Laboratorium nr 4

Temat: Tworzenie aplikacji Blazor

1 Szablon SSR static

Utworzenie nowego projektu Blazor SSR

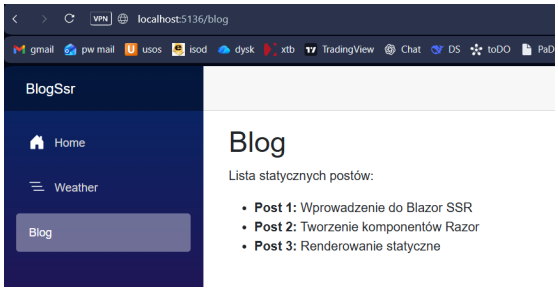
```
PS D:\stud\sem 5\Desktop\z4> dotnet new blazor -n BlogSsr --interactivity None
PS D:\stud\sem 5\Desktop\z4> cd BlogSsr
PS D:\stud\sem 5\Desktop\z4\BlogSsr> dotnet run
```

1.1 Dodanie strony blog

W folderze `Pages` dodano stronę `Blog.razor`:

```
@page "/blog"
<h1>Blog</h1>
<ul>
<li>Post 1: Wprowadzenie do Blazor SSR</li>
<li>Post 2: Tworzenie komponentów Razor</li>
<li>Post 3: Renderowanie statyczne</li>
</ul>
```

Po uruchomieniu aplikacji pod adresem `https://localhost:7045/blog` wyświetliła się lista postów.

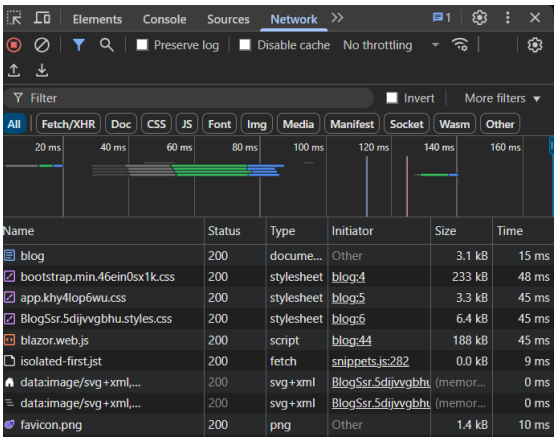


Rysunek 1: Widok działającej strony Blog

1.2 Zmierzenie czasu ładowania strony

Za pomocą narzędzi DevTools (zakładka Network) zmierzono czas pierwszego ładowania strony:

Wnioski: Strona renderuje się w pełni po stronie serwera (SSR), bez ładowania skryptów Blazora. Dzięki temu czas pierwszego renderowania jest minimalny.



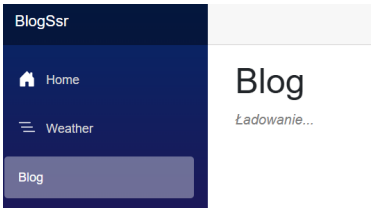
Rysunek 2: Pomiar czasu pierwszego ładowania (ok. 15ms lokalnie)

2 Stream Rendering

2.1 Dodanie delayu w Blog.razor

```
protected override async Task OnInitializedAsync()
{
 await Task.Delay(1500);
 posts = new()
 {
  "Post 1: Wprowadzenie do Blazor SSR",
  "Post 2: Komponenty Razor",
  "Post 3: Stream rendering (manualny)"
 };
}
```

2.2 Placeholder Ładowanie, który znika gdy lista się załaduje.



Rysunek 3: Widok komponentu Blog podczas ładowania (placeholder)

2.3 Porównajnie UX vs bez stream.

W wersji bez stream renderingu strona pozostaje pusta do czasu załadowania danych, co daje wrażenie opóźnienia.

