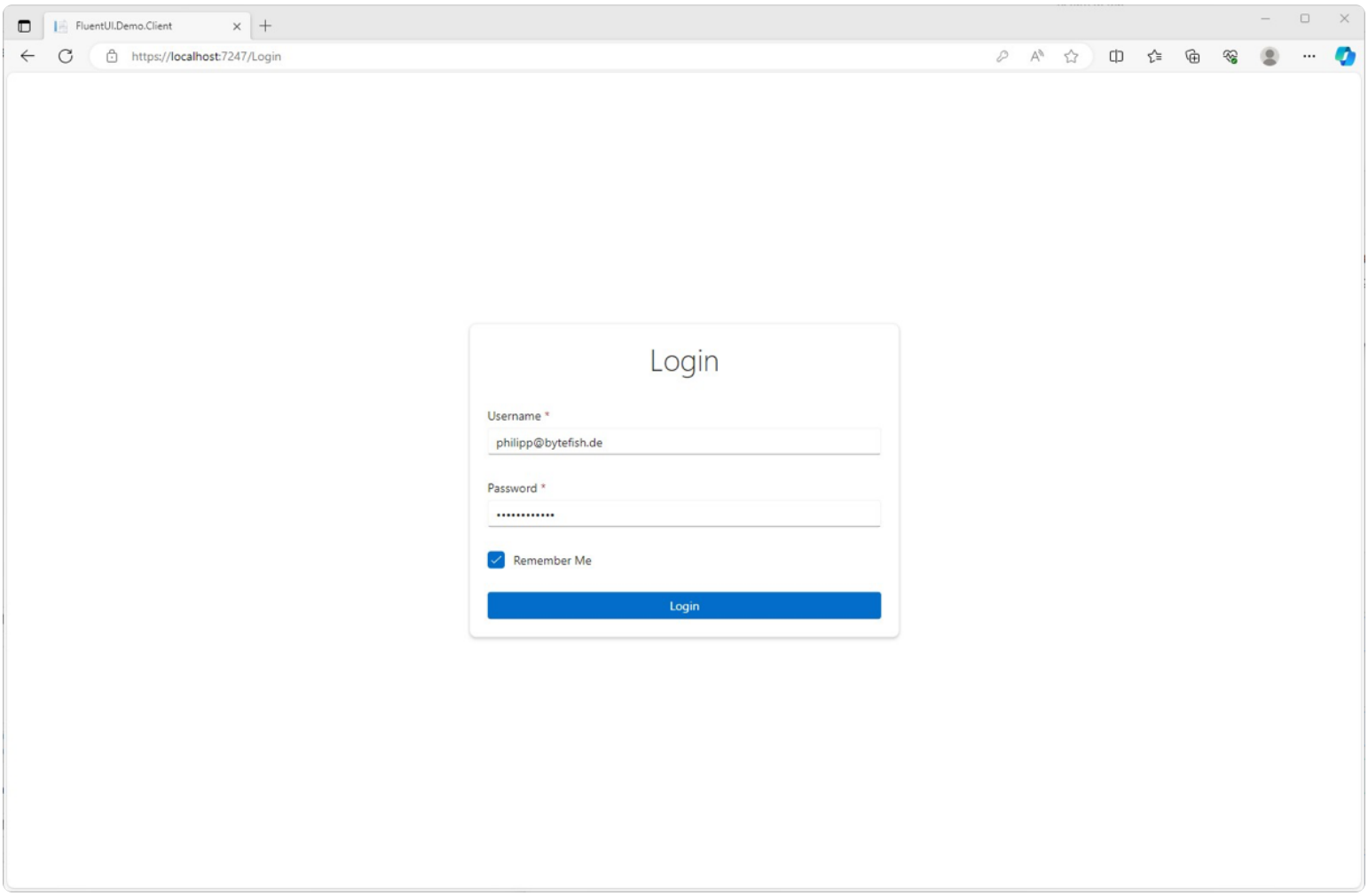
Blazor WebAssembly with Cookie Authentication

January 11, 2024 by Philipp Wagner

I've recently added Cookie Authentication to a Blazor WebAssembly application and there had been a lot of small parts to configure. I think it's a good idea to share my approach, so others can benefit.

