ASP.NET Core Blazor JavaScript with static server-side rendering (static SSR)

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.

This article explains how to load JavaScript (JS) in a Blazor Web App with static server-side rendering (static SSR) and enhanced navigation.

Some apps depend on JS to perform initialization tasks that are specific to each page. When using Blazor's enhanced navigation feature, which allows the user to avoid reloading the entire page, page-specific JS may not be executed again as expected each time an enhanced page navigation occurs.

To avoid this problem, we don't recommended relying on page-specific `<script>` elements placed outside of the layout file applied to the component. Instead, scripts should register an afterWebStarted JS initializer to perform initialization logic and use an event listener to listen for page updates caused by enhanced navigation.

Events

In the following event listener examples, the `{CALLBACK}` placeholder is the callback function.

- Enhanced navigation start (`enhancednavigationstart`) triggers a callback before an enhanced navigation occurs:

  JavaScript

  ```javascript
  blazor.addEventListener("enhancednavigationstart", {CALLBACK});
  ```

- Enhanced navigation end (`enhancednavigationend`) triggers a callback after an enhanced navigation occurs:

JavaScript

```javascript
blazor.addEventListener("enhancednavigationend", {CALLBACK});
```

- Enhanced navigation page load (`enhancedload`) triggers a callback each time the page updates due to an enhanced navigation, including streaming updates:

JavaScript

```javascript
blazor.addEventListener("enhancedload", {CALLBACK});
```

Enhanced page load script example

The following example demonstrates one way to configure JS code to run when a statically-rendered page with enhanced navigation is initially loaded or updated.

The following `PageWithScript` component example is a component in the app that requires scripts to run with static SSR and enhanced navigation. The following component example includes a `PageScript` component from a Razor class library (RCL) that's added to the solution later in this article.

`Components/Pages/PageWithScript.razor`:

razor

```razor
@page "/page-with-script"
@using BlazorPageScript

<PageTitle>Enhanced Load Script Example</PageTitle>

<PageScript Src="./Components/Pages/PageWithScript.razor.js" />

Welcome to my page.
```

In the Blazor Web App, add the following collocated JS file:

- `onLoad` is called when the script is added to the page.
- `onUpdate` is called when the script still exists on the page after an enhanced update.
- `onDispose` is called when the script is removed from the page after an enhanced update.

`Components/Pages/PageWithScript.razor.js`:

JavaScript

```
export function onLoad() {
  console.log('Loaded');
}

export function onUpdate() {
  console.log('Updated');
}

export function onDispose() {
  console.log('Disposed');
}
```

In a Razor Class Library (RCL) (the example RCL is named `BlazorPageScript`), add the following module.

`wwwroot/BlazorPageScript.lib.module.js`:

JavaScript

```
const pageScriptInfoBySrc = new Map();

function registerPageScriptElement(src) {
  if (!src) {
    throw new Error('Must provide a non-empty value for the "src"
attribute.');
  }

  let pageScriptInfo = pageScriptInfoBySrc.get(src);

  if (pageScriptInfo) {
    pageScriptInfo.referenceCount++;
  } else {
    pageScriptInfo = { referenceCount: 1, module: null };
    pageScriptInfoBySrc.set(src, pageScriptInfo);
    initializePageScriptModule(src, pageScriptInfo);
  }
}

function unregisterPageScriptElement(src) {
  if (!src) {
    return;
  }

  const pageScriptInfo = pageScriptInfoBySrc.get(src);

  if (!pageScriptInfo) {
    return;
  }

  pageScriptInfo.referenceCount--;
}

async function initializePageScriptModule(src, pageScriptInfo) {
  if (src.startsWith("./")) {
    src = new URL(src.substr(2), document.baseURI).toString();
  }

  const module = await import(src);

  if (pageScriptInfo.referenceCount <= 0) {
    return;
  }

  pageScriptInfo.module = module;
  module.onLoad?.();
  module.onUpdate?.();
}

function onEnhancedLoad() {
  for (const [src, { module, referenceCount }] of pageScriptInfoBySrc) {
    if (referenceCount <= 0) {
      module?.onDispose?.();
      pageScriptInfoBySrc.delete(src);
    }
  }

  for (const { module } of pageScriptInfoBySrc.values()) {
    module?.onUpdate?.();
  }
}

export function afterWebStarted(blazor) {
  customElements.define('page-script', class extends HTMLElement {
    static observedAttributes = ['src'];

    attributeChangedCallback(name, oldValue, newValue) {
      if (name !== 'src') {
        return;
      }

      this.src = newValue;
      unregisterPageScriptElement(oldValue);
      registerPageScriptElement(newValue);
    }

    disconnectedCallback() {
      unregisterPageScriptElement(this.src);
    }
  });

  blazor.addEventListener('enhancedload', onEnhancedLoad);
}
```

In the RCL, add the following `PageScript` component.

`PageScript.razor`:

```razor
<page-script src="@Src"></page-script>

@code {
    [Parameter]
    [EditorRequired]
    public string Src { get; set; } = default!;
}
```

The `PageScript` component functions normally on the top-level of a page.

If you place the `PageScript` component in an app's layout (for example, `MainLayout.razor`), which results in a shared `PageScript` among pages that use the layout, then the component only runs `onLoad` after a full page reload and `onUpdate` when any enhanced page update occurs, including enhanced navigation.

To reuse the same module among pages, but have the `onLoad` and `onDispose` callbacks invoked on each page change, append a query string to the end of the script so that it's recognized as a different module. An app could adopt the convention of using the component's name as the query string value. In the following example, the query string is "`counter`" because this `PageScript` component reference is placed in a `Counter` component. This is merely a suggestion, and you can use whatever query string scheme that you prefer.

```razor
<PageScript Src="./Components/Pages/PageWithScript.razor.js?counter" />
```

To monitor changes in specific DOM elements, use the MutationObserver pattern in JS on the client. For more information, see ASP.NET Core Blazor JavaScript interoperability (JS interop).

Example implementation without using an RCL

The approach described in this article can be implemented directly in a Blazor Web App without using a Razor class library (RCL). For an example, see Enable QR code generation for TOTP authenticator apps in an ASP.NET Core Blazor Web App.

Call a web API from ASP.NET Core Blazor

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 call a web API from a Blazor app.

Package

The System.Net.Http.Json package provides extension methods for System.Net.Http.HttpClient and System.Net.Http.HttpContent that perform automatic serialization and deserialization using System.Text.Json. The `System.Net.Http.Json` package is provided by the .NET shared framework and doesn't require adding a package reference to the app.

Sample apps

See the sample apps in the dotnet/blazor-samples GitHub repository.

`BlazorWebAppCallWebApi`

Call an external (not in the Blazor Web App) todo list web API from a Blazor Web App:

- `Backend`: A web API app for maintaining a todo list, based on Minimal APIs. The web API app is a separate app from the Blazor Web App, possibly hosted on a different server.
- `BlazorApp`/`BlazorApp.Client`: A Blazor Web App that calls the web API app with an HttpClient for todo list operations, such as creating, reading, updating, and deleting (CRUD) items from the todo list.

For client-side rendering (CSR), which includes Interactive WebAssembly components and Auto components that have adopted CSR, calls are made with a preconfigured HttpClient registered in the `Program` file of the client project (`BlazorApp.Client`):

```C#
builder.Services.AddScoped(sp =>
    new HttpClient
    {
        BaseAddress = new Uri(builder.Configuration["FrontendUrl"] ??
"https://localhost:5002")
    });
```

For server-side rendering (SSR), which includes prerendered and interactive Server components, prerendered WebAssembly components, and Auto components that are prerendered or have adopted SSR, calls are made with an HttpClient registered in the `Program` file of the server project (`BlazorApp`):

```C#
builder.Services.AddHttpClient();
```

Call an internal (inside the Blazor Web App) movie list API, where the API resides in the server project of the Blazor Web App:

- `BlazorApp`: A Blazor Web App that maintains a movie list:
  - When operations are performed on the movie list within the app on the server, ordinary API calls are used.
  - When API calls are made by a web-based client, a web API is used for movie list operations, based on Minimal APIs.
- `BlazorApp.Client`: The client project of the Blazor Web App, which contains Interactive WebAssembly and Auto components for user management of the movie list.

For CSR, which includes Interactive WebAssembly components and Auto components that have adopted CSR, calls to the API are made via a client-based service (`ClientMovieService`) that uses a preconfigured HttpClient registered in the `Program` file of the client project (`BlazorApp.Client`). Because these calls are made over a public or private web, the movie list API is a *web API*.

The following example obtains a list of movies from the `/movies` endpoint:

```C#
```

```
public class ClientMovieService(HttpClient http) : IMovieService
{
    public async Task<Movie[]> GetMoviesAsync(bool watchedMovies) =>
        await http.GetFromJsonAsync<Movie[]>("movies") ?? [];
}
```

For SSR, which includes prerendered and interactive Server components, prerendered WebAssembly components, and Auto components that are prerendered or have adopted SSR, calls are made directly via a server-based service (`ServerMovieService`). The API doesn't rely on a network, so it's a standard API for movie list CRUD operations.

The following example obtains a list of movies:

C#

```
public class ServerMovieService(MovieContext db) : IMovieService
{
    public async Task<Movie[]> GetMoviesAsync(bool watchedMovies) =>
        watchedMovies ?
        await db.Movies.Where(t => t.IsWatched).ToArrayAsync() :
        await db.Movies.ToArrayAsync();
}
```

`BlazorWebAppCallWebApi_Weather`

A weather data sample app that uses streaming rendering for weather data.

`BlazorWebAssemblyCallWebApi`

Calls a todo list web API from a Blazor WebAssembly app:

- `Backend`: A web API app for maintaining a todo list, based on Minimal APIs.
- `BlazorTodo`: A Blazor WebAssembly app that calls the web API with a preconfigured HttpClient for todo list CRUD operations.

Server-side scenarios for calling external web APIs

Server-based components call external web APIs using HttpClient instances, typically created using IHttpClientFactory. For guidance that applies to server-side apps, see Make HTTP requests using IHttpClientFactory in ASP.NET Core.

A server-side app doesn't include an HttpClient service. Provide an HttpClient to the app using the HttpClient factory infrastructure.

In the `Program` file:

C#

```
builder.Services.AddHttpClient();
```

The following Razor component makes a request to a web API for GitHub branches similar to the *Basic Usage* example in the Make HTTP requests using IHttpClientFactory in ASP.NET Core article.

`CallWebAPI.razor`:

razor

```
@page "/call-web-api"
@using System.Text.Json
@using System.Text.Json.Serialization
@inject IHttpClientFactory ClientFactory

<h1>Call web API from a Blazor Server Razor component</h1>

@if (getBranchesError || branches is null)
{
    <p>Unable to get branches from GitHub. Please try again later.</p>
}
else
{
    <ul>
        @foreach (var branch in branches)
        {
            <li>@branch.Name</li>
        }
    </ul>
}

@code {
    private IEnumerable<GitHubBranch>? branches = [];
    private bool getBranchesError;
    private bool shouldRender;

    protected override bool ShouldRender() => shouldRender;

    protected override async Task OnInitializedAsync()
    {
        var request = new HttpRequestMessage(HttpMethod.Get,
            "https://api.github.com/repos/dotnet/AspNetCore.Docs/branches");
        request.Headers.Add("Accept", "application/vnd.github.v3+json");
        request.Headers.Add("User-Agent", "HttpClientFactory-Sample");
        var client = ClientFactory.CreateClient();

        var response = await client.SendAsync(request);

        if (response.IsSuccessStatusCode)
        {
            using var responseStream = await 
response.Content.ReadAsStreamAsync();
            branches = await JsonSerializer.DeserializeAsync
                <IEnumerable<GitHubBranch>>(responseStream);
        }
        else
        {
            getBranchesError = true;
        }

        shouldRender = true;
    }

    public class GitHubBranch
    {
        [JsonPropertyName("name")]
        public string? Name { get; set; }
    }
}
```

In the preceding example for C# 12 or later, an empty array (`[]`) is created for the `branches` variable. For earlier versions of C#, create an empty array (`Array.Empty<GitHubBranch>()`).

For an additional working example, see the server-side file upload example that uploads files to a web API controller in the ASP.NET Core Blazor file uploads article.

Service abstractions for web API calls

This section applies to Blazor Web Apps that maintain a web API in the server project or transform web API calls to an external web API.

When using the interactive WebAssembly and Auto render modes, components are prerendered by default. Auto components are also initially rendered interactively from the server before the Blazor bundle downloads to the client and the client-side runtime activates. This means that components using these render modes should be designed so that they run successfully from both the client and the server. If the component must call a server project-based API or transform a request to an external web API (one that's outside of the Blazor Web App) when running on the client, the recommended approach is to abstract that API call behind a service interface and implement client and server versions of the service:

- The client version calls the web API with a preconfigured HttpClient.
- The server version can typically access the server-side resources directly. Injecting an HttpClient on the server that makes calls back to the server isn't recommended, as the network request is typically unnecessary. Alternatively, the API might be external to the server project, but a service abstraction for the server is required to transform the request in some way, for example to add an access token to a proxied request.

When using the WebAssembly render mode, you also have the option of disabling prerendering, so the components only render from the client. For more information, see ASP.NET Core Blazor render modes.

Examples (sample apps):

- Movie list web API in the `BlazorWebAppCallWebApi` sample app.
- Streaming rendering weather data web API in the `BlazorWebAppCallWebApi_Weather` sample app.
- Weather data returned to the client in either the `BlazorWebAppOidc` (non-BFF pattern) or `BlazorWebAppOidcBff` (BFF pattern) sample apps. These apps demonstrate secure (web) API calls. For more information, see Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC).

Blazor Web App external web APIs

This section applies to Blazor Web Apps that call a web API maintained by a separate (external) project, possibly hosted on a different server.

Blazor Web Apps normally prerender client-side WebAssembly components, and Auto components render on the server during static or interactive server-side rendering (SSR). HttpClient services aren't registered by default in a Blazor Web App's main project. If the app is run with only the HttpClient services registered in the `.Client` project, as described in the Add the HttpClient service section, executing the app results in a runtime error:

> InvalidOperationException: Cannot provide a value for property 'Http' on type '...{COMPONENT}'. There is no registered service of type 'System.Net.Http.HttpClient'.

Use ***either*** of the following approaches:

- Add the HttpClient services to the server project to make the HttpClient available during SSR. Use the following service registration in the server project's `Program` file:

  ```C#
  builder.Services.AddHttpClient();
  ```

  HttpClient services are provided by the shared framework, so a package reference in the app's project file isn't required.

  Example: Todo list web API in the `BlazorWebAppCallWebApi` sample app

- If prerendering isn't required for a WebAssembly component that calls the web API, disable prerendering by following the guidance in ASP.NET Core Blazor render modes. If you adopt this approach, you don't need to add HttpClient services to the main project of the Blazor Web App because the component isn't prerendered on the server.

For more information, see Client-side services fail to resolve during prerendering.

Prerendered data

When prerendering, components render twice: first statically, then interactively. State doesn't automatically flow from the prerendered component to the interactive one. If a component performs asynchronous initialization operations and renders different content for different states during initialization, such as a "Loading..." progress indicator, you may see a flicker when the component renders twice.

You can address this by flowing prerendered state using the Persistent Component State API, which the `BlazorWebAppCallWebApi` and `BlazorWebAppCallWebApi_Weather` sample apps demonstrate. When the component renders interactively, it can render the same way using the same state. However, the API doesn't currently work with enhanced navigation, which you can work around by disabling enhanced navigation on links to the page (`data-enhanced-nav=false`). For more information, see the following resources:

- Prerender ASP.NET Core Razor components
- ASP.NET Core Blazor routing and navigation
- Support persistent component state across enhanced page navigations (dotnet/aspnetcore #51584)

Add the `HttpClient` service

The guidance in this section applies to client-side scenarios.

Client-side components call web APIs using a preconfigured HttpClient service, which is focused on making requests back to the server of origin. Additional HttpClient service configurations for other web APIs can be created in developer code. Requests are composed using Blazor JSON helpers or with HttpRequestMessage. Requests can include Fetch API option configuration.

The configuration examples in this section are only useful when a single web API is called for a single HttpClient instance in the app. When the app must call multiple web APIs, each with its own base address and configuration, you can adopt the following approaches, which are covered later in this article:

- Named HttpClient with IHttpClientFactory: Each web API is provided a unique name. When app code or a Razor component calls a web API, it uses a named HttpClient instance to make the call.
- Typed HttpClient: Each web API is typed. When app code or a Razor component calls a web API, it uses a typed HttpClient instance to make the call.

In the `Program` file, add an HttpClient service if it isn't already present from a Blazor project template used to create the app:

C#

```csharp
builder.Services.AddScoped(sp =>
    new HttpClient
    {
        BaseAddress = new Uri(builder.HostEnvironment.BaseAddress)
    });
```

The preceding example sets the base address with `builder.HostEnvironment.BaseAddress` (IWebAssemblyHostEnvironment.BaseAddress), which gets the base address for the app and is typically derived from the `<base>` tag's `href` value in the host page.

The most common use cases for using the client's own base address are:

- The client project (`.Client`) of a Blazor Web App (.NET 8 or later) makes web API calls from WebAssembly components or code that runs on the client in WebAssembly to APIs in the server app.
- The client project (**Client**) of a hosted Blazor WebAssembly app makes web API calls to the server project (**Server**). Note that the hosted Blazor WebAssembly

project template is no longer available in .NET 8 or later. However, hosted Blazor WebAssembly apps remain supported for .NET 8.

If you're calling an external web API (not in the same URL space as the client app), set the URI to the web API's base address. The following example sets the base address of the web API to `https://localhost:5001`, where a separate web API app is running and ready to respond to requests from the client app:

C#

```
builder.Services.AddScoped(sp =>
    new HttpClient
    {
        BaseAddress = new Uri("https://localhost:5001")
    });
```

JSON helpers

HttpClient is available as a preconfigured service for making requests back to the origin server.

HttpClient and JSON helpers (System.Net.Http.Json.HttpClientJsonExtensions) are also used to call third-party web API endpoints. HttpClient is implemented using the browser's Fetch API and is subject to its limitations, including enforcement of the same-origin policy, which is discussed later in this article in the *Cross-Origin Resource Sharing (CORS)* section.

The client's base address is set to the originating server's address. Inject an HttpClient instance into a component using the @inject directive:

razor

```
@using System.Net.Http
@inject HttpClient Http
```

Use the System.Net.Http.Json namespace for access to HttpClientJsonExtensions, including GetFromJsonAsync, PutAsJsonAsync, and PostAsJsonAsync:

razor

```
@using System.Net.Http.Json
```

The following sections cover JSON helpers:

- GET
- POST
- PUT
- PATCH

System.Net.Http includes additional methods for sending HTTP requests and receiving HTTP responses, for example to send a DELETE request. For more information, see the DELETE and additional extension methods section.

GET from JSON (`GetFromJsonAsync`)

GetFromJsonAsync sends an HTTP GET request and parses the JSON response body to create an object.

In the following component code, the `todoItems` are displayed by the component. GetFromJsonAsync is called when the component is finished initializing (OnInitializedAsync).

C#

```
todoItems = await Http.GetFromJsonAsync<TodoItem[]>("todoitems");
```

POST as JSON (`PostAsJsonAsync`)

PostAsJsonAsync sends a POST request to the specified URI containing the value serialized as JSON in the request body.

In the following component code, `newItemName` is provided by a bound element of the component. The `AddItem` method is triggered by selecting a `<button>` element.

C#

```
await Http.PostAsJsonAsync("todoitems", addItem);
```

PostAsJsonAsync returns an HttpResponseMessage. To deserialize the JSON content from the response message, use the ReadFromJsonAsync extension method. The following example reads JSON weather data as an array:

C#

```
var content = await response.Content.ReadFromJsonAsync<WeatherForecast[]>()
??
```

```
    Array.Empty<WeatherForecast>();
```

PUT as JSON (`PutAsJsonAsync`)

PutAsJsonAsync sends an HTTP PUT request with JSON-encoded content.

In the following component code, `editItem` values for `Name` and `IsCompleted` are provided by bound elements of the component. The item's `Id` is set when the item is selected in another part of the UI (not shown) and `EditItem` is called. The `SaveItem` method is triggered by selecting the `<button>` element. The following example doesn't show loading `todoItems` for brevity. See the GET from JSON (GetFromJsonAsync) section for an example of loading items.

```C#
await Http.PutAsJsonAsync($"todoitems/{editItem.Id}", editItem);
```

PutAsJsonAsync returns an HttpResponseMessage. To deserialize the JSON content from the response message, use the ReadFromJsonAsync extension method. The following example reads JSON weather data as an array:

```C#
var content = await response.Content.ReadFromJsonAsync<WeatherForecast[]>()
??
    Array.Empty<WeatherForecast>();
```

PATCH as JSON (`PatchAsJsonAsync`)

PatchAsJsonAsync sends an HTTP PATCH request with JSON-encoded content.

> ⓘ Note
>
> For more information, see JsonPatch in ASP.NET Core web API.

In the following example, PatchAsJsonAsync receives a JSON PATCH document as a plain text string with escaped quotes:

```C#
```

```
await Http.PatchAsJsonAsync(
    $"todoitems/{id}",
    "
[{\"operationType\":2,\"path\":\"/IsComplete\",\"op\":\"replace\",\"value\":
true}]");
```

PatchAsJsonAsync returns an HttpResponseMessage. To deserialize the JSON content from the response message, use the ReadFromJsonAsync extension method. The following example reads JSON todo item data as an array. An empty array is created if no item data is returned by the method, so `content` isn't null after the statement executes:

```C#
var response = await Http.PatchAsJsonAsync(...);
var content = await response.Content.ReadFromJsonAsync<TodoItem[]>() ??
    Array.Empty<TodoItem>();
```

Laid out with indentation, spacing, and unescaped quotes, the unencoded PATCH document appears as the following JSON:

```JSON
[
  {
    "operationType": 2,
    "path": "/IsComplete",
    "op": "replace",
    "value": true
  }
]
```

To simplify the creation of PATCH documents in the app issuing PATCH requests, an app can use .NET JSON PATCH support, as the following guidance demonstrates.

Install the Microsoft.AspNetCore.JsonPatch NuGet package and use the API features of the package to compose a JsonPatchDocument for a PATCH request.

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 `@using` directives for the System.Text.Json, System.Text.Json.Serialization, and Microsoft.AspNetCore.JsonPatch namespaces to the top of the Razor component:

```razor
@using System.Text.Json
@using System.Text.Json.Serialization
@using Microsoft.AspNetCore.JsonPatch
```

Compose the JsonPatchDocument for a `TodoItem` with `IsComplete` set to `true` using the Replace method:

```C#
var patchDocument = new JsonPatchDocument<TodoItem>()
    .Replace(p => p.IsComplete, true);
```

Pass the document's operations (`patchDocument.Operations`) to the PatchAsJsonAsync call:

```C#
private async Task UpdateItem(long id)
{
    await Http.PatchAsJsonAsync(
        $"todoitems/{id}",
        patchDocument.Operations,
        new JsonSerializerOptions()
        {
            DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingDefault
        });
}
```

JsonSerializerOptions.DefaultIgnoreCondition is set to JsonIgnoreCondition.WhenWritingDefault to ignore a property only if it equals the default value for its type.

Add JsonSerializerOptions.WriteIndented set to `true` if you want to present the JSON payload in a pleasant format for display. Writing indented JSON has no bearing on processing PATCH requests and isn't typically performed in production apps for web API requests.

Follow the guidance in the JsonPatch in ASP.NET Core web API article to add a PATCH controller action to the web API. Alternatively, PATCH request processing can be implemented as a Minimal API with the following steps.

Add a package reference for the Microsoft.AspNetCore.Mvc.NewtonsoftJson NuGet package to the web API app.

> **Note**
>
> There's no need to add a package reference for the Microsoft.AspNetCore.JsonPatch package to the app because the reference to the `Microsoft.AspNetCore.Mvc.NewtonsoftJson` package automatically transitively adds a package reference for `Microsoft.AspNetCore.JsonPatch`.

In the `Program` file add an `@using` directive for the Microsoft.AspNetCore.JsonPatch namespace:

```C#
using Microsoft.AspNetCore.JsonPatch;
```

Provide the endpoint to the request processing pipeline of the web API:

```C#
app.MapPatch("/todoitems/{id}", async (long id, TodoContext db) =>
{
    if (await db.TodoItems.FindAsync(id) is TodoItem todo)
    {
        var patchDocument =
            new JsonPatchDocument<TodoItem>().Replace(p => p.IsComplete,
true);
        patchDocument.ApplyTo(todo);
        await db.SaveChangesAsync();

        return TypedResults.Ok(todo);
    }

    return TypedResults.NoContent();
});
```

> **Warning**
>
> As with the other examples in the JsonPatch in ASP.NET Core web API article, the preceding PATCH API doesn't protect the web API from over-posting attacks. For more information, see Tutorial: Create a web API with ASP.NET Core.

For a fully working PATCH experience, see the `BlazorWebAppCallWebApi` sample app.

DELETE (`DeleteAsync`) and additional extension methods

System.Net.Http includes additional extension methods for sending HTTP requests and receiving HTTP responses. HttpClient.DeleteAsync is used to send an HTTP DELETE request to a web API.

In the following component code, the `<button>` element calls the `DeleteItem` method. The bound `<input>` element supplies the `id` of the item to delete.