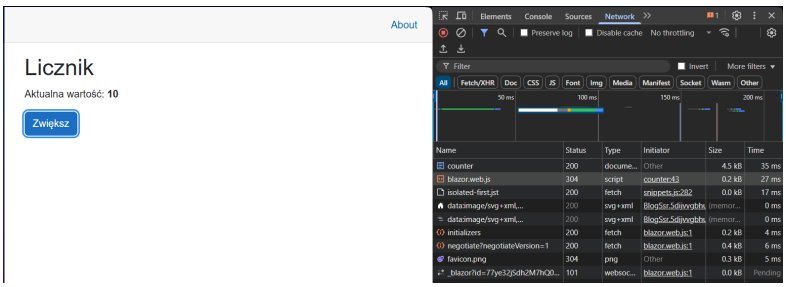
W wersji z stream renderingiem natychmiast wyświetla się placeholder „Ładowanie...", co poprawia wrażenia użytkownika i zwiększa interaktywność strony.

3 Dodanie interaktywności (Server)

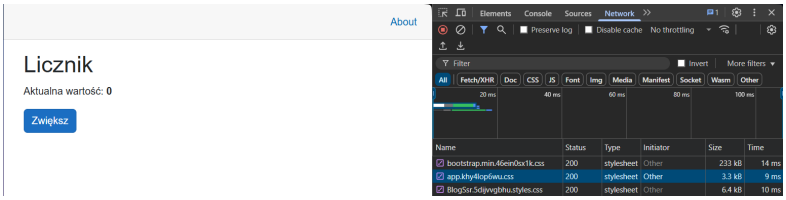
3.1 Utworzenie strony Counter.razor

```
@page "/counter"
@rendermode InteractiveServer

<PageTitle>Counter</PageTitle>

<h1>Licznik</h1>

<p>Aktualna warto Ż : <strong>@count</strong></p>

<button class="btn␣btn-primary" @onclick="IncrementCount">
    Zwi Źksz</button>

@code {
        private int count = 0;

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

Opis działania



Rysunek 4: Widok strony `/counter` z aktywnym połączeniem WebSocket — przycisk działa, licznik rośnie.



Rysunek 5: Widok strony `/counter` po usunięciu `@rendermode` — brak połączenia WebSocket, licznik nie reaguje.

Wnioski

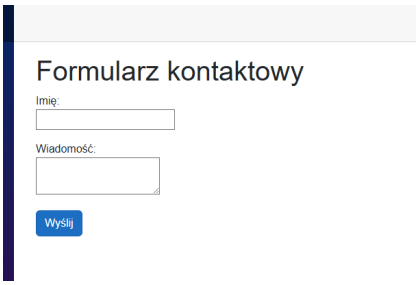
Tryb InteractiveServer umożliwia dynamiczną interakcję użytkownika z aplikacją Blazor poprzez utrzymywane połączenie WebSocket, co pozwala aktualizować widok bez przeładowania strony.

4 Formularz bez interaktywności

W tym zadaniu utworzono stronę Contact.razor działającą w trybie SSR (bez interaktywności). Formularz został zbudowany przy użyciu standardowego znacznika HTML `<form method="post">`, z obsługą atrybutu @formnam oraz adnotacji [SupplyParameterFromForm] w kodzie komponentu.

```
<form method="post" @formname="contactForm">
    <AntiforgeryToken />
    <input name="Name" />
    <textarea name="Message"></textarea>
    <button type="submit">Wyślij</button>
</form>
```

Dane wprowadzone do formularza są przekazywane do obiektu `ContactForm` po stronie serwera, a następnie renderowane ponownie po wykonaniu żądania POST. Po wysłaniu formularza wyświetlany jest komunikat: *„Dziękujemy za wiadomość!”* wraz z przesłanymi danymi.



Na zrzucie ekranu widać formularz kontaktowy.

Wnioski

Formularz w trybie SSR działa bez interaktywności Blazora (bez WebSocket). Odświeżenie strony po POST powoduje pełne renderowanie komponentu po stronie serwera, co jest prostsze, lecz mniej dynamiczne w porównaniu do wersji interaktywnej.

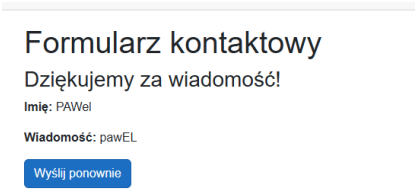
5 Migracja do interaktywnego formularza

W tym zadaniu zmigrowano formularz kontaktowy z wersji SSR (statycznej) do wersji interaktywnej, opartej na EditForm. Dzięki temu walidacja oraz obsługa zdarzeń odbywają się po stronie serwera, bez konieczności przeładowywania strony.