At the end of the article you will have Cookie Authentication and a nice login form:



The code has been taken from the Git Repository at:

- https://github.com/bytefish/OpenFgaExperiments

Table of contents

Enabling Cookie Authentication in the ASP.NET Core Backend

In the Backend I've started by adding Cookie Authentication in the startup and override the `OnRedirectToLogin` event handlers, so they are going to return a `HTTP Status Code 401` to the consumer. This is handled in the Exception Handling Middleware and not shown here.

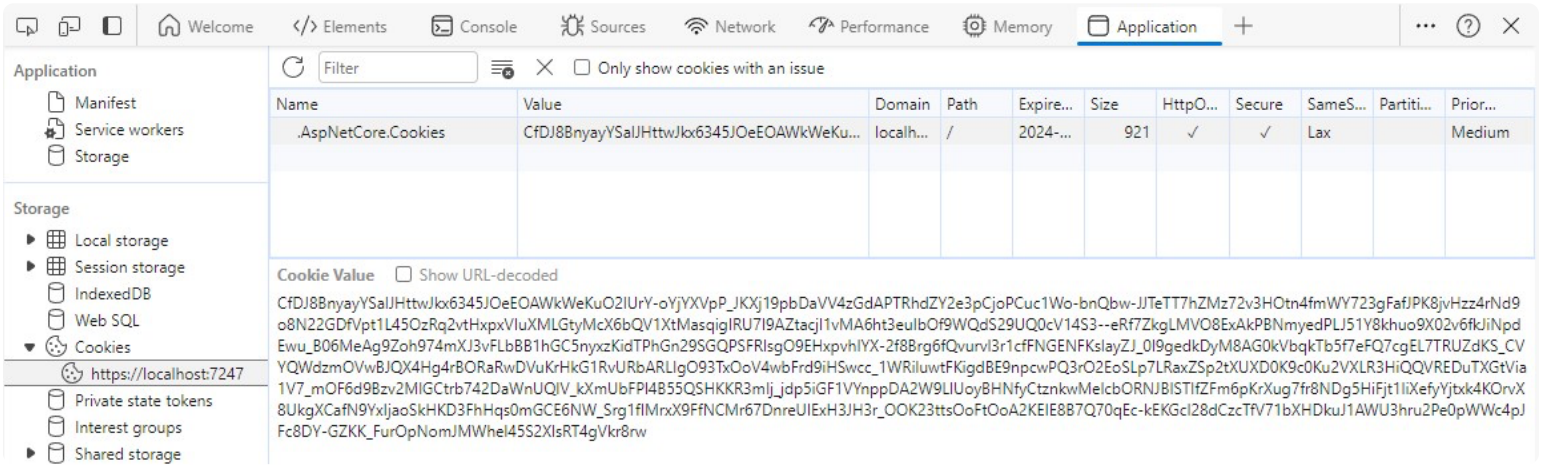
```
// Cookie Authentication
builder.Services
    .AddAuthentication(CookieAuthenticationDefaults.AuthenticationScheme)
    .AddCookie(options =>
    {
        options.Cookie.HttpOnly = true;
        options.Cookie.SameSite = SameSiteMode.Lax; // We don't want to deal with CSRF Tokens

        options.Events.OnRedirectToAccessDenied = (context) => throw new AuthenticationFailedEx
        options.Events.OnRedirectToLogin = (context) => throw new AuthenticationFailedException
    });
```

The user is signed in using `HttpContext#SignInAsync` with something along the lines of a `AuthenticationController`:

```
// Licensed under the MIT license. See LICENSE file in the project root for full license informa

// ...

namespace RebacExperiments.Server.Api.Controllers
{
    public class AuthenticationController : ODataController
    {
        // ...

        [HttpPost("odata/SignInUser")]
        public async Task<IActionResult> SignInUser([FromServices] IUserService userService, [F
        {
            // ...

            // Create the ClaimsPrincipal
            var claimsIdentity = new ClaimsIdentity(userClaims, CookieAuthenticationDefaults.Au
            var claimsPrincipal = new ClaimsPrincipal(claimsIdentity);
```

```
            // It's a valid ClaimsPrincipal, sign in
            await HttpContext.SignInAsync(claimsPrincipal, new AuthenticationProperties { IsPer
            // ...
        }
    }
}
```

You can then open your Browsers Developer Tools and see, that an (encrypted) Cookie has been created.