C#

```csharp
await Http.DeleteAsync($"todoitems/{id}");
```

Named `HttpClient` with `IHttpClientFactory`

IHttpClientFactory services and the configuration of a named HttpClient are supported.

> ⓘ **Note**
>
> An alternative to using a named HttpClient from an IHttpClientFactory is to use a typed HttpClient. For more information, see the Typed HttpClient section.

Add the Microsoft.Extensions.Http NuGet package to the app.

> ⓘ **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 of a client project:

C#

```csharp
builder.Services.AddHttpClient("WebAPI", client =>
    client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress));
```

If the named client is to be used by prerendered client-side components of a Blazor Web App, the preceding service registration should appear in both the server project and the `.Client` project. On the server, `builder.HostEnvironment.BaseAddress` is replaced by the web API's base address, which is described further below.

The preceding client-side example sets the base address with `builder.HostEnvironment.BaseAddress` (IWebAssemblyHostEnvironment.BaseAddress), which gets the base address for the client-side app and is typically derived from the `<base>` tag's `href` value in the host page.

The most common use cases for using the client's own base address are:

- The client project (`.Client`) of a Blazor Web App that makes web API calls from WebAssembly/Auto components or code that runs on the client in WebAssembly to APIs in the server app at the same host address.
- The client project (**Client**) of a hosted Blazor WebAssembly app that makes web API calls to the server project (**Server**).

If you're calling an external web API (not in the same URL space as the client app) or you're configuring the services in a server-side app (for example to deal with prerendering of client-side components on the server), set the URI to the web API's base address. The following example sets the base address of the web API to `https://localhost:5001`, where a separate web API app is running and ready to respond to requests from the client app:

C#

```csharp
builder.Services.AddHttpClient("WebAPI", client =>
    client.BaseAddress = new Uri("https://localhost:5001"));
```

In the following component code:

- An instance of IHttpClientFactory creates a named HttpClient.
- The named HttpClient is used to issue a GET request for JSON weather forecast data from the web API at `/forecast`.

razor

```razor
@inject IHttpClientFactory ClientFactory

...

@code {
    private Forecast[]? forecasts;

    protected override async Task OnInitializedAsync()
    {
        var client = ClientFactory.CreateClient("WebAPI");
        forecasts = await client.GetFromJsonAsync<Forecast[]>("forecast") ??
[];
    }
}
```

The `BlazorWebAppCallWebApi` sample app demonstrates calling a web API with a named HttpClient in its `CallTodoWebApiCsrNamedClient` component. For an additional working demonstration in a client app based on calling Microsoft Graph with a named HttpClient, see Use Graph API with ASP.NET Core Blazor WebAssembly.

Typed `HttpClient`

Typed HttpClient uses one or more of the app's HttpClient instances, default or named, to return data from one or more web API endpoints.

> **Note**
>
> An alternative to using a typed HttpClient is to use a named HttpClient from an IHttpClientFactory. For more information, see the Named HttpClient with IHttpClientFactory section.

Add the Microsoft.Extensions.Http NuGet package to the app.

> **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.

The following example issues a GET request for JSON weather forecast data from the web API at `/forecast`.

`ForecastHttpClient.cs`:

```C#
using System.Net.Http.Json;

namespace BlazorSample.Client;

public class ForecastHttpClient(HttpClient http)
{
    public async Task<Forecast[]> GetForecastAsync() =>
        await http.GetFromJsonAsync<Forecast[]>("forecast") ?? [];
}
```

In the `Program` file of a client project:

```C#
builder.Services.AddHttpClient<ForecastHttpClient>(client =>
    client.BaseAddress = new Uri(builder.HostEnvironment.BaseAddress));
```

If the typed client is to be used by prerendered client-side components of a Blazor Web App, the preceding service registration should appear in both the server project and the `.Client` project. On the server, `builder.HostEnvironment.BaseAddress` is replaced by the web API's base address, which is described further below.

The preceding example sets the base address with `builder.HostEnvironment.BaseAddress` (IWebAssemblyHostEnvironment.BaseAddress), which gets the base address for the client-side app and is typically derived from the `<base>` tag's `href` value in the host page.

The most common use cases for using the client's own base address are:

- The client project (`.Client`) of a Blazor Web App that makes web API calls from WebAssembly/Auto components or code that runs on the client in WebAssembly to APIs in the server app at the same host address.
- The client project (**Client**) of a hosted Blazor WebAssembly app that makes web API calls to the server project (**Server**).

If you're calling an external web API (not in the same URL space as the client app) or you're configuring the services in a server-side app (for example to deal with prerendering of client-side components on the server), set the URI to the web API's base address. The following example sets the base address of the web API to `https://localhost:5001`, where a separate web API app is running and ready to respond to requests from the client app:

```C#
builder.Services.AddHttpClient<ForecastHttpClient>(client =>
    client.BaseAddress = new Uri("https://localhost:5001"));
```

Components inject the typed HttpClient to call the web API.

In the following component code:

- An instance of the preceding `ForecastHttpClient` is injected, which creates a typed HttpClient.
- The typed HttpClient is used to issue a GET request for JSON weather forecast data from the web API.

razor

```razor
@inject ForecastHttpClient Http

...

@code {
    private Forecast[]? forecasts;

    protected override async Task OnInitializedAsync()
    {
        forecasts = await Http.GetForecastAsync();
    }
}
```

The `BlazorWebAppCallWebApi` sample app demonstrates calling a web API with a typed HttpClient in its `CallTodoWebApiCsrTypedClient` component. Note that the component adopts and client-side rendering (CSR) (`InteractiveWebAssembly` render mode) ***with prerendering***, so the typed client service registration appears in the `Program` file of both the server project and the `.Client` project.

Cookie-based request credentials

The guidance in this section applies to client-side scenarios that rely upon an authentication cookie.

For cookie-based authentication, which is considered more secure than bearer token authentication, cookie credentials can be sent with each web API request by calling AddHttpMessageHandler with a DelegatingHandler on a preconfigured HttpClient. The handler configures SetBrowserRequestCredentials with BrowserRequestCredentials.Include, which advises the browser to send credentials with each request, such as cookies or HTTP authentication headers, including for cross-origin requests.

`CookieHandler.cs`:

C#

```csharp
public class CookieHandler : DelegatingHandler
{
    protected override Task<HttpResponseMessage> SendAsync(
        HttpRequestMessage request, CancellationToken cancellationToken)
    {

request.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);
        request.Headers.Add("X-Requested-With", ["XMLHttpRequest"]);

        return base.SendAsync(request, cancellationToken);
    }
}
```

The `CookieHandler` is registered in the `Program` file:

```C#
builder.Services.AddTransient<CookieHandler>();
```

The message handler is added to any preconfigured HttpClient that requires cookie authentication:

```C#
builder.Services.AddHttpClient(...)
    .AddHttpMessageHandler<CookieHandler>();
```

For a demonstration, see Secure ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity.

When composing an HttpRequestMessage, set the browser request credentials and header directly:

```C#
var requestMessage = new HttpRequestMessage() { ... };

requestMessage.SetBrowserRequestCredentials(BrowserRequestCredentials.Includ
e);
requestMessage.Headers.Add("X-Requested-With", ["XMLHttpRequest"]);
```

`HttpClient` and `HttpRequestMessage` with Fetch API request options

The guidance in this section applies to client-side scenarios that rely upon bearer token authentication.

HttpClient (API documentation) and HttpRequestMessage can be used to customize requests. For example, you can specify the HTTP method and request headers. The following component makes a `POST` request to a web API endpoint and shows the response body.

`TodoRequest.razor`:

razor

```razor
@page "/todo-request"
@using System.Net.Http.Headers
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@inject HttpClient Http
@inject IAccessTokenProvider TokenProvider

<h1>ToDo Request</h1>

<h1>ToDo Request Example</h1>

<button @onclick="PostRequest">Submit POST request</button>

<p>Response body returned by the server:</p>

<p>@responseBody</p>

@code {
    private string? responseBody;

    private async Task PostRequest()
    {
        var requestMessage = new HttpRequestMessage()
        {
            Method = new HttpMethod("POST"),
            RequestUri = new Uri("https://localhost:10000/todoitems"),
            Content =
                JsonContent.Create(new TodoItem
                {
                    Name = "My New Todo Item",
                    IsComplete = false
                })
        };

        var tokenResult = await TokenProvider.RequestAccessToken();

        if (tokenResult.TryGetToken(out var token))
        {
            requestMessage.Headers.Authorization =
                new AuthenticationHeaderValue("Bearer", token.Value);

            requestMessage.Content.Headers.TryAddWithoutValidation(
                "x-custom-header", "value");

            var response = await Http.SendAsync(requestMessage);
            var responseStatusCode = response.StatusCode;

            responseBody = await response.Content.ReadAsStringAsync();
        }
    }

    public class TodoItem
    {
        public long Id { get; set; }
        public string? Name { get; set; }
        public bool IsComplete { get; set; }
    }
}
```

Blazor's client-side implementation of HttpClient uses Fetch API and configures the underlying request-specific Fetch API options via HttpRequestMessage extension methods and WebAssemblyHttpRequestMessageExtensions. Set additional options using the generic SetBrowserRequestOption extension method. Blazor and the underlying Fetch API don't directly add or modify request headers. For more information on how user agents, such as browsers, interact with headers, consult external user agent documentation sets and other web resources.

The HTTP response is typically buffered to enable support for synchronous reads on the response content. To enable support for response streaming, use the SetBrowserResponseStreamingEnabled extension method on the request.

To include credentials in a cross-origin request, use the SetBrowserRequestCredentials extension method:

```C#
requestMessage.SetBrowserRequestCredentials(BrowserRequestCredentials.Includ
e);
```

For more information on Fetch API options, see MDN web docs: WindowOrWorkerGlobalScope.fetch(): Parameters.

Handle errors

Handle web API response errors in developer code when they occur. For example, GetFromJsonAsync expects a JSON response from the web API with a `Content-Type` of `application/json`. If the response isn't in JSON format, content validation throws a NotSupportedException.

In the following example, the URI endpoint for the weather forecast data request is misspelled. The URI should be to `WeatherForecast` but appears in the call as `WeatherForcast`, which is missing the letter `e` in `Forecast`.

The GetFromJsonAsync call expects JSON to be returned, but the web API returns HTML for an unhandled exception with a `Content-Type` of `text/html`. The unhandled exception occurs because the path to `/WeatherForcast` isn't found and middleware can't serve a page or view for the request.

In OnInitializedAsync on the client, NotSupportedException is thrown when the response content is validated as non-JSON. The exception is caught in the `catch` block, where custom logic could log the error or present a friendly error message to the user.

`ReturnHTMLOnException.razor`:

```razor
@page "/return-html-on-exception"
@using {PROJECT NAME}.Shared
@inject HttpClient Http

<h1>Fetch data but receive HTML on unhandled exception</h1>

@if (forecasts == null)
{
    <p><em>Loading...</em></p>
}
else
{
    <h2>Temperatures by Date</h2>

    <ul>
        @foreach (var forecast in forecasts)
        {
            <li>
                @forecast.Date.ToShortDateString():
                @forecast.TemperatureC &#8451;
                @forecast.TemperatureF &#8457;
            </li>
        }
    </ul>
}

<p>
    @exceptionMessage
</p>

@code {
    private WeatherForecast[]? forecasts;
    private string? exceptionMessage;
    protected override async Task OnInitializedAsync()
    {
        try
        {
            // The URI endpoint "WeatherForecast" is misspelled on purpose 
on the
            // next line. See the preceding text for more information.
            forecasts = await Http.GetFromJsonAsync<WeatherForecast[]>
("WeatherForcast");
        }
        catch (NotSupportedException exception)
        {
            exceptionMessage = exception.Message;
        }
    }
}
```

Note

The preceding example is for demonstration purposes. A web API can be configured to return JSON even when an endpoint doesn't exist or an unhandled exception occurs on the server.

For more information, see Handle errors in ASP.NET Core Blazor apps.

Cross-Origin Resource Sharing (CORS)

Browser security restricts a webpage from making requests to a different domain than the one that served the webpage. This restriction is called the *same-origin policy*. The same-origin policy restricts (but doesn't prevent) a malicious site from reading sensitive data from another site. To make requests from the browser to an endpoint with a different origin, the *endpoint* must enable Cross-Origin Resource Sharing (CORS).

For more information on server-side CORS, see Enable Cross-Origin Requests (CORS) in ASP.NET Core. The article's examples don't pertain directly to Razor component scenarios, but the article is useful for learning general CORS concepts.

For information on client-side CORS requests, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Antiforgery support

To add antiforgery support to an HTTP request, inject the `AntiforgeryStateProvider` and add a `RequestToken` to the headers collection as a `RequestVerificationToken`:

razor

```razor
@inject AntiforgeryStateProvider Antiforgery
```

C#

```csharp
private async Task OnSubmit()
{
    var antiforgery = Antiforgery.GetAntiforgeryToken();
    var request = new HttpRequestMessage(HttpMethod.Post, "action");
    request.Headers.Add("RequestVerificationToken",
antiforgery.RequestToken);
    var response = await client.SendAsync(request);
    ...
}
```

For more information, see ASP.NET Core Blazor authentication and authorization.

Blazor framework component examples for testing web API access

Various network tools are publicly available for testing web API backend apps directly, such as Firefox Browser Developer. Blazor framework's reference source includes HttpClient test assets that are useful for testing:

HttpClientTest assets 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).

Additional resources

General

- Cross-Origin Resource Sharing (CORS) at W3C
- Enable Cross-Origin Requests (CORS) in ASP.NET Core: Although the content applies to ASP.NET Core apps, not Razor components, the article covers general CORS concepts.

Mitigation of overposting attacks

Web APIs can be vulnerable to an *overposting* attack, also known as a *mass assignment* attack. An overposting attack occurs when a malicious user issues an HTML form POST to the server that processes data for properties that aren't part of the rendered form and that the developer doesn't wish to allow users to modify. The term "overposting" literally means that the malicious user has *over*-POSTed with the form.

For guidance on mitigating overposting attacks, see Tutorial: Create a web API with ASP.NET Core.

Server-side

- Server-side ASP.NET Core Blazor additional security scenarios: Includes coverage on using HttpClient to make secure web API requests.
- Make HTTP requests using IHttpClientFactory in ASP.NET Core
- Enforce HTTPS in ASP.NET Core
- Kestrel HTTPS endpoint configuration

Client-side

- ASP.NET Core Blazor WebAssembly additional security scenarios: Includes coverage on using HttpClient to make secure web API requests.
- Use Graph API with ASP.NET Core Blazor WebAssembly
- Fetch API

Display images and documents in ASP.NET Core Blazor

Article • 10/18/2024

ⓘ **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 approaches for displaying images and documents in Blazor apps.

The examples in this article are available for inspection and use in the Blazor sample apps:

dotnet/blazor-samples GitHub repository: Navigate to the app named `BlazorSample_BlazorWebApp` (8.0 or later), `BlazorSample_Server` (7.0 or earlier), or `BlazorSample_WebAssembly`.

Dynamically set an image source

The following example demonstrates how to dynamically set an image's source with a C# field.

The example in this section uses three image files, named `image1.png`, `image2.png`, and `image3.png`. The images are placed in a folder named `images` in the app's web root (`wwwroot`). The use of the `images` folder is only for demonstration purposes. You can organize static assets in any folder layout that you prefer, including serving assets directly from the `wwwroot` folder.

In the following `ShowImage1` component:

- The image's source (`src`) is dynamically set to the value of `imageSource` in C#.
- The `ShowImage` method updates the `imageSource` field based on an image `id` argument passed to the method.
- Rendered buttons call the `ShowImage` method with an image argument for each of the three available images in the `images` folder. The file name is composed using

the argument passed to the method and matches one of the three images in the `images` folder.

`ShowImage1.razor`:

razor

```razor
@page "/show-image-1"

<PageTitle>Show Image 1</PageTitle>

<h1>Show Image Example 1</h1>

@if (imageSource is not null)
{
    <p>
        <img src="@imageSource" />
    </p>
}

@for (var i = 1; i <= 3; i++)
{
    var imageId = i;
    <button @onclick="() => ShowImage(imageId)">
        Image @imageId
    </button>
}

@code {
    private string? imageSource;

    private void ShowImage(int id) => imageSource = $"images/image{id}.png";
}
```

The preceding example uses a C# field to hold the image's source data, but you can also use a C# property to hold the data.

Avoid using a loop variable directly in a lambda expression, such as `i` in the preceding `for` loop example. Otherwise, the same variable is used by all lambda expressions, which results in use of the same value in all lambdas. Capture the variable's value in a local variable. In the preceding example:

- The loop variable `i` is assigned to `imageId`.
- `imageId` is used in the lambda expression.

Alternatively, use a `foreach` loop with Enumerable.Range, which doesn't suffer from the preceding problem:

razor

```
@foreach (var imageId in Enumerable.Range(1, 3))
{
    <button @onclick="() => ShowImage(imageId)">
        Image @imageId
    </button>
}
```

For more information on lambda expressions with event handling, see ASP.NET Core Blazor event handling.

Stream image or document data

An image or other document type, such as a PDF, can be directly transmitted to the client using Blazor's streaming interop features instead of hosting the file at a public URL.

The example in this section streams source data using JavaScript (JS) interop. The following `setSource` JS function:

- Can be used to stream content for the following elements: `<body>`, `<embed>`, `<iframe>`, `<img>`, `<link>`, `<object>`, `<script>`, `<style>`, and `<track>`.
- Accepts an element `id` to display the file's contents, a data stream for the document, the content type, and a title for the display element.

The function:

- Reads the provided stream into an ArrayBuffer.
- Creates a Blob to wrap the `ArrayBuffer`, setting the blob's content type.
- Creates an object URL to serve as the address for the document to be shown.
- Set's the element's title (`title`) from the `title` parameter and sets the element's source (`src`) from the created object URL.
- To prevent memory leaks, the function calls revokeObjectURL to dispose of the object URL after the element loads the resource (load event).

HTML

```html
<script>
  window.setSource = async (elementId, stream, contentType, title) => {
    const arrayBuffer = await stream.arrayBuffer();
    let blobOptions = {};
    if (contentType) {
      blobOptions['type'] = contentType;
    }
    const blob = new Blob([arrayBuffer], blobOptions);
    const url = URL.createObjectURL(blob);
    const element = document.getElementById(elementId);
    element.title = title;
    element.onload = () => {
      URL.revokeObjectURL(url);
    }
    element.src = url;
  }
</script>
```

ⓘ **Note**

For general guidance on JS location and our recommendations for production apps, see JavaScript location in ASP.NET Core Blazor apps.

The following `ShowImage2` component:

- Injects services for an System.Net.Http.HttpClient and Microsoft.JSInterop.IJSRuntime.
- Includes an `<img>` tag to display an image.
- Has a `GetImageStreamAsync` C# method to retrieve a Stream for an image. A production app may dynamically generate an image based on the specific user or retrieve an image from storage. The following example retrieves the .NET avatar for the `dotnet` GitHub repository.
- Has a `SetImageAsync` method that's triggered on the button's selection by the user. `SetImageAsync` performs the following steps:
  - Retrieves the Stream from `GetImageStreamAsync`.
  - Wraps the Stream in a DotNetStreamReference, which allows streaming the image data to the client.
  - Invokes the `setSource` JavaScript function, which accepts the data on the client.

ⓘ **Note**

Server-side apps use a dedicated HttpClient service to make requests, so no action is required by the developer of a server-side Blazor app to register an HttpClient service. Client-side apps have a default HttpClient service registration when the app is created from a Blazor project template. If an HttpClient service registration isn't present in the `Program` file of a client-side app, provide one by adding `builder.Services.AddHttpClient();`. For more information, see Make HTTP requests using IHttpClientFactory in ASP.NET Core.

`ShowImage2.razor`:

```razor
@page "/show-image-2"
@inject HttpClient Http
@inject IJSRuntime JS

<PageTitle>Show Image 2</PageTitle>

<h1>Show Image Example 2</h1>

<button @onclick="SetImageAsync">
    Set Image
</button>

<div class="p-3">
    <img id="avatar" />
</div>

@code {
    private async Task<Stream> GetImageStreamAsync() =>
        await
Http.GetStreamAsync("https://avatars.githubusercontent.com/u/9141961");

    private async Task SetImageAsync()
    {
        var imageStream = await GetImageStreamAsync();
        var strRef = new DotNetStreamReference(imageStream);
        await JS.InvokeVoidAsync("setSource", "avatar", strRef, "image/png",
            ".NET GitHub avatar");
    }
}
```

The following `ShowFile` component loads either a text file (`files/quote.txt`) or a PDF file (`files/quote.pdf`) into an <iframe> element (MDN documentation).

Caution

Use of the `<iframe>` element in the following example is safe and doesn't require sandboxing because content is loaded from the app, which is a trusted source.

When loading content from an untrusted source or user input, an improperly implemented `<iframe>` element risks creating security vulnerabilities.

`ShowFile.razor`:

```razor
```

```
@page "/show-file"
@inject NavigationManager Navigation
@inject HttpClient Http
@inject IJSRuntime JS

<PageTitle>Show File</PageTitle>

<div class="d-flex flex-column">
    <h1>Show File Example</h1>
    <div class="mb-4">
        <button @onclick="@(() => ShowFileAsync("files/quote.txt",
                "General Ravon quote (text file)"))">
            Show text ('quote.txt')
        </button>
        <button @onclick="@(() => ShowFileAsync("files/quote.pdf",
                "General Ravon quote (PDF file)"))">
            Show PDF ('quote.pdf')
        </button>
    </div>
    <iframe id="iframe" style="height: calc(100vh - 200px)" />
</div>

@code
{
    private async Task<(Stream, string?)> DownloadFileAsync(string url)
    {
        var absoluteUrl = Navigation.ToAbsoluteUri(url);
        Console.WriteLine($"Downloading file from {absoluteUrl}");

        var response = await Http.GetAsync(absoluteUrl);
        string? contentType = null;

        if (response.Content.Headers.TryGetValues("Content-Type", out var
values))
        {
            contentType = values.FirstOrDefault();
        }

        return (await response.Content.ReadAsStreamAsync(), contentType);
    }

    private async Task ShowFileAsync(string url, string title)
    {
        var (fileStream, contentType) = await DownloadFileAsync(url);
        var strRef = new DotNetStreamReference(fileStream);
        await JS.InvokeVoidAsync("setSource", "iframe", strRef, contentType,
title);
    }
}
```

Additional resources

- ASP.NET Core Blazor file uploads
- File uploads: Upload image preview
- ASP.NET Core Blazor file downloads
- Call .NET methods from JavaScript functions in ASP.NET Core Blazor
- Call JavaScript functions from .NET methods in ASP.NET Core Blazor
- Blazor samples GitHub repository (dotnet/blazor-samples) (how to download)

ASP.NET Core Blazor authentication and authorization

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 describes ASP.NET Core's support for the configuration and management of security in Blazor apps.

Blazor uses the existing ASP.NET Core authentication mechanisms to establish the user's identity. The exact mechanism depends on how the Blazor app is hosted, server-side or client-side.

Security scenarios differ between authorization code running server-side and client-side in Blazor apps. For authorization code that runs on the server, authorization checks are able to enforce access rules for areas of the app and components. Because client-side code execution can be tampered with, authorization code executing on the client can't be trusted to absolutely enforce access rules or control the display of client-side content.

If authorization rule enforcement must be guaranteed, don't implement authorization checks in client-side code. Build a Blazor Web App that only relies on server-side rendering (SSR) for authorization checks and rule enforcement.

Razor Pages authorization conventions don't apply to routable Razor components. If a non-routable Razor component is embedded in a page of a Razor Pages app, the page's authorization conventions indirectly affect the Razor component along with the rest of the page's content.

ⓘ **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 examples in this article.

Securely maintain sensitive data and credentials

Don't store app secrets, connection strings, credentials, passwords, personal identification numbers (PINs), private .NET/C# code, or private keys/tokens in client-side code, which is ***always insecure***. Client-side Blazor code should access secure services and databases through a secure web API that you control.

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 the following resources:

- Secure authentication flows (ASP.NET Core documentation)
- Managed identities for Microsoft Azure services (this article)

For client-side and server-side local development and testing, use the Secret Manager tool to secure sensitive credentials.

Managed identities for Microsoft Azure services

For Microsoft Azure services, we recommend using *managed identities*. Managed identities securely authenticate to Azure services without storing credentials in app code. For more information, see the following resources:

- What are managed identities for Azure resources? (Microsoft Entra documentation)
- Azure services documentation
  - Managed identities in Microsoft Entra for Azure SQL
  - How to use managed identities for App Service and Azure Functions

Antiforgery support

The Blazor template:

- Adds antiforgery services automatically when AddRazorComponents is called in the `Program` file.
- Adds Antiforgery Middleware by calling UseAntiforgery in its request processing pipeline in the `Program` file and requires endpoint antiforgery protection to mitigate the threats of Cross-Site Request Forgery (CSRF/XSRF). UseAntiforgery is called after UseHttpsRedirection. A call to UseAntiforgery must be placed after calls, if present, to UseAuthentication and UseAuthorization.

The AntiforgeryToken component renders an antiforgery token as a hidden field, and this component is automatically added to form (EditForm) instances. For more information, see ASP.NET Core Blazor forms overview.

The AntiforgeryStateProvider service provides access to an antiforgery token associated with the current session. Inject the service and call its GetAntiforgeryToken() method to obtain the current AntiforgeryRequestToken. For more information, see Call a web API from an ASP.NET Core Blazor app.