```
@page "/contact"
@rendermode InteractiveServer

<EditForm Model="@formData" OnValidSubmit="HandleSubmit">
<DataAnnotationsValidator />
<ValidationSummary />
<div>
<label>Imię:</label>
<InputText @bind-Value="formData.Name" />
<ValidationMessage For="@(() => formData.Name)" />
</div>
<div>
<label>Wiadomość:</label>
<InputTextArea @bind-Value="formData.Message" />
<ValidationMessage For="@(() => formData.Message)" />
```

```
</div>
<button type="submit">Wyślij</button>
</EditForm>
```

Do walidacji danych wykorzystano atrybuty DataAnnotations, np. [Required] oraz [StringLength], które automatycznie wyświetlają komunikaty o błędach. Po wysłaniu poprawnych danych użytkownikowi wyświetlany jest komunikat „Dziękujemy za wiadomość!.

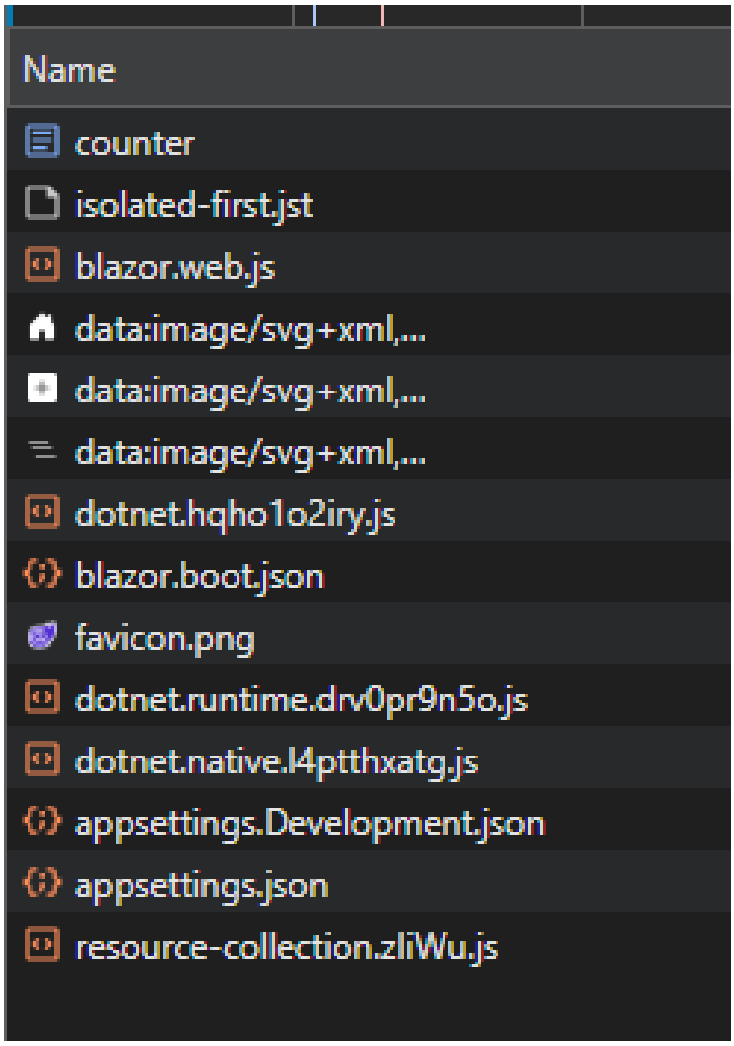


Wnioski

Przejście na EditForm pozwala na interaktywną obsługę formularza z wykorzystaniem połączenia WebSocket. Aplikacja nie wymaga odświeżania strony, a błędy walidacji są wyświetlane natychmiastowo, co znacząco poprawia wrażenia użytkownika.

6 Dodanie Client (WebAssembly)

Po przeniesieniu komponentu Counter do projektu Blazor WebAssembly:



Rysunek 6: Network - WASM

- Kluczowe pliki WASM:
  - **dotnet.wasm** – .NET runtime (2-3MB)
  - **dotnet.native.js** – JavaScript loader dla WASM
  - **dotnet.runtime.js** – runtime helper
  - **blazor.web.js** – loader frameworku Blazor
  - **blazor.boot.json** – konfiguracja bootstrapper'a
- Porównanie payload:

  - **BlogSsr/counter:** 50KB (HTML + CSS + mały JS)
  - **BlogWasmNew/counter:** 3-5MB (HTML + CSS + .wasm + DLL + JS)

- Wnioski:
  - Większy payload na początku ładowania (załadowanie całego .NET runtime)
  - Interaktywność działa w pełni po stronie klienta
  - Brak połączenia WebSocket – komunikacja z serwerem nie jest wymagana po załadowaniu

7 Model domenowy + EF Core

W tym zadaniu utworzono projekt BlogShared, w którym zdefiniowano model domenowy VideoGame oraz skonfigurowano Entity Framework Core wraz z migracjami i seedingiem przykładowych danych.

7.1 Klasa VideoGame

W katalogu BlogShared/Models dodano klasę VideoGame.cs:

```
namespace BlogShared.Models
{
 public class VideoGame
 {
  public int Id { get; set; }
  public string Title { get; set; } = string.Empty;
  public string Publisher { get; set; } = string.Empty;
  public int ReleaseYear { get; set; }
 }
}
```

7.2 DbContext

W katalogu BlogShared/Data utworzono klasę AppDbContext.cs:

```
using Microsoft.EntityFrameworkCore;
using BlogShared.Models;

namespace BlogShared.Data
{
 public class AppDbContext : DbContext
 {
  public AppDbContext(DbContextOptions<AppDbContext> options) : base(options) { }
```

```
  public DbSet<VideoGame> VideoGames => Set<VideoGame>();

  protected override void OnModelCreating(ModelBuilder modelBuilder)
  {
   base.OnModelCreating(modelBuilder);

   // Seeding przykładowych danych
   modelBuilder.Entity<VideoGame>().HasData(
   new VideoGame { Id = 1, Title = "The Legend of Zelda: Breath of the Wild", Publishe
   new VideoGame { Id = 2, Title = "Cyberpunk 2077", Publisher = "CD Projekt", Release
   new VideoGame { Id = 3, Title = "God of War", Publisher = "Sony", ReleaseYear = 201
   );
  }
 }
}
```

7.3 Konfiguracja w projekcie Server

W Program.cs projektu BlogServer dodano rejestrację DbContext oraz prosty endpoint zwracający wszystkie gry:

```
using BlogShared.Data;
using Microsoft.EntityFrameworkCore;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<AppDbContext>(options =>
options.UseSqlite("Data Source=videogames.db"));

var app = builder.Build();

app.MapGet("/games", async (AppDbContext db) => await db.VideoGames.ToListAsync());

app.Run();
```

7.4 Migracje i baza danych

Migracje utworzono w katalogu `BlogServer`:

```
dotnet ef migrations add InitialCreate --project ../BlogShared --startup-project .
dotnet ef database update --project ../BlogShared --startup-project .
```

Po uruchomieniu serwera dane są dostępne pod adresem `/games`.

7.5 Przykładowy wynik w przeglądarce

```
[{"id":1,"title":"The Legend of Zelda: Breath of the Wild","publisher":"Nintendo","rele
```

8 Warstwa usług.

8.1 Interfejs IGameService i implementacja GameService

W projekcie BlogShared utworzono warstwę usług dla zarządzania grami. Zdefiniowano interfejs IGameService oraz klasę GameService, która korzysta bezpośrednio z AppDbContext.