Enabling Cookie Authentication in Blazor WebAssembly

Once we have successfully logged in and got our Cookie, we need to send the Authorization Cookie on every request to the API. So we start by adding a `CookieDelegatingHandler`, that does just that:

```
// Licensed under the MIT license. See LICENSE file in the project root for full license informa

using Microsoft.AspNetCore.Components.WebAssembly.Http;
using RebacExperiments.Blazor.Shared.Logging;

namespace RebacExperiments.Blazor.Infrastructure
{
    public class CookieDelegatingHandler : DelegatingHandler
    {
        private readonly ILogger<CookieDelegatingHandler> _logger;

        public CookieDelegatingHandler(ILogger<CookieDelegatingHandler> logger)
        {
            _logger = logger;
```

```
        }

        protected override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage request
        {
            _logger.TraceMethodEntry();

            request.SetBrowserRequestCredentials(BrowserRequestCredentials.Include);

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

The `CookieDelegatingHandler` needs to be registered for the `HttpClient`, so we use the `IHttpClientBuilder#AddHttpMessageHandler` extension method like this:

```
builder.Services
    .AddHttpClient<IRequestAdapter, HttpClientRequestAdapter>(client => client.BaseAddress = new
    .AddHttpMessageHandler<CookieDelegatingHandler>();
```

The Blazor Authorization Infrastructure uses an `AuthenticationStateProvider` to pass the user information into the components. We want to persist the user information across page refreshes, so the local storage of a Browser seems to be a good place to persist it.

We don't need to take additional dependencies, just write a small `LocalStorageService`.

```
// Licensed under the MIT license. See LICENSE file in the project root for full license informa

using Microsoft.JSInterop;
using System.Text.Json;

namespace RebacExperiments.Blazor.Infrastructure
{
    public class LocalStorageService
    {
        private IJSRuntime _jsRuntime;

        public LocalStorageService(IJSRuntime jsRuntime)
        {
            _jsRuntime = jsRuntime;
        }

        public async Task<T?> GetItemAsync<T>(string key)
        {
```

```
            var json = await _jsRuntime.InvokeAsync<string>("localStorage.getItem", key);

            if (json == null)
            {
                return default;
            }

            return JsonSerializer.Deserialize<T>(json);
        }

        public async Task SetItem<T>(string key, T value)
        {
            await _jsRuntime.InvokeVoidAsync("localStorage.setItem", key, JsonSerializer.Serial
        }

        public async Task RemoveItemAsync(string key)
        {
            await _jsRuntime.InvokeVoidAsync("localStorage.removeItem", key);
        }
    }
}
```

And register it in the `Program.cs`.

```
// LocalStorage
builder.Services.AddSingleton<LocalStorageService>();
```

We can then implement an `AuthenticationStateProvider`, that allows us to set a `User` (think of User Profile) and notify subscribers about the new `AuthenticationState`. The `User` is persisted using our `LocalStorageService`.

```
// Licensed under the MIT license. See LICENSE file in the project root for full license inform

using Microsoft.AspNetCore.Components.Authorization;
using RebacExperiments.Shared.ApiSdk.Models;
using System.Security.Claims;

namespace RebacExperiments.Blazor.Infrastructure
{
    public class CustomAuthenticationStateProvider : AuthenticationStateProvider
    {
        private const string LocalStorageKey = "currentUser";

        private readonly LocalStorageService _localStorageService;
```

```
        public CustomAuthenticationStateProvider(LocalStorageService localStorageService)
        {
            _localStorageService = localStorageService;
        }

        public override async Task<AuthenticationState> GetAuthenticationStateAsync()
        {
            var currentUser = await GetCurrentUserAsync();

            if(currentUser == null)
            {
                return new AuthenticationState(new ClaimsPrincipal(new ClaimsIdentity()));
            }

            Claim[] claims = [
                new Claim(ClaimTypes.NameIdentifier, currentUser.Id!.ToString()!),
                new Claim(ClaimTypes.Name, currentUser.LogonName!.ToString()!),
                new Claim(ClaimTypes.Email, currentUser.LogonName!.ToString()!)
            ];

            var authenticationState = new AuthenticationState(new ClaimsPrincipal(new ClaimsIde

            return authenticationState;
        }

        public async Task SetCurrentUserAsync(User? currentUser)
        {
            await _localStorageService.SetItem(LocalStorageKey, currentUser);

            NotifyAuthenticationStateChanged(GetAuthenticationStateAsync());
        }

        public Task<User?> GetCurrentUserAsync() => _localStorageService.GetItemAsync<User>(Loc
    }
}
```

Don't forget to register all authentication related services.

