.

In addition to the safeguards that the framework implements, the app must be coded by the developer to safeguard against threats and take appropriate actions:

- Always validate data when handling events.
- Take appropriate action upon receiving invalid data:
 - Ignore the data and return. This allows the app to continue processing requests.
 - If the app determines that the input is illegitimate and couldn't be produced by legitimate client, throw an exception. Throwing an exception tears down the circuit and ends the session.
- Don't trust the error message provided by render batch completions included in the logs. The error is ***provided by the client*** and can't generally be trusted, as the client might be compromised.

- Don't trust the input on JS interop calls in either direction between JavaScript and .NET methods.
- The app is responsible for validating that the content of arguments and results are valid, even if the arguments or results are correctly deserialized.

For a XSS vulnerability to exist, the app must incorporate user input in the rendered page. Blazor executes a compile-time step where the markup in a `.razor` file is transformed into procedural C# logic. At runtime, the C# logic builds a *render tree* describing the elements, text, and child components. This is applied to the browser's DOM via a sequence of JavaScript instructions (or is serialized to HTML in the case of prerendering):

- User input rendered via normal Razor syntax (for example, `@someStringValue`) doesn't expose a XSS vulnerability because the Razor syntax is added to the DOM via commands that can only write text. Even if the value includes HTML markup, the value is displayed as static text. When prerendering, the output is HTML-encoded, which also displays the content as static text.
- Script tags aren't allowed and shouldn't be included in the app's component render tree. If a script tag is included in a component's markup, a compile-time error is generated.
- Component authors can author components in C# without using Razor. The component author is responsible for using the correct APIs when emitting output. For example, use `builder.AddContent(0, someUserSuppliedString)` and *not* `builder.AddMarkupContent(0, someUserSuppliedString)`, as the latter could create a XSS vulnerability.

Consider further mitigating XSS vulnerabilities. For example, implement a restrictive [Content Security Policy \(CSP\)](#). For more information, see [Enforce a Content Security Policy for ASP.NET Core Blazor](#).

For more information, see [Prevent Cross-Site Scripting \(XSS\) in ASP.NET Core](#).

Cross-origin protection

Cross-origin attacks involve a client from a different origin performing an action against the server. The malicious action is typically a GET request or a form POST (Cross-Site Request Forgery, CSRF), but opening a malicious WebSocket is also possible. Blazor apps offer [the same guarantees that any other SignalR app using the hub protocol offer](#):

- Apps can be accessed cross-origin unless additional measures are taken to prevent it. To disable cross-origin access, either disable CORS in the endpoint by adding the CORS Middleware to the pipeline and adding the [DisableCorsAttribute](#) to the

Blazor endpoint metadata or limit the set of allowed origins by [configuring SignalR for Cross-Origin Resource Sharing](#). For guidance on WebSocket origin restrictions, see [WebSockets support in ASP.NET Core](#).

- If CORS is enabled, extra steps might be required to protect the app depending on the CORS configuration. If CORS is globally enabled, CORS can be disabled for the Blazor SignalR hub by adding the [DisableCorsAttribute](#) metadata to the endpoint metadata after calling [MapBlazorHub](#) on the endpoint route builder.

For more information, see [Prevent Cross-Site Request Forgery \(XSRF/CSRF\) attacks in ASP.NET Core](#).

Click-jacking

Click-jacking involves rendering a site as an `<iframe>` inside a site from a different origin in order to trick the user into performing actions on the site under attack.

To protect an app from rendering inside of an `<iframe>`, use [Content Security Policy \(CSP\)](#) [↗](#) and the `X-Frame-Options` header.

For more information, see the following resources:

- [Enforce a Content Security Policy for ASP.NET Core Blazor](#)
- [MDN web docs: X-Frame-Options](#) [↗](#)

Open redirects

When an app session starts, the server performs basic validation of the URLs sent as part of starting the session. The framework checks that the base URL is a parent of the current URL before establishing the circuit. No additional checks are performed by the framework.

When a user selects a link on the client, the URL for the link is sent to the server, which determines what action to take. For example, the app may perform a client-side navigation or indicate to the browser to go to the new location.

Components can also trigger navigation requests programmatically through the use of [NavigationManager](#). In such scenarios, the app might perform a client-side navigation or indicate to the browser to go to the new location.

Components must:

- Avoid using user input as part of the navigation call arguments.
- Validate arguments to ensure that the target is allowed by the app.

Otherwise, a malicious user can force the browser to go to a cyberattacker-controlled site. In this scenario, the cyberattacker tricks the app into using some user input as part of the invocation of the [NavigationManager.NavigateTo](#) method.

This advice also applies when rendering links as part of the app:

- If possible, use relative links.
- Validate that absolute link destinations are valid before including them in a page.

For more information, see [Prevent open redirect attacks in ASP.NET Core](#).

Security checklist

The following list of security considerations isn't comprehensive:

- Validate arguments from events.
- Validate inputs and results from JS interop calls.
- Avoid using (or validate beforehand) user input for .NET to JS interop calls.
- Prevent the client from allocating an unbound amount of memory.
 - Data within the component.
 - [DotNetObjectReference](#) objects returned to the client.
- Protect against multiple dispatches.
- Cancel long-running operations when the component is disposed.
- Avoid events that produce large amounts of data.
- Avoid using user input as part of calls to [NavigationManager.NavigateTo](#) and validate user input for URLs against a set of allowed origins first if unavoidable.
- Don't make authorization decisions based on the state of the UI but only from component state.
- Consider using [Content Security Policy \(CSP\)](#) [↗](#) to protect against XSS attacks. For more information, see [Enforce a Content Security Policy for ASP.NET Core Blazor](#).
- Consider using CSP and [X-Frame-Options](#) [↗](#) to protect against click-jacking.
- Ensure CORS settings are appropriate when enabling CORS or explicitly disable CORS for Blazor apps.
- Test to ensure that the server-side limits for the Blazor app provide an acceptable user experience without unacceptable levels of risk.

Account confirmation and password recovery in ASP.NET Core Blazor

Article • 11/19/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article explains how to configure an ASP.NET Core Blazor Web App with email confirmation and password recovery.

Note

This article only applies to Blazor Web Apps. To implement email confirmation and password recovery for standalone Blazor WebAssembly apps with ASP.NET Core Identity, see [Account confirmation and password recovery in ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity](#).

Namespace

The app's namespace used by the example in this article is `BlazorSample`. Update the code examples to use the namespace of your app.

Select and configure an email provider

In this article, [Mailchimp's Transactional API](#) is used via [Mandrill.net](#) to send email. We recommend using an email service to send email rather than SMTP. SMTP is difficult to configure and secure properly. Whichever email service you use, access their guidance for .NET apps, create an account, configure an API key for their service, and install any NuGet packages required.

Create a class to hold the secret email provider API key. The example in this article uses a class named `AuthMessageSenderOptions` with an `EmailAuthKey` property to hold the key.

AuthMessageSenderOptions.cs:

```
C#  
  
namespace BlazorSample;  
  
public class AuthMessageSenderOptions  
{  
    public string? EmailAuthKey { get; set; }  
}
```

Register the `AuthMessageSenderOptions` configuration instance in the `Program` file:

```
C#  
  
builder.Services.Configure<AuthMessageSenderOptions>(builder.Configuration);
```

Configure a user secret for the provider's security key

If the project has already been initialized for the [Secret Manager tool](#), it will already have an app secrets identifier (`<AppSecretsId>`) in its project file (`.csproj`). In Visual Studio, you can tell if the app secrets ID is present by looking at the **Properties** panel when the project is selected in **Solution Explorer**. If the app hasn't been initialized, execute the following command in a command shell opened to the project's directory. In Visual Studio, you can use the Developer PowerShell command prompt.

.NET CLI

```
dotnet user-secrets init
```

Set the API key with the Secret Manager tool. In the following example, the key name is `EmailAuthKey` to match `AuthMessageSenderOptions.EmailAuthKey`, and the key is represented by the `{KEY}` placeholder. Execute the following command with the API key:

.NET CLI

```
dotnet user-secrets set "EmailAuthKey" "{KEY}"
```

If using Visual Studio, you can confirm the secret is set by right-clicking the server project in **Solution Explorer** and selecting **Manage User Secrets**.

For more information, see [Safe storage of app secrets in development in ASP.NET Core](#).

Warning

Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private C#/.NET code, or private keys/tokens in client-side code, which is ***always insecure***. In test/staging and production environments, server-side Blazor code and web APIs should use secure authentication flows that avoid maintaining credentials within project code or configuration files. Outside of local development testing, we recommend avoiding the use of environment variables to store sensitive data, as environment variables aren't the most secure approach. For local development testing, the [Secret Manager tool](#) is recommended for securing sensitive data. For more information, see [Securely maintain sensitive data and credentials](#).

Implement `ISender`

The following example is based on Mailchimp's Transactional API using [Mandrill.net](#). For a different provider, refer to their documentation on how to implement sending an email message.

Add the [Mandrill.net](#) NuGet package to the project.

Add the following `ISender` class to implement `ISender<TUser>`. In the following example, `ApplicationUser` is an [IdentityUser](#). The message HTML markup can be further customized. As long as the `message` passed to `MandrillMessage` starts with the `<` character, the Mandrill.net API assumes that the message body is composed in HTML.

`Components/Account/ISender.cs`:

C#

```
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;
using Mandrill;
using Mandrill.Model;
using BlazorSample.Data;

namespace BlazorSample.Components.Account;

public class ISender(IOptions<AuthMessageSenderOptions> optionsAccessor,
    ILogger<ISender> logger) : ISender<ApplicationUser>
{
    private readonly ILogger logger = logger;
```

```

    public AuthMessageSenderOptions Options { get; } =
optionsAccessor.Value;

    public Task SendConfirmationLinkAsync(AppUser user, string email,
        string confirmationLink) => SendEmailAsync(email, "Confirm your
email",
        "<html lang=\"en\"><head></head><body>Please confirm your account by
" +
        $"<a href='{confirmationLink}'>clicking here</a>.</body></html>");

    public Task SendPasswordResetLinkAsync(AppUser user, string email,
        string resetLink) => SendEmailAsync(email, "Reset your password",
        "<html lang=\"en\"><head></head><body>Please reset your password by
" +
        $"<a href='{resetLink}'>clicking here</a>.</body></html>");

    public Task SendPasswordResetCodeAsync(AppUser user, string email,
        string resetCode) => SendEmailAsync(email, "Reset your password",
        "<html lang=\"en\"><head></head><body>Please reset your password " +
        $"using the following code:<br>{resetCode}</body></html>");

    public async Task SendEmailAsync(string toEmail, string subject, string
message)
    {
        if (string.IsNullOrEmpty(Options.EmailAuthKey))
        {
            throw new Exception("Null EmailAuthKey");
        }

        await Execute(Options.EmailAuthKey, subject, message, toEmail);
    }

    public async Task Execute(string apiKey, string subject, string message,
string toEmail)
    {
        var api = new MandrillApi(apiKey);
        var mandrillMessage = new MandrillMessage("sarah@contoso.com",
toEmail,
            subject, message);
        await api.Messages.SendAsync(mandrillMessage);

        logger.LogInformation("Email to {EmailAddress} sent!", toEmail);
    }
}

```

ⓘ Note

Body content for messages might require special encoding for the email service provider. If links in the message body can't be followed in the email message, consult the service provider's documentation to troubleshoot the problem.

Configure app to support email

In the `Program` file, change the email sender implementation to the `EmailSender`:

diff

```
- builder.Services.AddSingleton<IEmailSender<ApplicationUser>,  
IdentityNoOpEmailSender>();  
+ builder.Services.AddSingleton<IEmailSender<ApplicationUser>, EmailSender>  
(());
```

Remove the `IdentityNoOpEmailSender` (`Components/Account/IdentityNoOpEmailSender.cs`) from the app.