Blazor stores request tokens in component state, which guarantees that antiforgery tokens are available to interactive components, even when they don't have access to the request.

ⓘ Note

Antiforgery mitigation is only required when submitting form data to the server encoded as `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain` since these are the **only valid form enctypes**.

For more information, see the following resources:

- Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core: This article is the primary ASP.NET Core article on the subject, which applies to server-side Blazor Server, the server project of Blazor Web Apps, and Blazor integration with MVC/Razor Pages.
- ASP.NET Core Blazor forms overview: The *Antiforgery support* section of the article pertains to Blazor forms antiforgery support.

Server-side Blazor authentication

Server-side Blazor apps are configured for security in the same manner as ASP.NET Core apps. For more information, see the articles under ASP.NET Core security topics.

The authentication context is only established when the app starts, which is when the app first connects to the WebSocket over a SignalR connection with the client. Authentication can be based on a cookie or some other bearer token, but authentication is managed via the SignalR hub and entirely within the circuit. The authentication context is maintained for the lifetime of the circuit. Apps periodically revalidate the user's authentication state every 30 minutes.

If the app must capture users for custom services or react to updates to the user, see ASP.NET Core server-side and Blazor Web App additional security scenarios.

Blazor differs from a traditional server-rendered web apps that make new HTTP requests with cookies on every page navigation. Authentication is checked during navigation events. However, cookies aren't involved. Cookies are only sent when making an HTTP request to a server, which isn't what happens when the user navigates in a Blazor app. During navigation, the user's authentication state is checked within the Blazor circuit, which you can update at any time on the server using the RevalidatingAuthenticationStateProvider abstraction.

ⓘ **Important**

Implementing a custom `NavigationManager` to achieve authentication validation during navigation isn't recommended. If the app must execute custom authentication state logic during navigation, use a custom AuthenticationStateProvider.

ⓘ **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 examples in this article.

The built-in or custom AuthenticationStateProvider service obtains authentication state data from ASP.NET Core's HttpContext.User. This is how authentication state integrates with existing ASP.NET Core authentication mechanisms.

For more information on server-side authentication, see ASP.NET Core Blazor authentication and authorization.

`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.

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.

Server-side security of sensitive data and credentials

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 the following resources:

- Secure authentication flows (ASP.NET Core documentation)
- Managed identities for Microsoft Azure services (Blazor documentation)

For client-side and server-side local development and testing, use the Secret Manager tool to secure sensitive credentials.

Project template

Create a new server-side Blazor app by following the guidance in Tooling for ASP.NET Core Blazor.

Visual Studio

After choosing the server-side app template and configuring the project, select the app's authentication under **Authentication type**:

- **None** (default): No authentication.
- **Individual Accounts**: User accounts are stored within the app using ASP.NET Core Identity.

Blazor Identity UI (Individual Accounts)

Blazor supports generating a full Blazor-based Identity UI when you choose the authentication option for *Individual Accounts*.

The Blazor Web App template scaffolds Identity code for a SQL Server database. The command line version uses SQLite and includes a SQLite database for Identity.

The template:

- Supports interactive server-side rendering (interactive SSR) and client-side rendering (CSR) scenarios with authenticated users.

- Adds Identity Razor components and related logic for routine authentication tasks, such as signing users in and out. The Identity components also support advanced Identity features, such as account confirmation and password recovery and multifactor authentication using a third-party app. Note that the Identity components themselves don't support interactivity.
- Adds the Identity-related packages and dependencies.
- References the Identity packages in `_Imports.razor`.
- Creates a custom user Identity class (`ApplicationUser`).
- Creates and registers an EF Core database context (`ApplicationDbContext`).
- Configures routing for the built-in Identity endpoints.
- Includes Identity validation and business logic.

To inspect the Blazor framework's Identity components, access them in the `Pages` and `Shared` folders of the Account folder in the Blazor Web App project template (reference source).

When you choose the Interactive WebAssembly or Interactive Auto render modes, the server handles all authentication and authorization requests, and the Identity components render statically on the server in the Blazor Web App's main project.

The framework provides a custom AuthenticationStateProvider in both the server and client (`.Client`) projects to flow the user's authentication state to the browser. The server project calls `AddAuthenticationStateSerialization`, while the client project calls `AddAuthenticationStateDeserialization`. Authenticating on the server rather than the client allows the app to access authentication state during prerendering and before the .NET WebAssembly runtime is initialized. The custom AuthenticationStateProvider implementations use the Persistent Component State service (PersistentComponentState) to serialize the authentication state into HTML comments and then read it back from WebAssembly to create a new AuthenticationState instance. For more information, see the Manage authentication state in Blazor Web Apps section.

Only for Interactive Server solutions, IdentityRevalidatingAuthenticationStateProvider (reference source) is a server-side AuthenticationStateProvider that revalidates the security stamp for the connected user every 30 minutes an interactive circuit is connected.

Blazor Identity depends on DbContext instances not created by a factory, which is intentional because DbContext is sufficient for the project template's Identity components to render statically without supporting interactivity.

For a description on how global interactive render modes are applied to non-Identity components while at the same time enforcing static SSR for the Identity components, see ASP.NET Core Blazor render modes.

For more information on persisting prerendered state, see Prerender ASP.NET Core Razor components.

For more information on the Blazor Identity UI and guidance on integrating external logins through social websites, see What's new with identity in .NET 8.

ⓘ **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).

Manage authentication state in Blazor Web Apps

This section applies to Blazor Web Apps that adopt:

- Individual Accounts
- *Client-side rendering (CSR, WebAssembly-based interactivity).*

A client-side authentication state provider is only used within Blazor and isn't integrated with the ASP.NET Core authentication system. During prerendering, Blazor respects the metadata defined on the page and uses the ASP.NET Core authentication system to determine if the user is authenticated. When a user navigates from one page to another, a client-side authentication provider is used. When the user refreshes the page (full-page reload), the client-side authentication state provider isn't involved in the authentication decision on the server. Since the user's state isn't persisted by the server, any authentication state maintained client-side is lost.

To address this, the best approach is to perform authentication within the ASP.NET Core authentication system. The client-side authentication state provider only takes care of reflecting the user's authentication state. Examples for how to accomplish this with authentication state providers are demonstrated by the Blazor Web App project template and described below.

In the server project's `Program` file, call `AddAuthenticationStateSerialization`, which serializes the AuthenticationState returned by the server-side AuthenticationStateProvider using the Persistent Component State service (PersistentComponentState):

C#

```
builder.Services.AddRazorComponents()
    .AddInteractiveWebAssemblyComponents()
    .AddAuthenticationStateSerialization();
```

The API only serializes the server-side name and role claims for access in the browser. To include all claims, set `SerializeAllClaims` to `true` in the server-side call to `AddAuthenticationStateSerialization`:

C#

```
builder.Services.AddRazorComponents()
    .AddInteractiveWebAssemblyComponents()
    .AddAuthenticationStateSerialization(
        options => options.SerializeAllClaims = true);
```

In the client (`.Client`) project's `Program` file, call `AddAuthenticationStateDeserialization`, which adds an AuthenticationStateProvider where the AuthenticationState is deserialized from the server using `AuthenticationStateData` and the Persistent Component State service (PersistentComponentState). There should be a corresponding call to `AddAuthenticationStateSerialization` in the server project.

C#

```
builder.Services.AddAuthorizationCore();
builder.Services.AddCascadingAuthenticationState();
builder.Services.AddAuthenticationStateDeserialization();
```

Additional claims and tokens from external providers

To store additional claims from external providers, see Persist additional claims and tokens from external providers in ASP.NET Core.

Azure App Service on Linux with Identity Server

Specify the issuer explicitly when deploying to Azure App Service on Linux with Identity Server. For more information, see Use Identity to secure a Web API backend for SPAs.

Inject `AuthenticationStateProvider` for services scoped to a component

Don't attempt to resolve AuthenticationStateProvider within a custom scope because it results in the creation of a new instance of the AuthenticationStateProvider that isn't correctly initialized.

To access the AuthenticationStateProvider within a service scoped to a component, inject the AuthenticationStateProvider with the @inject directive or the [Inject] attribute and pass it to the service as a parameter. This approach ensures that the correct, initialized instance of the AuthenticationStateProvider is used for each user app instance.

`ExampleService.cs`:

```C#
public class ExampleService
{
    public async Task<string> ExampleMethod(AuthenticationStateProvider
authStateProvider)
    {
        var authState = await
authStateProvider.GetAuthenticationStateAsync();
        var user = authState.User;

        if (user.Identity is not null && user.Identity.IsAuthenticated)
        {
            return $"{user.Identity.Name} is authenticated.";
        }
        else
        {
            return "The user is NOT authenticated.";
        }
    }
}
```

Register the service as scoped. In a server-side Blazor app, scoped services have a lifetime equal to the duration of the client connection circuit.

In the `Program` file:

```C#
builder.Services.AddScoped<ExampleService>();
```

In the following `InjectAuthStateProvider` component:

- The component inherits OwningComponentBase.
- The AuthenticationStateProvider is injected and passed to `ExampleService.ExampleMethod`.

- `ExampleService` is resolved with OwningComponentBase.ScopedServices and GetRequiredService, which returns the correct, initialized instance of `ExampleService` that exists for the lifetime of the user's circuit.

`InjectAuthStateProvider.razor`:

```razor
@page "/inject-auth-state-provider"
@inherits OwningComponentBase
@inject AuthenticationStateProvider AuthenticationStateProvider

<h1>Inject <code>AuthenticationStateProvider</code> Example</h1>

<p>@message</p>

@code {
    private string? message;
    private ExampleService? ExampleService { get; set; }

    protected override async Task OnInitializedAsync()
    {
        ExampleService = ScopedServices.GetRequiredService<ExampleService>
();

        message = await
ExampleService.ExampleMethod(AuthenticationStateProvider);
    }
}
```

For more information, see the guidance on OwningComponentBase in ASP.NET Core Blazor dependency injection.

Unauthorized content display while prerendering with a custom `AuthenticationStateProvider`

To avoid showing unauthorized content, for example content in an AuthorizeView component, while prerendering with a custom AuthenticationStateProvider, adopt ***one*** of the following approaches:

- Implement IHostEnvironmentAuthenticationStateProvider for the custom AuthenticationStateProvider to support prerendering: For an example implementation of IHostEnvironmentAuthenticationStateProvider, see the Blazor framework's ServerAuthenticationStateProvider implementation in ServerAuthenticationStateProvider.cs (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).

- 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

  ```razor
  <Routes @rendermode="new InteractiveServerRenderMode(prerender: false)"
  />
  ```

  Also, disable prerendering for the `HeadOutlet` component:

  razor

  ```razor
  <HeadOutlet @rendermode="new InteractiveServerRenderMode(prerender:
  false)" />
  ```

  You can also selectively control the render mode applied to the `Routes` component instance. For example, see ASP.NET Core Blazor render modes.

- Authenticate the user on the server before the app starts: To adopt this approach, the app must respond to a user's initial request with the Identity-based sign-in page or view and prevent any requests to Blazor endpoints until they're authenticated. For more information, see Create an ASP.NET Core app with user data protected by authorization. After authentication, unauthorized content in prerendered Razor components is only shown when the user is truly unauthorized to view the content.

User state management

In spite of the word "state" in the name, AuthenticationStateProvider isn't for storing *general user state*. AuthenticationStateProvider only indicates the user's authentication state to the app, whether they are signed into the app and who they are signed in as.

Authentication uses the same ASP.NET Core Identity authentication as Razor Pages and MVC apps. The user state stored for ASP.NET Core Identity flows to Blazor without adding additional code to the app. Follow the guidance in the ASP.NET Core Identity articles and tutorials for the Identity features to take effect in the Blazor parts of the app.

For guidance on general state management outside of ASP.NET Core Identity, see ASP.NET Core Blazor state management.

Additional security abstractions

Two additional abstractions participate in managing authentication state:

- ServerAuthenticationStateProvider (reference source): An AuthenticationStateProvider used by the Blazor framework to obtain authentication state from the server.

- RevalidatingServerAuthenticationStateProvider (reference source): A base class for AuthenticationStateProvider services used by the Blazor framework to receive an authentication state from the host environment and revalidate it at regular intervals.

  The default 30 minute revalidation interval can be adjusted in RevalidatingIdentityAuthenticationStateProvider (Areas/Identity/RevalidatingIdentityAuthenticationStateProvider.cs). The following example shortens the interval to 20 minutes:

  C#

  ```csharp
  protected override TimeSpan RevalidationInterval =>
  TimeSpan.FromMinutes(20);
  ```

ⓘ 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).

Authentication state management at sign out

Server-side Blazor persists user authentication state for the lifetime of the circuit, including across browser tabs. To proactively sign off a user across browser tabs when the user signs out on one tab, you must implement a RevalidatingServerAuthenticationStateProvider (reference source) with a short RevalidationInterval.

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).

Temporary redirection URL validity duration

This section applies to Blazor Web Apps.

Use the RazorComponentsServiceOptions.TemporaryRedirectionUrlValidityDuration option to get or set the lifetime of ASP.NET Core Data Protection validity for temporary redirection URLs emitted by Blazor server-side rendering. These are only used transiently, so the lifetime only needs to be long enough for a client to receive the URL and begin navigation to it. However, it should also be long enough to allow for clock skew across servers. The default value is five minutes.

In the following example the value is extended to seven minutes:

C#

```
builder.Services.AddRazorComponents(options =>
    options.TemporaryRedirectionUrlValidityDuration =
        TimeSpan.FromMinutes(7));
```

Client-side Blazor authentication

In client-side Blazor apps, client-side authentication checks can be bypassed because all client-side code can be modified by users. The same is true for all client-side app technologies, including JavaScript SPA frameworks and native apps for any operating system.

Add the following:

- A package reference for the Microsoft.AspNetCore.Components.Authorization 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**.

- The Microsoft.AspNetCore.Components.Authorization namespace to the app's `_Imports.razor` file.

To handle authentication, use the built-in or custom AuthenticationStateProvider service.

For more information on client-side authentication, see Secure ASP.NET Core Blazor WebAssembly.

`AuthenticationStateProvider` service

AuthenticationStateProvider is the underlying service used by the AuthorizeView component and cascading authentication services to obtain the authentication state for a user.

You don't typically use AuthenticationStateProvider directly. Use the AuthorizeView component or Task<AuthenticationState> approaches described later in this article. The main drawback to using AuthenticationStateProvider directly is that the component isn't notified automatically if the underlying authentication state data changes.

To implement a custom AuthenticationStateProvider, see ASP.NET Core Blazor authentication state, which includes guidance on implementing user authentication state change notifications.

Obtain a user's claims principal data

The AuthenticationStateProvider service can provide the current user's ClaimsPrincipal data, as shown in the following example.

`ClaimsPrincipalData.razor`:

razor

```razor
@page "/claims-principal-data"
@using System.Security.Claims
@inject AuthenticationStateProvider AuthenticationStateProvider

<h1>ClaimsPrincipal Data</h1>

<button @onclick="GetClaimsPrincipalData">Get ClaimsPrincipal Data</button>

<p>@authMessage</p>

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

<p>@surname</p>

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

    private async Task GetClaimsPrincipalData()
    {
        var authState = await AuthenticationStateProvider
            .GetAuthenticationStateAsync();
        var user = authState.User;

        if (user.Identity is not null && user.Identity.IsAuthenticated)
        {
            authMessage = $"{user.Identity.Name} is authenticated.";
            claims = user.Claims;
            surname = user.FindFirst(c => c.Type ==
ClaimTypes.Surname)?.Value;
        }
        else
        {
            authMessage = "The user is NOT authenticated.";
        }
    }
}
```

In the preceding example:

- ClaimsPrincipal.Claims returns the user's claims (`claims`) for display in the UI.
- The line that obtains the user's surname (`surname`) calls ClaimsPrincipal.FindAll with a predicate to filter the user's claims.

If `user.Identity.IsAuthenticated` is `true` and because the user is a ClaimsPrincipal, claims can be enumerated and membership in roles evaluated.

For more information on dependency injection (DI) and services, see ASP.NET Core Blazor dependency injection and Dependency injection in ASP.NET Core. For information on how to implement a custom AuthenticationStateProvider, see ASP.NET Core Blazor authentication state.

Expose the authentication state as a cascading parameter

If authentication state data is required for procedural logic, such as when performing an action triggered by the user, obtain the authentication state data by defining a cascading parameter of type `Task<AuthenticationState>`, as the following example demonstrates.

`CascadeAuthState.razor`:

razor

```razor
@page "/cascade-auth-state"

<h1>Cascade Auth State</h1>

<p>@authMessage</p>

@code {
    private string authMessage = "The user is NOT authenticated.";

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

    protected override async Task OnInitializedAsync()
    {
        if (authenticationState is not null)
        {
            var authState = await authenticationState;
            var user = authState?.User;

            if (user?.Identity is not null && user.Identity.IsAuthenticated)
            {
                authMessage = $"{user.Identity.Name} is authenticated.";
            }
        }
    }
}
```

If `user.Identity.IsAuthenticated` is `true`, claims can be enumerated and membership in roles evaluated.

Set up the `Task<`AuthenticationState`>` cascading parameter using the AuthorizeRouteView and cascading authentication state services.

When you create a Blazor app from one of the Blazor project templates with authentication enabled, the app includes the AuthorizeRouteView and the call to AddCascadingAuthenticationState shown in the following example. A client-side Blazor app includes the required service registrations as well. Additional information is presented in the Customize unauthorized content with the Router component section.

razor

```razor
<Router ...>
    <Found ...>
        <AuthorizeRouteView RouteData="routeData"
            DefaultLayout="typeof(Layout.MainLayout)" />
        ...
    </Found>
</Router>
```

In the `Program` file, register cascading authentication state services:

C#

```csharp
builder.Services.AddCascadingAuthenticationState();
```

In a client-side Blazor app, add authorization services to the `Program` file:

C#

```csharp
builder.Services.AddAuthorizationCore();
```

In a server-side Blazor app, services for options and authorization are already present, so no further steps are required.

Authorization

After a user is authenticated, *authorization* rules are applied to control what the user can do.

Access is typically granted or denied based on whether:

- A user is authenticated (signed in).
- A user is in a *role*.
- A user has a *claim*.
- A *policy* is satisfied.

Each of these concepts is the same as in an ASP.NET Core MVC or Razor Pages app. For more information on ASP.NET Core security, see the articles under ASP.NET Core Security and Identity.

`AuthorizeView` component

The AuthorizeView component selectively displays UI content depending on whether the user is authorized. This approach is useful when you only need to *display* data for the user and don't need to use the user's identity in procedural logic.

The component exposes a `context` variable of type AuthenticationState (`@context` in Razor syntax), which you can use to access information about the signed-in user:

razor

```razor
<AuthorizeView>
    <p>Hello, @context.User.Identity?.Name!</p>
</AuthorizeView>
```

You can also supply different content for display if the user isn't authorized with a combination of the Authorized and NotAuthorized parameters:

razor

```razor
<AuthorizeView>
    <Authorized>
        <p>Hello, @context.User.Identity?.Name!</p>
        <p><button @onclick="HandleClick">Authorized Only Button</button>
</p>
    </Authorized>
    <NotAuthorized>
        <p>You're not authorized.</p>
    </NotAuthorized>
```

```
</AuthorizeView>

@code {
    private void HandleClick() { ... }
}
```

Although the AuthorizeView component controls the visibility of elements based on the user’s authorization status, it doesn't enforce security on the event handler itself. In the preceding example, the `HandleClick` method is only associated with a button visible to authorized users, but nothing prevents invoking this method from other places. To ensure method-level security, implement additional authorization logic within the handler itself or in the relevant API.

Razor components of Blazor Web Apps never display `<NotAuthorized>` content when authorization fails server-side during static server-side rendering (static SSR). The server-side ASP.NET Core pipeline processes authorization on the server. Use server-side techniques to handle unauthorized requests. For more information, see ASP.NET Core Blazor render modes.

> ⚠ **Warning**
>
> Client-side markup and methods associated with an AuthorizeView are only protected from view and execution in the ***rendered UI*** in client-side Blazor apps. In order to protect authorized content and secure methods in client-side Blazor, the content is usually supplied by a secure, authorized web API call to a server API and never stored in the app. For more information, see Call a web API from an ASP.NET Core Blazor app and ASP.NET Core Blazor WebAssembly additional security scenarios.

The content of Authorized and NotAuthorized can include arbitrary items, such as other interactive components.

Authorization conditions, such as roles or policies that control UI options or access, are covered in the Authorization section.

If authorization conditions aren't specified, AuthorizeView uses a default policy:

- Authenticated (signed-in) users are authorized.
- Unauthenticated (signed-out) users are unauthorized.

The AuthorizeView component can be used in the `NavMenu` component (`Shared/NavMenu.razor`) to display a NavLink component (NavLink), but note that this approach only removes the list item from the rendered output. It doesn't prevent the user from navigating to the component. Implement authorization separately in the destination component.

Role-based and policy-based authorization

The AuthorizeView component supports *role-based* or *policy-based* authorization.

For role-based authorization, use the Roles parameter. In the following example, the user must have a role claim for either the `Admin` or `Superuser` roles:

razor

```razor
<AuthorizeView Roles="Admin, Superuser">
    <p>You have an 'Admin' or 'Superuser' role claim.</p>
</AuthorizeView>
```

To require a user have both `Admin` and `Superuser` role claims, nest AuthorizeView components:

razor

```razor
<AuthorizeView Roles="Admin">
    <p>User: @context.User</p>
    <p>You have the 'Admin' role claim.</p>
    <AuthorizeView Roles="Superuser" Context="innerContext">
        <p>User: @innerContext.User</p>
        <p>You have both 'Admin' and 'Superuser' role claims.</p>
    </AuthorizeView>
</AuthorizeView>
```

The preceding code establishes a `Context` for the inner AuthorizeView component to prevent an AuthenticationState context collision. The AuthenticationState context is accessed in the outer AuthorizeView with the standard approach for accessing the context (`@context.User`). The context is accessed in the inner AuthorizeView with the named `innerContext` context (`@innerContext.User`).

For more information, including configuration guidance, see Role-based authorization in ASP.NET Core.

For policy-based authorization, use the Policy parameter with a single policy name:

razor

```razor
<AuthorizeView Policy="Over21">
    <p>You satisfy the 'Over21' policy.</p>
```

```
</AuthorizeView>
```

To handle the case where the user should satisfy one of several policies, create a policy that confirms that the user satisfies other policies.

To handle the case where the user must satisfy several policies simultaneously, take *either* of the following approaches:

- Create a policy for AuthorizeView that confirms that the user satisfies several other policies.

- Nest the policies in multiple AuthorizeView components:

  razor

  ```razor
  <AuthorizeView Policy="Over21">
      <AuthorizeView Policy="LivesInCalifornia">
          <p>You satisfy the 'Over21' and 'LivesInCalifornia' policies.
  </p>
      </AuthorizeView>
  </AuthorizeView>
  ```

Claims-based authorization is a special case of policy-based authorization. For example, you can define a policy that requires users to have a certain claim. For more information, see Policy-based authorization in ASP.NET Core.

If neither Roles nor Policy is specified, AuthorizeView uses the default policy:

- Authenticated (signed-in) users are authorized.
- Unauthenticated (signed-out) users are unauthorized.

Because .NET string comparisons are case-sensitive, matching role and policy 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 and policy 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 and policy names is unusual but permitted by the framework. For example, `billing administrator` is an unusual role or policy name format in .NET apps, but it's a valid role or policy name.

Content displayed during asynchronous authentication

Blazor allows for authentication state to be determined *asynchronously*. The primary scenario for this approach is in client-side Blazor apps that make a request to an external endpoint for authentication.

While authentication is in progress, AuthorizeView displays no content. To display content while authentication occurs, assign content to the Authorizing parameter:

razor

```razor
<AuthorizeView>
    <Authorized>
        <p>Hello, @context.User.Identity?.Name!</p>
    </Authorized>
    <Authorizing>
        <p>You can only see this content while authentication is in
progress.</p>
    </Authorizing>
</AuthorizeView>
```

This approach isn't normally applicable to server-side Blazor apps. Server-side Blazor apps know the authentication state as soon as the state is established. Authorizing content can be provided in an app's AuthorizeView component, but the content is never displayed.

`[Authorize]` attribute

The [Authorize] attribute is available in Razor components:

razor

```razor
@page "/"
@attribute [Authorize]

You can only see this if you're signed in.
```

ⓘ **Important**

Only use **[Authorize]** on `@page` components reached via the Blazor router. Authorization is only performed as an aspect of routing and *not* for child components rendered within a page. To authorize the display of specific parts within a page, use **AuthorizeView** instead.

The [Authorize] attribute also supports role-based or policy-based authorization. For role-based authorization, use the Roles parameter:

razor

```razor
@page "/"
@attribute [Authorize(Roles = "Admin, Superuser")]

<p>You can only see this if you're in the 'Admin' or 'Superuser' role.</p>
```

For policy-based authorization, use the Policy parameter:

razor

```razor
@page "/"
@attribute [Authorize(Policy = "Over21")]

<p>You can only see this if you satisfy the 'Over21' policy.</p>
```

If neither Roles nor Policy is specified, [Authorize] uses the default policy:

- Authenticated (signed-in) users are authorized.
- Unauthenticated (signed-out) users are unauthorized.

When the user isn't authorized and if the app doesn't customize unauthorized content with the Router component, the framework automatically displays the following fallback message:

HTML

```html
Not authorized.
```

Resource authorization

To authorize users for resources, pass the request's route data to the Resource parameter of AuthorizeRouteView.