```
// Auth
builder.Services.AddAuthorizationCore();
builder.Services.AddCascadingAuthenticationState();
builder.Services.AddSingleton<CustomAuthenticationStateProvider>();
builder.Services.AddSingleton<AuthenticationStateProvider>(s => s.GetRequiredService<CustomAuth
```

In the `App.razor` add the `CascadingAuthenticationState` and `AuthorizeRouteView` components, so the `AuthenticationState` flows down to the components automagically.

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

<FluentDesignTheme StorageName="theme" />
<CascadingAuthenticationState>
    <Router AppAssembly="@typeof(App).Assembly">
        <Found Context="routeData">
            <AuthorizeRouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)">
            </AuthorizeRouteView>
        </Found>
        <NotFound>
            <PageTitle>Not found</PageTitle>
            <LayoutView Layout="@typeof(MainLayout)">
                <p role="alert">Sorry, there's nothing at this address.</p>
            </LayoutView>
        </NotFound>
    </Router>
</CascadingAuthenticationState>
```

In the `MainLayout`, you can then use the `<AuthorizeView>` component, that allows to check, if a given user is authorized or not. If the User is not authorized, we are redirecting to the `Login` page using a `<RedirectToLogin>` component.

```
@using Microsoft.AspNetCore.Components
@using System.Runtime.InteropServices
@using RebacExperiments.Blazor.Components
@using RebacExperiments.Blazor.Components.RedirectToLogin
@namespace RebacExperiments.Blazor.Shared

<PageTitle>Relationship-based Experiments with ASP.NET Core OData</PageTitle>
<AuthorizeView>
    <Authorized>
        <!-- Authorized Main Content -->
    </Authorized>
    <NotAuthorized>
        <RedirectToLogin></RedirectToLogin>
    </NotAuthorized>
</AuthorizeView>
```

The `<RedirectToLogin>` component simply uses the `NavigationManager` to navigate to the Login Page.

```
@inject NavigationManager Navigation
```

```
@code {

    protected override void OnInitialized()
    {
        var baseRelativePath = Navigation.ToBaseRelativePath(Navigation.Uri);

        if(string.IsNullOrWhiteSpace(baseRelativePath))
        {
            Navigation.NavigateTo($"Login", true);
        } else
        {
            Navigation.NavigateTo($"Login?returnUrl={Uri.EscapeDataString(baseRelativePath)}", t
        }
    }
}
```

Now what happens, if the Web service returns a HTTP Status Code `401` (Unauthorized) and we still have the `User` in the Local Storage? Yes, it will be out of sync. So we need to update the `AuthenticationState` and clear the `User` information, if the service returns a HTTP Status Code `401`.

This can be done by using a `DelegatingHandler`, that takes a dependency on our `CustomAuthenticationStateProvider`, and sets the current `User` to `null`. This should inform all subscribers, that we are now unauthorized to perform actions.

```
// Licensed under the MIT license. See LICENSE file in the project root for full license informa

using RebacExperiments.Blazor.Shared.Logging;

namespace RebacExperiments.Blazor.Infrastructure
{
    public class UnauthorizedDelegatingHandler : DelegatingHandler
    {
        private readonly ILogger<UnauthorizedDelegatingHandler> _logger;

        private readonly CustomAuthenticationStateProvider _customAuthenticationStateProvider;

        public UnauthorizedDelegatingHandler(ILogger<UnauthorizedDelegatingHandler> logger, Cus
        {
            _logger = logger;
            _customAuthenticationStateProvider = customAuthenticationStateProvider;
        }

        protected override async Task<HttpResponseMessage> SendAsync(HttpRequestMessage request
        {
```

```
            _logger.TraceMethodEntry();

            var response = await base.SendAsync(request, cancellationToken);

            if (response.StatusCode == System.Net.HttpStatusCode.Unauthorized)
            {
                var currentUser = await _customAuthenticationStateProvider.GetCurrentUserAsync(

                if(currentUser != null)
                {
                    await _customAuthenticationStateProvider.SetCurrentUserAsync(null);
                }
            }

            return response;
        }
    }
}
```