In the `RegisterConfirmation` component

(`Components/Account/Pages/RegisterConfirmation.razor`), remove the conditional block in the `@code` block that checks if the `EmailSender` is an `IdentityNoOpEmailSender`:

diff

```
- else if (EmailSender is IdentityNoOpEmailSender)  
- {  
-     ...  
- }
```

Also in the `RegisterConfirmation` component, remove the Razor markup and code for checking the `emailConfirmationLink` field, leaving just the line instructing the user to check their email ...

diff

```
- @if (emailConfirmationLink is not null)  
- {  
-     ...  
- }  
- else  
- {  
    <p>Please check your email to confirm your account.</p>  
- }  
  
@code {  
-     private string? emailConfirmationLink;  
  
    ...  
}
```

Enable account confirmation after a site has users

Enabling account confirmation on a site with users locks out all the existing users. Existing users are locked out because their accounts aren't confirmed. To work around existing user lockout, use one of the following approaches:

- Update the database to mark all existing users as confirmed.
- Confirm existing users. For example, batch-send emails with confirmation links.

Email and activity timeout

The default inactivity timeout is 14 days. The following code sets the inactivity timeout to five days with sliding expiration:

C#

```
builder.Services.ConfigureApplicationCookie(options => {  
    options.ExpireTimeSpan = TimeSpan.FromDays(5);  
    options.SlidingExpiration = true;  
});
```

Change all ASP.NET Core Data Protection token lifespans

The following code changes Data Protection tokens' timeout period to three hours:

C#

```
builder.Services.Configure<DataProtectionTokenProviderOptions>(options =>  
    options.TokenLifespan = TimeSpan.FromHours(3));
```

The built-in Identity user tokens

([AspNetCore/src/Identity/Extensions.Core/src/TokenOptions.cs](#)) have a [one day timeout](#).

ⓘ Note

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list.

For more information, see [How to select a version tag of ASP.NET Core source code \(dotnet/AspNetCore.Docs #26205\)](#).[↗].

Change the email token lifespan

The default token lifespan of the [Identity user tokens](#)[↗] is [one day](#)[↗].

ⓘ Note

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see [How to select a version tag of ASP.NET Core source code \(dotnet/AspNetCore.Docs #26205\)](#).[↗].

To change the email token lifespan, add a custom [DataProtectorTokenProvider<TUser>](#) and [DataProtectionTokenProviderOptions](#).

CustomTokenProvider.cs:

C#

```
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;

namespace BlazorSample;

public class CustomEmailConfirmationTokenProvider<TUser>
    : DataProtectorTokenProvider<TUser> where TUser : class
{
    public CustomEmailConfirmationTokenProvider(
        IDataProtectionProvider dataProtectionProvider,
        IOptions<EmailConfirmationTokenProviderOptions> options,
        ILogger<DataProtectorTokenProvider<TUser>> logger)
        : base(dataProtectionProvider, options, logger)
    {
    }
}

public class EmailConfirmationTokenProviderOptions
    : DataProtectionTokenProviderOptions
{
    public EmailConfirmationTokenProviderOptions()
    {
        Name = "EmailDataProtectorTokenProvider";
        TokenLifespan = TimeSpan.FromHours(4);
    }
}
```



```

    }
}

public class CustomPasswordResetTokenProvider<TUser>
    : DataProtectorTokenProvider<TUser> where TUser : class
{
    public CustomPasswordResetTokenProvider(
        IDataProtectionProvider dataProtectionProvider,
        IOptions<PasswordResetTokenProviderOptions> options,
        ILogger<DataProtectorTokenProvider<TUser>> logger)
        : base(dataProtectionProvider, options, logger)
    {
    }
}

public class PasswordResetTokenProviderOptions :
    DataProtectionTokenProviderOptions
{
    public PasswordResetTokenProviderOptions()
    {
        Name = "PasswordResetDataProtectorTokenProvider";
        TokenLifespan = TimeSpan.FromHours(3);
    }
}

```

Configure the services to use the custom token provider in the `Program` file:

```

C#

builder.Services.AddIdentityCore<ApplicationUser>(options =>
{
    options.SignIn.RequireConfirmedAccount = true;
    options.Tokens.ProviderMap.Add("CustomEmailConfirmation",
        new TokenProviderDescriptor(
            typeof(CustomEmailConfirmationTokenProvider<ApplicationUser>)));
    options.Tokens.EmailConfirmationTokenProvider =
        "CustomEmailConfirmation";
})
.AddEntityFrameworkStores<ApplicationDbContext>()
.AddSignInManager()
.AddDefaultTokenProviders();

builder.Services
    .AddTransient<CustomEmailConfirmationTokenProvider<ApplicationUser>>();



```

Troubleshoot

If you can't get email working:

- Set a breakpoint in `EmailSender.Execute` to verify `SendEmailAsync` is called.
- Create a console app to send email using code similar to `EmailSender.Execute` to debug the problem.
- Review the account email history pages at the email provider's website.
- Check your spam folder for messages.
- Try another email alias on a different email provider, such as Microsoft, Yahoo, or Gmail.
- Try sending to different email accounts.

Additional resources

- [Mandrill.net \(GitHub repository\)](#) 
- [Mailchimp developer: Transactional API](#) 

Enable QR code generation for TOTP authenticator apps in an ASP.NET Core Blazor Web App

Article • 12/11/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article explains how to configure an ASP.NET Core Blazor Web App for two-factor authentication (2FA) with QR codes generated by Time-based One-time Password Algorithm (TOTP) authenticator apps.

For an introduction to 2FA with TOTP authenticator apps, see [Enable QR code generation for TOTP authenticator apps in ASP.NET Core](#).

The guidance in this article relies upon either creating the app with **Individual Accounts** for the new app's **Authentication type** or [scaffolding Identity into an existing app](#). For guidance on using the .NET CLI instead of Visual Studio for scaffolding Identity into an existing app, see [ASP.NET Core code generator tool \(`aspnet-codegenerator`\)](#).

Warning

TOTP codes should be kept secret because they can be used to authenticate multiple times before they expire.

Adding QR codes to the 2FA configuration page

A QR code generated by the app to set up 2FA with an TOTP authenticator app must be generated by a QR code library.

The guidance in this article uses [manuelbl/QrCodeGenerator](#), but you can use any QR code generation library.

Add a package reference for the [Net.Codecrete.QrCodeGenerator](#) NuGet package.

❗ Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at [Package consumption workflow \(NuGet documentation\)](#). Confirm correct package versions at [NuGet.org](#).

Open the `EnableAuthenticator` component in the `Components/Account/Pages/Manage` folder. At the top of the file under the `@page` directive, add an `@using` directive for the `QrCodeGenerator` namespace:

razor

```
@using Net.Codecrete.QrCodeGenerator
```

Delete the `<div>` element that contains the QR code instructions and the two `<div>` elements where the QR code should appear and where the QR code data is stored in the page:

diff

```
- <div class="alert alert-info">
-     Learn how to <a href="https://go.microsoft.com/fwlink/?
Linkid=852423">enable
-     QR code generation</a>.
- </div>
- <div></div>
- <div data-url="@authenticatorUri"></div>
```

Replace the deleted elements with the following markup:

razor

```
<div>
    <svg xmlns="http://www.w3.org/2000/svg" height="300" width="300"
stroke="none"
        version="1.1" viewBox="0 0 50 50">
        <rect width="300" height="300" fill="#ffffff" />
        <path d="@svgGraphicsPath" fill="#000000" />
    </svg>
</div>
```

Just inside the opening `@code` block, change the variable declaration for `authenticatorUri` to `svgGraphicsPath`:

diff

```
- private string? authenticatorUri;  
+ private string? svgGraphicsPath;
```

Change the site name in the `GenerateQrCodeUri` method. The default value is `Microsoft.AspNetCore.Identity.UI`. Change the value to a meaningful site name that users can identify easily in their authenticator app. Developers usually set a site name that matches the company's name. We recommend limiting the site name length to 30 characters or less to allow the site name to display on narrow mobile device screens.

In the following example, the default value `Microsoft.AspNetCore.Identity.UI` is changed to the company name `Weyland-Yutani Corporation` (©1986 20th Century Studios [Aliens](#) [↗]).

In the `GenerateQrCodeUri` method:

diff

```
- UrlEncoder.Encode("Microsoft.AspNetCore.Identity.UI"),  
+ UrlEncoder.Encode("Weyland-Yutani Corporation"),
```

At the bottom of the `LoadSharedKeyAndQrCodeUriAsync` method, add the `var` keyword to the line that sets `authenticatorUri`, making it an implicitly-typed local variable:

diff

```
- authenticatorUri = GenerateQrCodeUri(email!, unformattedKey!);  
+ var authenticatorUri = GenerateQrCodeUri(email!, unformattedKey!);
```

Add the following lines of code at the bottom of the `LoadSharedKeyAndQrCodeUriAsync` method:

C#

```
var qr = QrCode.EncodeText(authenticatorUri, QrCode.Ecc.Medium);  
svgGraphicsPath = qr.ToGraphicsPath();
```

Run the app and ensure that the QR code is scannable and that the code validates.

Warning


An ASP.NET Core TOTP code should be kept secret because it can be used to authenticate successfully multiple times before it expires.

EnableAuthenticator component in reference source

The `EnableAuthenticator` component can be inspected in reference source:

[EnableAuthenticator component in reference source](#) 

Note

Documentation links to .NET reference source usually load the repository's default branch, which represents the current development for the next release of .NET. To select a tag for a specific release, use the **Switch branches or tags** dropdown list. For more information, see [How to select a version tag of ASP.NET Core source code \(dotnet/AspNetCore.Docs #26205\)](#) .

Additional resources

- [Using a different QR code library](#)
- [TOTP client and server time skew](#)

Enforce a Content Security Policy for ASP.NET Core Blazor

Article • 09/27/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article explains how to use a [Content Security Policy \(CSP\)](#) with ASP.NET Core Blazor apps to help protect against [Cross-Site Scripting \(XSS\)](#) attacks.

[Cross-Site Scripting \(XSS\)](#) is a security vulnerability where a cyberattacker places one or more malicious client-side scripts into an app's rendered content. A CSP helps protect against XSS attacks by informing the browser of valid:

- Sources for loaded content, including scripts, stylesheets, images, and plugins.
- Actions taken by a page, specifying permitted URL targets of forms.

To apply a CSP to an app, the developer specifies several CSP content security *directives* in one or more `Content-Security-Policy` headers or `<meta>` tags. For guidance on applying a CSP to an app in C# code at startup, see [ASP.NET Core Blazor startup](#).

Policies are evaluated by the browser while a page is loading. The browser inspects the page's sources and determines if they meet the requirements of the content security directives. When policy directives aren't met for a resource, the browser doesn't load the resource. For example, consider a policy that doesn't allow third-party scripts. When a page contains a `<script>` tag with a third-party origin in the `src` attribute, the browser prevents the script from loading.

CSP is supported in most modern desktop and mobile browsers, including Chrome, Edge, Firefox, Opera, and Safari. CSP is recommended for Blazor apps.

Policy directives

Minimally, specify the following directives and sources for Blazor apps. Add additional directives and sources as needed. The following directives are used in the *Apply the*

policy section of this article, where example security policies for Blazor apps are provided:

- [base-uri](#): Restricts the URLs for a page's `<base>` tag. Specify `self` to indicate that the app's origin, including the scheme and port number, is a valid source.
- [default-src](#): Indicates a fallback for source directives that aren't explicitly specified by the policy. Specify `self` to indicate that the app's origin, including the scheme and port number, is a valid source.
- [img-src](#): Indicates valid sources for images.
 - Specify `data:` to permit loading images from `data:` URLs.
 - Specify `https:` to permit loading images from HTTPS endpoints.
- [object-src](#): Indicates valid sources for the `<object>`, `<embed>`, and `<applet>` tags. Specify `none` to prevent all URL sources.
- [script-src](#): Indicates valid sources for scripts.
 - Specify `self` to indicate that the app's origin, including the scheme and port number, is a valid source.
 - In a client-side Blazor app:
 - Specify [wasm-unsafe-eval](#) to permit the client-side Blazor Mono runtime to function.
 - Specify any additional hashes to permit your required *non-framework scripts* to load.
 - In a server-side Blazor app, specify hashes to permit required scripts to load.
- [style-src](#): Indicates valid sources for stylesheets.
 - Specify `self` to indicate that the app's origin, including the scheme and port number, is a valid source.
 - If the app uses inline styles, specify `unsafe-inline` to allow the use of your inline styles.
- [upgrade-insecure-requests](#): Indicates that content URLs from insecure (HTTP) sources should be acquired securely over HTTPS.

The preceding directives are supported by all browsers except Microsoft Internet Explorer.