In the Router.Found content for a requested route:

razor

```razor
<AuthorizeRouteView Resource="routeData" RouteData="routeData"
    DefaultLayout="typeof(MainLayout)" />
```

For more information on how authorization state data is passed and used in procedural logic, see the Expose the authentication state as a cascading parameter section.

When the AuthorizeRouteView receives the route data for the resource, authorization policies have access to RouteData.PageType and RouteData.RouteValues that permit custom logic to make authorization decisions.

In the following example, an `EditUser` policy is created in AuthorizationOptions for the app's authorization service configuration (AddAuthorizationCore) with the following logic:

- Determine if a route value exists with a key of `id`. If the key exists, the route value is stored in `value`.
- In a variable named `id`, store `value` as a string or set an empty string value (`string.Empty`).
- If `id` isn't an empty string, assert that the policy is satisfied (return `true`) if the string's value starts with `EMP`. Otherwise, assert that the policy fails (return `false`).

In the `Program` file:

- Add namespaces for Microsoft.AspNetCore.Components and System.Linq:

  C#

  ```
  using Microsoft.AspNetCore.Components;
  using System.Linq;
  ```

- Add the policy:

  C#

  ```
  options.AddPolicy("EditUser", policy =>
      policy.RequireAssertion(context =>
      {
          if (context.Resource is RouteData rd)
          {
              var routeValue = rd.RouteValues.TryGetValue("id", out var
  value);
              var id = Convert.ToString(value,
                  System.Globalization.CultureInfo.InvariantCulture) ??
  string.Empty;

              if (!string.IsNullOrEmpty(id))
              {
                  return id.StartsWith("EMP",
  StringComparison.InvariantCulture);
              }
          }

          return false;
  ```

```
        })
    );
```

The preceding example is an oversimplified authorization policy, merely used to demonstrate the concept with a working example. For more information on creating and configuring authorization policies, see Policy-based authorization in ASP.NET Core.

In the following `EditUser` component, the resource at `/users/{id}/edit` has a route parameter for the user's identifier (`{id}`). The component uses the preceding `EditUser` authorization policy to determine if the route value for `id` starts with `EMP`. If `id` starts with `EMP`, the policy succeeds and access to the component is authorized. If `id` starts with a value other than `EMP` or if `id` is an empty string, the policy fails, and the component doesn't load.

`EditUser.razor`:

```razor
@page "/users/{id}/edit"
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize(Policy = "EditUser")]

<h1>Edit User</h1>

<p>The "EditUser" policy is satisfied! <code>Id</code> starts with 'EMP'.
</p>

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

Customize unauthorized content with the `Router` component

The Router component, in conjunction with the AuthorizeRouteView component, allows the app to specify custom content if:

- The user fails an [Authorize] condition applied to the component. The markup of the <NotAuthorized> element is displayed. The [Authorize] attribute is covered in the [Authorize] attribute section.
- Asynchronous authorization is in progress, which usually means that the process of authenticating the user is in progress. The markup of the <Authorizing> element is displayed.

ⓘ **Important**

Blazor router features that display `<NotAuthorized>` and `<NotFound>` content aren't operational during static server-side rendering (static SSR) because request processing is entirely handled by ASP.NET Core middleware pipeline request processing and Razor components aren't rendered at all for unauthorized or bad requests. Use server-side techniques to handle unauthorized and bad requests during static SSR. For more information, see ASP.NET Core Blazor render modes.

razor

```razor
<Router ...>
    <Found ...>
        <AuthorizeRouteView ...>
            <NotAuthorized>
                ...
            </NotAuthorized>
            <Authorizing>
                ...
            </Authorizing>
        </AuthorizeRouteView>
    </Found>
</Router>
```

The content of Authorized and NotAuthorized can include arbitrary items, such as other interactive components.

Note

The preceding requires cascading authentication state services registration in the app's `Program` file:

C#

```csharp
builder.Services.AddCascadingAuthenticationState();
```

If NotAuthorized content isn't specified, the AuthorizeRouteView uses the following fallback message:

HTML

```html
Not authorized.
```

An app created from the Blazor WebAssembly project template with authentication enabled includes a `RedirectToLogin` component, which is positioned in the `<NotAuthorized>` content of the Router component. When a user isn't authenticated (`context.User.Identity?.IsAuthenticated != true`), the `RedirectToLogin` component redirects the browser to the `authentication/login` endpoint for authentication. The user is returned to the requested URL after authenticating with the identity provider.

Procedural logic

If the app is required to check authorization rules as part of procedural logic, use a cascaded parameter of type `Task<`AuthenticationState`>` to obtain the user's ClaimsPrincipal. `Task<`AuthenticationState`>` can be combined with other services, such as `IAuthorizationService`, to evaluate policies.

In the following example:

- The `user.Identity.IsAuthenticated` executes code for authenticated (signed-in) users.
- The `user.IsInRole("admin")` executes code for users in the 'Admin' role.
- The `(await AuthorizationService.AuthorizeAsync(user, "content-editor")).Succeeded` executes code for users satisfying the 'content-editor' policy.

A server-side Blazor app includes the appropriate namespaces when created from the project template. In a client-side Blazor app, confirm the presence of the Microsoft.AspNetCore.Authorization and Microsoft.AspNetCore.Components.Authorization namespaces either in the component or in the app's `_Imports.razor` file:

razor

```razor
@using Microsoft.AspNetCore.Authorization
@using Microsoft.AspNetCore.Components.Authorization
```

`ProceduralLogic.razor`:

razor

```razor
@page "/procedural-logic"
@inject IAuthorizationService AuthorizationService

<h1>Procedural Logic Example</h1>

<button @onclick="@DoSomething">Do something important</button>
```

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

    private async Task DoSomething()
    {
        if (authenticationState is not null)
        {
            var authState = await authenticationState;
            var user = authState?.User;

            if (user is not null)
            {
                if (user.Identity is not null &&
user.Identity.IsAuthenticated)
                {
                    // ...
                }

                if (user.IsInRole("Admin"))
                {
                    // ...
                }

                if ((await AuthorizationService.AuthorizeAsync(user,
"content-editor"))
                    .Succeeded)
                {
                    // ...
                }
            }
        }
    }
}
```

Troubleshoot errors

Common errors:

- **Authorization requires a cascading parameter of type `Task<AuthenticationState>`. Consider using `CascadingAuthenticationState` to supply this.**
- **`null` value is received for `authenticationStateTask`**

It's likely that the project wasn't created using a server-side Blazor template with authentication enabled.

In .NET 7 or earlier, wrap a `<CascadingAuthenticationState>` around some part of the UI tree, for example around the Blazor router:

```razor
<CascadingAuthenticationState>
    <Router ...>
        ...
    </Router>
</CascadingAuthenticationState>
```

In .NET 8 or later, don't use the CascadingAuthenticationState component:

```diff
- <CascadingAuthenticationState>
      <Router ...>
          ...
      </Router>
- </CascadingAuthenticationState>
```

Instead, add cascading authentication state services to the service collection in the `Program` file:

```C#
builder.Services.AddCascadingAuthenticationState();
```

The CascadingAuthenticationState component (.NET 7 or earlier) or services provided by AddCascadingAuthenticationState (.NET 8 or later) supplies the `Task<`AuthenticationState`>` cascading parameter, which in turn it receives from the underlying AuthenticationStateProvider dependency injection service.

Personally Identifiable Information (PII)

Microsoft uses the GDPR definition for 'personal data' (GDPR 4.1) when documentation discusses Personally Identifiable Information (PII).

PII refers any information relating to an identified or identifiable natural person. An identifiable natural person is one who can be identified, directly or indirectly, with any of the following:

- Name
- Identification number

- Location coordinates
- Online identifier
- Other specific factors
  - Physical
  - Physiological
  - Genetic
  - Mental (psychological)
  - Economic
  - Cultural
  - Social identity

Additional resources

- Server-side and Blazor Web App resources
  - Quickstart: Add sign-in with Microsoft to an ASP.NET Core web app
  - Quickstart: Protect an ASP.NET Core web API with Microsoft identity platform
  - Configure ASP.NET Core to work with proxy servers and load balancers: Includes guidance on:
    - Using Forwarded Headers Middleware to preserve HTTPS scheme information across proxy servers and internal networks.
    - Additional scenarios and use cases, including manual scheme configuration, request path changes for correct request routing, and forwarding the request scheme for Linux and non-IIS reverse proxies.
- Microsoft identity platform documentation
  - Overview
  - OAuth 2.0 and OpenID Connect protocols on the Microsoft identity platform
  - Microsoft identity platform and OAuth 2.0 authorization code flow
  - Microsoft identity platform ID tokens
  - Microsoft identity platform access tokens
- ASP.NET Core security topics
- Configure Windows Authentication in ASP.NET Core
- Build a custom version of the Authentication.MSAL JavaScript library
- Awesome Blazor: Authentication community sample links
- ASP.NET Core Blazor Hybrid authentication and authorization

ASP.NET Core Blazor authentication state

Article • 09/23/2024

ⓘ **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 create a custom authentication state provider and receive user authentication state change notifications in code.

The general approaches taken for server-side and client-side Blazor apps are similar but differ in their exact implementations, so this article pivots between server-side Blazor apps and client-side Blazor apps. Use the pivot selector at the top of the article to change the article's pivot to match the type of Blazor project that you're working with:

- Server-side Blazor apps (**Server** pivot): Blazor Server for .NET 7 or earlier and the server project of a Blazor Web App for .NET 8 or later.
- Client-side Blazor apps (**Blazor WebAssembly** pivot): Blazor WebAssembly for all versions of .NET or the `.Client` project of a Blazor Web App for .NET 8 or later.

Abstract `AuthenticationStateProvider` class

The Blazor framework includes an abstract AuthenticationStateProvider class to provide information about the authentication state of the current user with the following members:

- GetAuthenticationStateAsync: Asynchronously gets the authentication state of the current user.
- AuthenticationStateChanged: An event that provides notification when the authentication state has changed. For example, this event may be raised if a user signs in or out of the app.
- NotifyAuthenticationStateChanged: Raises an authentication state changed event.

Implement a custom `AuthenticationStateProvider`

The app must reference the Microsoft.AspNetCore.Components.Authorization NuGet package, which provides authentication and authorization support for Blazor apps.

> **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.

Configure the following authentication, authorization, and cascading authentication state services in the `Program` file.

When you create a Blazor app from one of the Blazor project templates with authentication enabled, the app is preconfigured with the following service registrations, which includes exposing the authentication state as a cascading parameter. For more information, see ASP.NET Core Blazor authentication and authorization with additional information presented in the article's Customize unauthorized content with the Router component section.

C#

```csharp
using Microsoft.AspNetCore.Components.Authorization;

...

builder.Services.AddAuthorization();
builder.Services.AddCascadingAuthenticationState();
```

Subclass AuthenticationStateProvider and override GetAuthenticationStateAsync to create the user's authentication state. In the following example, all users are authenticated with the username `mrfibuli`.

`CustomAuthStateProvider.cs`:

C#

```csharp
using System.Security.Claims;
using Microsoft.AspNetCore.Components.Authorization;

public class CustomAuthStateProvider : AuthenticationStateProvider
{
    public override Task<AuthenticationState> GetAuthenticationStateAsync()
    {
        var identity = new ClaimsIdentity(
        [
            new Claim(ClaimTypes.Name, "mrfibuli"),
        ], "Custom Authentication");

        var user = new ClaimsPrincipal(identity);

        return Task.FromResult(new AuthenticationState(user));
    }
}
```

ⓘ **Note**

The preceding code that creates a new ClaimsIdentity uses simplified collection initialization introduced with C# 12 (.NET 8). For more information, see Collection expressions - C# language reference.

The `CustomAuthStateProvider` service is registered in the `Program` file. Register the service *scoped* with AddScoped:

C#

```
builder.Services.AddScoped<AuthenticationStateProvider,
CustomAuthStateProvider>();
```

If it isn't present, add an @using statement to the `_Imports.razor` file to make the Microsoft.AspNetCore.Components.Authorization namespace available across components:

razor

```
@using Microsoft.AspNetCore.Components.Authorization;
```

Confirm or change the route view component to an AuthorizeRouteView in the Router component definition. The location of the `Router` component differs depending on the type of app. Use search to locate the component if you're unaware of its location in the project.

razor

```
<Router ...>
    <Found ...>
        <AuthorizeRouteView RouteData="routeData"
```

```
            DefaultLayout="typeof(Layout.MainLayout)" />
        ...
    </Found>
</Router>
```

ⓘ Note

When you create a Blazor app from one of the Blazor project templates with authentication enabled, the app includes the AuthorizeRouteView component. For more information, see ASP.NET Core Blazor authentication and authorization with additional information presented in the article's Customize unauthorized content with the Router component section.

The following example AuthorizeView component demonstrates the authenticated user's name:

razor

```razor
<AuthorizeView>
    <Authorized>
        <p>Hello, @context.User.Identity?.Name!</p>
    </Authorized>
    <NotAuthorized>
        <p>You're not authorized.</p>
    </NotAuthorized>
</AuthorizeView>
```

For guidance on the use of AuthorizeView, see ASP.NET Core Blazor authentication and authorization.

Authentication state change notifications

A custom AuthenticationStateProvider can invoke NotifyAuthenticationStateChanged on the AuthenticationStateProvider base class to notify consumers of the authentication state change to rerender.

The following example is based on implementing a custom AuthenticationStateProvider by following the guidance in the Implement a custom AuthenticationStateProvider section earlier in this article. If you already followed the guidance in that section, the following `CustomAuthStateProvider` replaces the one shown in the section.

The following `CustomAuthStateProvider` implementation exposes a custom method, `AuthenticateUser`, to sign in a user and notify consumers of the authentication state change.

`CustomAuthStateProvider.cs`:

```C#
using System.Security.Claims;
using Microsoft.AspNetCore.Components.Authorization;

public class CustomAuthStateProvider : AuthenticationStateProvider
{
    public override Task<AuthenticationState> GetAuthenticationStateAsync()
    {
        var identity = new ClaimsIdentity();
        var user = new ClaimsPrincipal(identity);

        return Task.FromResult(new AuthenticationState(user));
    }

    public void AuthenticateUser(string userIdentifier)
    {
        var identity = new ClaimsIdentity(
        [
            new Claim(ClaimTypes.Name, userIdentifier),
        ], "Custom Authentication");

        var user = new ClaimsPrincipal(identity);

        NotifyAuthenticationStateChanged(
            Task.FromResult(new AuthenticationState(user)));
    }
}
```

Note

The preceding code that creates a new ClaimsIdentity uses simplified collection initialization introduced with C# 12 (.NET 8). For more information, see Collection expressions - C# language reference.

In a component:

- Inject AuthenticationStateProvider.
- Add a field to hold the user's identifier.
- Add a button and a method to cast the AuthenticationStateProvider to `CustomAuthStateProvider` and call `AuthenticateUser` with the user's identifier.

razor

```
@inject AuthenticationStateProvider AuthenticationStateProvider

<input @bind="userIdentifier" />
<button @onclick="SignIn">Sign in</button>

<AuthorizeView>
    <Authorized>
        <p>Hello, @context.User.Identity?.Name!</p>
    </Authorized>
    <NotAuthorized>
        <p>You're not authorized.</p>
    </NotAuthorized>
</AuthorizeView>

@code {
    public string userIdentifier = string.Empty;

    private void SignIn()
    {
        ((CustomAuthStateProvider)AuthenticationStateProvider)
            .AuthenticateUser(userIdentifier);
    }
}
```

The preceding approach can be enhanced to trigger notifications of authentication state changes via a custom service. The following `CustomAuthenticationService` class maintains the current user's claims principal in a backing field (`currentUser`) with an event (`UserChanged`) that the authentication state provider can subscribe to, where the event invokes NotifyAuthenticationStateChanged. With the additional configuration later in this section, the `CustomAuthenticationService` can be injected into a component with logic that sets the `CurrentUser` to trigger the `UserChanged` event.

`CustomAuthenticationService.cs`:

C#

```csharp
using System.Security.Claims;

public class CustomAuthenticationService
{
    public event Action<ClaimsPrincipal>? UserChanged;
    private ClaimsPrincipal? currentUser;

    public ClaimsPrincipal CurrentUser
    {
        get { return currentUser ?? new(); }
        set
        {
            currentUser = value;
            if (UserChanged is not null)
            {
                UserChanged(currentUser);
            }
        }
    }
}
```

In the `Program` file, register the `CustomAuthenticationService` in the dependency injection container:

```C#
builder.Services.AddScoped<CustomAuthenticationService>();
```

The following `CustomAuthStateProvider` subscribes to the `CustomAuthenticationService.UserChanged` event. The `GetAuthenticationStateAsync` method returns the user's authentication state. Initially, the authentication state is based on the value of the `CustomAuthenticationService.CurrentUser`. When the user changes, a new authentication state is created for the new user (`new AuthenticationState(newUser)`) for calls to `GetAuthenticationStateAsync`:

```C#
using Microsoft.AspNetCore.Components.Authorization;

public class CustomAuthStateProvider : AuthenticationStateProvider
{
    private AuthenticationState authenticationState;

    public CustomAuthStateProvider(CustomAuthenticationService service)
    {
        authenticationState = new AuthenticationState(service.CurrentUser);

        service.UserChanged += (newUser) =>
        {
            authenticationState = new AuthenticationState(newUser);

NotifyAuthenticationStateChanged(Task.FromResult(authenticationState));
        };
    }

    public override Task<AuthenticationState> GetAuthenticationStateAsync()
=>
        Task.FromResult(authenticationState);
}
```

The following component's `SignIn` method creates a claims principal for the user's identifier to set on `CustomAuthenticationService.CurrentUser`:

razor

```
@using System.Security.Claims
@inject CustomAuthenticationService AuthService

<input @bind="userIdentifier" />
<button @onclick="SignIn">Sign in</button>

<AuthorizeView>
    <Authorized>
        <p>Hello, @context.User.Identity?.Name!</p>
    </Authorized>
    <NotAuthorized>
        <p>You're not authorized.</p>
    </NotAuthorized>
</AuthorizeView>

@code {
    public string userIdentifier = string.Empty;

    private void SignIn()
    {
        var currentUser = AuthService.CurrentUser;

        var identity = new ClaimsIdentity(
            [
                new Claim(ClaimTypes.Name, userIdentifier),
            ],
            "Custom Authentication");

        var newUser = new ClaimsPrincipal(identity);

        AuthService.CurrentUser = newUser;
    }
}
```

ⓘ Note

The preceding code that creates a new ClaimsIdentity uses simplified collection initialization introduced with C# 12 (.NET 8). For more information, see Collection expressions - C# language reference.

Additional resources

- Server-side unauthorized content display while prerendering with a custom AuthenticationStateProvider
- How to access an AuthenticationStateProvider from a DelegatingHandler set up using an IHttpClientFactory
- Secure an ASP.NET Core Blazor Web App with OpenID Connect (OIDC)
- Secure ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity

Secure ASP.NET Core Blazor WebAssembly

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.

Blazor WebAssembly apps are secured in the same manner as single-page applications (SPAs). There are several approaches for authenticating users to SPAs, but the most common and comprehensive approach is to use an implementation based on the OAuth 2.0 protocol, such as OpenID Connect (OIDC).

The Blazor WebAssembly security documentation primarily focuses on how to accomplish user authentication and authorization tasks. For OAuth 2.0/OIDC general concept coverage, see the resources in the main overview article's *Additional resources* section.

Client-side/SPA security of sensitive data and credentials

A Blazor WebAssembly app's .NET/C# codebase is served to clients, and the app's code can't be protected from inspection and tampering by users. Never place credentials or secrets into a Blazor WebAssembly app, such as app secrets, connection strings, passwords, private .NET/C# code, or other sensitive data.

To protect .NET/C# code and use ASP.NET Core Data Protection features to secure data, use a server-side ASP.NET Core web API. Have the client-side Blazor WebAssembly app call the server-side web API for secure app features and data processing. For more information, see Call a web API from an ASP.NET Core Blazor app and the articles in this node.

For local development testing, the Secret Manager tool is recommended for securing sensitive data.

Authentication library

Blazor WebAssembly supports authenticating and authorizing apps using OIDC via the Microsoft.AspNetCore.Components.WebAssembly.Authentication library using the Microsoft identity platform. The library provides a set of primitives for seamlessly authenticating against ASP.NET Core backends. The library can authenticate against any third-party Identity Provider (IP) that supports OIDC, which are called OpenID Providers (OP).

The authentication support in the Blazor WebAssembly Library (`Authentication.js`) is built on top of the Microsoft Authentication Library (MSAL, msal.js), which is used to handle the underlying authentication protocol details. The Blazor WebAssembly Library only supports the Proof Key for Code Exchange (PKCE) authorization code flow. Implicit grant isn't supported.

Other options for authenticating SPAs exist, such as the use of SameSite cookies. However, the engineering design of Blazor WebAssembly uses OAuth and OIDC as the best option for authentication in Blazor WebAssembly apps. Token-based authentication based on JSON Web Tokens (JWTs) was chosen over cookie-based authentication for functional and security reasons:

- Using a token-based protocol offers fewer vulnerabilities, as the tokens aren't sent in all requests.
- The tokens are explicitly sent to the server, so server endpoints don't require protection against Cross-Site Request Forgery (CSRF). This allows you to host Blazor WebAssembly apps alongside MVC or Razor pages apps.
- Tokens have narrower permissions than cookies. For example, tokens can't be used to manage the user account or change a user's password unless such functionality is explicitly implemented.
- Tokens have a short lifetime, one hour, which limits the attack window. Tokens can also be revoked at any time.
- Self-contained JWTs offer guarantees to the client and server about the authentication process. For example, a client has the means to detect and validate that the tokens it receives are legitimate and were emitted as part of a given authentication process. If a third party attempts to switch a token in the middle of the authentication process, the client can detect the switched token and avoid using it.
- Tokens with OAuth and OIDC don't rely on the user agent behaving correctly to ensure that the app is secure.
- Token-based protocols, such as OAuth and OIDC, allow for authenticating and authorizing users in standalone Blazor Webassembly apps with the same set of

security characteristics.

> ⓘ **Important**
>
> For versions of ASP.NET Core that adopt Duende Identity Server in Blazor project templates, Duende Software might require you to pay a license fee for production use of Duende Identity Server. For more information, see Migrate from ASP.NET Core 5.0 to 6.0.

Authentication process with OIDC

The Microsoft.AspNetCore.Components.WebAssembly.Authentication library offers several primitives to implement authentication and authorization using OIDC. In broad terms, authentication works as follows:

- When an anonymous user selects the login button or requests a Razor component or page with the [Authorize] attribute applied, the user is redirected to the app's login page (`/authentication/login`).
- In the login page, the authentication library prepares for a redirect to the authorization endpoint. The authorization endpoint is outside of the Blazor WebAssembly app and can be hosted at a separate origin. The endpoint is responsible for determining whether the user is authenticated and for issuing one or more tokens in response. The authentication library provides a login callback to receive the authentication response.
  - If the user isn't authenticated, the user is redirected to the underlying authentication system, which is usually ASP.NET Core Identity.
  - If the user was already authenticated, the authorization endpoint generates the appropriate tokens and redirects the browser back to the login callback endpoint (`/authentication/login-callback`).
- When the Blazor WebAssembly app loads the login callback endpoint (`/authentication/login-callback`), the authentication response is processed.
  - If the authentication process completes successfully, the user is authenticated and optionally sent back to the original protected URL that the user requested.
  - If the authentication process fails for any reason, the user is sent to the login failed page (`/authentication/login-failed`), where an error is displayed.

`Authentication` component

The `Authentication` component (`Authentication.razor`) handles remote authentication operations and permits the app to:

- Configure app routes for authentication states.
- Set UI content for authentication states.
- Manage authentication state.

Authentication actions, such as registering or signing in a user, are passed to the Blazor framework's RemoteAuthenticatorViewCore<TAuthenticationState> component, which persists and controls state across authentication operations.

For more information and examples, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Authorization

In Blazor WebAssembly apps, authorization checks can be bypassed because all client-side code can be modified by users. The same is true for all client-side app technologies, including JavaScript SPA frameworks or native apps for any operating system.

Always perform authorization checks on the server within any API endpoints accessed by your client-side app.

Customize authentication

Blazor WebAssembly provides methods to add and retrieve additional parameters for the underlying Authentication library to conduct remote authentication operations with external identity providers.

To pass additional parameters, NavigationManager supports passing and retrieving history entry state when performing external location changes. For more information, see the following resources:

- Blazor *Fundamentals* > *Routing and navigation* article
  - Navigation history state
  - Navigation options
- MDN documentation: History API

The state stored by the History API provides the following benefits for remote authentication:

- The state passed to the secured app endpoint is tied to the navigation performed to authenticate the user at the `authentication/login` endpoint.

- Extra work encoding and decoding data is avoided.
- Vulnerabilities are reduced. Unlike using the query string to store navigation state, a top-level navigation or influence from a different origin can't set the state stored by the History API.
- The history entry is replaced upon successful authentication, so the state attached to the history entry is removed and doesn't require clean up.

InteractiveRequestOptions represents the request to the identity provider for logging in or provisioning an access token.

NavigationManagerExtensions provides the NavigateToLogin method for a login operation and NavigateToLogout for a logout operation. The methods call NavigationManager.NavigateTo, setting the history entry state with a passed InteractiveRequestOptions or a new InteractiveRequestOptions instance created by the method for:

- A user signing in (InteractionType.SignIn) with the current URI for the return URL.
- A user signing out (InteractionType.SignOut) with the return URL.