```
public interface IGameService
{
        Task<List<VideoGame>> GetAllAsync();
        Task<VideoGame?> GetByIdAsync(int id);
        Task<VideoGame?> UpdateAsync(int id, VideoGame game)
            ;
        Task<bool> DeleteAsync(int id);
}

public class GameService : IGameService
{
        private readonly AppDbContext _context;
        public GameService(AppDbContext context)
        {
                _context = context;
        }

        public async Task<List<VideoGame>> GetAllAsync() =>
        await _context.VideoGames.ToListAsync();

        public async Task<VideoGame?> GetByIdAsync(int id)
            =>
        await _context.VideoGames.FindAsync(id);

        public async Task<VideoGame> AddAsync(VideoGame game
            )
        {
                _context.VideoGames.Add(game);
                await _context.SaveChangesAsync();
                return game;
        }

        public async Task<VideoGame?> UpdateAsync(int id,
            VideoGame game)
        {
                var existing = await _context.VideoGames.
                    FindAsync(id);
                if (existing == null) return null;
                existing.Title = game.Title;
                existing.Publisher = game.Publisher;
                existing.ReleaseYear = game.ReleaseYear;
                await _context.SaveChangesAsync();
```

```
                return existing;
        }

        public async Task<bool> DeleteAsync(int id)
        {
                var game = await _context.VideoGames.
                    FindAsync(id);
                if (game == null) return false;
                _context.VideoGames.Remove(game);
                await _context.SaveChangesAsync();
                return true;
        }
}
```

8.2 Rejestracja usług w DI

W pliku Program.cs zarejestrowano warstwę usług i DbContext:

```
builder.Services.AddScoped<IGameService, GameService>();
builder.Services.AddDbContext<AppDbContext>(options =>
options.UseSqlite("Data Source=videogames.db"));
```

8.3 Minimal API / GameController

Stworzono kontroler REST API dla gier, obsługujący wszystkie operacje CRUD:

```
app.MapGet("/games", async (IGameService service) => await
    service.GetAllAsync());
app.MapGet("/games/{id}", async (IGameService service, int
    id) =>
await service.GetByIdAsync(id) is VideoGame game ? Results.
    Ok(game) : Results.NotFound());
app.MapPost("/games", async (IGameService service, VideoGame
     game) =>
Results.Created($"/games/{game.Id}", await service.AddAsync(
    game)));
app.MapPut("/games/{id}", async (IGameService service, int
    id, VideoGame game) =>
await service.UpdateAsync(id, game) is VideoGame updated ?
    Results.Ok(updated) : Results.NotFound());
app.MapDelete("/games/{id}", async (IGameService service,
    int id) =>
await service.DeleteAsync(id) ? Results.NoContent() :
    Results.NotFound());
```

8.4 Testowanie API

Do testów użyto pliku games.http w VS Code. Przykładowa odpowiedź z endpointu GET /games:

```
HTTP/1.1 200 OK
Connection: close
Content-Type: application/json; charset=utf-8
Date: Fri, 07 Nov 2025 16:26:00 GMT
Server: Kestrel
Transfer-Encoding: chunked

[
{
 "id": 2,
 "title": "Cyberpunk 2077",
 "publisher": "CD Projekt",
 "releaseYear": 2020
},
{
 "id": 3,
 "title": "God of War",
 "publisher": "Sony",
 "releaseYear": 2018
},
{
 "id": 4,
 "title": "Hollow Knight",
 "publisher": "Team Cherry",
 "releaseYear": 2017
}
]
```

9 Druga implementacja (HTTP Client)

9.1 Cel zadania

Celem było stworzenie klienta HTTP w projekcie Blazor WebAssembly (BlogWasmNew), który pobiera dane gier z API serwera BlogServer. Interfejs `IGameService` pozostaje ten sam, co w warstwie serwera, dzięki czemu komponenty Blazora nie wymagają zmian.

9.2 Implementacja ClientGameService

W katalogu `BlogWasmNew/Client/Services` utworzono klasę `ClientGameService.cs`:

```
using System.Net.Http.Json;
using BlogShared.Models;

public class ClientGameService : IGameService
{
        private readonly HttpClient _http;

        public ClientGameService(HttpClient http)
        {
                _http = http;
        }

        public async Task<List<VideoGame>> GetAllAsync() =>
        await _http.GetFromJsonAsync<List<VideoGame>>("/api/
            games") ?? new();

        public async Task<VideoGame?> GetByIdAsync(int id)
            =>
        await _http.GetFromJsonAsync<VideoGame>($"/api/games
            /{id}");

        public async Task<VideoGame?> UpdateAsync(int id,
            VideoGame game)
        {
                var response = await _http.PutAsJsonAsync($
                    "/api/games/{id}", game);
                return response.IsSuccessStatusCode ? game :
                     null;
        }

        public async Task<VideoGame?> AddAsync(VideoGame
            game)
        {
                var response = await _http.PostAsJsonAsync
                    ("/api/games", game);
                return response.IsSuccessStatusCode ? game :
                     null;
        }

        public async Task<bool> DeleteAsync(int id)
        {
                var response = await _http.DeleteAsync($"/
                    api/games/{id}");
                return response.IsSuccessStatusCode;
        }
}
```

9.3 Rejestracja w DI (warunkowo dla WASM)

W pliku `Program.cs` projektu Blazor WASM dodano wstrzykiwanie zależności:

```
#if WASM
builder.Services.AddScoped<IGameService, ClientGameService
    >();
#endif
```

Dzięki temu komponenty używające `IGameService` mogą działać bez zmian, niezależnie od tego, czy dane pochodzą z lokalnego DbContext (Server) czy z API (WASM).

9.4 Testowanie

Do testów użyto komponentu `GameList.razor` w projekcie Blazor WASM. Po uruchomieniu aplikacji i połączeniu z działającym serwerem BlogServer, metoda `GetAllAsync()` zwróciła listę gier w formacie JSON, identyczną jak w zadaniu 8.


```
[
{"id":1,"title":"The Legend of Zelda: Breath of the Wild","publisher":"Nintendo","releaseYear":2
{"id":2,"title":"Cyberpunk 2077","publisher":"CD Projekt","releaseYear":2020},
{"id":3,"title":"God of War","publisher":"Sony","releaseYear":2018}
]
```


9.5 Wnioski

- Interfejs `IGameService` umożliwia jednolitą obsługę danych zarówno po stronie serwera, jak i w kliencie WASM.
- Wstrzykiwanie `ClientGameService` do DI pozwala na łatwą wymianę źródła danych.
- Komponenty Blazor pozostają niezmienione, co zwiększa modularność i testowalność aplikacji.

10 Komponenty CRUD - Games i EditGame

Celem tego zadania było stworzenie interaktywnych komponentów Razor do zarządzania listą gier wideo (Create, Read, Update, Delete) z komunikacją z API serwera BlogServer.

10.1 Architektura

Aplikacja składa się z dwóch serwów:

- **BlogServer** (http://localhost:5021) – Minimal API z endpointami CRUD

- **BlogWasmNew** (http://localhost:5197) – Frontend Blazor Server z komponentami

10.2 Komponent Games.razor – Lista gier

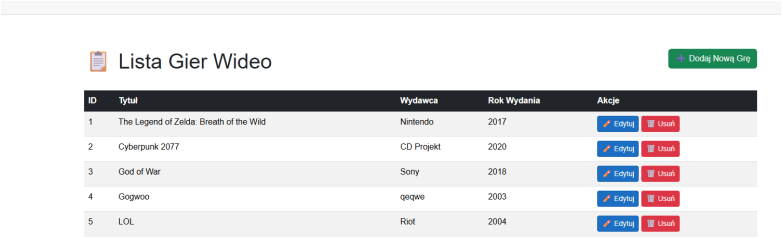
Komponent wyświetla tabelę wszystkich gier pobranych z API oraz umożliwia edycję i usunięcie rekordów.

10.2.1 Routing