To obtain SHA hashes for additional inline scripts:

- Apply the CSP shown in the *Apply the policy* section.
- Access the browser's developer tools console while running the app locally. The browser calculates and displays hashes for blocked scripts when a CSP header or `meta` tag is present.
- Copy the hashes provided by the browser to the `script-src` sources. Use single quotes around each hash.

For a Content Security Policy Level 2 browser support matrix, see [Can I use: Content Security Policy Level 2](#).

Apply the policy

Use a `<meta>` tag to apply the policy:

- Set the value of the `http-equiv` attribute to `Content-Security-Policy`.
- Place the directives in the `content` attribute value. Separate directives with a semicolon (;).
- Always place the `meta` tag in the `<head>` content.

The following sections show example policies. These examples are versioned with this article for each release of Blazor. To use a version appropriate for your release, select the document version with the **Version** dropdown selector on this webpage.

Server-side Blazor apps

In the `<head>` content, apply the directives described in the *Policy directives* section:

HTML

```
<meta http-equiv="Content-Security-Policy"
      content="base-uri 'self';
              default-src 'self';
              img-src data: https:;
              object-src 'none';
              script-src 'self';
              style-src 'self';
              upgrade-insecure-requests;">
```

Add additional `script-src` and `style-src` hashes as required by the app. During development, use an online tool or browser developer tools to have the hashes calculated for you. For example, the following browser tools console error reports the hash for a required script not covered by the policy:

Refused to execute inline script because it violates the following Content Security Policy directive: " ... ". Either the 'unsafe-inline' keyword, a hash ('sha256-v8v3RKRPmN4odZ1CWM5gw80QKPCCWMcpNeOmimNL2AA='), or a nonce ('nonce-...') is required to enable inline execution.

The particular script associated with the error is displayed in the console next to the error.

Client-side Blazor apps

In the `<head>` content, apply the directives described in the *Policy directives* section:

HTML

```
<meta http-equiv="Content-Security-Policy"
      content="base-uri 'self';
              default-src 'self';
              img-src data: https;;
              object-src 'none';
              script-src 'self'
                      'wasm-unsafe-eval';
              style-src 'self';
              upgrade-insecure-requests;">
```

Add additional `script-src` and `style-src` hashes as required by the app. During development, use an online tool or browser developer tools to have the hashes calculated for you. For example, the following browser tools console error reports the hash for a required script not covered by the policy:

Refused to execute inline script because it violates the following Content Security Policy directive: " ... ". Either the 'unsafe-inline' keyword, a hash ('sha256-v8v3RKRPmN4odZ1CWM5gw80QKPCWmcpNeOmimNL2AA='), or a nonce ('nonce-...') is required to enable inline execution.

The particular script associated with the error is displayed in the console next to the error.

Apply a CSP in non-Development environments

When a CSP is applied to a Blazor app's `<head>` content, it interferes with local testing in the `Development` environment. For example, [Browser Link](#) and the browser refresh script fail to load. The following examples demonstrate how to apply the CSP's `<meta>` tag in non-`Development` environments.

ⓘ Note

The examples in this section don't show the full `<meta>` tag for the CSPs. The complete `<meta>` tags are found in the subsections of the [Apply the policy](#) section earlier in this article.

Three general approaches are available:

- Apply the CSP via the `App` component, which applies the CSP to all layouts of the app.
- If you need to apply CSPs to different areas of the app, for example a custom CSP for only the admin pages, apply the CSPs on a per-layout basis using the `<HeadContent>` tag. For complete effectiveness, every app layout file must adopt the approach.
- The hosting service or server can provide a CSP via a [Content-Security-Policy header](#) added to an app's outgoing responses. Because this approach varies by hosting service or server, it isn't addressed in the following examples. If you wish to adopt this approach, consult the documentation for your hosting service provider or server.

Blazor Web App approaches

In the `App` component (`Components/App.razor`), inject `IHostEnvironment`:

```
razor

@inject IHostEnvironment Env
```

In the `App` component's `<head>` content, apply the CSP when not in the `Development` environment:

```
razor

@if (!Env.IsDevelopment())
{
    <meta ...>
}
```

Alternatively, apply CSPs on a per-layout basis in the `Components/Layout` folder, as the following example demonstrates. Make sure that every layout specifies a CSP.

```
razor

@inject IHostEnvironment Env

@if (!Env.IsDevelopment())
{
    <HeadContent>
        <meta ...>
    </HeadContent>
}
```

```
</HeadContent>
}
```

Blazor WebAssembly app approaches

In the `App` component (`App.razor`), inject `IWebAssemblyHostEnvironment`:

razor

```
@using Microsoft.AspNetCore.Components.WebAssembly.Hosting
@inject IWebAssemblyHostEnvironment Env
```

In the `App` component's `<head>` content, apply the CSP when not in the `Development` environment:





razor

```
@if (!Env.IsDevelopment())
{
    <HeadContent>
        <meta ...>
    </HeadContent>
}
```

Alternatively, use the preceding code but apply CSPs on a per-layout basis in the `Layout` folder. Make sure that every layout specifies a CSP.

Meta tag limitations

A `<meta>` tag policy doesn't support the following directives:

- [frame-ancestors](#) 
- [report-to](#) 
- [report-uri](#) 
- [sandbox](#) 

To support the preceding directives, use a header named `Content-Security-Policy`. The directive string is the header's value.

Test a policy and receive violation reports

Testing helps confirm that third-party scripts aren't inadvertently blocked when building an initial policy.

To test a policy over a period of time without enforcing the policy directives, set the `<meta>` tag's `http-equiv` attribute or header name of a header-based policy to `Content-Security-Policy-Report-Only`. Failure reports are sent as JSON documents to a specified URL. For more information, see [MDN web docs: Content-Security-Policy-Report-Only](#).

For reporting on violations while a policy is active, see the following articles:

- [report-to](#)
- [report-uri](#)

Although `report-uri` is no longer recommended for use, both directives should be used until `report-to` is supported by all of the major browsers. Don't exclusively use `report-uri` because support for `report-uri` is subject to being dropped *at any time* from browsers. Remove support for `report-uri` in your policies when `report-to` is fully supported. To track adoption of `report-to`, see [Can I use: report-to](#).

Test and update an app's policy every release.

Troubleshoot

- Errors appear in the browser's developer tools console. Browsers provide information about:
 - Elements that don't comply with the policy.
 - How to modify the policy to allow for a blocked item.
- A policy is only completely effective when the client's browser supports all of the included directives. For a current browser support matrix, see [Can I use: Content-Security-Policy](#).

Additional resources

- [Apply a CSP in C# code at startup](#)
- [MDN web docs: Content Security Policy \(CSP\)](#)
- [MDN web docs: Content-Security-Policy response header](#)
- [Content Security Policy Level 2](#)
- [Google CSP Evaluator](#)

ASP.NET Core server-side and Blazor Web App additional security scenarios

Article • 11/19/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.



For the current release, see the [.NET 9 version of this article](#).

This article explains how to configure server-side Blazor for additional security scenarios, including how to pass tokens to a Blazor app.

Note

The code examples in this article adopt [nullable reference types \(NRTs\)](#) and [.NET compiler null-state static analysis](#), which are supported in ASP.NET Core in .NET 6 or later. When targeting ASP.NET Core 5.0 or earlier, remove the null type designation (?) from the `string?`, `TodoItem[]?`, `WeatherForecast[]?`, and `IEnumerable<GitHubBranch>?` types in the article's examples.

Pass tokens to a server-side Blazor app

Updating this section for Blazor Web Apps is pending [Update section on passing tokens in Blazor Web Apps \(dotnet/AspNetCore.Docs #31691\)](#) . For more information, see [Problem providing Access Token to HttpClient in Interactive Server mode \(dotnet/aspnetcore #52390\)](#) .

For Blazor Server, view the [7.0 version of this article section](#).

Set the authentication scheme

For an app that uses more than one Authentication Middleware and thus has more than one authentication scheme, the scheme that Blazor uses can be explicitly set in the endpoint configuration of the `Program` file. The following example sets the OpenID Connect (OIDC) scheme:

C#

```
using Microsoft.AspNetCore.Authentication.OpenIdConnect;

...

app.MapRazorComponents<App>().RequireAuthorization(
    new AuthorizeAttribute
    {
        AuthenticationSchemes = OpenIdConnectDefaults.AuthenticationScheme
    })
    .AddInteractiveServerRenderMode();
```

Circuit handler to capture users for custom services

Use a [CircuitHandler](#) to capture a user from the [AuthenticationStateProvider](#) and set the user in a service. If you want to update the user, register a callback to [AuthenticationStateChanged](#) and enqueue a [Task](#) to obtain the new user and update the service. The following example demonstrates the approach.

In the following example:

- [OnConnectionUpAsync](#) is called every time the circuit reconnects, setting the user for the lifetime of the connection. Only the [OnConnectionUpAsync](#) method is required unless you implement updates via a handler for authentication changes ([AuthenticationChanged](#) in the following example).
- [OnCircuitOpenedAsync](#) is called to attach the authentication changed handler, [AuthenticationChanged](#), to update the user.
- The `catch` block of the [UpdateAuthentication](#) task takes no action on exceptions because there's no way to report the exceptions at this point in code execution. If an exception is thrown from the task, the exception is reported elsewhere in app.

`UserService.cs`:

C#

```
using System.Security.Claims;
using Microsoft.AspNetCore.Components.Authorization;
using Microsoft.AspNetCore.Components.Server.Circuits;

public class UserService
{
    private ClaimsPrincipal currentUser = new(new ClaimsIdentity());
```

```

public ClaimsPrincipal GetUser() => currentUser;

internal void SetUser(ClaimsPrincipal user)
{
    if (currentUser != user)
    {
        currentUser = user;
    }
}

internal sealed class UserCircuitHandler(
    AuthenticationStateProvider authenticationStateProvider,
    UserService userService)
    : CircuitHandler, IDisposable
{
    public override Task OnCircuitOpenedAsync(Circuit circuit,
        Cancellation_token cancellationToken)
    {
        authenticationStateProvider.AuthenticationStateChanged +=
            AuthenticationChanged;

        return base.OnCircuitOpenedAsync(circuit, cancellationToken);
    }

    private void AuthenticationChanged(Task<AuthenticationState> task)
    {
        _ = UpdateAuthentication(task);

        async Task UpdateAuthentication(Task<AuthenticationState> task)
        {
            try
            {
                var state = await task;
                userService.SetUser(state.User);
            }
            catch
            {
            }
        }
    }

    public override async Task OnConnectionUpAsync(Circuit circuit,
        Cancellation_token cancellationToken)
    {
        var state = await
authenticationStateProvider.GetAuthenticationStateAsync();
        userService.SetUser(state.User);
    }

    public void Dispose()
    {
        authenticationStateProvider.AuthenticationStateChanged -=
            AuthenticationChanged;
    }
}

```



```
}  
}
```

In the `Program` file:

C#

```
using Microsoft.AspNetCore.Components.Server.Circuits;  
using Microsoft.Extensions.DependencyInjection.Extensions;  
  
...  
  
builder.Services.AddScoped<UserService>();  
builder.Services.TryAddEnumerable(  
    ServiceDescriptor.Scoped<CircuitHandler, UserCircuitHandler>());
```

Use the service in a component to obtain the user:

razor

```
@inject UserService UserService  
  
<h1>Hello, @(UserService.GetUser().Identity?.Name ?? "world")!</h1>
```

To set the user in middleware for MVC, Razor Pages, and in other ASP.NET Core scenarios, call `SetUser` on the `UserService` in custom middleware after the Authentication Middleware runs, or set the user with an [IClaimsTransformation](#) implementation. The following example adopts the middleware approach.

`UserServiceMiddleware.cs`:

C#

```
public class UserServiceMiddleware  
{  
    private readonly RequestDelegate next;  
  
    public UserServiceMiddleware(RequestDelegate next)  
    {  
        this.next = next ?? throw new ArgumentNullException(nameof(next));  
    }  
  
    public async Task InvokeAsync(HttpContext context, UserService service)  
    {  
        service.SetUser(context.User);  
        await next(context);  
    }  
}
```

Immediately before the call to `app.MapRazorComponents<App>()` in the `Program` file, call the middleware:

C#

```
app.UseMiddleware<UserServiceMiddleware>();
```

Access `AuthenticationStateProvider` in outgoing request middleware

The `AuthenticationStateProvider` from a `DelegatingHandler` for `HttpClient` created with `IHttpClientFactory` can be accessed in outgoing request middleware using a `circuit activity handler`.