The following authentication scenarios are covered in the ASP.NET Core Blazor WebAssembly additional security scenarios article:

- Customize the login process
- Logout with a custom return URL
- Customize options before obtaining a token interactively
- Customize options when using an IAccessTokenProvider
- Obtain the login path from authentication options

Require authorization for the entire app

Apply the [Authorize] attribute (API documentation) to each Razor component of the app using ***one*** of the following approaches:

- In the app's Imports file, add an @using directive for the Microsoft.AspNetCore.Authorization namespace with an @attribute directive for the [Authorize] attribute.

  `_Imports.razor`:

  razor

  ```
  @using Microsoft.AspNetCore.Authorization
  @attribute [Authorize]
  ```

Allow anonymous access to the `Authentication` component to permit redirection to the identity provider. Add the following Razor code to the `Authentication` component under its @page directive.

`Authentication.razor`:

```razor
@using Microsoft.AspNetCore.Components.WebAssembly.Authentication
@attribute [AllowAnonymous]
```

- Add the attribute to each Razor component under the @page directive:

```razor
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]
```

ⓘ **Note**

Setting an AuthorizationOptions.FallbackPolicy to a policy with RequireAuthenticatedUser is **not** supported.

Use one identity provider app registration per app

Some of the articles under this *Overview* pertain to Blazor hosting scenarios that involve two or more apps. A standalone Blazor WebAssembly app uses web API with authenticated users to access server resources and data provided by a server app.

When this scenario is implemented in documentation examples, ***two*** identity provider registrations are used, one for the client app and one for the server app. Using separate registrations, for example in Microsoft Entra ID, isn't strictly required. However, using two registrations is a security best practice because it isolates the registrations by app. Using separate registrations also allows independent configuration of the client and server registrations.

Refresh tokens

Although refresh tokens can't be secured in Blazor WebAssembly apps, they can be used if you implement them with appropriate security strategies.

For standalone Blazor WebAssembly apps in ASP.NET Core in .NET 6 or later, we recommend using:

- The OAuth 2.0 Authorization Code flow (Code) with Proof Key for Code Exchange (PKCE).
- A refresh token that has a short expiration.
- A rotated refresh token.
- A refresh token with an expiration after which a new interactive authorization flow is required to refresh the user's credentials.

For more information, see the following resources:

- Microsoft identity platform refresh tokens: Refresh token lifetime
- OAuth 2.0 for Browser-Based Apps (IETF specification)

Establish claims for users

Apps often require claims for users based on a web API call to a server. For example, claims are frequently used to establish authorization in an app. In these scenarios, the app requests an access token to access the service and uses the token to obtain user data for creating claims.

For examples, see the following resources:

- Additional scenarios: Customize the user
- ASP.NET Core Blazor WebAssembly with Microsoft Entra ID groups and roles

Prerendering support

Prerendering isn't supported for authentication endpoints (`/authentication/` path segment).

For more information, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Azure App Service on Linux with Identity Server

Specify the issuer explicitly when deploying to Azure App Service on Linux with Identity Server.

For more information, see Use Identity to secure a Web API backend for SPAs.

Windows Authentication

We don't recommend using Windows Authentication with Blazor Webassembly or with any other SPA framework. We recommend using token-based protocols instead of Windows Authentication, such as OIDC with Active Directory Federation Services (ADFS).

If Windows Authentication is used with Blazor Webassembly or with any other SPA framework, additional measures are required to protect the app from cross-site request forgery (CSRF) tokens. The same concerns that apply to cookies apply to Windows Authentication with the addition that Windows Authentication doesn't offer a mechanism to prevent sharing of the authentication context across origins. Apps using Windows Authentication without additional protection from CSRF should at least be restricted to an organization's intranet and not be used on the open Internet.

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

Secure a SignalR hub

To secure a SignalR hub in the server API project, apply the [Authorize] attribute to the hub class or to methods of the hub class.

In a client project with prerendering, such as hosted Blazor WebAssembly (ASP.NET Core in .NET 7 or earlier) or a Blazor Web App (ASP.NET Core in .NET 8 or later), see the guidance in ASP.NET Core Blazor SignalR guidance.

In a client project component without prerendering, such as standalone Blazor WebAssembly, or non-browser apps, supply an access token to the hub connection, as the following example demonstrates. For more information, see Authentication and authorization in ASP.NET Core SignalR.

razor

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

...

var tokenResult = await TokenProvider.RequestAccessToken();

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

```
{
    hubConnection = new HubConnectionBuilder()
        .WithUrl(Navigation.ToAbsoluteUri("/chathub"),
            options => { options.AccessTokenProvider = () =>
Task.FromResult(token?.Value); })
        .Build();

  ...
}
```

Logging

This section applies to Blazor WebAssembly apps in ASP.NET Core in .NET 7 or later.

To enable debug or trace logging, see the *Authentication logging (Blazor WebAssembly)* section in a 7.0 or later version of the ASP.NET Core Blazor logging article.

The WebAssembly sandbox

The WebAssembly *sandbox* restricts access to the environment of the system executing WebAssembly code, including access to I/O subsystems, system storage and resources, and the operating system. The isolation between WebAssembly code and the system that executes the code makes WebAssembly a safe coding framework for systems. However, WebAssembly is vulnerable to side-channel attacks at the hardware level. Normal precautions and due diligence in sourcing hardware and placing limitations on accessing hardware apply.

WebAssembly isn't owned or maintained by Microsoft.

For more information, see the following W3C resources:

- WebAssembly: Security
- WebAssembly Specification: Security Considerations
- W3C WebAssembly Community Group: Feedback and issues : The W3C WebAssembly Community Group link is only provided for reference, making it clear that WebAssembly security vulnerabilities and bugs are patched on an ongoing basis, often reported and addressed by browser. ***Don't send feedback or bug reports on Blazor to the W3C WebAssembly Community Group.*** Blazor feedback should be reported to the Microsoft ASP.NET Core product unit . If the Microsoft product unit determines that an underlying problem with WebAssembly exists, they take the appropriate steps to report the problem to the W3C WebAssembly Community Group.

Implementation guidance

Articles under this *Overview* provide information on authenticating users in Blazor WebAssembly apps against specific providers.

Standalone Blazor WebAssembly apps:

- General guidance for OIDC providers and the WebAssembly Authentication Library
- Microsoft Accounts
- Microsoft Entra ID (ME-ID)
- Azure Active Directory (AAD) B2C

Further configuration guidance is found in the following articles:

- ASP.NET Core Blazor WebAssembly additional security scenarios
- Use Graph API with ASP.NET Core Blazor WebAssembly

Use the Authorization Code flow with PKCE

Microsoft identity platform's Microsoft Authentication Library for JavaScript (MSAL) v2.0 or later provides support for the Authorization Code flow with Proof Key for Code Exchange (PKCE) and Cross-Origin Resource Sharing (CORS) for single-page applications, including Blazor.

Microsoft doesn't recommend using Implicit grant.

For more information, see the following resources:

- Authentication flow support in MSAL: Implicit grant
- Microsoft identity platform and implicit grant flow: Prefer the auth code flow
- Microsoft identity platform and OAuth 2.0 authorization code flow

Additional resources

- Microsoft identity platform documentation
  - General documentation
  - Access tokens
- Configure ASP.NET Core to work with proxy servers and load balancers
  - Using Forwarded Headers Middleware to preserve HTTPS scheme information across proxy servers and internal networks.
  - Additional scenarios and use cases, including manual scheme configuration, request path changes for correct request routing, and forwarding the request

scheme for Linux and non-IIS reverse proxies.

- Prerendering with authentication
- WebAssembly: Security
- WebAssembly Specification: Security Considerations

Secure ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity

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.

Standalone Blazor WebAssembly apps can be secured with ASP.NET Core Identity by following the guidance in this article.

Endpoints for registering, logging in, and logging out

Instead of using the default UI provided by ASP.NET Core Identity for SPA and Blazor apps, which is based on Razor Pages, call MapIdentityApi in a backend API to add JSON API endpoints for registering and logging in users with ASP.NET Core Identity. Identity API endpoints also support advanced features, such as two-factor authentication and email verification.

On the client, call the `/register` endpoint to register a user with their email address and password:

```C#
var result = await _httpClient.PostAsJsonAsync(
    "register", new
    {
        email,
        password
    });
```

On the client, log in a user with cookie authentication using the `/login` endpoint with `useCookies` query string set to `true`:

C#

```
var result = await _httpClient.PostAsJsonAsync(
    "login?useCookies=true", new
    {
        email,
        password
    });
```

The backend server API establishes cookie authentication with a call to AddIdentityCookies on the authentication builder:

C#

```
builder.Services
    .AddAuthentication(IdentityConstants.ApplicationScheme)
    .AddIdentityCookies();
```

Token authentication

For native and mobile scenarios where some clients don't support cookies, the login API provides a parameter to request tokens.

⚠ **Warning**

We recommend using cookies for browser-based apps instead of tokens because the browser handles cookies without exposing them to JavaScript. If you opt to use token-based security in web apps, you're responsible for ensuring the tokens are kept secure.

A custom token (one that is proprietary to the ASP.NET Core Identity platform) is issued that can be used to authenticate subsequent requests. The token should be passed in the `Authorization` header as a bearer token. A refresh token is also provided. This token allows the app to request a new token when the old one expires without forcing the user to log in again.

The tokens are not standard JSON Web Tokens (JWTs). The use of custom tokens is intentional, as the built-in Identity API is meant primarily for simple scenarios. The token option is not intended to be a fully-featured identity service provider or token server, but instead an alternative to the cookie option for clients that can't use cookies.

The following guidance begins the process of implementing token-based authentication with the login API. Custom code is required to complete the implementation. For more information, see Use Identity to secure a Web API backend for SPAs.

Instead of the backend server API establishing cookie authentication with a call to AddIdentityCookies on the authentication builder, the server API sets up bearer token auth with the AddBearerToken extension method. Specify the scheme for bearer authentication tokens with IdentityConstants.BearerScheme.

In `Backend/Program.cs`, change the authentication services and configuration to the following:

```C#
builder.Services
    .AddAuthentication()
    .AddBearerToken(IdentityConstants.BearerScheme);
```

In `BlazorWasmAuth/Identity/CookieAuthenticationStateProvider.cs`, remove the `useCookies` query string parameter in the `LoginAsync` method of the `CookieAuthenticationStateProvider`:

```diff
- login?useCookies=true
+ login
```

At this point, you must provide custom code to parse the AccessTokenResponse on the client and manage the access and refresh tokens. For more information, see Use Identity to secure a Web API backend for SPAs.

Additional Identity scenarios

Scenarios covered by the Blazor documentation set:

- Account confirmation, password management, and recovery codes
- Two-factor authentication (2FA): *Implementation guidance coming soon!* This work is tracked by Add 2FA/TOTP coverage to the Standalone+Identity article+sample (dotnet/AspNetCore.Docs #33772). In the meantime, general information to aid you with a custom implementation is available in Use Identity to secure a Web API backend for SPAs.

For information on additional Identity scenarios provided by the API, see Use Identity to secure a Web API backend for SPAs:

- Secure selected endpoints
- Two-factor authentication (2FA) and recovery codes
- User info management

Use secure authentication flows to maintain sensitive data and credentials

⚠ 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.

Sample apps

In this article, sample apps serve as a reference for standalone Blazor WebAssembly apps that access ASP.NET Core Identity through a backend web API. The demonstration includes two apps:

- `Backend`: A backend web API app that maintains a user identity store for ASP.NET Core Identity.
- `BlazorWasmAuth`: A standalone Blazor WebAssembly frontend app with user authentication.

Access the sample apps through the latest version folder from the repository's root with the following link. The samples are provided for .NET 8 or later. See the `README` file in the `BlazorWebAssemblyStandaloneWithIdentity` folder for steps on how to run the sample apps.

View or download sample code (how to download)

Backend web API app packages and code

The backend web API app maintains a user identity store for ASP.NET Core Identity.

Packages

The app uses the following NuGet packages:

- Microsoft.AspNetCore.Identity.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.InMemory
- NSwag.AspNetCore

If your app is to use a different EF Core database provider than the in-memory provider, don't create a package reference in your app for `Microsoft.EntityFrameworkCore.InMemory`.

In the app's project file (`.csproj`), invariant globalization is configured.

Sample app code

App settings configure backend and frontend URLs:

- `Backend` app (`BackendUrl`): `https://localhost:7211`
- `BlazorWasmAuth` app (`FrontendUrl`): `https://localhost:7171`

The Backend.http file can be used for testing the weather data request. Note that the `BlazorWasmAuth` app 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.

The following setup and configuration is found in the app's Program file.

User identity with cookie authentication is added by calling AddAuthentication and AddIdentityCookies. Services for authorization checks are added by a call to AddAuthorizationBuilder.

Only recommended for demonstrations, the app uses the EF Core in-memory database provider for the database context registration (AddDbContext). The in-memory database provider makes it easy to restart the app and test the registration and login user flows. Each run starts with a fresh database, but the app includes test user seeding demonstration code, which is described later in this article. If the database is changed to SQLite, users are saved between sessions, but the database must be created through migrations, as shown in the EF Core getting started tutorial†. You can use other relational providers such as SQL Server for your production code.

ⓘ Note

†The EF Core getting started tutorial uses PowerShell commands to execute database migrations when using Visual Studio. An alternative approach in Visual Studio is to use the Connected Services UI:

In **Solution Explorer**, double-click **Connected Services**. In **Service Dependencies** > **SQL Server Express LocalDB**, select the ellipsis (`...`) followed by either **Add migration** to create a migration or **Update database** to update the database.

Configure Identity to use the EF Core database and expose the Identity endpoints via the calls to AddIdentityCore, AddEntityFrameworkStores, and AddApiEndpoints.

A Cross-Origin Resource Sharing (CORS) policy is established to permit requests from the frontend and backend apps. Fallback URLs are configured for the CORS policy if app settings don't provide them:

- `Backend` app (`BackendUrl`): `https://localhost:5001`
- `BlazorWasmAuth` app (`FrontendUrl`): `https://localhost:5002`

Services and endpoints for Swagger/OpenAPI are included for web API documentation and development testing. For more information on NSwag, see Get started with NSwag and ASP.NET Core.

User role claims are sent from a Minimal API at the `/roles` endpoint.

Routes are mapped for Identity endpoints by calling `MapIdentityApi<AppUser>()`.

A logout endpoint (`/Logout`) is configured in the middleware pipeline to sign users out.

To secure an endpoint, add the RequireAuthorization extension method to the route definition. For a controller, add the [Authorize] attribute to the controller or action.

For more information on basic patterns for initialization and configuration of a DbContext instance, see DbContext Lifetime, Configuration, and Initialization in the EF Core documentation.

Frontend standalone Blazor WebAssembly app packages and code

A standalone Blazor WebAssembly frontend app demonstrates user authentication and authorization to access a private webpage.

Packages

The app uses the following NuGet packages:

- Microsoft.AspNetCore.Components.WebAssembly.Authentication
- Microsoft.Extensions.Http
- Microsoft.AspNetCore.Components.WebAssembly
- Microsoft.AspNetCore.Components.WebAssembly.DevServer

Sample app code

The `Models` folder contains the app's models:

- FormResult (Identity/Models/FormResult.cs): Response for login and registration.
- UserInfo (Identity/Models/UserInfo.cs): User info from identity endpoint to establish claims.

The IAccountManagement interface (Identity/CookieHandler.cs) provides account management services.

The CookieAuthenticationStateProvider class (Identity/CookieAuthenticationStateProvider.cs) handles state for cookie-based authentication and provides account management service implementations described by the `IAccountManagement` interface. The `LoginAsync` method explicitly enables cookie authentication via the `useCookies` query string value of `true`. The class also manages creating role claims for authenticated users.

The CookieHandler class (Identity/CookieHandler.cs) ensures cookie credentials are sent with each request to the backend web API, which handles Identity and maintains the Identity data store.

The wwwroot/appsettings.file provides backend and frontend URL endpoints.

The App component exposes the authentication state as a cascading parameter. For more information, see ASP.NET Core Blazor authentication and authorization.

The MainLayout component and NavMenu component use the AuthorizeView component to selectively display content based on the user's authentication status.

The following components handle common user authentication tasks, making use of `IAccountManagement` services:

- Register component (Components/Identity/Register.razor)
- Login component (Components/Identity/Login.razor)

- Logout component (Components/Identity/Logout.razor)

The PrivatePage component (Components/Pages/PrivatePage.razor) requires authentication and shows the user's claims.

Services and configuration is provided in the Program file (Program.cs):

- The cookie handler is registered as a scoped service.
- Authorization services are registered.
- The custom authentication state provider is registered as a scoped service.
- The account management interface (`IAccountManagement`) is registered.
- The base host URL is configured for a registered HTTP client instance.
- The base backend URL is configured for a registered HTTP client instance that's used for auth interactions with the backend web API. The HTTP client uses the cookie handler to ensure that cookie credentials are sent with each request.

Call AuthenticationStateProvider.NotifyAuthenticationStateChanged when the user's authentication state changes. For an example, see the `LoginAsync` and `LogoutAsync` methods of the CookieAuthenticationStateProvider class (Identity/CookieAuthenticationStateProvider.cs).

⚠ Warning

The AuthorizeView component selectively displays UI content depending on whether the user is authorized. All content within a Blazor WebAssembly app placed in an AuthorizeView component is discoverable without authentication, so sensitive content should be obtained from a backend server-based web API after authentication succeeds. For more information, see the following resources:

- ASP.NET Core Blazor authentication and authorization
- Call a web API from an ASP.NET Core Blazor app
- ASP.NET Core Blazor WebAssembly additional security scenarios

Test user seeding demonstration

The `SeedData` class (SeedData.cs) demonstrates how to create test users for development. The test user, named Leela, signs into the app with the email address `leela@contoso.com`. The user's password is set to `Passw0rd!`. Leela is given `Administrator` and `Manager` roles for authorization, which enables the user to access the manager page at `/private-manager-page` but not the editor page at `/private-editor-page`.

⚠ Warning

Never allow test user code to run in a production environment. `SeedData.InitializeAsync` is only called in the `Development` environment in the `Program` file:

```C#
if (builder.Environment.IsDevelopment())
{
    await using var scope = app.Services.CreateAsyncScope();
    await SeedData.InitializeAsync(scope.ServiceProvider);
}
```

Roles

Role claims aren't sent back from the `manage/info` endpoint to create user claims for users of the `BlazorWasmAuth` app. Role claims are managed independently via a separate request in the `GetAuthenticationStateAsync` method of the CookieAuthenticationStateProvider class (Identity/CookieAuthenticationStateProvider.cs) after the user is authenticated in the `Backend` project.

In the `CookieAuthenticationStateProvider`, a roles request is made to the `/roles` endpoint of the `Backend` server API project. The response is read into a string by calling ReadAsStringAsync(). JsonSerializer.Deserialize deserializes the string into a custom `RoleClaim` array. Finally, the claims are added to the user's claims collection.

In the `Backend` server API's `Program` file, a Minimal API manages the `/roles` endpoint. Claims of RoleClaimType are selected into an anonymous type and serialized for return to the `BlazorWasmAuth` project with TypedResults.Json.

The roles endpoint requires authorization by calling RequireAuthorization. If you decide not to use Minimal APIs in favor of controllers for secure server API endpoints, be sure to set the [Authorize] attribute on controllers or actions.

Cross-domain hosting (same-site configuration)

The sample apps are configured for hosting both apps at the same domain. If you host the `Backend` app at a different domain than the `BlazorWasmAuth` app, uncomment the code that configures the cookie (ConfigureApplicationCookie) in the `Backend` app's `Program` file. The default values are:

- Same-site mode: SameSiteMode.Lax
- Secure policy: CookieSecurePolicy.SameAsRequest

Change the values to:

- Same-site mode: SameSiteMode.None
- Secure policy: CookieSecurePolicy.Always

diff

```diff
- options.Cookie.SameSite = SameSiteMode.Lax;
- options.Cookie.SecurePolicy = CookieSecurePolicy.SameAsRequest;
+ options.Cookie.SameSite = SameSiteMode.None;
+ options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
```

For more information on same-site cookie settings, see the following resources:

- Set-Cookie: SameSite=<samesite-value> (MDN documentation)
- Cookies: HTTP State Management Mechanism (RFC Draft 6265, Section 4.1)

Antiforgery support

Only the logout endpoint (`/logout`) in the `Backend` app requires attention to mitigate the threat of Cross-Site Request Forgery (CSRF).

The logout endpoint checks for an empty body to prevent CSRF attacks. By requiring a body, the request must be made from JavaScript, which is the only way to access the authentication cookie. The logout endpoint can't be accessed by a form-based POST. This prevents a malicious site from logging the user out.

Furthermore, the endpoint is protected by authorization (RequireAuthorization) to prevent anonymous access.

The `BlazorWasmAuth` client app is simply required to pass an empty object `{}` in the body of the request.

Outside of the logout endpoint, antiforgery mitigation is only required when submitting form data to the server encoded as `application/x-www-form-urlencoded`, `multipart/form-data`, or `text/plain`. Blazor manages CSRF mitigation for forms in most cases. For more information, see ASP.NET Core Blazor authentication and authorization and ASP.NET Core Blazor forms overview.

Requests to other server API endpoints (web API) with `application/json`-encoded content and CORS enabled doesn't require CSRF protection. This is why no CSRF protection is required for the `Backend` app's data processing (`/data-processing`) endpoint. The roles (`/roles`) endpoint doesn't need CSRF protection because it's a GET endpoint that doesn't modify any state.

Troubleshoot

Logging

To enable debug or trace logging for Blazor WebAssembly authentication, see ASP.NET Core Blazor logging.

Common errors

Check each project's configuration. Confirm that the URLs are correct:

- `Backend` project
  - `appsettings.json`
    - `BackendUrl`
    - `FrontendUrl`
  - `Backend.http`: `Backend_HostAddress`
- `BlazorWasmAuth` project: `wwwroot/appsettings.json`
  - `BackendUrl`
  - `FrontendUrl`

If the configuration appears correct:

- Analyze application logs.

- Examine the network traffic between the `BlazorWasmAuth` app and `Backend` 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 backend 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 using the **Open a documentation issue** link at the bottom of the article. The team isn't able 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) .

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, or 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's claims

To troubleshoot problems with user claims, the following `UserClaims` component can be used directly in apps or serve as the basis for further customization.

`UserClaims.razor`:

razor

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

<PageTitle>User Claims</PageTitle>

<h1>User Claims</h1>

**Name**: @AuthenticatedUser?.Identity?.Name

<h2>Claims</h2>

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

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

    public ClaimsPrincipal? AuthenticatedUser { get; set; }

    protected override async Task OnInitializedAsync()
    {
        if (AuthenticationState is not null)
        {
            var state = await AuthenticationState;
            AuthenticatedUser = state.User;
        }
    }
}
```

Additional resources

- AuthenticationStateProvider service
- Client-side SignalR cross-origin negotiation for authentication

Account confirmation and password recovery in ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity

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 WebAssembly app with ASP.NET Core Identity with email confirmation and password recovery.

ⓘ **Note**

This article only applies standalone Blazor WebAssembly apps with ASP.NET Core Identity. To implement email confirmation and password recovery for Blazor Web Apps, see Account confirmation and password recovery in ASP.NET Core Blazor.

Namespaces and article code examples

The namespaces used by the examples in this article are:

- `Backend` for the backend server web API project ("server project" in this article).
- `BlazorWasmAuth` for the front-end client standalone Blazor WebAssembly app ("client project" in this article).

These namespaces correspond to the projects in the `BlazorWebAssemblyStandaloneWithIdentity` sample solution in the dotnet/blazor-samples GitHub repository. For more information, see Secure ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity.

If you aren't using the `BlazorWebAssemblyStandaloneWithIdentity` sample solution, change the namespaces in the code examples to use the namespaces of your projects.

Select and configure an email provider for the server project

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.

In the server project, 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 Backend;

public class AuthMessageSenderOptions
{
    public string? EmailAuthKey { get; set; }
}
```

Register the `AuthMessageSenderOptions` configuration instance in the server project's `Program` file:

```C#
builder.Services.Configure<AuthMessageSenderOptions>(builder.Configuration);
```

Configure a user secret for the provider's security key

If the server project has already been initialized for the Secret Manager tool, it will already have a 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 server project's directory. In Visual Studio, you can use the Developer PowerShell command prompt (use the `cd` command to change the directory to the server project after you open the command shell).

.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 `IEmailSender` in the server project

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 server project.

Add the following `EmailSender` class to implement IEmailSender<TUser>. In the following example, `AppUser` 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.

`EmailSender.cs`:

```
C#
```

```csharp
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.Options;
using Mandrill;
using Mandrill.Model;

namespace Backend;

public class EmailSender(IOptions<AuthMessageSenderOptions> optionsAccessor,
    ILogger<EmailSender> logger) : IEmailSender<AppUser>
{
    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.

Add the following IEmailSender<TUser> service registration to the server project's `Program` file:

C#

```
builder.Services.AddTransient<IEmailSender<AppUser>, EmailSender>();
```

Configure the server project to require email confirmation

In the server project's `Program` file, require a confirmed email address to sign in to the app.

Locate the line that calls AddIdentityCore and set the RequireConfirmedEmail property to `true`:

diff

```
- builder.Services.AddIdentityCore<AppUser>()
+ builder.Services.AddIdentityCore<AppUser>(o => 
o.SignIn.RequireConfirmedEmail = true)
```

Update the client project's account registration response

In the client project's `Register` component (`Components/Identity/Register.razor`), change the message to users on a successful account registration to instruct them to confirm their account. The following example includes a link to trigger Identity on the server to resend the confirmation email.