You need to add the `UnauthorizedDelegatingHandler` to the `HttpClient` .

```
builder.Services
    .AddHttpClient<IRequestAdapter, HttpClientRequestAdapter>(client => client.BaseAddress = new
    .AddHttpMessageHandler<CookieDelegatingHandler>()
    .AddHttpMessageHandler<UnauthorizedDelegatingHandler>();
```

Now let's connect everything!

I want the Login Page to have its own layout and don't want to use the `MainLayout` . So I am adding an `<EmptyLayout>` component.

```
@inherits LayoutComponentBase

@Body
```

This `EmptyLayout` is then used as the Layout for the Login Page, so I can style it to my needs. The example uses a `<SimpleValidator>` for validation, that has been developed in a previous article. You could easily replace it with a `<DataAnnotationsValidator>` , to use Blazors built-in validations.

```
@page "/Login"
@layout EmptyLayout
```

```
@using RebacExperiments.Shared.ApiSdk

@inject ApiClient ApiClient
@inject IStringLocalizer<SharedResource> Loc
@inject NavigationManager NavigationManager
@inject CustomAuthenticationStateProvider AuthStateProvider

<div class="container">
    <FluentCard Width="500px">
        <EditForm Model="@Input" OnValidSubmit="SignInUserAsync" FormName="login_form" novalida
            <SimpleValidator TModel=InputModel ValidationFunc="ValidateInputModel" />
            <FluentValidationSummary />
            <FluentStack Orientation="Orientation.Vertical">
                <FluentGrid Spacing="3" Justify="JustifyContent.Center">
                    <FluentGridItem xs="12">
                        <h1>Login</h1>
                     </FluentGridItem>
                    <FluentGridItem xs="12">
                        <FluentTextField Name="login_eMail" Style="width: 100%" @bind-Value="In
                         <FluentValidationMessage For="@(() => Input.Email)" />
                     </FluentGridItem>
                     <FluentGridItem xs="12">
                         <FluentTextField Name="login_password" Style="width: 100%" TextFieldTy
                         <FluentValidationMessage For="@(() => Input.Password)" />
                     </FluentGridItem>
                     <FluentGridItem xs="12">
                         <FluentCheckbox Name="login_rememberMe" @bind-Value="Input.RememberMe"
                         <FluentValidationMessage For="@(() => Input.RememberMe)" />
                     </FluentGridItem>
                     <FluentGridItem xs="12">
                         <FluentButton Type="ButtonType.Submit" Appearance="Appearance.Accent" 
                     </FluentGridItem>
                     @if(!string.IsNullOrWhiteSpace(ErrorMessage)) {
                        <FluentGridItem xs="12">
                            <FluentMessageBar Style="width: 100%" Title=@ErrorMessage Intent="@
                            </FluentMessageBar>
                        </FluentGridItem>
                     }
                </FluentGrid>
            </FluentStack>
        </EditForm>
    </FluentCard>
</div>
```

Let's take a look at the `Login.razor.cs` Code-Behind.

The `Login#SignInUserAsync` methods starts by logging the User in. The Server will return the HttpOnly Cookie, that's going to be sent with every request to the API. To get the `User`