ⓘ Note

For general guidance on defining delegating handlers for HTTP requests by `HttpClient` instances created using `IHttpClientFactory` in ASP.NET Core apps, see the following sections of [Make HTTP requests using IHttpClientFactory in ASP.NET Core](#):

- [Outgoing request middleware](#)
- [Use DI in outgoing request middleware](#)

The following example uses `AuthenticationStateProvider` to attach a custom user name header for authenticated users to outgoing requests.

First, implement the `CircuitServicesAccessor` class in the following section of the Blazor dependency injection (DI) article:

Access server-side Blazor services from a different DI scope

Use the `CircuitServicesAccessor` to access the `AuthenticationStateProvider` in the `DelegatingHandler` implementation.

`AuthenticationStateHandler.cs`:

C#

```
public class AuthenticationStateHandler(  
    CircuitServicesAccessor circuitServicesAccessor)  
    : DelegatingHandler
```

```

{
    protected override async Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {
        var authStateProvider = circuitServicesAccessor.Services
            .GetRequiredService<AuthenticationStateProvider>();
        var authState = await
authStateProvider.GetAuthenticationStateAsync();
        var user = authState.User;

        if (user.Identity is not null && user.Identity.IsAuthenticated)
        {
            request.Headers.Add("X-USER-IDENTITY-NAME", user.Identity.Name);
        }

        return await base.SendAsync(request, cancellationToken);
    }
}

```

In the `Program` file, register the `AuthenticationStateHandler` and add the handler to the `IHttpClientFactory` that creates `HttpClient` instances:

```

C#

builder.Services.AddTransient<AuthenticationStateHandler>();

builder.Services.AddHttpClient("HttpMessageHandler")
    .AddHttpMessageHandler<AuthenticationStateHandler>();

```

ASP.NET Core Blazor state management

Article • 09/12/2024

📘 Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article describes common approaches for maintaining a user's data (state) while they use an app and across browser sessions.

📌 Note

The code examples in this article adopt [nullable reference types \(NRTs\)](#) and [.NET compiler null-state static analysis](#), which are supported in ASP.NET Core in .NET 6 or later. When targeting ASP.NET Core 5.0 or earlier, remove the null type designation (`?`) from types in the article's examples.

Maintain user state

Server-side Blazor is a stateful app framework. Most of the time, the app maintains a connection to the server. The user's state is held in the server's memory in a *circuit*.

Examples of user state held in a circuit include:

- The hierarchy of component instances and their most recent render output in the rendered UI.
- The values of fields and properties in component instances.
- Data held in [dependency injection \(DI\)](#) service instances that are scoped to the circuit.

User state might also be found in JavaScript variables in the browser's memory set via [JavaScript interop](#) calls.

If a user experiences a temporary network connection loss, Blazor attempts to reconnect the user to their original circuit with their original state. However, reconnecting a user to their original circuit in the server's memory isn't always possible:

- The server can't retain a disconnected circuit forever. The server must release a disconnected circuit after a timeout or when the server is under memory pressure.
- In multi-server, load-balanced deployment environments, individual servers may fail or be automatically removed when no longer required to handle the overall volume of requests. The original server processing requests for a user may become unavailable when the user attempts to reconnect.
- The user might close and reopen their browser or reload the page, which removes any state held in the browser's memory. For example, JavaScript variable values set through JavaScript interop calls are lost.

When a user can't be reconnected to their original circuit, the user receives a new circuit with an empty state. This is equivalent to closing and reopening a desktop app.

Persist state across circuits

Generally, maintain state across circuits where users are actively creating data, not simply reading data that already exists.

To preserve state across circuits, the app must persist the data to some other storage location than the server's memory. State persistence isn't automatic. You must take steps when developing the app to implement stateful data persistence.

Data persistence is typically only required for high-value state that users expended effort to create. In the following examples, persisting state either saves time or aids in commercial activities:

- Multi-step web forms: It's time-consuming for a user to re-enter data for several completed steps of a multi-step web form if their state is lost. A user loses state in this scenario if they navigate away from the form and return later.
- Shopping carts: Any commercially important component of an app that represents potential revenue can be maintained. A user who loses their state, and thus their shopping cart, may purchase fewer products or services when they return to the site later.

An app can only persist *app state*. UIs can't be persisted, such as component instances and their render trees. Components and render trees aren't generally serializable. To persist UI state, such as the expanded nodes of a tree view control, the app must use custom code to model the behavior of the UI state as serializable app state.

Where to persist state

Common locations exist for persisting state:

- [Server-side storage](#)
- [URL](#)
- [Browser storage](#)
- [In-memory state container service](#)


Server-side storage

For permanent data persistence that spans multiple users and devices, the app can use server-side storage. Options include:

- Blob storage
- Key-value storage
- Relational database
- Table storage

After data is saved, the user's state is retained and available in any new circuit.

For more information on Azure data storage options, see the following:

- [Azure Databases](#) 
- [Azure Storage Documentation](#)

URL

For transient data representing navigation state, model the data as a part of the URL. Examples of user state modeled in the URL include:



- The ID of a viewed entity.
- The current page number in a paged grid.

The contents of the browser's address bar are retained:

- If the user manually reloads the page.
- If the web server becomes unavailable, and the user is forced to reload the page in order to connect to a different server.

For information on defining URL patterns with the `@page` directive, see [ASP.NET Core Blazor routing and navigation](#).

Browser storage

For transient data that the user is actively creating, a commonly used storage location is the browser's [localStorage](#)  and [sessionStorage](#)  collections:

- `localStorage` is scoped to the browser's window. If the user reloads the page or closes and reopens the browser, the state persists. If the user opens multiple browser tabs, the state is shared across the tabs. Data persists in `localStorage` until explicitly cleared.
- `sessionStorage` is scoped to the browser tab. If the user reloads the tab, the state persists. If the user closes the tab or the browser, the state is lost. If the user opens multiple browser tabs, each tab has its own independent version of the data.

Generally, `sessionStorage` is safer to use. `sessionStorage` avoids the risk that a user opens multiple tabs and encounters the following:

- Bugs in state storage across tabs.
- Confusing behavior when a tab overwrites the state of other tabs.

`localStorage` is the better choice if the app must persist state across closing and reopening the browser.

Caveats for using browser storage:

- Similar to the use of a server-side database, loading and saving data are asynchronous.
- The requested page doesn't exist in the browser during prerendering, so local storage isn't available during prerendering.
- Storage of a few kilobytes of data is reasonable to persist for server-side Blazor apps. Beyond a few kilobytes, you must consider the performance implications because the data is loaded and saved across the network.
- Users may view or tamper with the data. [ASP.NET Core Data Protection](#) can mitigate the risk. For example, [ASP.NET Core Protected Browser Storage](#) uses ASP.NET Core Data Protection.

Third-party NuGet packages provide APIs for working with `localStorage` and `sessionStorage`. It's worth considering choosing a package that transparently uses [ASP.NET Core Data Protection](#). Data Protection encrypts stored data and reduces the potential risk of tampering with stored data. If JSON-serialized data is stored in plain text, users can see the data using browser developer tools and also modify the stored data. Securing trivial data isn't a problem. For example, reading or modifying the stored color of a UI element isn't a significant security risk to the user or the organization. Avoid allowing users to inspect or tamper with *sensitive data*.

ASP.NET Core Protected Browser Storage

ASP.NET Core Protected Browser Storage leverages [ASP.NET Core Data Protection](#) for [localStorage](#) and [sessionStorage](#).

ⓘ Note

Protected Browser Storage relies on ASP.NET Core Data Protection and is only supported for server-side Blazor apps.

Save and load data within a component

In any component that requires loading or saving data to browser storage, use the [@inject](#) directive to inject an instance of either of the following:

- `ProtectedLocalStorage`
- `ProtectedSessionStorage`

The choice depends on which browser storage location you wish to use. In the following example, `sessionStorage` is used:

razor

```
@using Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage
@inject ProtectedSessionStorage ProtectedSessionStore
```

The `@using` directive can be placed in the app's `_Imports.razor` file instead of in the component. Use of the `_Imports.razor` file makes the namespace available to larger segments of the app or the whole app.

To persist the `currentCount` value in the `Counter` component of an app based on the [Blazor project template](#), modify the `IncrementCount` method to use

`ProtectedSessionStore.SetAsync`:

C#

```
private async Task IncrementCount()
{
    currentCount++;
    await ProtectedSessionStore.SetAsync("count", currentCount);
}
```

In larger, more realistic apps, storage of individual fields is an unlikely scenario. Apps are more likely to store entire model objects that include complex state.

`ProtectedSessionStore` automatically serializes and deserializes JSON data to store complex state objects.

In the preceding code example, the `currentCount` data is stored as `sessionStorage['count']` in the user's browser. The data isn't stored in plain text but rather is protected using ASP.NET Core Data Protection. The encrypted data can be inspected if `sessionStorage['count']` is evaluated in the browser's developer console.

To recover the `currentCount` data if the user returns to the `Counter` component later, including if the user is on a new circuit, use `ProtectedSessionStore.GetAsync`:

C#

```
protected override async Task OnInitializedAsync()
{
    var result = await ProtectedSessionStore.GetAsync<int>("count");
    currentCount = result.Success ? result.Value : 0;
}
```

If the component's parameters include navigation state, call

`ProtectedSessionStore.GetAsync` and assign a non-`null` result in

`OnParametersSetAsync`, not `OnInitializedAsync`. `OnInitializedAsync` is only called once when the component is first instantiated. `OnInitializedAsync` isn't called again later if the user navigates to a different URL while remaining on the same page. For more information, see [ASP.NET Core Razor component lifecycle](#).

Warning

The examples in this section only work if the server doesn't have prerendering enabled. With prerendering enabled, an error is generated explaining that JavaScript interop calls cannot be issued because the component is being prerendered.

Either disable prerendering or add additional code to work with prerendering. To learn more about writing code that works with prerendering, see the [Handle prerendering](#) section.

Handle the loading state

Since browser storage is accessed asynchronously over a network connection, there's always a period of time before the data is loaded and available to a component. For the

best results, render a message while loading is in progress instead of displaying blank or default data.

One approach is to track whether the data is `null`, which means that the data is still loading. In the default `Counter` component, the count is held in an `int`. [Make `currentCount` nullable](#) by adding a question mark (?) to the type (`int`):

C#

```
private int? currentCount;
```

Instead of unconditionally displaying the count and `Increment` button, display these elements only if the data is loaded by checking [HasValue](#):

razor

```
@if (currentCount.HasValue)
{
    <p>Current count: <strong>@currentCount</strong></p>
    <button @onclick="IncrementCount">Increment</button>
}
else
{
    <p>Loading...</p>
}
```

Handle prerendering

During prerendering:

- An interactive connection to the user's browser doesn't exist.
- The browser doesn't yet have a page in which it can run JavaScript code.

`localStorage` or `sessionStorage` aren't available during prerendering. If the component attempts to interact with storage, an error is generated explaining that JavaScript interop calls cannot be issued because the component is being prerendered.

One way to resolve the error is to disable prerendering. This is usually the best choice if the app makes heavy use of browser-based storage. Prerendering adds complexity and doesn't benefit the app because the app can't prerender any useful content until `localStorage` or `sessionStorage` are available.

To disable prerendering, indicate the render mode with the `prerender` parameter set to `false` at the highest-level component in the app's component hierarchy that isn't a root

component.

❗ Note

Making a root component interactive, such as the `App` component, isn't supported. Therefore, prerendering can't be disabled directly by the `App` component.

For apps based on the Blazor Web App project template, prerendering is typically disabled where the `Routes` component is used in the `App` component

(`Components/App.razor`):

razor

```
<Routes @rendermode="new InteractiveServerRenderMode(prerender: false)" />
```

Also, disable prerendering for the `HeadOutlet` component:

razor

```
<HeadOutlet @rendermode="new InteractiveServerRenderMode(prerender: false)" />
```

For more information, see [ASP.NET Core Blazor render modes](#).