```diff
- You successfully registered and can <a href="login">login</a> to the app.
+ You successfully registered. You must now confirm your account by clicking
+ the link in the email that was sent to you. After confirming your account,
+ you can <a href="login">login</a> to the app.
+ <a href="resendConfirmationEmail">Resend confirmation email</a>
```

Update seed data code to confirm seeded accounts

In the server project's seed data class (`SeedData.cs`), change the code in the `InitializeAsync` method to confirm the seeded accounts, which avoids requiring email address confirmation for each test run of the solution with one of the accounts:

```diff
- if (appUser is not null && user.RoleList is not null)
- {
-     await userManager.AddToRolesAsync(appUser, user.RoleList);
- }
+ if (appUser is not null)
+ {
+     if (user.RoleList is not null)
+     {
+         await userManager.AddToRolesAsync(appUser, user.RoleList);
+     }
+
+     var token = await
userManager.GenerateEmailConfirmationTokenAsync(appUser);
+     await userManager.ConfirmEmailAsync(appUser, token);
+ }
```

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. Use one of the following approaches, which are beyond the scope of this article:

- Update the database to mark all existing users as confirmed.
- Batch-send emails with confirmation links to all existing users, which requires each user to confirm their own account.

Password recovery

Password recovery requires the server app to adopt an email provider in order to send password reset codes to users. Therefore, follow the guidance earlier in this article to adopt an email provider:

- Select and configure an email provider for the server project
- Configure a user secret for the provider's security key
- Implement IEmailSender in the server project

Password recovery is a two-step process:

1. A POST request is made to the `/forgotPassword` endpoint provided by MapIdentityApi in the server project. A message in the UI instructs the user to check their email for a reset code.
2. A POST request is made to the `/resetPassword` endpoint of the server project with the user's email address, password reset code, and new password.

The preceding steps are demonstrated by the following implementation guidance for the sample solution.

In the client project, add the following method signatures to the `IAccountManagement` class (`Identity/IAccountManagement.cs`):

```
C#

public Task<bool> ForgotPasswordAsync(string email);

public Task<FormResult> ResetPasswordAsync(string email, string resetCode,
    string newPassword);
```

In the client project, add implementations for the preceding methods in the `CookieAuthenticationStateProvider` class (`Identity/CookieAuthenticationStateProvider.cs`):

C#

```
public async Task<bool> ForgotPasswordAsync(string email)
{
    try
    {
        var result = await httpClient.PostAsJsonAsync(
            "forgotPassword", new
            {
                email
            });

        if (result.IsSuccessStatusCode)
        {
            return true;
        }
    }
    catch { }

    return false;
}

public async Task<FormResult> ResetPasswordAsync(string email, string
resetCode,
    string newPassword)
{
    string[] defaultDetail = ["An unknown error prevented resetting the
password."];

    try
    {
        var result = await httpClient.PostAsJsonAsync(
            "resetPassword", new
            {
                email,
                resetCode,
                newPassword
            });

        if (result.IsSuccessStatusCode)
        {
            return new FormResult { Succeeded = true };
        }

        var details = await result.Content.ReadAsStringAsync();
        var problemDetails = JsonDocument.Parse(details);
        var errors = new List<string>();
        var errorList = problemDetails.RootElement.GetProperty("errors");
```

```
            foreach (var errorEntry in errorList.EnumerateObject())
            {
                if (errorEntry.Value.ValueKind == JsonValueKind.String)
                {
                    errors.Add(errorEntry.Value.GetString()!);
                }
                else if (errorEntry.Value.ValueKind == JsonValueKind.Array)
                {
                    errors.AddRange(
                        errorEntry.Value.EnumerateArray().Select(
                            e => e.GetString() ?? string.Empty)
                        .Where(e => !string.IsNullOrEmpty(e)));
                }
            }

            return new FormResult
            {
                Succeeded = false,
                ErrorList = problemDetails == null ? defaultDetail : [.. errors]
            };
        }
        catch { }

        return new FormResult
        {
            Succeeded = false,
            ErrorList = defaultDetail
        };
    }
```

In the client project, add the following `ForgotPassword` component.

`Components/Identity/ForgotPassword.razor`:

razor

```razor
@page "/forgot-password"
@using System.ComponentModel.DataAnnotations
@using BlazorWasmAuth.Identity
@inject IAccountManagement Acct

<PageTitle>Forgot your password?</PageTitle>

<h1>Forgot your password?</h1>
<p>Provide your email address and select the <b>Reset password</b> button.
</p>
<hr />
<div class="row">
    <div class="col-md-4">
        @if (!passwordResetCodeSent)
        {
            <EditForm Model="Input" FormName="forgot-password"
                      OnValidSubmit="OnValidSubmitStep1Async" method="post">
```

```
                <DataAnnotationsValidator />
                <ValidationSummary class="text-danger" role="alert" />

                <div class="form-floating mb-3">
                    <InputText @bind-Value="Input.Email"
                        id="Input.Email" class="form-control"
                        autocomplete="username" aria-required="true"
                        placeholder="name@example.com" />
                    <label for="Input.Email" class="form-label">
                        Email
                    </label>
                    <ValidationMessage For="() => Input.Email"
                        class="text-danger" />
                </div>
                <button type="submit" class="w-100 btn btn-lg btn-primary">
                    Request reset code
                </button>
            </EditForm>
        }
        else
        {
            if (passwordResetSuccess)
            {
                if (errors)
                {
                    foreach (var error in errorList)
                    {
                        <div class="alert alert-danger">@error</div>
                    }
                }
                else
                {
                    <div>
                        Your password was reset. You may <a
href="login">login</a>
                        to the app with your new password.
                    </div>
                }
            }
            else
            {
                <div>
                    A password reset code has been sent to your email
address.
                    Obtain the code from the email for this form.
                </div>
                <EditForm Model="Reset" FormName="reset-password"
                    OnValidSubmit="OnValidSubmitStep2Async" method="post">
                    <DataAnnotationsValidator />
                    <ValidationSummary class="text-danger" role="alert" />

                    <div class="form-floating mb-3">
                        <InputText @bind-Value="Reset.ResetCode"
                            id="Reset.ResetCode" class="form-control"
                            autocomplete="username" aria-required="true" />
                        <label for="Reset.ResetCode" class="form-label">
                            Reset code
                        </label>
                        <ValidationMessage For="() => Reset.ResetCode"
                            class="text-danger" />
                    </div>
                    <div class="form-floating mb-3">
                        <InputText type="password" @bind-
Value="Reset.NewPassword"
                            id="Reset.NewPassword" class="form-control"
                            autocomplete="new-password" aria-required="true"
                            placeholder="password" />
                        <label for="Reset.NewPassword" class="form-label">
                            New Password
                        </label>
                        <ValidationMessage For="() => Reset.NewPassword"
                            class="text-danger" />
                    </div>
                    <div class="form-floating mb-3">
                        <InputText type="password"
                            @bind-Value="Reset.ConfirmPassword"
                            id="Reset.ConfirmPassword" class="form-control"
                            autocomplete="new-password" aria-required="true"
                            placeholder="password" />
                        <label for="Reset.ConfirmPassword" class="form-
label">
                            Confirm Password
                        </label>
                        <ValidationMessage For="() => Reset.ConfirmPassword"
                            class="text-danger" />
                    </div>
                    <button type="submit" class="w-100 btn btn-lg btn-
primary">
                        Reset password
                    </button>
                </EditForm>
            }
        }
    </div>
</div>

@code {
    private bool passwordResetCodeSent, passwordResetSuccess, errors;
    private string[] errorList = [];

    [SupplyParameterFromForm(FormName = "forgot-password")]
    private InputModel Input { get; set; } = new();

    [SupplyParameterFromForm(FormName = "reset-password")]
    private ResetModel Reset { get; set; } = new();

    private async Task OnValidSubmitStep1Async()
    {
        passwordResetCodeSent = await Acct.ForgotPasswordAsync(Input.Email);
    }
    private async Task OnValidSubmitStep2Async()
    {
        var result = await Acct.ResetPasswordAsync(Input.Email,
Reset.ResetCode,
            Reset.NewPassword);

        if (result.Succeeded)
        {
            passwordResetSuccess = true;

        }
        else
        {
            errors = true;
            errorList = result.ErrorList;
        }
    }

    private sealed class InputModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email")]
        public string Email { get; set; } = string.Empty;
    }

    private sealed class ResetModel
    {
        [Required]
        [Base64String]
        public string ResetCode { get; set; } = string.Empty;

        [StringLength(100, ErrorMessage = "The {0} must be at least {2} and
at " +
            "max {1} characters long.", MinimumLength = 6)]
        [DataType(DataType.Password)]
        [Display(Name = "Password")]
        public string NewPassword { get; set; } = string.Empty;

        [DataType(DataType.Password)]
        [Display(Name = "Confirm password")]
        [Compare("NewPassword", ErrorMessage = "The new password and
confirmation " +
            "password don't match.")]
        public string ConfirmPassword { get; set; } = string.Empty;
    }
}
```

In the `Login` component (`Components/Identity/Login.razor`) of the client project immediately before the closing `</NotAuthorized>` tag, add a forgot password link to reach the `ForgotPassword` component:

HTML

```html
<div>
    <a href="forgot-password">Forgot password</a>
</div>
```

Email and activity timeout

The default inactivity timeout is 14 days. In the server project, the following code sets the inactivity timeout to five days with sliding expiration:

C#

```csharp
builder.Services.ConfigureApplicationCookie(options => {
    options.ExpireTimeSpan = TimeSpan.FromDays(5);
    options.SlidingExpiration = true;
});
```

Change all ASP.NET Core Data Protection token lifespans

In the server project, the following code changes Data Protection tokens' timeout period to three hours:

C#

```csharp
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 to the server project.

`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 server project's `Program` file:

```C#
builder.Services.AddIdentityCore<AppUser>(options =>
    {
        options.SignIn.RequireConfirmedAccount = true;
        options.Tokens.ProviderMap.Add("CustomEmailConfirmation",
            new TokenProviderDescriptor(
                typeof(CustomEmailConfirmationTokenProvider<AppUser>)));
        options.Tokens.EmailConfirmationTokenProvider =
            "CustomEmailConfirmation";
    })
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddSignInManager()
    .AddDefaultTokenProviders();

builder.Services
    .AddTransient<CustomEmailConfirmationTokenProvider<AppUser>>();
```

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 ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity

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 WebAssembly app with Identity 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.

⚠ **Warning**

TOTP codes should be kept secret because they can be used to authenticate multiple times before they expire.

Namespaces and article code examples

The namespaces used by the examples in this article are:

- `Backend` for the backend server web API project, described as the "server project" in this article.
- `BlazorWasmAuth` for the front-end client standalone Blazor WebAssembly app, described as the "client project" in this article.

These namespaces correspond to the projects in the `BlazorWebAssemblyStandaloneWithIdentity` sample solution in the dotnet/blazor-samples

GitHub repository. For more information, see Secure ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity.

If you aren't using the `BlazorWebAssemblyStandaloneWithIdentity` sample, change the namespaces in the code examples to use the namespaces of your projects.

All of the changes to the solution covered by this article take place in the `BlazorWasmAuth` project of the `BlazorWebAssemblyStandaloneWithIdentity` solution.

In article examples, code lines are split to reduce horizontal scrolling. These breaks don't affect execution but can be removed when pasting into your project.

Optional account confirmation and password recovery

Although apps that implement 2FA usually adopt account confirmation and password recovery features, 2FA doesn't require it. The guidance in this article can be followed to implement 2FA without following the guidance in Account confirmation and password recovery in ASP.NET Core Blazor WebAssembly with ASP.NET Core Identity.

Add a QR code library to the app

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 to the client project 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.

Set the TOTP organization name

Set the site name in the app settings file of the client project. Use 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 the company name is `Weyland-Yutani Corporation` (©1986 20th Century Studios *Aliens*).

Added to `wwwroot/appsettings.json`:

```json
"TotpOrganizationName": "Weyland-Yutani Corporation"
```

The app settings file after the TOTP organization name configuration is added:

```json
{
  "BackendUrl": "https://localhost:7211",
  "FrontendUrl": "https://localhost:7171",
  "TotpOrganizationName": "Weyland-Yutani Corporation"
}
```

Add model classes

Add the following `LoginResponse` class to the `Models` folder. This class is populated for requests to the `/login` endpoint of MapIdentityApi in the server app.

`Identity/Models/LoginResponse.cs`:

```csharp
namespace BlazorWasmAuth.Identity.Models;

public class LoginResponse
{
    public string? Type { get; set; }
    public string? Title { get; set; }
    public int Status { get; set; }
    public string? Detail { get; set; }
}
```

Add the following `TwoFactorRequest` class to the `Models` folder. This class is populated for requests to the `/manage/2fa` endpoint of MapIdentityApi in the server app.

`Identity/Models/TwoFactorRequest.cs`:

```csharp
namespace BlazorWasmAuth.Identity.Models;

public class TwoFactorRequest
{
    public bool? Enable { get; set; }
    public string? TwoFactorCode { get; set; }
    public bool? ResetSharedKey { get; set; }
    public bool? ResetRecoveryCodes { get; set; }
    public bool? ForgetMachine {  get; set; }
}
```

Add the following `TwoFactorResponse` class to the `Models` folder. This class is populated by the response to a 2FA request made to the `/manage/2fa` endpoint of MapIdentityApi in the server app.

`Identity/Models/TwoFactorResponse.cs`:

```csharp
namespace BlazorWasmAuth.Identity.Models;

public class TwoFactorResponse
{
    public string SharedKey { get; set; } = string.Empty;
    public int RecoveryCodesLeft { get; set; } = 0;
    public string[] RecoveryCodes { get; set; } = [];
    public bool IsTwoFactorEnabled { get; set; }
    public bool IsMachineRemembered { get; set; }
    public string[] ErrorList { get; set; } = [];
}
```

`IAccountManagement` interface

Add the following class signatures to the `IAccountManagement` interface. The class signatures represent methods added to the cookie authentication state provider for the following client requests:

- Log in with a 2FA TOTP code (`/login` endpoint): `LoginTwoFactorCodeAsync`

- Log in with a 2FA recovery code (`/login` endpoint): `LoginTwoFactorRecoveryCodeAsync`
- Make a 2FA management request (`/manage/2fa` endpoint): `TwoFactorRequestAsync`

`Identity/IAccountManagement.cs` (paste the following code at the bottom of the file):

```C#
public Task<FormResult> LoginTwoFactorCodeAsync(
    string email,
    string password,
    string twoFactorCode);

public Task<FormResult> LoginTwoFactorRecoveryCodeAsync(
    string email,
    string password,
    string twoFactorRecoveryCode);

public Task<TwoFactorResponse> TwoFactorRequestAsync(
    TwoFactorRequest twoFactorRequest);
```

Update the cookie authentication state provider

Update the `CookieAuthenticationStateProvider` with features to add the following features:

- Authenticate users with either a TOTP authenticator app code or a recovery code.
- Manage 2FA in the app.

At the top of the `CookieAuthenticationStateProvider.cs` file, add a `using` statement for System.Text.Json.Serialization:

```C#
using System.Text.Json.Serialization;
```

In the JsonSerializerOptions, add the DefaultIgnoreCondition option set to JsonIgnoreCondition.WhenWritingNull, which avoids serializing null properties:

```diff
private readonly JsonSerializerOptions jsonSerializerOptions =
    new()
    {
        PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
+       DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull,
    };
```

The `LoginAsync` method is updated with the following logic:

- Attempt a normal login at the `/login` endpoint with an email address and password.
- If the server responds with a success status code, the method returns a `FormResult` with the `Succeeded` property set to `true`.
- If the server responds with the *401 - Unauthorized* status code and a detail code of "`RequiresTwoFactor`," a `FormResult` is returned with `Succeeded` set to `false` and the `RequiresTwoFactor` detail in the error list.

In `Identity/CookieAuthenticationStateProvider.cs`, replace the `LoginAsync` method with the following code:

C#

```csharp
public async Task<FormResult> LoginAsync(string email, string password)
{
    try
    {
        var result = await httpClient.PostAsJsonAsync(
            "login?useCookies=true", new
            {
                email,
                password
            });

        if (result.IsSuccessStatusCode)
        {
            NotifyAuthenticationStateChanged(GetAuthenticationStateAsync());

            return new FormResult { Succeeded = true };
        }
        else if (result.StatusCode == HttpStatusCode.Unauthorized)
        {
            var responseJson = await result.Content.ReadAsStringAsync();
            var response = JsonSerializer.Deserialize<LoginResponse>(
                responseJson, jsonSerializerOptions);

            if (response?.Detail == "RequiresTwoFactor")
            {
                return new FormResult
                {
                    Succeeded = false,
                    ErrorList = [ "RequiresTwoFactor" ]
                };
            }
        }
    }
    catch (Exception ex)
    {
        logger.LogError(ex, "App error");
    }

    return new FormResult
    {
        Succeeded = false,
        ErrorList = [ "Invalid email and/or password." ]
    };
}
```

A `LoginTwoFactorCodeAsync` method is added, which sends a request to the `/login` endpoint with a 2FA TOTP code (`twoFactorCode`). The method processes the response in a similar fashion to a normal, non-2FA login request.

Add the following method and class to `Identity/CookieAuthenticationStateProvider.cs` (paste the following code at the bottom of the class file):

C#

```csharp
public async Task<FormResult> LoginTwoFactorCodeAsync(
    string email, string password, string twoFactorCode)
{
    try
    {
        var result = await httpClient.PostAsJsonAsync(
            "login?useCookies=true", new
            {
                email,
                password,
                twoFactorCode
            });

        if (result.IsSuccessStatusCode)
        {
            NotifyAuthenticationStateChanged(GetAuthenticationStateAsync());

            return new FormResult { Succeeded = true };
        }
    }
    catch (Exception ex)
    {
        logger.LogError(ex, "App error");
    }

    return new FormResult
    {
        Succeeded = false,
        ErrorList = [ "Invalid two-factor code." ]
    };
}
```

A `LoginTwoFactorRecoveryCodeAsync` method is added, which sends a request to the `/login` endpoint with a 2FA recovery code (`twoFactorRecoveryCode`). The method processes the response in a similar fashion to a normal, non-2FA login request.

Add the following method and class to `Identity/CookieAuthenticationStateProvider.cs` (paste the following code at the bottom of the class file):

```csharp
public async Task<FormResult> LoginTwoFactorRecoveryCodeAsync(string email,
    string password, string twoFactorRecoveryCode)
{
    try
    {
        var result = await httpClient.PostAsJsonAsync(
            "login?useCookies=true", new
            {
                email,
                password,
                twoFactorRecoveryCode
            });

        if (result.IsSuccessStatusCode)
        {
            NotifyAuthenticationStateChanged(GetAuthenticationStateAsync());

            return new FormResult { Succeeded = true };
        }
    }
    catch (Exception ex)
    {
        logger.LogError(ex, "App error");
    }

    return new FormResult
    {
        Succeeded = false,
        ErrorList = [ "Invalid recovery code." ]
    };
}
```

A `TwoFactorRequestAsync` method is added, which is used to manage 2FA for the user:

- Reset the shared 2FA key when `TwoFactorRequest.ResetSharedKey` is `true`. Resetting the shared key implicitly disables 2FA. This forces the user to prove that

they can provide a valid TOTP code from their authenticator app to enable 2FA after receiving a new shared key.
- Reset the user's recovery codes when `TwoFactorRequest.ResetRecoveryCodes` is `true`.
- Forget the machine when `TwoFactorRequest.ForgetMachine` is `true`, which means that a new 2FA TOTP code is required on the next login attempt.
- Enable 2FA using a TOTP code from a TOTP authenticator app when `TwoFactorRequest.Enable` is `true` and `TwoFactorRequest.TwoFactorCode` has a valid TOTP value.
- Obtain 2FA status with an empty request when all of `TwoFactorRequest`'s properties are `null`.

Add the following `TwoFactorRequestAsync` method to `Identity/CookieAuthenticationStateProvider.cs` (paste the following code at the bottom of the class file):

```
C#

public async Task<TwoFactorResponse> TwoFactorRequestAsync(TwoFactorRequest
twoFactorRequest)
{
    string[] defaultDetail =
        [ "An unknown error prevented two-factor authentication." ];

    var response = await httpClient.PostAsJsonAsync("manage/2fa",
twoFactorRequest,
        jsonSerializerOptions);

    // successful?
    if (response.IsSuccessStatusCode)
    {
        return await response.Content
            .ReadFromJsonAsync<TwoFactorResponse>() ??
            new()
            {
                ErrorList = [ "There was an error processing the request." ]
            };
    }

    // body should contain details about why it failed
    var details = await response.Content.ReadAsStringAsync();
    var problemDetails = JsonDocument.Parse(details);
    var errors = new List<string>();
    var errorList = problemDetails.RootElement.GetProperty("errors");

    foreach (var errorEntry in errorList.EnumerateObject())
    {
        if (errorEntry.Value.ValueKind == JsonValueKind.String)
        {
            errors.Add(errorEntry.Value.GetString()!);
        }
        else if (errorEntry.Value.ValueKind == JsonValueKind.Array)
        {
            errors.AddRange(
                errorEntry.Value.EnumerateArray().Select(
                    e => e.GetString() ?? string.Empty)
                .Where(e => !string.IsNullOrEmpty(e)));
        }
    }

    // return the error list
    return new TwoFactorResponse
    {
        ErrorList = problemDetails == null ? defaultDetail : [.. errors]
    };
}
```

Replace `Login` component

Replace the `Login` component. The following version of the `Login` component:

- Accepts a user's email address and password for an initial login attempt.
- If login is successful (2FA is disabled), the component notifies the user that they're authenticated.
- If the login attempt results in a response indicating that 2FA is required, a 2FA input element appears to receive either a 2FA TOTP code from an authenticator app or a recovery code. Depending on which code the user enters, login is attempted again by calling either `LoginTwoFactorCodeAsync` for a TOTP code or `LoginTwoFactorRecoveryCodeAsync` for a recovery code.