information for populating the `AuthenticationState` the `/Me` endpoint is called. The `User` is the set in the `AuthStateProvider` and we navigate to our application.

```
// Licensed under the MIT license. See LICENSE file in the project root for full license informa

using Microsoft.AspNetCore.Components;
using RebacExperiments.Shared.ApiSdk.Odata.SignInUser;
using System.ComponentModel.DataAnnotations;
using RebacExperiments.Blazor.Infrastructure;
using Microsoft.Extensions.Localization;

namespace RebacExperiments.Blazor.Pages
{
    public partial class Login
    {
        /// <summary>
        /// Data Model for binding to the Form.
        /// </summary>
        private sealed class InputModel
        {
            /// <summary>
            /// Gets or sets the Email.
            /// </summary>
            [Required]
            [EmailAddress]
            public required string Email { get; set; }

            /// <summary>
            /// Gets or sets the Password.
            /// </summary>
            [Required]
            [DataType(DataType.Password)]
            public required string Password { get; set; }

            /// <summary>
            /// Gets or sets the RememberMe Flag.
            /// </summary>
            [Required]
            public bool RememberMe { get; set; } = false;
        }

        // Default Values.
        private static class Defaults
        {
            public static class Philipp
            {
                public const string Email = "philipp@bytefish.de";
                public const string Password = "5!F25GbKwU3P";
```

```
        public const bool RememberMe = true;
    }

    public static class MaxMustermann
    {
        public const string Email = "max@mustermann.local";
        public const string Password = "5!F25GbKwU3P";
        public const bool RememberMe = true;
    }
}


/// <summary>
/// If a Return URL is given, we will navigate there after login.
/// </summary>
[SupplyParameterFromQuery(Name = "returnUrl")]
private string? ReturnUrl { get; set; }

/// <summary>
/// The Model the Form is going to bind to.
/// </summary>
[SupplyParameterFromForm]
private InputModel Input { get; set; } = new()
{
    Email = Defaults.Philipp.Email,
    Password = Defaults.Philipp.Password,
    RememberMe = Defaults.Philipp.RememberMe
};

/// <summary>
/// Error Message.
/// </summary>
private string? ErrorMessage;

/// <summary>
/// Signs in the User to the Service using Cookie Authentication.
/// </summary>
/// <returns></returns>
public async Task SignInUserAsync()
{
    ErrorMessage = null;

    try
    {
        await ApiClient.Odata.SignInUser.PostAsync(new SignInUserPostRequestBody
        {
            Username = Input.Email,
            Password = Input.Password,
            RememberMe = true
        });
```

```
            // Now refresh the Authentication State:
            var me = await ApiClient.Odata.Me.GetAsync();

            await AuthStateProvider.SetCurrentUserAsync(me);

            var navigationUrl = GetNavigationUrl();

            NavigationManager.NavigateTo(navigationUrl);
        }
        catch
        {
            ErrorMessage = Loc["Login_Failed"];
        }
    }

    private string GetNavigationUrl()
    {
        if(string.IsNullOrWhiteSpace(ReturnUrl))
        {
            return "/";
        }

        return ReturnUrl;
    }

    /// <summary>
    /// Validates an <see cref="InputModel"/>.
    /// </summary>
    /// <param name="model">InputModel to validate</param>
    /// <returns>The list of validation errors for the EditContext model fields</returns>
    private IEnumerable<ValidationError> ValidateInputModel(InputModel model)
    {
        if (string.IsNullOrWhiteSpace(model.Email))
        {
            yield return new ValidationError
            {
                PropertyName = nameof(model.Email),
                ErrorMessage = Loc.GetString("Validation_IsRequired", nameof(model.Email))
            };
        }

        if (string.IsNullOrWhiteSpace(model.Password))
        {
            yield return new ValidationError
            {
                PropertyName = nameof(model.Password),
                ErrorMessage = Loc.GetString("Validation_IsRequired", nameof(model.Password
            };
        }
```

```
        }
    }
}
```

In the `Login.razor.css` we add a bit of styling.

```
@keyframes fade {
    from {
        opacity: 0;
    }

    to {
        opacity: 1;
    }
}

.container {
    position: absolute;
    top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
    animation: fade 0.2s ease-in-out forwards;
}

h1 {
    font-size: 35px;
    font-weight: 100;
    text-align: center;
}
```

Conclusion

And that's it! You will now be able to use Cookie Authentication in your Blazor Application.

It would be interesting to see, how other people tackle Cookie Authentication in Blazor WebAssembly.

How to contribute

One of the easiest ways to contribute is to participate in discussions. You can also contribute by submitting pull requests.

General feedback and discussions?

Do you have questions or feedback on this article? Please create an issue on the Repositories issue tracker.

Something is wrong or missing?

There may be something wrong or missing in this article. If you want to help fixing it, then please make a Pull Request to this file.