When prerendering is disabled, [prerendering of <head> content](#) is disabled.

Prerendering might be useful for other pages that don't use `localStorage` or `sessionStorage`. To retain prerendering, defer the loading operation until the browser is connected to the circuit. The following is an example for storing a counter value:

razor

```
@using Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage
@inject ProtectedLocalStorage ProtectedLocalStore

@if (isConnected)
{
    <p>Current count: <strong>@currentCount</strong></p>
    <button @onclick="IncrementCount">Increment</button>
}
else
{
    <p>Loading...</p>
}

@code {
```

```

private int currentCount;
private bool isConnected;

protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender)
    {
        isConnected = true;
        await LoadStateAsync();
        StateHasChanged();
    }
}

private async Task LoadStateAsync()
{
    var result = await ProtectedLocalStorage.GetAsync<int>("count");
    currentCount = result.Success ? result.Value : 0;
}

private async Task IncrementCount()
{
    currentCount++;
    await ProtectedLocalStorage.SetAsync("count", currentCount);
}
}

```

Factor out the state preservation to a common location

If many components rely on browser-based storage, implementing state provider code many times creates code duplication. One option for avoiding code duplication is to create a *state provider parent component* that encapsulates the state provider logic. Child components can work with persisted data without regard to the state persistence mechanism.

In the following example of a `CounterStateProvider` component, counter data is persisted to `sessionStorage`:

razor

```

@using Microsoft.AspNetCore.Components.Server.ProtectedBrowserStorage
@inject ProtectedSessionStorage ProtectedSessionStore

@if (isLoading)
{
    <CascadingValue Value="this">
        @ChildContent
    </CascadingValue>
}
else
{

```

```

    <p>Loading...</p>
}

@code {
    private bool isLoading;

    [Parameter]
    public RenderFragment? ChildContent { get; set; }

    public int CurrentCount { get; set; }

    protected override async Task OnInitializedAsync()
    {
        var result = await ProtectedSessionStore.GetAsync<int>("count");
        CurrentCount = result.Success ? result.Value : 0;
        isLoading = true;
    }

    public async Task SaveChangesAsync()
    {
        await ProtectedSessionStore.SetAsync("count", CurrentCount);
    }
}

```

ⓘ Note

For more information on [RenderFragment](#), see [ASP.NET Core Razor components](#).

The `CounterStateProvider` component handles the loading phase by not rendering its child content until state loading is complete.

To make the state accessible to all components in an app, wrap the `CounterStateProvider` component around the `Router` (`<Router>...</Router>`) in the `Routes` component with global interactive server-side rendering (interactive SSR).

In the `App` component (`Components/App.razor`):

```

razor

<Routes @rendermode="InteractiveServer" />

```

In the `Routes` component (`Components/Routes.razor`):

```

razor

<CounterStateProvider>
    <Router ...>
    ...

```

```
</Router>
</CounterStateProvider>
```

Wrapped components receive and can modify the persisted counter state. The following `Counter` component implements the pattern:

razor

```
@page "/counter"

<p>Current count: <strong>@CounterStateProvider?.CurrentCount</strong></p>
<button @onclick="IncrementCount">Increment</button>

@code {
    [CascadingParameter]
    private CounterStateProvider? CounterStateProvider { get; set; }

    private async Task IncrementCount()
    {
        if (CounterStateProvider is not null)
        {
            CounterStateProvider.CurrentCount++;
            await CounterStateProvider.SaveChangesAsync();
        }
    }
}
```

The preceding component isn't required to interact with `ProtectedBrowserStorage`, nor does it deal with a "loading" phase.

To deal with prerendering as described earlier, `CounterStateProvider` can be amended so that all of the components that consume the counter data automatically work with prerendering. For more information, see the [Handle prerendering](#) section.

In general, the *state provider parent component* pattern is recommended:

- To consume state across many components.
- If there's just one top-level state object to persist.

To persist many different state objects and consume different subsets of objects in different places, it's better to avoid persisting state globally.

In-memory state container service

Nested components typically bind data using *chained bind* as described in [ASP.NET Core Blazor data binding](#). Nested and unnested components can share access to data using a

registered in-memory state container. A custom state container class can use an assignable [Action](#) to notify components in different parts of the app of state changes. In the following example:

- A pair of components uses a state container to track a property.
- One component in the following example is nested in the other component, but nesting isn't required for this approach to work.

Important

The example in this section demonstrates how to create an in-memory state container service, register the service, and use the service in components. The example doesn't persist data without further development. For persistent storage of data, the state container must adopt an underlying storage mechanism that survives when browser memory is cleared. This can be accomplished with `localStorage` / `sessionStorage` or some other technology.

StateContainer.cs:

C#

```
public class StateContainer
{
    private string? savedString;

    public string Property
    {
        get => savedString ?? string.Empty;
        set
        {
            savedString = value;
            NotifyStateChanged();
        }
    }

    public event Action? OnChange;

    private void NotifyStateChanged() => OnChange?.Invoke();
}
```

Client-side apps (`Program` file):

C#

```
builder.Services.AddSingleton<StateContainer>();
```

Server-side apps (Program file, ASP.NET Core in .NET 6 or later):

C#

```
builder.Services.AddScoped<StateContainer>();
```

Server-side apps (Startup.ConfigureServices of Startup.cs, ASP.NET Core earlier than 6.0):

C#

```
services.AddScoped<StateContainer>();
```

Shared/Nested.razor:

razor

```
@implements IDisposable
@inject StateContainer StateContainer

<h2>Nested component</h2>

<p>Nested component Property: <b>@StateContainer.Property</b></p>

<p>
    <button @onclick="ChangePropertyValue">
        Change the Property from the Nested component
    </button>
</p>

@code {
    protected override void OnInitialized()
    {
        StateContainer.OnChange += StateHasChanged;
    }

    private void ChangePropertyValue()
    {
        StateContainer.Property =
            $"New value set in the Nested component: {DateTime.Now}";
    }

    public void Dispose()
    {
        StateContainer.OnChange -= StateHasChanged;
    }
}
```

StateContainerExample.razor:


```

@page "/state-container-example"
@implements IDisposable
@inject StateContainer StateContainer

<h1>State Container Example component</h1>

<p>State Container component Property: <b>@StateContainer.Property</b></p>

<p>
    <button @onclick="ChangePropertyValue">
        Change the Property from the State Container Example component
    </button>
</p>

<Nested />

@code {
    protected override void OnInitialized()
    {
        StateContainer.OnChange += StateHasChanged;
    }

    private void ChangePropertyValue()
    {
        StateContainer.Property = "New value set in the State " +
            $"Container Example component: {DateTime.Now}";
    }

    public void Dispose()
    {
        StateContainer.OnChange -= StateHasChanged;
    }
}

```

The preceding components implement [IDisposable](#), and the `OnChange` delegates are unsubscribed in the `Dispose` methods, which are called by the framework when the components are disposed. For more information, see [ASP.NET Core Razor component lifecycle](#).

Additional approaches

When implementing custom state storage, a useful approach is to adopt [cascading values and parameters](#):

- To consume state across many components.
- If there's just one top-level state object to persist.

Troubleshoot

In a custom state management service, a callback invoked outside of Blazor's synchronization context must wrap the logic of the callback in [ComponentBase.InvokeAsync](#) to move it onto the renderer's synchronization context.

When the state management service doesn't call [StateHasChanged](#) on Blazor's synchronization context, the following error is thrown:

```
System.InvalidOperationException: 'The current thread is not associated with the Dispatcher. Use InvokeAsync() to switch execution to the Dispatcher when triggering rendering or component state.'
```

For more information and an example of how to address this error, see [ASP.NET Core Razor component rendering](#).

Additional resources

- [Save app state before an authentication operation \(Blazor WebAssembly\)](#)
- Managing state via an external server API
 - [Call a web API from an ASP.NET Core Blazor app](#)
 - [Secure ASP.NET Core Blazor WebAssembly](#)

Debug ASP.NET Core apps

Article • 10/21/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

This article describes how to debug Blazor apps, including debugging Blazor WebAssembly apps with browser developer tools or an integrated development environment (IDE).

Blazor Web Apps can be debugged in Visual Studio or Visual Studio Code.


Blazor WebAssembly apps can be debugged:

- In Visual Studio or Visual Studio Code.
- Using browser developer tools in Chromium-based browsers, including Microsoft Edge, Google Chrome, and Firefox.

Available scenarios for Blazor WebAssembly debugging include:

- Set and remove breakpoints.
- Run the app with debugging support in IDEs.
- Single-step through the code.
- Resume code execution with a keyboard shortcut in IDEs.
- In the *Locals* window, observe the values of local variables.
- See the call stack, including call chains between JavaScript and .NET.
- Use a [symbol server](#) for debugging, configured by Visual Studio preferences.

Unsupported scenarios include:

- Debug in non-local scenarios (for example, [Windows Subsystem for Linux \(WSL\)](#) or [Visual Studio Codespaces](#) .
- Debug in Firefox from Visual Studio or Visual Studio Code.

Prerequisites

This section explains the prerequisites for debugging.

Browser prerequisites

The latest version of the following browsers:

- Google Chrome
- Microsoft Edge
- Firefox (browser developer tools only)

Ensure that firewalls or proxies don't block communication with the debug proxy (`NodeJS` process). For more information, see the [Firewall configuration](#) section.

ⓘ Note

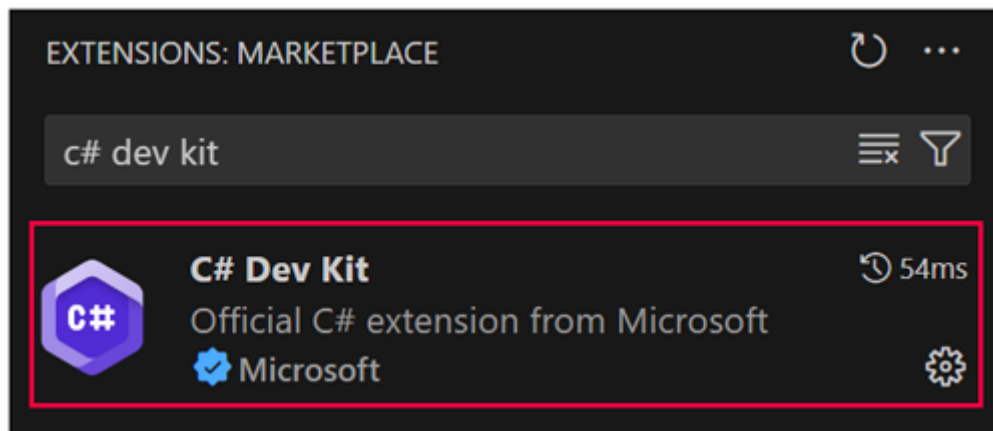
Apple Safari on macOS isn't currently supported.

IDE prerequisites

The latest version of Visual Studio or Visual Studio Code is required.

Visual Studio Code prerequisites

Visual Studio Code requires the [C# Dev Kit for Visual Studio Code](#) (Getting Started with C# in VS Code). In the Visual Studio Code Extensions Marketplace, filter the extension list with "`c# dev kit`" to locate the extension:



Installing the C# Dev Kit automatically installs the following additional extensions:

- [.NET Install Tool](#)
- [C#](#)
- [IntelliCode for C# Dev Kit](#)

If you encounter warnings or errors, you can [open an issue \(microsoft/vscode-dotnettools GitHub repository\)](#) describing the problem.

App configuration prerequisites

The guidance in this subsection applies to client-side debugging.

Open the `Properties/launchSettings.json` file of the startup project. Confirm the presence of the following `inspectUri` property in each launch profile of the file's `profiles` node. If the following property isn't present, add it to each profile:

JSON

```
"inspectUri": "{wsProtocol}://{url.hostname}:{url.port}/_framework/debug/ws-proxy?browser={browserInspectUri}"
```

The `inspectUri` property:

- Enables the IDE to detect that the app is a Blazor app.
- Instructs the script debugging infrastructure to connect to the browser through Blazor's debugging proxy.

The placeholder values for the WebSocket protocol (`wsProtocol`), host (`url.hostname`), port (`url.port`), and inspector URI on the launched browser (`browserInspectUri`) are provided by the framework.