`Components/Identity/Login.razor`:

razor

```razor
@page "/login"
@using System.ComponentModel.DataAnnotations
@using BlazorWasmAuth.Identity
@using BlazorWasmAuth.Identity.Models
@inject IAccountManagement Acct
@inject ILogger<Login> Logger
@inject NavigationManager Navigation

<PageTitle>Login</PageTitle>

<h1>Login</h1>

<AuthorizeView>
```

```
    <Authorized>
        <div class="alert alert-success">
            You're logged in as @context.User.Identity?.Name.
        </div>
    </Authorized>
    <NotAuthorized>
        @foreach (var error in formResult.ErrorList)
        {
            <div class="alert alert-danger">@error</div>
        }
        <div class="row">
            <div class="col">
                <section>
                    <EditForm Model="Input" method="post"
OnValidSubmit="LoginUser"
                            FormName="login" Context="editform_context">
                        <DataAnnotationsValidator />
                        <h2>Use a local account to log in.</h2>
                        <hr />
                        <div style="display:@(requiresTwoFactor ? "none" :
"block")">
                            <div class="form-floating mb-3">
                                <InputText @bind-Value="Input.Email"
                                    id="Input.Email"
                                    class="form-control"
                                    autocomplete="username"
                                    aria-required="true"
                                    placeholder="name@example.com" />
                                <label for="Input.Email" class="form-label">
                                    Email
                                </label>
                                <ValidationMessage For="() => Input.Email"
                                    class="text-danger" />
                            </div>
                            <div class="form-floating mb-3">
                                <InputText type="password"
                                    @bind-Value="Input.Password"
                                    id="Input.Password"
                                    class="form-control"
                                    autocomplete="current-password"
                                    aria-required="true"
                                    placeholder="password" />
                                <label for="Input.Password" class="form-
label">

                                    Password
                                </label>
                                <ValidationMessage For="() =>
Input.Password"

                                    class="text-danger" />
                            </div>
                        </div>
                        <div style="display:@(requiresTwoFactor ? "block" :
"none")">
                            <div class="form-floating mb-3">
                                <InputText @bind-
```

```
Value="Input.TwoFactorCodeOrRecoveryCode"
                                    id="Input.TwoFactorCodeOrRecoveryCode"
                                    class="form-control"
                                    autocomplete="off"
                                    placeholder="###### or #####-#####" />
                                <label
for="Input.TwoFactorCodeOrRecoveryCode" class="form-label">
                                    Two-factor Code or Recovery Code
                                </label>
                                <ValidationMessage For="() =>
Input.TwoFactorCodeOrRecoveryCode"
                                    class="text-danger" />
                            </div>
                        </div>
                        <div>
                            <button type="submit" class="w-100 btn btn-lg
btn-primary">
                                Log in
                            </button>
                        </div>
                        <div class="mt-3">
                            <p>
                                <a href="forgot-password">Forgot
password</a>
                            </p>
                            <p>
                                <a href="register">Register as a new
user</a>
                            </p>
                        </div>
                    </EditForm>
                </section>
            </div>
        </div>
    </NotAuthorized>
</AuthorizeView>

@code {
    private FormResult formResult = new();
    private bool requiresTwoFactor;

    [SupplyParameterFromForm]
    private InputModel Input { get; set; } = new();

    [SupplyParameterFromQuery]
    private string? ReturnUrl { get; set; }

    public async Task LoginUser()
    {
        if (requiresTwoFactor)
        {
            if (!string.IsNullOrEmpty(Input.TwoFactorCodeOrRecoveryCode))
            {
                // The [RegularExpression] data annotation ensures that the
input
```

```
                // is either a six-digit authenticator code (######) or an
                // eleven-character alphanumeric recovery code (#####-#####)
                if (Input.TwoFactorCodeOrRecoveryCode.Length == 6)
                {
                    formResult = await Acct.LoginTwoFactorCodeAsync(
                        Input.Email, Input.Password,
                        Input.TwoFactorCodeOrRecoveryCode);
                }
                else
                {
                    formResult = await Acct.LoginTwoFactorRecoveryCodeAsync(
                        Input.Email, Input.Password,
                        Input.TwoFactorCodeOrRecoveryCode);

                    if (formResult.Succeeded)
                    {
                        var twoFactorResponse = await
Acct.TwoFactorRequestAsync(new());
                    }
                }
            }
            else
            {
                formResult =
                    new FormResult
                    {
                        Succeeded = false,
                        ErrorList = [ "Invalid two-factor code." ]
                    };
            }
        }
        else
        {
            formResult = await Acct.LoginAsync(Input.Email, Input.Password);
            requiresTwoFactor =
formResult.ErrorList.Contains("RequiresTwoFactor");
            Input.TwoFactorCodeOrRecoveryCode = string.Empty;

            if (requiresTwoFactor)
            {
                formResult.ErrorList = [];
            }
        }

        if (formResult.Succeeded && !string.IsNullOrEmpty(ReturnUrl))
        {
            Navigation.NavigateTo(ReturnUrl);
        }
    }

    private sealed class InputModel
    {
        [Required]
        [EmailAddress]
        [Display(Name = "Email")]
        public string Email { get; set; } = string.Empty;

        [Required]
        [DataType(DataType.Password)]
        [Display(Name = "Password")]
        public string Password { get; set; } = string.Empty;

        [RegularExpression(@"^([0-9]{6})|([A-Z0-9]{5}[-]{1}[A-Z0-9]{5})$",
            ErrorMessage = "Must be a six-digit authenticator code (######)
or " +
            "eleven-character alphanumeric recovery code (#####-#####, dash
" +
            "required)")]
        [Display(Name = "Two-factor Code or Recovery Code")]
        public string TwoFactorCodeOrRecoveryCode { get; set; } =
string.Empty;
    }
}
```

Using the preceding component, the user is remembered after a successful login with a valid TOTP code from an authenticator app. If you want to always require a TOTP code for login and not remember the machine, call the `TwoFactorRequestAsync` method with `TwoFactorRequest.ForgetMachine` set to `true` immediately after a successful two-factor login:

```diff
if (Input.TwoFactorCodeOrRecoveryCode.Length == 6)
{
    formResult = await Acct.LoginTwoFactorCodeAsync(Input.Email,
Input.Password,
        Input.TwoFactorCodeOrRecoveryCode);

+    if (formResult.Succeeded)
+    {
+        var forgetMachine =
+            await Acct.TwoFactorRequestAsync(new() { ForgetMachine = true
});
+    }
}
```

Add a component to display recovery codes

Add the following `ShowRecoveryCodes` component to the app to display recovery codes to the user.

`Components/Identity/ShowRecoveryCodes.razor`:

razor

```razor
<h3>Recovery codes</h3>

<div class="alert alert-warning" role="alert">
    <p>
        <strong>Put these codes in a safe place.</strong>
    </p>
    <p>
        If you lose your device and don't have an unused
        recovery code, you can't access your account.
    </p>
</div>
<div class="row">
    <div class="col-md-12">
        @foreach (var recoveryCode in RecoveryCodes)
        {
            <div>
                <code class="recovery-code">@recoveryCode</code>
            </div>
        }
    </div>
</div>

@code {
    [Parameter]
    public string[] RecoveryCodes { get; set; } = [];
}
```

Manage 2FA page

Add the following `Manage2fa` component to the app to manage 2FA for users.

If 2FA isn't enabled, the component loads a form with a QR code to enable 2FA with a TOTP authenticator app. The user adds the app to their authenticator app and then verifies the authenticator app and enables 2FA by providing a TOTP code from the authenticator app.

If 2FA is enabled, buttons appear to disable 2FA and regenerate recovery codes.

`Components/Identity/Manage2fa.razor`:

razor

```razor
@page "/manage-2fa"
@using System.ComponentModel.DataAnnotations
@using System.Globalization
@using System.Text
@using System.Text.Encodings.Web
```

```
@using Net.Codecrete.QrCodeGenerator
@using BlazorWasmAuth.Identity
@using BlazorWasmAuth.Identity.Models
@attribute [Authorize]
@inject IAccountManagement Acct
@inject IAuthorizationService AuthorizationService
@inject IConfiguration Config
@inject ILogger<Manage2fa> Logger

<PageTitle>Manage 2FA</PageTitle>

<h1>Manage Two-factor Authentication</h1>
<hr />
<div class="row">
    <div class="col">
        @if (loading)
        {
            <p>Loading ...</p>
        }
        else
        {
            @if (twoFactorResponse is not null)
            {
                @foreach (var error in twoFactorResponse.ErrorList)
                {
                    <div class="alert alert-danger">@error</div>
                }
                @if (twoFactorResponse.IsTwoFactorEnabled)
                {
                    <div class="alert alert-success" role="alert">
                        Two-factor authentication is enabled for your
account.
                    </div>

                    <div class="m-1">
                        <button @onclick="Disable2FA" class="btn btn-lg btn-
primary">
                            Disable 2FA
                        </button>
                    </div>

                    @if (twoFactorResponse.RecoveryCodes is null)
                    {
                        <div class="m-1">
                            Recovery Codes Remaining:
                            @twoFactorResponse.RecoveryCodesLeft
                        </div>
                        <div class="m-1">
                            <button @onclick="GenerateNewCodes"
                                    class="btn btn-lg btn-primary">
                                Generate New Recovery Codes
                            </button>
                        </div>
                    }
                    else
                    {
                        <ShowRecoveryCodes
                            RecoveryCodes="twoFactorResponse.RecoveryCodes"
/>
                    }
                }
                else
                {
                    <h3>Configure authenticator app</h3>
                    <div>
                        <p>To use an authenticator app:</p>
                        <ol class="list">
                            <li>
                                <p>
                                    Download a two-factor authenticator app,
such
                                    as either of the following:
                                    <ul>
                                        <li>
                                            Microsoft Authenticator for
                                            <a
href="https://go.microsoft.com/fwlink/?Linkid=825072">
                                                Android
                                            </a> and
                                            <a
href="https://go.microsoft.com/fwlink/?Linkid=825073">
                                                iOS
                                            </a>
                                        </li>
                                        <li>
                                            Google Authenticator for
                                            <a
href="https://play.google.com/store/apps/details?
id=com.google.android.apps.authenticator2">
                                                Android
                                            </a> and
                                            <a
href="https://itunes.apple.com/us/app/google-authenticator/id388497605?
mt=8">
                                                iOS
                                            </a>
                                        </li>
                                    </ul>
                                </p>
                            </li>
                            <li>
                                <p>
                                    Scan the QR Code or enter this key
                                    <kbd>@twoFactorResponse.SharedKey</kbd>
into your
                                    two-factor authenticator app. Spaces and
casing
                                    don't matter.
                                </p>
                                <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>
                            </li>
                            <li>
                                <p>
                                    After you have scanned the QR code or
input the
                                    key above, your two-factor authenticator
app
                                    will provide you with a unique two-
factor code.
                                    Enter the code in the confirmation box
below.
                                </p>
                                <div class="row">
                                    <div class="col-xl-6">
                                        <EditForm Model="Input"
                                                FormName="send-code"

OnValidSubmit="OnValidSubmitAsync"
                                                method="post">
                                            <DataAnnotationsValidator />
                                            <div class="form-floating mb-3">
                                                <InputText
                                                    @bind-Value="Input.Code"
                                                    id="Input.Code"
                                                    class="form-control"
                                                    autocomplete="off"
                                                    placeholder="Enter the
code" />
                                                <label for="Input.Code"
                                                        class="control-label
form-label">
                                                    Verification Code
                                                </label>
                                                <ValidationMessage
                                                    For="() => Input.Code"
                                                    class="text-danger" />
                                            </div>
                                            <button type="submit"
                                                    class="w-100 btn btn-lg
btn-primary">
                                                Verify
                                            </button>
                                        </EditForm>
                                    </div>
                                </div>
                            </li>
                        </ol>
                    </div>
                }
            }
        }
    </div>
</div>

@code {
    private TwoFactorResponse twoFactorResponse = new();
    private bool loading = true;
    private string? svgGraphicsPath;

    [SupplyParameterFromForm]
    private InputModel Input { get; set; } = new();

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

    protected override async Task OnInitializedAsync()
    {
        twoFactorResponse = await Acct.TwoFactorRequestAsync(new());
        svgGraphicsPath = await GetQrCode(twoFactorResponse.SharedKey);
        loading = false;
    }

    private async Task<string> GetQrCode(string sharedKey)
    {
        if (authenticationState is not null &&
!string.IsNullOrEmpty(sharedKey))
        {
            var authState = await authenticationState;
            var email = authState?.User?.Identity?.Name!;
            var uri = string.Format(
                CultureInfo.InvariantCulture,
                "otpauth://totp/{0}:{1}?secret={2}&issuer={0}&digits=6",
                UrlEncoder.Default.Encode(Config["TotpOrganizationName"]!),
                email,
                twoFactorResponse.SharedKey);
            var qr = QrCode.EncodeText(uri, QrCode.Ecc.Medium);

            return qr.ToGraphicsPath();
        }

        return string.Empty;
    }

    private async Task Disable2FA()
    {
        await Acct.TwoFactorRequestAsync(new() { ForgetMachine = true });
        twoFactorResponse =
            await Acct.TwoFactorRequestAsync(new() { ResetSharedKey = true
});
```

```
        svgGraphicsPath = await GetQrCode(twoFactorResponse.SharedKey);
    }

    private async Task GenerateNewCodes()
    {
        twoFactorResponse =
            await Acct.TwoFactorRequestAsync(new() { ResetRecoveryCodes =
true });
    }

    private async Task OnValidSubmitAsync()
    {
        twoFactorResponse = await Acct.TwoFactorRequestAsync(
            new()
            {
                Enable = true,
                TwoFactorCode = Input.Code
            });
        Input.Code = string.Empty;

        // When 2FA is first enabled, recovery codes are returned.
        // However, subsequently disabling and re-enabling 2FA
        // leaves the existing codes in place and doesn't generate
        // a new set of recovery codes. The following code ensures
        // that a new set of recovery codes is generated each
        // time 2FA is enabled.
        if (twoFactorResponse.RecoveryCodes is null ||
            twoFactorResponse.RecoveryCodes.Length == 0)
        {
            await GenerateNewCodes();
        }
    }

    private sealed class InputModel
    {
        [Required]
        [RegularExpression(@"^([0-9]{6})$",
            ErrorMessage = "Must be a six-digit authenticator code
(######)")]
        [DataType(DataType.Text)]
        [Display(Name = "Verification Code")]
        public string Code { get; set; } = string.Empty;
    }
}
```

Link to the the Manage 2FA page

Add a link to the navigation menu for users to reach the `Manage2fa` component page.

In the `<Authorized>` content of the `<AuthorizeView>` in `Components/Layout/NavMenu.razor`, add the following markup:

diff

```
<AuthorizeView>
    <Authorized>

        ...

+       <div class="nav-item px-3">
+           <NavLink class="nav-link" href="manage-2fa">
+               <span class="bi bi-key" aria-hidden="true"></span> Manage
2FA
+           </NavLink>
+       </div>

        ...

    </Authorized>
</AuthorizeView>
```

Additional resources

- nimiq/qr-creator
- MapIdentityApi

Secure an ASP.NET Core Blazor WebAssembly standalone app with the Authentication library

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 explains how to secure an ASP.NET Core Blazor WebAssembly standalone app with the Blazor WebAssembly Authentication library.

The Blazor WebAssembly Authentication library (`Authentication.js`) only supports the Proof Key for Code Exchange (PKCE) authorization code flow via the Microsoft Authentication Library (MSAL, msal.js). To implement other grant flows, access the MSAL guidance to implement MSAL directly, but we don't support or recommend the use of grant flows other than PKCE for Blazor apps.

For Microsoft Entra (ME-ID) and Azure Active Directory B2C (AAD B2C) guidance, don't follow the guidance in this topic. See Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID or Secure an ASP.NET Core Blazor WebAssembly standalone app with Azure Active Directory B2C.

For additional security scenario coverage after reading this article, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Walkthrough

The subsections of the walkthrough explain how to:

- Register an app
- Create the Blazor app
- Run the app

Register an app

Register an app with an OpenID Connect (OIDC) Identity Provider (IP) following the guidance provided by the maintainer of the IP.

Record the following information:

- Authority (for example, `https://accounts.google.com/`).
- Application (client) ID (for example, `2...7-e...q.apps.googleusercontent.com`).
- Additional IP configuration (see the IP's documentation).

> **Note**
>
> The IP must use OIDC. For example, Facebook's IP isn't an OIDC-compliant provider, so the guidance in this topic doesn't work with the Facebook IP. For more information, see Secure ASP.NET Core Blazor WebAssembly.

Create the Blazor app

To create a standalone Blazor WebAssembly app that uses the Microsoft.AspNetCore.Components.WebAssembly.Authentication library, follow the guidance for your choice of tooling. If adding support for authentication, see the Parts of the app section of this article for guidance on setting up and configuring the app.

Visual Studio

To create a new Blazor WebAssembly project with an authentication mechanism:

After choosing the **Blazor WebAssembly App** template, set the **Authentication type** to **Individual Accounts**.

The **Individual Accounts** selection uses ASP.NET Core's Identity system. This selection adds authentication support and doesn't result in storing users in a database. The following sections of this article provide further details.

Configure the app

Configure the app following the IP's guidance. At a minimum, the app requires the `Local:Authority` and `Local:ClientId` configuration settings in the app's `wwwroot/appsettings.json` file:

JSON

```
{
  "Local": {
    "Authority": "{AUTHORITY}",
    "ClientId": "{CLIENT ID}"
  }
}
```

Google OAuth 2.0 OIDC example for an app that runs on the `localhost` address at port 5001:

```json
{
  "Local": {
    "Authority": "https://accounts.google.com/",
    "ClientId": "2...7-e...q.apps.googleusercontent.com",
    "PostLogoutRedirectUri": "https://localhost:5001/authentication/logout-
callback",
    "RedirectUri": "https://localhost:5001/authentication/login-callback",
    "ResponseType": "code"
  }
}
```

The redirect URI (`https://localhost:5001/authentication/login-callback`) is registered in the Google APIs console in **Credentials** > `{NAME}` > **Authorized redirect URIs**, where `{NAME}` is the app's client name in the **OAuth 2.0 Client IDs** app list of the Google APIs console.

Note

Supplying the port number for a `localhost` redirect URI isn't required for some OIDC IPs per the OAuth 2.0 specification. Some IPs permit the redirect URI for loopback addresses to omit the port. Others allow the use of a wildcard for the port number (for example, `*`). For additional information, see the IP's documentation.

Run the app

Use one of the following approaches to run the app:

- Visual Studio
  - Select the **Run** button.
  - Use **Debug** > **Start Debugging** from the menu.
  - Press `F5`.

- .NET CLI command shell: Execute the `dotnet watch` (or `dotnet run`) command from the app's folder.

Parts of the app

This section describes the parts of an app generated from the Blazor WebAssembly project template and how the app is configured. There's no specific guidance to follow in this section for a basic working application if you created the app using the guidance in the Walkthrough section. The guidance in this section is helpful for updating an app to authenticate and authorize users. However, an alternative approach to updating an app is to create a new app from the guidance in the Walkthrough section and moving the app's components, classes, and resources to the new app.

Authentication package

When an app is created to use Individual User Accounts, the app automatically receives a package reference for the Microsoft.AspNetCore.Components.WebAssembly.Authentication package. The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the Microsoft.AspNetCore.Components.WebAssembly.Authentication package to the app.

> **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.

Authentication service support

Support for authenticating users using OpenID Connect (OIDC) is registered in the service container with the AddOidcAuthentication extension method provided by the Microsoft.AspNetCore.Components.WebAssembly.Authentication package.

The AddOidcAuthentication method accepts a callback to configure the parameters required to authenticate an app using OIDC. The values required for configuring the app can be obtained from the OIDC-compliant IP. Obtain the values when you register the app, which typically occurs in their online portal.

For a new app, provide values for the `{AUTHORITY}` and `{CLIENT ID}` placeholders in the following configuration. Provide other configuration values that are required for use with the app's IP. The example is for Google, which requires `PostLogoutRedirectUri`, `RedirectUri`, and `ResponseType`. If adding authentication to an app, manually add the following code and configuration to the app with values for the placeholders and other configuration values.

In the `Program` file:

C#

```
builder.Services.AddOidcAuthentication(options =>
{
    builder.Configuration.Bind("Local", options.ProviderOptions);
});
```

`wwwroot/appsettings.json` configuration

Configuration is supplied by the `wwwroot/appsettings.json` file:

JSON

```
{
  "Local": {
    "Authority": "{AUTHORITY}",
    "ClientId": "{CLIENT ID}"
  }
}
```

Access token scopes

The Blazor WebAssembly template automatically configures default scopes for `openid` and `profile`.

The Blazor WebAssembly template doesn't automatically configure the app to request an access token for a secure API. To provision an access token as part of the sign-in flow, add the scope to the default token scopes of the OidcProviderOptions. If adding authentication to an app, manually add the following code and configure the scope URI.

In the `Program` file:

C#

```
builder.Services.AddOidcAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultScopes.Add("{SCOPE URI}");
});
```

For more information, see the following sections of the *Additional scenarios* article:

- Request additional access tokens
- Attach tokens to outgoing requests

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

razor

```razor
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {APPLICATION ASSEMBLY}
@using {APPLICATION ASSEMBLY}.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

HTML

```html
<script
src="_content/Microsoft.AspNetCore.Components.WebAssembly.Authentication/Aut
henticationService.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `App` component (`App.razor`) isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `App` component (`App.razor`) in the generated app.

- Inspect the `App` component (`App.razor`) in reference source. Select the version from the branch selector, and search for the component in the `ProjectTemplates` folder of the repository because it has moved over the years.

  > **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).

RedirectToLogin component

The `RedirectToLogin` component (`RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- The current URL that the user is attempting to access is maintained by so that they can be returned to that page if authentication is successful using:
  - Navigation history state in ASP.NET Core in .NET 7 or later.
  - A query string in ASP.NET Core in .NET 6 or earlier.

Inspect the `RedirectToLogin` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component.

> 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).

LoginDisplay component

The `LoginDisplay` component (`LoginDisplay.razor`) is rendered in the `MainLayout` component (`MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current user name.
  - Offers a link to the user profile page in ASP.NET Core Identity.
  - Offers a button to log out of the app.
- For anonymous users:
  - Offers the option to register.
  - Offers the option to log in.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `LoginDisplay` component isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `LoginDisplay` component in the generated app.

- Inspect the `LoginDisplay` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component. The templated content for `Hosted` equal to `true` is used.

  > 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).

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the Microsoft.AspNetCore.Components.WebAssembly.Authentication package.
- Manages performing the appropriate actions at each stage of authentication.

razor

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

<RemoteAuthenticatorView Action="@Action" />

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

ⓘ Note

Nullable reference types (NRTs) and .NET compiler null-state static analysis is supported in ASP.NET Core in .NET 6 or later. Prior to the release of ASP.NET Core in .NET 6, the `string` type appears without the null type designation (`?`).

Troubleshoot

Logging

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 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):

> eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ilg1ZVhrNHh5b2pORnVtMWtsMll0djhkbE5QNC1j ... bQdHBHGcQQRbW7Wmo6SWYG4V_bU55Ug_PW4pLPr20tTS8Ct7_uwy9DWrzCMzpD-EiwT5IjXwlGX3IXVjHIlX50IVIydBoPQtadvT7saKo1G5Jmutgq41o-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-c57dO6QGTVBwaNk"
}.{
  "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_signupsignin"
}.[Signature]
```

Additional resources

- ASP.NET Core Blazor WebAssembly additional security scenarios
- Unauthenticated or unauthorized web API requests in an app with a secure default client

- Configure ASP.NET Core to work with proxy servers and load balancers: Includes guidance on:
  - Using Forwarded Headers Middleware to preserve HTTPS scheme information across proxy servers and internal networks.
  - Additional scenarios and use cases, including manual scheme configuration, request path changes for correct request routing, and forwarding the request scheme for Linux and non-IIS reverse proxies.

Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Accounts

Article • 10/20/2024

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 create a standalone Blazor WebAssembly app that uses Microsoft Accounts with Microsoft Entra (ME-ID) for authentication.

For additional security scenario coverage after reading this article, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Walkthrough

The subsections of the walkthrough explain how to:

- Create a tenant in Azure
- Register an app in Azure
- Create the Blazor app
- Run the app

Create a tenant in Azure

Follow the guidance in Quickstart: Set up a tenant to create a tenant in ME-ID.

Register an app in Azure

Register an ME-ID app:

1. Navigate to **Microsoft Entra ID** in the Azure portal. Select **App registrations** in the sidebar. Select the **New registration** button.
2. Provide a **Name** for the app (for example, **Blazor Standalone ME-ID MS Accounts**).

3. In **Supported account types**, select **Accounts in any organizational directory (Any Microsoft Entra ID directory – Multitenant)**.
4. Set the **Redirect URI** dropdown list to **Single-page application (SPA)** and provide the following redirect URI: `https://localhost/authentication/login-callback`. If you know the production redirect URI for the Azure default host (for example, `azurewebsites.net`) or the custom domain host (for example, `contoso.com`), you can also add the production redirect URI at the same time that you're providing the `localhost` redirect URI. Be sure to include the port number for non-`:443` ports in any production redirect URIs that you add.
5. If you're using an unverified publisher domain, clear the **Permissions** > **Grant admin consent to openid and offline_access permissions** checkbox. If the publisher domain is verified, this checkbox isn't present.
6. Select **Register**.

Note

Supplying the port number for a `localhost` ME-ID redirect URI isn't required. For more information, see Redirect URI (reply URL) restrictions and limitations: Localhost exceptions (Entra documentation).

Record the Application (client) ID (for example, `00001111-aaaa-2222-bbbb-3333cccc4444`).

In **Authentication** > **Platform configurations** > **Single-page application**:

1. Confirm the redirect URI of `https://localhost/authentication/login-callback` is present.
2. In the **Implicit grant** section, ensure that the checkboxes for **Access tokens** and **ID tokens** aren't selected. **Implicit grant isn't recommended for Blazor apps using MSAL v2.0 or later.** For more information, see Secure ASP.NET Core Blazor WebAssembly.
3. The remaining defaults for the app are acceptable for this experience.
4. Select the **Save** button if you made changes.

Create the Blazor app

Create the app. Replace the placeholders in the following command with the information recorded earlier and execute the following command in a command shell:

.NET CLI

```
dotnet new blazorwasm -au SingleOrg --client-id "{CLIENT ID}" --tenant-id
"common" -o {PROJECT NAME}
```

Expand table

Placeholder	Azure portal name	Example
`{PROJECT NAME}`	—	`BlazorSample`
`{CLIENT ID}`	Application (client) ID	`00001111-aaaa-2222-bbbb-3333cccc4444`

The output location specified with the `-o|--output` option creates a project folder if it doesn't exist and becomes part of the project's name.

Add a pair of MsalProviderOptions for `openid` and `offline_access` DefaultAccessTokenScopes:

```C#
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("openid");
    options.ProviderOptions.DefaultAccessTokenScopes.Add("offline_access");
});
```

Run the app

Use one of the following approaches to run the app:

- Visual Studio
  - Select the **Run** button.
  - Use **Debug** > **Start Debugging** from the menu.
  - Press F5.
- .NET CLI command shell: Execute the `dotnet watch` (or `dotnet run`) command from the app's folder.

Parts of the app

This section describes the parts of an app generated from the Blazor WebAssembly project template and how the app is configured. There's no specific guidance to follow in this section for a basic working application if you created the app using the guidance in the Walkthrough section. The guidance in this section is helpful for updating an app to authenticate and authorize users. However, an alternative approach to updating an app is to create a new app from the guidance in the Walkthrough section and moving the app's components, classes, and resources to the new app.

Authentication package

When an app is created to use Work or School Accounts (`SingleOrg`), the app automatically receives a package reference for the Microsoft Authentication Library (Microsoft.Authentication.WebAssembly.Msal). The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the Microsoft.Authentication.WebAssembly.Msal package to the app.

> **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.

The Microsoft.Authentication.WebAssembly.Msal package transitively adds the Microsoft.AspNetCore.Components.WebAssembly.Authentication package to the app.

Authentication service support

Support for authenticating users is registered in the service container with the AddMsalAuthentication extension method provided by the Microsoft.Authentication.WebAssembly.Msal package. This method sets up all of the services required for the app to interact with the Identity Provider (IP).

In the `Program` file:

```
C#
```

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

The AddMsalAuthentication method accepts a callback to configure the parameters required to authenticate an app. The values required for configuring the app can be obtained from the ME-ID configuration when you register the app.

`wwwroot/appsettings.json` configuration

Configuration is supplied by the `wwwroot/appsettings.json` file:

JSON

```json
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/common",
    "ClientId": "{CLIENT ID}",
    "ValidateAuthority": true
  }
}
```

Example:

JSON

```json
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/common",
    "ClientId": "00001111-aaaa-2222-bbbb-3333cccc4444",
    "ValidateAuthority": true
  }
}
```

Access token scopes

The Blazor WebAssembly template doesn't automatically configure the app to request an access token for a secure API. To provision an access token as part of the sign-in flow, add the scope to the default access token scopes of the MsalProviderOptions:

C#

```csharp
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("{SCOPE URI}");
});
```

Specify additional scopes with `AdditionalScopesToConsent`:

C#

```csharp
options.ProviderOptions.AdditionalScopesToConsent.Add("{ADDITIONAL SCOPE
URI}");
```

ⓘ **Note**

AdditionalScopesToConsent 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.

For more information, see the following sections of the *Additional scenarios* article:

- Request additional access tokens
- Attach tokens to outgoing requests
- Quickstart: Configure an application to expose web APIs

Login mode

The framework defaults to pop-up login mode and falls back to redirect login mode if a pop-up can't be opened. Configure MSAL to use redirect login mode by setting the `LoginMode` property of MsalProviderOptions to `redirect`:

C#

```csharp
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.LoginMode = "redirect";
});
```

The default setting is `popup`, and the string value isn't case-sensitive.

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

razor

```razor
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
```

```
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {APPLICATION ASSEMBLY}
@using {APPLICATION ASSEMBLY}.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

HTML

```html
<script
src="_content/Microsoft.Authentication.WebAssembly.Msal/AuthenticationServic
e.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `App` component (`App.razor`) isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `App` component (`App.razor`) in the generated app.

- Inspect the `App` component (`App.razor`) in reference source. Select the version from the branch selector, and search for the component in the `ProjectTemplates` folder of the repository because it has moved over the years.

> 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).

RedirectToLogin component

The `RedirectToLogin` component (`RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- The current URL that the user is attempting to access is maintained by so that they can be returned to that page if authentication is successful using:
  - Navigation history state in ASP.NET Core in .NET 7 or later.
  - A query string in ASP.NET Core in .NET 6 or earlier.

Inspect the `RedirectToLogin` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component.

> 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).

LoginDisplay component

The `LoginDisplay` component (`LoginDisplay.razor`) is rendered in the `MainLayout` component (`MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current user name.
  - Offers a link to the user profile page in ASP.NET Core Identity.
  - Offers a button to log out of the app.

- For anonymous users:
  - Offers the option to register.
  - Offers the option to log in.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `LoginDisplay` component isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `LoginDisplay` component in the generated app.

- Inspect the `LoginDisplay` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component. The templated content for `Hosted` equal to `true` is used.

  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).

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the Microsoft.AspNetCore.Components.WebAssembly.Authentication package.
- Manages performing the appropriate actions at each stage of authentication.

razor

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

<RemoteAuthenticatorView Action="@Action" />

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

> Note
>
> Nullable reference types (NRTs) and .NET compiler null-state static analysis is supported in ASP.NET Core in .NET 6 or later. Prior to the release of ASP.NET Core in .NET 6, the `string` type appears without the null type designation (`?`).

Troubleshoot

Logging

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 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):

> eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ilg1ZVhrNHh5b2pORnVtMWtsMll0djhkbE5QNC1j ...
> bQdHBHGcQQRbW7Wmo6SWYG4V_bU55Ug_PW4pLPr20tTS8Ct7_uwy9DWrzCMzpD-EiwT5IjXwlGX3IXVjHIlX50IVIydBoPQtadvT7saKo1G5Jmutgq41o-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-c57dO6QGTVBwaNk"
}.{
  "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_signupsignin"
}.[Signature]
```

Additional resources

- 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
- ASP.NET Core Blazor WebAssembly with Microsoft Entra ID groups and roles
- Quickstart: Register an application with the Microsoft identity platform
- Quickstart: Configure an application to expose web APIs

Secure an ASP.NET Core Blazor WebAssembly standalone app with Microsoft Entra ID

Article • 10/20/2024

> ⓘ **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 create a standalone Blazor WebAssembly app that uses Microsoft Entra ID (ME-ID) for authentication.

For additional security scenario coverage after reading this article, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Walkthrough

The subsections of the walkthrough explain how to:

- Create a tenant in Azure
- Register an app in Azure
- Create the Blazor app
- Run the app

Create a tenant in Azure

Follow the guidance in Quickstart: Set up a tenant to create a tenant in ME-ID.

Register an app in Azure

Register an ME-ID app:

1. Navigate to **Microsoft Entra ID in the Azure portal**. Select **Applications** > **App registrations** in the sidebar. Select the **New registration** button.
2. Provide a **Name** for the app (for example, **Blazor Standalone ME-ID**).

3. Choose a **Supported account types**. You may select **Accounts in this organizational directory only** for this experience.
4. Set the **Redirect URI** dropdown list to **Single-page application (SPA)** and provide the following redirect URI: `https://localhost/authentication/login-callback`. If you know the production redirect URI for the Azure default host (for example, `azurewebsites.net`) or the custom domain host (for example, `contoso.com`), you can also add the production redirect URI at the same time that you're providing the `localhost` redirect URI. Be sure to include the port number for non-`:443` ports in any production redirect URIs that you add.
5. If you're using an unverified publisher domain, clear the **Permissions** > **Grant admin consent to openid and offline_access permissions** checkbox. If the publisher domain is verified, this checkbox isn't present.
6. Select **Register**.

ⓘ Note

Supplying the port number for a `localhost` ME-ID redirect URI isn't required. For more information, see Redirect URI (reply URL) restrictions and limitations: Localhost exceptions (Entra documentation).

Record the following information:

- Application (client) ID (for example, `00001111-aaaa-2222-bbbb-3333cccc4444`)
- Directory (tenant) ID (for example, `aaaabbbb-0000-cccc-1111-dddd2222eeee`)

In **Authentication** > **Platform configurations** > **Single-page application**:

1. Confirm the redirect URI of `https://localhost/authentication/login-callback` is present.
2. In the **Implicit grant** section, ensure that the checkboxes for **Access tokens** and **ID tokens** aren't selected. **Implicit grant isn't recommended for Blazor apps using MSAL v2.0 or later.** For more information, see Secure ASP.NET Core Blazor WebAssembly.
3. The remaining defaults for the app are acceptable for this experience.
4. Select the **Save** button if you made changes.

Create the Blazor app

Create the app in an empty folder. Replace the placeholders in the following command with the information recorded earlier and execute the command in a command shell:

.NET CLI

```
dotnet new blazorwasm -au SingleOrg --client-id "{CLIENT ID}" -o {PROJECT
NAME} --tenant-id "{TENANT ID}"
```

Expand table

Placeholder	Azure portal name	Example
`{PROJECT NAME}`	—	`BlazorSample`
`{CLIENT ID}`	Application (client) ID	`00001111-aaaa-2222-bbbb-3333cccc4444`
`{TENANT ID}`	Directory (tenant) ID	`aaaabbbb-0000-cccc-1111-dddd2222eeee`

The output location specified with the `-o|--output` option creates a project folder if it doesn't exist and becomes part of the project's name.

Add a MsalProviderOptions for `User.Read` permission with DefaultAccessTokenScopes:

C#

```csharp
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes
        .Add("https://graph.microsoft.com/User.Read");
});
```

Run the app

Use one of the following approaches to run the app:

- Visual Studio
  - Select the **Run** button.
  - Use **Debug** > **Start Debugging** from the menu.
  - Press F5.
- .NET CLI command shell: Execute the `dotnet watch` (or `dotnet run`) command from the app's folder.

Parts of the app

This section describes the parts of an app generated from the Blazor WebAssembly project template and how the app is configured. There's no specific guidance to follow in this section for a basic working application if you created the app using the guidance in the Walkthrough section. The guidance in this section is helpful for updating an app to authenticate and authorize users. However, an alternative approach to updating an app is to create a new app from the guidance in the Walkthrough section and moving the app's components, classes, and resources to the new app.

Authentication package

When an app is created to use Work or School Accounts (`SingleOrg`), the app automatically receives a package reference for the Microsoft Authentication Library (Microsoft.Authentication.WebAssembly.Msal). The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the Microsoft.Authentication.WebAssembly.Msal package to the app.

> **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.

The Microsoft.Authentication.WebAssembly.Msal package transitively adds the Microsoft.AspNetCore.Components.WebAssembly.Authentication package to the app.

Authentication service support

Support for authenticating users is registered in the service container with the AddMsalAuthentication extension method provided by the Microsoft.Authentication.WebAssembly.Msal package. This method sets up the services required for the app to interact with the Identity Provider (IP).

In the `Program` file:

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

The AddMsalAuthentication method accepts a callback to configure the parameters required to authenticate an app. The values required for configuring the app can be obtained from the ME-ID configuration when you register the app.

`wwwroot/appsettings.json` configuration

Configuration is supplied by the `wwwroot/appsettings.json` file:

JSON

```json
{
  "AzureAd": {
    "Authority": "https://login.microsoftonline.com/{TENANT ID}",
    "ClientId": "{CLIENT ID}",
    "ValidateAuthority": true
  }
}
```

Example:

JSON

```json
{
  "AzureAd": {
    "Authority":
"https://login.microsoftonline.com/e86c78e2-...-918e0565a45e",
    "ClientId": "00001111-aaaa-2222-bbbb-3333cccc4444",
    "ValidateAuthority": true
  }
}
```

Access token scopes

The Blazor WebAssembly template doesn't automatically configure the app to request an access token for a secure API. To provision an access token as part of the sign-in flow, add the scope to the default access token scopes of the MsalProviderOptions:

C#

```csharp
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("{SCOPE URI}");
});
```

Specify additional scopes with `AdditionalScopesToConsent`:

```C#
options.ProviderOptions.AdditionalScopesToConsent.Add("{ADDITIONAL SCOPE
URI}");
```

> ⓘ **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.

For more information, see the following resources:

- Request additional access tokens
- Attach tokens to outgoing requests
- Quickstart: Configure an application to expose web APIs
- Access token scopes for Microsoft Graph API

Login mode

The framework defaults to pop-up login mode and falls back to redirect login mode if a pop-up can't be opened. Configure MSAL to use redirect login mode by setting the `LoginMode` property of MsalProviderOptions to `redirect`:

```C#
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.LoginMode = "redirect";
});
```

The default setting is `popup`, and the string value isn't case-sensitive.

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

razor

```razor
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
@using {APPLICATION ASSEMBLY}
@using {APPLICATION ASSEMBLY}.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

HTML

```html
<script
src="_content/Microsoft.Authentication.WebAssembly.Msal/AuthenticationServic
e.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `App` component (`App.razor`) isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `App` component (`App.razor`) in the generated app.

- Inspect the `App` component (`App.razor`) in reference source. Select the version from the branch selector, and search for the component in the `ProjectTemplates` folder of the repository because it has moved over the years.

  > ⓘ 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).

RedirectToLogin component

The `RedirectToLogin` component (`RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- The current URL that the user is attempting to access is maintained by so that they can be returned to that page if authentication is successful using:
  - Navigation history state in ASP.NET Core in .NET 7 or later.
  - A query string in ASP.NET Core in .NET 6 or earlier.

Inspect the `RedirectToLogin` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component.

> ⓘ 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).

LoginDisplay component

The `LoginDisplay` component (`LoginDisplay.razor`) is rendered in the `MainLayout` component (`MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current user name.
  - Offers a link to the user profile page in ASP.NET Core Identity.
  - Offers a button to log out of the app.
- For anonymous users:
  - Offers the option to register.
  - Offers the option to log in.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `LoginDisplay` component isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `LoginDisplay` component in the generated app.

- Inspect the `LoginDisplay` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component. The templated content for `Hosted` equal to `true` is used.

  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).

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the Microsoft.AspNetCore.Components.WebAssembly.Authentication package.
- Manages performing the appropriate actions at each stage of authentication.

razor

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

<RemoteAuthenticatorView Action="@Action" />

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

ⓘ Note

Nullable reference types (NRTs) and .NET compiler null-state static analysis is supported in ASP.NET Core in .NET 6 or later. Prior to the release of ASP.NET Core in .NET 6, the `string` type appears without the null type designation (`?`).

Troubleshoot

Logging

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 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

```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):

> eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ilg1ZVhrNHh5b2pORnVtMWtsMll0
> djhkbE5QNC1j …
> bQdHBHGcQQRbW7Wmo6SWYG4V_bU55Ug_PW4pLPr20tTS8Ct7_uwy9DWrzCMzp
> D-EiwT5IjXwlGX3IXVjHIlX50IVIydBoPQtadvT7saKo1G5Jmutgq41o-dmz6-
> yBMKV2_nXA25Q

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

JSON

```json
{
  "typ": "JWT",
  "alg": "RS256",
  "kid": "X5eXk4xyojNFum1kl2Ytv8dlNP4-c57dO6QGTVBwaNk"
}.{
  "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_signupsignin"
}.[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
- ASP.NET Core Blazor WebAssembly with Microsoft Entra ID groups and roles
- Microsoft identity platform and Microsoft Entra ID with ASP.NET Core
- Microsoft identity platform documentation
- Security best practices for application properties in Microsoft Entra ID

Secure an ASP.NET Core Blazor WebAssembly standalone app with Azure Active Directory B2C

Article • 10/20/2024

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 create a standalone Blazor WebAssembly app that uses Azure Active Directory (AAD) B2C for authentication.

For additional security scenario coverage after reading this article, see ASP.NET Core Blazor WebAssembly additional security scenarios.

Walkthrough

The subsections of the walkthrough explain how to:

- Create a tenant in Azure
- Register an app in Azure
- Create the Blazor app
- Run the app

Create a tenant in Azure

Follow the guidance in Tutorial: Create an Azure Active Directory B2C tenant to create an AAD B2C tenant.

Before proceeding with this article's guidance, confirm that you've selected the correct directory for the AAD B2C tenant.

Register an app in Azure

Register an AAD B2C app:

1. Navigate to **Azure AD B2C** in the Azure portal. Select **App registrations** in the sidebar. Select the **New registration** button.
2. Provide a **Name** for the app (for example, **Blazor Standalone AAD B2C**).
3. For **Supported account types**, select the multi-tenant option: **Accounts in any organizational directory or any identity provider. For authenticating users with Azure AD B2C.**
4. Set the **Redirect URI** dropdown list to **Single-page application (SPA)** and provide the following redirect URI: `https://localhost/authentication/login-callback`. If you know the production redirect URI for the Azure default host (for example, `azurewebsites.net`) or the custom domain host (for example, `contoso.com`), you can also add the production redirect URI at the same time that you're providing the `localhost` redirect URI. Be sure to include the port number for non-`:443` ports in any production redirect URIs that you add.
5. If you're using an unverified publisher domain, confirm that **Permissions** > **Grant admin consent to openid and offline_access permissions** is selected. If the publisher domain is verified, this checkbox isn't present.
6. Select **Register**.

> ⓘ **Note**
>
> Supplying the port number for a `localhost` AAD B2C redirect URI isn't required. For more information, see Redirect URI (reply URL) restrictions and limitations: Localhost exceptions (Entra documentation).

Record the following information:

- Application (client) ID (for example, `00001111-aaaa-2222-bbbb-3333cccc4444`).
- AAD B2C instance (for example, `https://contoso.b2clogin.com/`, which includes the trailing slash): The instance is the scheme and host of an Azure B2C app registration, which can be found by opening the **Endpoints** window from the **App registrations** page in the Azure portal.
- AAD B2C Primary/Publisher/Tenant domain (for example, `contoso.onmicrosoft.com`): The domain is available as the **Publisher domain** in the **Branding** blade of the Azure portal for the registered app.

In **Authentication** > **Platform configurations** > **Single-page application**:

1. Confirm the redirect URI of `https://localhost/authentication/login-callback` is present.
2. In the **Implicit grant** section, ensure that the checkboxes for **Access tokens** and **ID tokens** aren't selected. **Implicit grant isn't recommended for Blazor apps using**

MSAL v2.0 or later. For more information, see Secure ASP.NET Core Blazor WebAssembly.
3. The remaining defaults for the app are acceptable for this experience.
4. Select the **Save** button if you made changes.

In **Home** > **Azure AD B2C** > **User flows**:

Create a sign-up and sign-in user flow

At a minimum, select the **Application claims** > **Display Name** user attribute to populate the `context.User.Identity?.Name` / `context.User.Identity.Name` in the `LoginDisplay` component (`Shared/LoginDisplay.razor`).

Record the sign-up and sign-in user flow name created for the app (for example, `B2C_1_signupsignin`).

Create the Blazor app

In an empty folder, replace the placeholders in the following command with the information recorded earlier and execute the command in a command shell:

.NET CLI

```
dotnet new blazorwasm -au IndividualB2C --aad-b2c-instance "{AAD B2C
INSTANCE}" --client-id "{CLIENT ID}" --domain "{TENANT DOMAIN}" -o {PROJECT
NAME} -ssp "{SIGN UP OR SIGN IN POLICY}"
```

Expand table

Placeholder	Azure portal name	Example
`{AAD B2C INSTANCE}`	Instance	`https://contoso.b2clogin.com/` (includes the trailing slash)
`{PROJECT NAME}`	—	`BlazorSample`
`{CLIENT ID}`	Application (client) ID	`00001111-aaaa-2222-bbbb-3333cccc4444`
`{SIGN UP OR SIGN IN POLICY}`	Sign-up/sign-in user flow	`B2C_1_signupsignin1`
`{TENANT DOMAIN}`	Primary/Publisher/Tenant domain	`contoso.onmicrosoft.com`

The output location specified with the `-o|--output` option creates a project folder if it doesn't exist and becomes part of the project's name.

Add a pair of MsalProviderOptions for `openid` and `offline_access` DefaultAccessTokenScopes:

```C#
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("openid");
    options.ProviderOptions.DefaultAccessTokenScopes.Add("offline_access");
});
```

After creating the app, you should be able to:

- Log into the app using an Microsoft Entra ID user account.
- Request access tokens for Microsoft APIs. For more information, see:
  - Access token scopes
  - Quickstart: Configure an application to expose web APIs.

Run the app

Use one of the following approaches to run the app:

- Visual Studio
  - Select the **Run** button.
  - Use **Debug** > **Start Debugging** from the menu.
  - Press F5.
- .NET CLI command shell: Execute the `dotnet watch` (or `dotnet run`) command from the app's folder.

Parts of the app

This section describes the parts of an app generated from the Blazor WebAssembly project template and how the app is configured. There's no specific guidance to follow in this section for a basic working application if you created the app using the guidance in the Walkthrough section. The guidance in this section is helpful for updating an app to authenticate and authorize users. However, an alternative approach to updating an app is to create a new app from the guidance in the Walkthrough section and moving the app's components, classes, and resources to the new app.

Authentication package

When an app is created to use an Individual B2C Account (`IndividualB2C`), the app automatically receives a package reference for the Microsoft Authentication Library (Microsoft.Authentication.WebAssembly.Msal). The package provides a set of primitives that help the app authenticate users and obtain tokens to call protected APIs.

If adding authentication to an app, manually add the Microsoft.Authentication.WebAssembly.Msal package to the app.

> ⓘ **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.

The Microsoft.Authentication.WebAssembly.Msal package transitively adds the Microsoft.AspNetCore.Components.WebAssembly.Authentication package to the app.

Authentication service support

Support for authenticating users is registered in the service container with the AddMsalAuthentication extension method provided by the Microsoft.Authentication.WebAssembly.Msal package. This method sets up all of the services required for the app to interact with the Identity Provider (IP).

In the `Program` file:

C#

```csharp
builder.Services.AddMsalAuthentication(options =>
{
    builder.Configuration.Bind("AzureAdB2C",
options.ProviderOptions.Authentication);
});
```

The AddMsalAuthentication method accepts a callback to configure the parameters required to authenticate an app. The values required for configuring the app can be obtained from the configuration when you register the app.

Configuration is supplied by the `wwwroot/appsettings.json` file:

JSON

```
{
  "AzureAdB2C": {
    "Authority": "{AAD B2C INSTANCE}{TENANT DOMAIN}/{SIGN UP OR SIGN IN
POLICY}",
    "ClientId": "{CLIENT ID}",
    "ValidateAuthority": false
  }
}
```

In the preceding configuration, the `{AAD B2C INSTANCE}` includes a trailing slash.

Example:

```JSON
{
  "AzureAdB2C": {
    "Authority":
"https://contoso.b2clogin.com/contoso.onmicrosoft.com/B2C_1_signupsignin1",
    "ClientId": "00001111-aaaa-2222-bbbb-3333cccc4444",
    "ValidateAuthority": false
  }
}
```

Access token scopes

The Blazor WebAssembly template doesn't automatically configure the app to request an access token for a secure API. To provision an access token as part of the sign-in flow, add the scope to the default access token scopes of the MsalProviderOptions:

```C#
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.DefaultAccessTokenScopes.Add("{SCOPE URI}");
});
```

Specify additional scopes with `AdditionalScopesToConsent`:

```C#
options.ProviderOptions.AdditionalScopesToConsent.Add("{ADDITIONAL SCOPE
URI}");
```

ⓘ 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.

For more information, see the following sections of the *Additional scenarios* article:

- Request additional access tokens
- Attach tokens to outgoing requests

Login mode

The framework defaults to pop-up login mode and falls back to redirect login mode if a pop-up can't be opened. Configure MSAL to use redirect login mode by setting the `LoginMode` property of MsalProviderOptions to `redirect`:

C#

```csharp
builder.Services.AddMsalAuthentication(options =>
{
    ...
    options.ProviderOptions.LoginMode = "redirect";
});
```

The default setting is `popup`, and the string value isn't case-sensitive.

Imports file

The Microsoft.AspNetCore.Components.Authorization namespace is made available throughout the app via the `_Imports.razor` file:

razor

```razor
@using System.Net.Http
@using System.Net.Http.Json
@using Microsoft.AspNetCore.Components.Authorization
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.AspNetCore.Components.Web.Virtualization
@using Microsoft.AspNetCore.Components.WebAssembly.Http
@using Microsoft.JSInterop
```

```
@using {APPLICATION ASSEMBLY}
@using {APPLICATION ASSEMBLY}.Shared
```

Index page

The Index page (`wwwroot/index.html`) page includes a script that defines the `AuthenticationService` in JavaScript. `AuthenticationService` handles the low-level details of the OIDC protocol. The app internally calls methods defined in the script to perform the authentication operations.

```HTML
<script
src="_content/Microsoft.Authentication.WebAssembly.Msal/AuthenticationServic
e.js"></script>
```

App component

The `App` component (`App.razor`) is similar to the `App` component found in Blazor Server apps:

- The AuthorizeRouteView component makes sure that the current user is authorized to access a given page or otherwise renders the `RedirectToLogin` component.
- The `RedirectToLogin` component manages redirecting unauthorized users to the login page.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `App` component (`App.razor`) isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `App` component (`App.razor`) in the generated app.

- Inspect the `App` component (`App.razor`) in reference source. Select the version from the branch selector, and search for the component in the `ProjectTemplates` folder of the repository because it has moved over the years.

> ⓘ 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).

RedirectToLogin component

The `RedirectToLogin` component (`RedirectToLogin.razor`):

- Manages redirecting unauthorized users to the login page.
- The current URL that the user is attempting to access is maintained by so that they can be returned to that page if authentication is successful using:
  - Navigation history state in ASP.NET Core in .NET 7 or later.
  - A query string in ASP.NET Core in .NET 6 or earlier.

Inspect the `RedirectToLogin` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component.

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).

LoginDisplay component

The `LoginDisplay` component (`LoginDisplay.razor`) is rendered in the `MainLayout` component (`MainLayout.razor`) and manages the following behaviors:

- For authenticated users:
  - Displays the current user name.
  - Offers a link to the user profile page in ASP.NET Core Identity.
  - Offers a button to log out of the app.
- For anonymous users:
  - Offers the option to register.
  - Offers the option to log in.

Due to changes in the framework across releases of ASP.NET Core, Razor markup for the `LoginDisplay` component isn't shown in this section. To inspect the markup of the component for a given release, use ***either*** of the following approaches:

- Create an app provisioned for authentication from the default Blazor WebAssembly project template for the version of ASP.NET Core that you intend to use. Inspect the `LoginDisplay` component in the generated app.

- Inspect the `LoginDisplay` component in reference source. The location of the component changed over time, so use GitHub search tools to locate the component. The templated content for `Hosted` equal to `true` is used.

  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).

Authentication component

The page produced by the `Authentication` component (`Pages/Authentication.razor`) defines the routes required for handling different authentication stages.

The RemoteAuthenticatorView component:

- Is provided by the Microsoft.AspNetCore.Components.WebAssembly.Authentication package.
- Manages performing the appropriate actions at each stage of authentication.

razor

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

<RemoteAuthenticatorView Action="@Action" />

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

ⓘ Note

Nullable reference types (NRTs) and .NET compiler null-state static analysis is supported in ASP.NET Core in .NET 6 or later. Prior to the release of ASP.NET Core in .NET 6, the `string` type appears without the null type designation (`?`).

Custom policies

The Microsoft Authentication Library (Microsoft.Authentication.WebAssembly.Msal, NuGet package) doesn't support AAD B2C custom policies.

Troubleshoot

Logging

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.