Packages

Blazor Web Apps: [Microsoft.AspNetCore.Components.WebAssembly.Server](#):

References an internal package ([Microsoft.NETCore.BrowserDebugHost.Transport](#)) for assemblies that share the browser debug host.

Standalone Blazor WebAssembly:

[Microsoft.AspNetCore.Components.WebAssembly.DevServer](#): Development server for use when building Blazor apps. Calls [UseWebAssemblyDebugging](#) internally to add middleware for debugging Blazor WebAssembly apps inside Chromium developer tools.

 **Note**

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at [Package consumption workflow \(NuGet documentation\)](#). Confirm correct package versions at [NuGet.org](#).

Debug a Blazor Web App in an IDE

Visual Studio

The example in this section assumes that you've created a Blazor Web App with an interactive render mode of Auto (Server and WebAssembly) and per-component interactivity location.

1. Open the app.
2. Set a breakpoint on the `currentCount++;` line in the `Counter` component (`Pages/Counter.razor`) of the client project (`.Client`).
3. With the server project selected in **Solution Explorer**, press `F5` to run the app in the debugger.
4. In the browser, navigate to `Counter` page at `/counter`. Wait a few seconds for the debug proxy to load and run. Select the **Click me** button to hit the breakpoint.
5. In Visual Studio, inspect the value of the `currentCount` field in the **Locals** window.
6. Press `F5` to continue execution.

Breakpoints can also be hit in the server project in statically-rendered and interactively-rendered server-side components.

1. Stop the debugger.
2. In the server app, open the statically-rendered `Weather` component (`Components/Pages/Weather.razor`) and set a breakpoint anywhere in the `OnInitializedAsync` method.
3. Press `F5` to run the app in the debugger.
4. In the browser, navigate to the `Weather` page at `/weather`. Wait a few seconds for the debug proxy to load and run. Application execution stops at the breakpoint.
5. Press `F5` to continue execution.

Breakpoints are **not** hit during app startup before the debug proxy is running. This includes breakpoints in the `Program` file and breakpoints in the `OnInitializedAsync`

[lifecycle methods](#) of components that are loaded by the first page requested from the app.

Debug a Blazor WebAssembly app in an IDE

Visual Studio

1. Open the app.
2. Set a breakpoint on the `currentCount++;` line in the `Counter` component (`Pages/Counter.razor`).
3. Press `F5` to run the app in the debugger.
4. In the browser, navigate to `Counter` page at `/counter`. Wait a few seconds for the debug proxy to load and run. Select the **Click me** button to hit the breakpoint.
5. In Visual Studio, inspect the value of the `currentCount` field in the **Locals** window.
6. Press `F5` to continue execution.

Breakpoints are **not** hit during app startup before the debug proxy is running. This includes breakpoints in the `Program` file and breakpoints in the `OnInitializedAsync` [lifecycle methods](#) of components that are loaded by the first page requested from the app.

Attach to an existing Visual Studio Code debugging session

To attach to a running Blazor app, open the `.vscode/launch.json` file and replace the `{URL}` placeholder with the URL where the app is running:

JSON

```
{
  "name": "Attach and Debug",
  "type": "blazorwasm",
  "request": "attach",
  "url": "{URL}"
}
```

Visual Studio Code launch options

The launch configuration options in the following table are supported for the `blazorwasm` debug type (`.vscode/launch.json`).

 Expand table

Option	Description
<code>browser</code>	The browser to launch for the debugging session. Set to <code>edge</code> or <code>chrome</code> . Defaults to <code>edge</code> .
<code>cwd</code>	The working directory to launch the app under.
<code>request</code>	Use <code>launch</code> to launch and attach a debugging session to a Blazor WebAssembly app or <code>attach</code> to attach a debugging session to an already-running app.
<code>timeout</code>	The number of milliseconds to wait for the debugging session to attach. Defaults to 30,000 milliseconds (30 seconds).
<code>trace</code>	Used to generate logs from the JS debugger. Set to <code>true</code> to generate logs.
<code>url</code>	The URL to open in the browser when debugging.
<code>webRoot</code>	Specifies the absolute path of the web server. Should be set if an app is served from a sub-route.

Debug Blazor WebAssembly with Google Chrome or Microsoft Edge

The guidance in this section applies debugging Blazor WebAssembly apps in:

- **Google Chrome** running on Windows or macOS.
- ***Microsoft Edge** running on Windows.

1. Run the app in a command shell with `dotnet watch` (Or `dotnet run`).
2. Launch a browser and navigate to the app's URL.
3. Start remote debugging by pressing:

- `Shift` + `Alt` + `d` on Windows.
- `Shift` + `⌘` + `d` on macOS.

The browser must be running with remote debugging enabled, which isn't the default. If remote debugging is disabled, an **Unable to find debuggable browser**

tab error page is rendered with instructions for launching the browser with the debugging port open. Follow the instructions for your browser.

After following the instructions to enable remote debugging, the app opens in a new browser window. Start remote debugging by pressing the HotKey combination in the new browser window:

- **Shift** + **Alt** + **d** on Windows.
- **Shift** + **⌘** + **d** on macOS.

A new window developer tools browser tab opens showing a ghosted image of the app.

ⓘ Note

If you followed the instructions to open a new browser tab with remote debugging enabled, you can close the original browser window, leaving the second window open with the first tab running the app and the second tab running the debugger.

4. After a moment, the **Sources** tab shows a list of the app's .NET assemblies and pages.
5. Open the `file://` node. In component code (`.razor` files) and C# code files (`.cs`), breakpoints that you set are hit when code executes in the app's browser tab (the initial tab opened after starting remote debugging). After a breakpoint is hit, single-step (**F10**) through the code or resume (**F8**) code execution normally in the debugging tab.

For Chromium-based browser debugging, Blazor provides a debugging proxy that implements the [Chrome DevTools Protocol](#) and augments the protocol with .NET-specific information. When debugging keyboard shortcut is pressed, Blazor points the Chrome DevTools at the proxy. The proxy connects to the browser window you're seeking to debug (hence the need to enable remote debugging).

Debug a Blazor WebAssembly app with Firefox

The guidance in this section applies debugging Blazor WebAssembly apps in Firefox running on Windows.

Debugging a Blazor WebAssembly app with Firefox requires configuring the browser for remote debugging and connecting to the browser using the browser developer tools

through the .NET WebAssembly debugging proxy.

ⓘ Note

Debugging in Firefox from Visual Studio isn't supported at this time.

To debug a Blazor WebAssembly app in Firefox during development:

1. Configure Firefox:

- Open `about:config` in a new browser tab. Read and dismiss the warning that appears.
- Enable `devtools.debugger.remote-enabled` by setting its value to `True`.
- Enable `devtools.chrome.enabled` by setting its value to `True`.
- Disable `devtools.debugger.prompt-connection` by setting its value to `False`.

2. Close all Firefox instances.

3. Run the app in a command shell with `dotnet watch` (or `dotnet run`).

4. Relaunch the Firefox browser and navigate to the app.

5. Open `about:debugging` in a new browser tab. **Leave this tab open.**

6. Go back to the tab where the app is running. Start remote debugging by pressing `Shift + Alt + d`.

7. In the `Debugger` tab, open the app source file you wish to debug under the `file://` node and set a breakpoint. For example, set a breakpoint on the `currentCount++;` line in the `IncrementCount` method of the `Counter` component (`Pages/Counter.razor`).

8. Navigate to the `Counter` component page (`/counter`) in the app's browser tab and select the counter button to hit the breakpoint.

9. Press `F5` to continue execution in the debugging tab.

Break on unhandled exceptions

The debugger doesn't break on unhandled exceptions because Blazor catches exceptions that are unhandled by developer code.

To break on unhandled exceptions:

- Open the debugger's exception settings (**Debug > Windows > Exception Settings**) in Visual Studio.
- Set the following **JavaScript Exceptions** settings:
 - **All Exceptions**

- Uncaught Exceptions

Browser source maps

Browser source maps allow the browser to map compiled files back to their original source files and are commonly used for client-side debugging. However, Blazor doesn't currently map C# directly to JavaScript/WASM. Instead, Blazor does IL interpretation within the browser, so source maps aren't relevant.

Firewall configuration

If a firewall blocks communication with the debug proxy, create a firewall exception rule that permits communication between the browser and the `NodeJS` process.

Warning

Modification of a firewall configuration must be made with care to avoid creating security vulnerabilities. Carefully apply security guidance, follow best security practices, and respect warnings issued by the firewall's manufacturer.

Permitting open communication with the `NodeJS` process:

- Opens up the Node server to any connection, depending on the firewall's capabilities and configuration.
- Might be risky depending on your network.
- **Is only recommended on developer machines.**

If possible, only allow open communication with the `NodeJS` process **on trusted or private networks**.

For [Windows Firewall](#) configuration guidance, see [Create an Inbound Program or Service Rule](#). For more information, see [Windows Defender Firewall with Advanced Security](#) and related articles in the Windows Firewall documentation set.

Troubleshoot

If you're running into errors, the following tips may help:

- Remove breakpoints:

- Google Chrome: In the **Debugger** tab, open the developer tools in your browser. In the console, execute `localStorage.clear()` to remove any breakpoints.
- Microsoft Edge: In the **Application** tab, open **Local storage**. Right-click the site and select **Clear**.
- Confirm that you've installed and trusted the ASP.NET Core HTTPS development certificate. For more information, see [Enforce HTTPS in ASP.NET Core](#).
- Visual Studio requires the **Enable JavaScript debugging for ASP.NET (Chrome and Edge)** option in **Tools > Options > Debugging > General**. This is the default setting for Visual Studio. If debugging isn't working, confirm that the option is selected.
- If your environment uses an HTTP proxy, make sure that `localhost` is included in the proxy bypass settings. This can be done by setting the `NO_PROXY` environment variable in either:
 - The `launchSettings.json` file for the project.
 - At the user or system environment variables level for it to apply to all apps. When using an environment variable, restart Visual Studio for the change to take effect.
- Ensure that firewalls or proxies don't block communication with the debug proxy (`NodeJS` process). For more information, see the [Firewall configuration](#) section.

Breakpoints in `OnInitialized{Async}` not hit

The Blazor framework's debugging proxy doesn't launch instantly on app startup, so breakpoints in the [OnInitialized{Async} lifecycle methods](#) might not be hit. We recommend adding a delay at the start of the method body to give the debug proxy some time to launch before the breakpoint is hit. You can include the delay based on an [if compiler directive](#) to ensure that the delay isn't present for a release build of the app.

[OnInitialized](#):

```
C#  
  
protected override void OnInitialized()  
{  
    #if DEBUG  
        Thread.Sleep(10000);  
    #endif  
  
    ...  
}
```

[OnInitializedAsync](#):

C#

```
protected override async Task OnInitializedAsync()
{
    #if DEBUG
        await Task.Delay(10000);
    #endif

    ...
}
```

Visual Studio (Windows) timeout

If Visual Studio throws an exception that the debug adapter failed to launch mentioning that the timeout was reached, you can adjust the timeout with a Registry setting:

Console

```
VsRegEdit.exe set "<VSInstallFolder>" HKCU JSDebugger\Options\Debugging
"BlazorTimeoutInMilliseconds" dword {TIMEOUT}
```

The {TIMEOUT} placeholder in the preceding command is in milliseconds. For example, one minute is assigned as 60000.

Lazy load assemblies in ASP.NET Core Blazor WebAssembly

Article • 12/02/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

Blazor WebAssembly app startup performance can be improved by waiting to load developer-created app assemblies until the assemblies are required, which is called *lazy loading*.

This article's initial sections cover the app configuration. For a working demonstration, see the [Complete example](#) section at the end of this article.

This article only applies to Blazor WebAssembly apps. Assembly lazy loading doesn't benefit server-side apps because server-rendered apps don't download assemblies to the client.

Lazy loading shouldn't be used for core runtime assemblies, which might be trimmed on publish and unavailable on the client when the app loads.

File extension placeholder (`{FILE EXTENSION}`) for assembly files

Assembly files use the [Webcil packaging format for .NET assemblies](#) with a `.wasm` file extension.

Throughout the article, the `{FILE EXTENSION}` placeholder represents "`wasm`".

Project file configuration

Mark assemblies for lazy loading in the app's project file (`.csproj`) using the `BlazorWebAssemblyLazyLoad` item. Use the assembly name with file extension. The Blazor framework prevents the assembly from loading at app launch.

XML

```
<ItemGroup>
  <BlazorWebAssemblyLazyLoad Include="{ASSEMBLY NAME}.{FILE EXTENSION}" />
</ItemGroup>
```

The `{ASSEMBLY NAME}` placeholder is the name of the assembly, and the `{FILE EXTENSION}` placeholder is the file extension. The file extension is required.

Include one `BlazorWebAssemblyLazyLoad` item for each assembly. If an assembly has dependencies, include a `BlazorWebAssemblyLazyLoad` entry for each dependency.

Router component configuration

The Blazor framework automatically registers a singleton service for lazy loading assemblies in client-side Blazor WebAssembly apps, [LazyAssemblyLoader](#). The [LazyAssemblyLoader.LoadAssembliesAsync](#) method:

- Uses [JS interop](#) to fetch assemblies via a network call.
- Loads assemblies into the runtime executing on WebAssembly in the browser.

Blazor's [Router](#) component designates the assemblies that Blazor searches for routable components and is also responsible for rendering the component for the route where the user navigates. The [Router](#) component's [OnNavigateAsync](#) method is used in conjunction with lazy loading to load the correct assemblies for endpoints that a user requests.

Logic is implemented inside [OnNavigateAsync](#) to determine the assemblies to load with [LazyAssemblyLoader](#). Options for how to structure the logic include:

- Conditional checks inside the [OnNavigateAsync](#) method.
- A lookup table that maps routes to assembly names, either injected into the component or implemented within the component's code.

In the following example:

- The namespace for [Microsoft.AspNetCore.Components.WebAssembly.Services](#) is specified.
- The [LazyAssemblyLoader](#) service is injected (`AssemblyLoader`).
- The `{PATH}` placeholder is the path where the list of assemblies should load. The example uses a conditional check for a single path that loads a single set of assemblies.

- The `{LIST OF ASSEMBLIES}` placeholder is the comma-separated list of assembly file name strings, including their file extensions (for example, `"Assembly1.{FILE EXTENSION}"`, `"Assembly2.{FILE EXTENSION}"`).

App.razor:

razor

```
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.WebAssembly.Services
@using Microsoft.Extensions.Logging
@inject LazyAssemblyLoader AssemblyLoader
@inject ILogger<App> Logger

<Router AppAssembly="typeof(App).Assembly"
    OnNavigateAsync="OnNavigateAsync">
    ...
</Router>

@code {
    private async Task OnNavigateAsync(NavigationContext args)
    {
        try
        {
            if (args.Path == "{PATH}")
            {
                var assemblies = await
AssemblyLoader.LoadAssembliesAsync(
                    new[] { {LIST OF ASSEMBLIES} });
            }
        }
        catch (Exception ex)
        {
            Logger.LogError("Error: {Message}", ex.Message);
        }
    }
}
```

ⓘ Note

The preceding example doesn't show the contents of the [Router](#) component's Razor markup (`...`). For a demonstration with complete code, see the [Complete example](#) section of this article.

Assemblies that include routable components

When the list of assemblies includes routable components, the assembly list for a given path is passed to the [Router](#) component's [AdditionalAssemblies](#) collection.

In the following example:

- The `List<Assembly>` in `lazyLoadedAssemblies` passes the assembly list to [AdditionalAssemblies](#). The framework searches the assemblies for routes and updates the route collection if new routes are found. To access the [Assembly](#) type, the namespace for [System.Reflection](#) is included at the top of the `App.razor` file.
- The `{PATH}` placeholder is the path where the list of assemblies should load. The example uses a conditional check for a single path that loads a single set of assemblies.
- The `{LIST OF ASSEMBLIES}` placeholder is the comma-separated list of assembly file name strings, including their file extensions (for example, `"Assembly1.{FILE EXTENSION}"`, `"Assembly2.{FILE EXTENSION}"`).

`App.razor`:

razor

```
@using System.Reflection
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.WebAssembly.Services
@using Microsoft.Extensions.Logging
@inject ILogger<App> Logger
@inject LazyAssemblyLoader AssemblyLoader

<Router AppAssembly="typeof(App).Assembly"
    AdditionalAssemblies="lazyLoadedAssemblies"
    OnNavigateAsync="OnNavigateAsync">
    ...
</Router>

@code {
    private List<Assembly> lazyLoadedAssemblies = new();

    private async Task OnNavigateAsync(NavigationContext args)
    {
        try
        {
            if (args.Path == "{PATH}")
            {
                var assemblies = await
AssemblyLoader.LoadAssembliesAsync(
                    new[] { {LIST OF ASSEMBLIES} });
                lazyLoadedAssemblies.AddRange(assemblies);
            }
        }
        catch (Exception ex)
```

```
    {  
        Logger.LogError("Error: {Message}", ex.Message);  
    }  
}
```

ⓘ Note

The preceding example doesn't show the contents of the [Router](#) component's Razor markup (...). For a demonstration with complete code, see the [Complete example](#) section of this article.

For more information, see [ASP.NET Core Blazor routing and navigation](#).

User interaction with `<Navigating>` content

While loading assemblies, which can take several seconds, the [Router](#) component can indicate to the user that a page transition is occurring with the router's [Navigating](#) property.

For more information, see [ASP.NET Core Blazor routing and navigation](#).

Handle cancellations in `OnNavigateAsync`

The [NavigationContext](#) object passed to the [OnNavigateAsync](#) callback contains a [CancellationToken](#) that's set when a new navigation event occurs. The [OnNavigateAsync](#) callback must throw when the cancellation token is set to avoid continuing to run the [OnNavigateAsync](#) callback on an outdated navigation.

For more information, see [ASP.NET Core Blazor routing and navigation](#).

`OnNavigateAsync` events and renamed assembly files

The resource loader relies on the assembly names that are defined in the `blazor.boot.json` file. If [assemblies are renamed](#), the assembly names used in an [OnNavigateAsync](#) callback and the assembly names in the `blazor.boot.json` file are out of sync.

To rectify this:

- Check to see if the app is running in the `Production` environment when determining which assembly names to use.
- Store the renamed assembly names in a separate file and read from that file to determine what assembly name to use with the [LazyAssemblyLoader](#) service and [OnNavigateAsync](#) callback.

Complete example

The demonstration in this section:

- Creates a robot controls assembly (`GrantImaharaRobotControls.{FILE_EXTENSION}`) as a [Razor class library \(RCL\)](#) that includes a `Robot` component (`Robot.razor` with a route template of `/robot`).
- Lazily loads the RCL's assembly to render its `Robot` component when the `/robot` URL is requested by the user.

Create a standalone Blazor WebAssembly app to demonstrate lazy loading of a Razor class library's assembly. Name the project `LazyLoadTest`.

Add an ASP.NET Core class library project to the solution:

- Visual Studio: Right-click the solution file in **Solution Explorer** and select **Add > New project**. From the dialog of new project types, select **Razor Class Library**. Name the project `GrantImaharaRobotControls`. Do **not** select the **Support pages and views** checkbox.
- Visual Studio Code/.NET CLI: Execute `dotnet new razorclasslib -o GrantImaharaRobotControls` from a command prompt. The `-o|--output` option creates a folder and names the project `GrantImaharaRobotControls`.

Create a `HandGesture` class in the RCL with a `ThumbUp` method that hypothetically makes a robot perform a thumbs-up gesture. The method accepts an argument for the axis, `Left` or `Right`, as an [enum](#). The method returns `true` on success.

`HandGesture.cs`:

C#

```
using Microsoft.Extensions.Logging;

namespace GrantImaharaRobotControls;

public static class HandGesture
{
    public static bool ThumbUp(Axis axis, ILogger logger)
```

```

    {
        logger.LogInformation("Thumb up gesture. Axis: {Axis}", axis);

        // Code to make robot perform gesture

        return true;
    }
}

public enum Axis { Left, Right }

```

Add the following component to the root of the RCL project. The component permits the user to submit a left or right hand thumb-up gesture request.

Robot.razor:

```

razor

@page "/robot"
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.Extensions.Logging
@inject ILogger<Robot> Logger

<h1>Robot</h1>

<EditForm FormName="RobotForm" Model="robotModel"
OnValidSubmit="HandleValidSubmit">
    <InputRadioGroup @bind-Value="robotModel.AxisSelection">
        @foreach (var entry in Enum.GetValues<Axis>())
        {
            <InputRadio Value="entry" />
            <text>&nbsp;</text>@entry<br>
        }
    </InputRadioGroup>

    <button type="submit">Submit</button>
</EditForm>

<p>
    @message
</p>

@code {
    private RobotModel robotModel = new() { AxisSelection = Axis.Left };
    private string? message;

    private void HandleValidSubmit()
    {
        Logger.LogInformation("HandleValidSubmit called");

        var result = HandGesture.ThumbUp(robotModel.AxisSelection, Logger);

        message = $"ThumbUp returned {result} at {DateTime.Now}.";
    }
}

```

```

    }

    public class RobotModel
    {
        public Axis AxisSelection { get; set; }
    }
}

```

In the `LazyLoadTest` project, create a project reference for the `GrantImaharaRobotControls` RCL:

- Visual Studio: Right-click the `LazyLoadTest` project and select **Add > Project Reference** to add a project reference for the `GrantImaharaRobotControls` RCL.
- Visual Studio Code/.NET CLI: Execute `dotnet add reference {PATH}` in a command shell from the project's folder. The `{PATH}` placeholder is the path to the RCL project.

Specify the RCL's assembly for lazy loading in the `LazyLoadTest` app's project file (`.csproj`):

XML

```

<ItemGroup>
    <BlazorWebAssemblyLazyLoad Include="GrantImaharaRobotControls.{FILE
EXTENSION}" />
</ItemGroup>

```

The following `Router` component demonstrates loading the `GrantImaharaRobotControls`. `{FILE EXTENSION}` assembly when the user navigates to `/robot`. Replace the app's default `App` component with the following `App` component.

During page transitions, a styled message is displayed to the user with the `<Navigating>` element. For more information, see the [User interaction with <Navigating> content](#) section.

The assembly is assigned to [AdditionalAssemblies](#), which results in the router searching the assembly for routable components, where it finds the `Robot` component. The `Robot` component's route is added to the app's route collection. For more information, see the [ASP.NET Core Blazor routing and navigation](#) article and the [Assemblies that include routable components](#) section of this article.

`App.razor`:

razor

```

@using System.Reflection
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.WebAssembly.Services
@using Microsoft.Extensions.Logging
@inject ILogger<App> Logger
@inject LazyAssemblyLoader AssemblyLoader

<Router AppAssembly="typeof(App).Assembly"
        AdditionalAssemblies="lazyLoadedAssemblies"
        OnNavigateAsync="OnNavigateAsync">
    <Navigating>
        <div style="padding:20px;background-color:blue;color:white">
            <p>Loading the requested page&hellip;</p>
        </div>
    </Navigating>
    <Found Context="routeData">
        <RouteView RouteData="routeData" DefaultLayout="typeof(MainLayout)"
    />
    </Found>
    <NotFound>
        <LayoutView Layout="typeof(MainLayout)">
            <p>Sorry, there's nothing at this address.</p>
        </LayoutView>
    </NotFound>
</Router>

@code {
    private List<Assembly> lazyLoadedAssemblies = new();

    private async Task OnNavigateAsync(NavigationContext args)
    {
        try
        {
            if (args.Path == "robot")
            {
                var assemblies = await AssemblyLoader.LoadAssembliesAsync(
                    new[] { "GrantImaharaRobotControls.{FILE_EXTENSION}" });
                lazyLoadedAssemblies.AddRange(assemblies);
            }
        }
        catch (Exception ex)
        {
            Logger.LogError("Error: {Message}", ex.Message);
        }
    }
}

```

Build and run the app.

When the `Robot` component from the RCL is requested at `/robot`, the `GrantImaharaRobotControls.{FILE_EXTENSION}` assembly is loaded and the `Robot`

component is rendered. You can inspect the assembly loading in the **Network** tab of the browser's developer tools.

Troubleshoot

- If unexpected rendering occurs, such as rendering a component from a previous navigation, confirm that the code throws if the cancellation token is set.
- If assemblies configured for lazy loading unexpectedly load at app start, check that the assembly is marked for lazy loading in the project file.

Additional resources

- [Handle asynchronous navigation events with OnNavigateAsync](#)
- [ASP.NET Core Blazor performance best practices](#)

ASP.NET Core Blazor WebAssembly native dependencies

Article • 04/12/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

Blazor WebAssembly apps can use native dependencies built to run on WebAssembly. You can statically link native dependencies into the .NET WebAssembly runtime using the **.NET WebAssembly build tools**, the same tools used to [ahead-of-time \(AOT\) compile](#) a Blazor app to WebAssembly and to [relink the runtime to remove unused features](#).

This article only applies to Blazor WebAssembly.

.NET WebAssembly build tools

The .NET WebAssembly build tools are based on [Emscripten](#), a compiler toolchain for the web platform. For more information on the build tools, including installation, see [ASP.NET Core Blazor WebAssembly build tools and ahead-of-time \(AOT\) compilation](#).

Add native dependencies to a Blazor WebAssembly app by adding `NativeFileReference` items in the app's project file. When the project is built, each `NativeFileReference` is passed to Emscripten by the .NET WebAssembly build tools so that they are compiled and linked into the runtime. Next, [p/invoke](#) into the native code from the app's .NET code.

Generally, any portable native code can be used as a native dependency with Blazor WebAssembly. You can add native dependencies to C/C++ code or code previously compiled using Emscripten:

- Object files (`.o`)
- Archive files (`.a`)
- Bitcode (`.bc`)

- Standalone WebAssembly modules (`.wasm`)

Prebuilt dependencies typically must be built using the same version of Emscripten used to build the .NET WebAssembly runtime.

ⓘ Note

For [Mono](#) /WebAssembly MSBuild properties and targets, see [WasmApp.targets \(dotnet/runtime GitHub repository\)](#). Official documentation for common MSBuild properties is planned per [Document blazor msbuild configuration options \(dotnet/docs #27395\)](#).

Use native code

Add a simple native C function to a Blazor WebAssembly app:

1. Create a new Blazor WebAssembly project.
2. Add a `Test.c` file to the project.
3. Add a C function for computing factorials.

`Test.c`:

```
C

int fact(int n)
{
    if (n == 0) return 1;
    return n * fact(n - 1);
}
```

4. Add a `NativeFileReference` for `Test.c` in the app's project file:

```
XML

<ItemGroup>
  <NativeFileReference Include="Test.c" />
</ItemGroup>
```

5. In a Razor component, add a `DllImportAttribute` for the `fact` function in the generated `Test` library and call the `fact` method from .NET code in the component.

Pages/NativeCTest.razor:

```
razor

@page "/native-c-test"
@using System.Runtime.InteropServices

<PageTitle>Native C</PageTitle>

<h1>Native C Test</h1>

<p>
    @@fact(3) result: @fact(3)
</p>

@code {
    [DllImport("Test")]
    static extern int fact(int n);
}
```

When you build the app with the .NET WebAssembly build tools installed, the native C code is compiled and linked into the .NET WebAssembly runtime (`dotnet.wasm`). After the app is built, run the app to see the rendered factorial value.

C++ managed method callbacks

Label managed methods that are passed to C++ with the `[UnmanagedCallersOnly]` attribute.

The method marked with the `[UnmanagedCallersOnly]` attribute must be `static`. To call an instance method in a Razor component, pass a `GCHandle` for the instance to C++ and then pass it back to native. Alternatively, use some other method to identify the instance of the component.

The method marked with `[DllImport]` must use a C# 9.0 function pointer rather than a delegate type for the callback argument.

ⓘ Note

For C# function pointer types in `[DllImport]` methods, use `IntPtr` in the method signature on the managed side instead of `delegate *unmanaged<int, void>`. For more information, see [\[WASM\] callback from native code to .NET: Parsing function pointer types in signatures is not supported \(dotnet/runtime #56145\)](#).[↗]

Package native dependencies in a NuGet package

NuGet packages can contain native dependencies for use on WebAssembly. These libraries and their native functionality are then available to any Blazor WebAssembly app. The files for the native dependencies should be built for WebAssembly and packaged in the `browser-wasm` [architecture-specific folder](#). WebAssembly-specific dependencies aren't referenced automatically and must be referenced manually as `NativeFileReferences`. Package authors can choose to add the native references by including a `.props` file in the package with the references.

SkiaSharp example library use

[SkiaSharp](#) is a cross-platform 2D graphics library for .NET based on the native [Skia graphics library](#) with support for Blazor WebAssembly.

To use SkiaSharp in a Blazor WebAssembly app:

1. Add a package reference to the [SkiaSharp.Views.Blazor](#) package in a Blazor WebAssembly project. Use Visual Studio's process for adding packages to an app (**Manage NuGet Packages** with **Include prerelease** selected) or execute the `dotnet add package` command in a command shell:

```
.NET CLI
```

```
dotnet add package --prerelease SkiaSharp.Views.Blazor
```

ⓘ Note

For guidance on adding packages to .NET apps, see the articles under *Install and manage packages* at [Package consumption workflow \(NuGet documentation\)](#). Confirm correct package versions at [NuGet.org](#).

2. Add a `SKCanvasView` component to the app with the following:

- `SkiaSharp` and `SkiaSharp.Views.Blazor` namespaces.
- Logic to draw in the SkiaSharp Canvas View component (`SKCanvasView`).

```
Pages/NativeDependencyExample.razor:
```

```
razor
```

```

@page "/native-dependency-example"
@using SkiaSharp
@using SkiaSharp.Views.Blazor

<PageTitle>Native dependency</PageTitle>

<h1>Native dependency example with SkiaSharp</h1>

<SKCanvasView OnPaintSurface="OnPaintSurface" />

@code {
    private void OnPaintSurface(SKPaintSurfaceEventArgs e)
    {
        var canvas = e.Surface.Canvas;

        canvas.Clear(SKColors.White);

        using var paint = new SKPaint
        {
            Color = SKColors.Black,
            IsAntialias = true,
            TextSize = 24
        };

        canvas.DrawText("SkiaSharp", 0, 24, paint);
    }
}

```

3. Build the app, which might take several minutes. Run the app and navigate to the `NativeDependencyExample` component at `/native-dependency-example`.

Additional resources

[.NET WebAssembly build tools](#)

ASP.NET Core Blazor performance best practices

Article • 09/12/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the [.NET 9 version of this article](#).

Blazor is optimized for high performance in most realistic application UI scenarios. However, the best performance depends on developers adopting the correct patterns and features.

Note

The code examples in this article adopt [nullable reference types \(NRTs\)](#) and [.NET compiler null-state static analysis](#), which are supported in ASP.NET Core in .NET 6 or later.

Optimize rendering speed

Optimize rendering speed to minimize rendering workload and improve UI responsiveness, which can yield a *ten-fold or higher improvement* in UI rendering speed.

Avoid unnecessary rendering of component subtrees

You might be able to remove the majority of a parent component's rendering cost by skipping the rerendering of child component subtrees when an event occurs. You should only be concerned about skipping the rerendering subtrees that are particularly expensive to render and are causing UI lag.

At runtime, components exist in a hierarchy. A root component (the first component loaded) has child components. In turn, the root's children have their own child components, and so on. When an event occurs, such as a user selecting a button, the following process determines which components to rerender:

1. The event is dispatched to the component that rendered the event's handler. After executing the event handler, the component is rerendered.
2. When a component is rerendered, it supplies a new copy of parameter values to each of its child components.
3. After a new set of parameter values is received, each component decides whether to rerender. Components rerender if the parameter values may have changed, for example, if they're mutable objects.

The last two steps of the preceding sequence continue recursively down the component hierarchy. In many cases, the entire subtree is rerendered. Events targeting high-level components can cause expensive rerendering because every component below the high-level component must rerender.

To prevent rendering recursion into a particular subtree, use either of the following approaches:

- Ensure that child component parameters are of primitive immutable types, such as `string`, `int`, `bool`, `DateTime`, and other similar types. The built-in logic for detecting changes automatically skips rerendering if the primitive immutable parameter values haven't changed. If you render a child component with `<Customer CustomerId="item.CustomerId" />`, where `CustomerId` is an `int` type, then the `Customer` component isn't rerendered unless `item.CustomerId` changes.
- Override `ShouldRender`:
 - To accept nonprimitive parameter values, such as complex custom model types, event callbacks, or `RenderFragment` values.
 - If authoring a UI-only component that doesn't change after the initial render, regardless of parameter value changes.

The following airline flight search tool example uses private fields to track the necessary information to detect changes. The previous inbound flight identifier (`prevInboundFlightId`) and previous outbound flight identifier (`prevOutboundFlightId`) track information for the next potential component update. If either of the flight identifiers change when the component's parameters are set in `OnParametersSet`, the component is rerendered because `shouldRender` is set to `true`. If `shouldRender` evaluates to `false` after checking the flight identifiers, an expensive rerender is avoided:

razor

```
@code {  
    private int prevInboundFlightId = 0;  
    private int prevOutboundFlightId = 0;  
    private bool shouldRender;
```

```

[Parameter]
public FlightInfo? InboundFlight { get; set; }

[Parameter]
public FlightInfo? OutboundFlight { get; set; }

protected override void OnParametersSet()
{
    shouldRender = InboundFlight?.FlightId != prevInboundFlightId
        || OutboundFlight?.FlightId != prevOutboundFlightId;

    prevInboundFlightId = InboundFlight?.FlightId ?? 0;
    prevOutboundFlightId = OutboundFlight?.FlightId ?? 0;
}

protected override bool ShouldRender() => shouldRender;
}

```

An event handler can also set `shouldRender` to `true`. For most components, determining rerendering at the level of individual event handlers usually isn't necessary.

For more information, see the following resources:

- [ASP.NET Core Razor component lifecycle](#)
- [ShouldRender](#)
- [ASP.NET Core Razor component rendering](#)

Virtualization

When rendering large amounts of UI within a loop, for example, a list or grid with thousands of entries, the sheer quantity of rendering operations can lead to a lag in UI rendering. Given that the user can only see a small number of elements at once without scrolling, it's often wasteful to spend time rendering elements that aren't currently visible.

Blazor provides the `Virtualize<TItem>` component to create the appearance and scroll behaviors of an arbitrarily-large list while only rendering the list items that are within the current scroll viewport. For example, a component can render a list with 100,000 entries but only pay the rendering cost of 20 items that are visible.

For more information, see [ASP.NET Core Razor component virtualization](#).

Create lightweight, optimized components

Most Razor components don't require aggressive optimization efforts because most components don't repeat in the UI and don't rerender at high frequency. For example,

routable components with an `@page` directive and components used to render high-level pieces of the UI, such as dialogs or forms, most likely appear only one at a time and only rerender in response to a user gesture. These components don't usually create high rendering workload, so you can freely use any combination of framework features without much concern about rendering performance.

However, there are common scenarios where components are repeated at scale and often result in poor UI performance:

- Large nested forms with hundreds of individual elements, such as inputs or labels.
- Grids with hundreds of rows or thousands of cells.
- Scatter plots with millions of data points.

If modelling each element, cell, or data point as a separate component instance, there are often so many of them that their rendering performance becomes critical. This section provides advice on making such components lightweight so that the UI remains fast and responsive.

Avoid thousands of component instances

Each component is a separate island that can render independently of its parents and children. By choosing how to split the UI into a hierarchy of components, you are taking control over the granularity of UI rendering. This can result in either good or poor performance.

By splitting the UI into separate components, you can have smaller portions of the UI rerender when events occur. In a table with many rows that have a button in each row, you may be able to have only that single row rerender by using a child component instead of the whole page or table. However, each component requires additional memory and CPU overhead to deal with its independent state and rendering lifecycle.

In a test performed by the ASP.NET Core product unit engineers, a rendering overhead of around 0.06 ms per component instance was seen in a Blazor WebAssembly app. The test app rendered a simple component that accepts three parameters. Internally, the overhead is largely due to retrieving per-component state from dictionaries and passing and receiving parameters. By multiplication, you can see that adding 2,000 extra component instances would add 0.12 seconds to the rendering time and the UI would begin feeling slow to users.

It's possible to make components more lightweight so that you can have more of them. However, a more powerful technique is often to avoid having so many components to render. The following sections describe two approaches that you can take.