```
@page "/games"
@rendermode InteractiveServer
@using Microsoft.JSInterop
```

Komponent działa w trybie `InteractiveServer`, co umożliwia interaktywne przyciski (Edytuj, Usuń) bez konieczności przeładowywania strony.

10.2.2 Interfejs użytkownika



Rysunek 7: Interfejs użytkownika

10.2.3 Logika komponentu

```
@code {
        private List<VideoGame>? games;
        private HttpClient http = null!;

        [Inject]
        protected HttpClient HttpClient { get; set; } = null
            !;

        [Inject]
        protected IJSRuntime JS { get; set; } = null!;

        protected override async Task OnInitializedAsync()
        {
                http = HttpClient;
                await LoadGames();
```

```
        }

        private async Task LoadGames()
        {
                try
                {
                        games = await http.GetFromJsonAsync<
                            List<VideoGame>>(
                                "http://localhost:5021/api/
                                    games");
                }
                catch (Exception ex)
                {
                        Console.WriteLine($"B Ć d
                             Ćadowania  gier: {ex.Message}");
                        games = new();
                }
        }

        private async Task DeleteGame(int id)
        {
                if (await ConfirmDelete())
                {
                        try
                        {
                                var response = await http.
                                    DeleteAsync(
                                $"http://localhost:5021/api/
                                    games/{id}");
                                if (response.
                                    IsSuccessStatusCode)
                                {
                                        await LoadGames();
                                }
                        }
                        catch (Exception ex)
                        {
                                Console.WriteLine($"B Ć d
                                    usuwania gry: {ex.Message
                                    }");
                        }
                }
        }

        private async Task<bool> ConfirmDelete()
        {
                return await JS.InvokeAsync<bool>("confirm",
                "Czy na pewno chcesz  u s u n    t Ź  gr Ź?");
        }
}
```

Opis działania:

- `OnInitializedAsync()` – Pobiera listę gier z API przy załadowaniu
- `LoadGames()` – Wykonuje HTTP GET na endpoint `/api/games`
- `DeleteGame(id)` – Wysyła HTTP DELETE i odświeża listę po powodzeniu
- `ConfirmDelete()` – Wyświetla dialog potwierdzenia JavaScript

10.2.4 Routing z parametrem opcjonalnym

```
@page "/games/edit"
@page "/games/edit/{id:int}"
@rendermode InteractiveServer
```

Parametr `id` jest opcjonalny. Jeśli nie jest podany, komponent wyświetla pusty formularz (tryb dodawania). Jeśli jest podany, pobiera dane gry z API i wypełnia formularz (tryb edycji).

10.2.5 Formularz

Formularz zawiera pola:

- **Tytuł Gry** – pole tekstowe wymagane
- **Wydawca** – pole tekstowe wymagane
- **Rok Wydania** – pole numeryczne wymagane

Przyciski:

- **Dodaj Grę / Zapisz Zmiany** – wysyła formularz
- **Anuluj** – wróć do listy gier
- **Spinner** – wyświetlany podczas wysyłania

Komunikaty:

- `errorMessage` – wyświetla błędy HTTP/sieciowe
- `successMessage` – potwierdza dodanie/edycję, następnie przechodzi do listy

10.3 Wnioski

- Komponenty **Games.razor** i **EditGame.razor** realizują pełny CRUD poprzez HTTP REST API
- Tryb **InteractiveServer** umożliwia dynamiczne przyciski bez przeładowania strony
- Parametr opcjonalny w routingu (`@page "/games/edit/{id:int}"`) umożliwia dwufunkcyjność komponentu
- **Komunikaty** sukcesu/błędu poprawiają UX
- **Dialog potwierdzenia** (JavaScript `confirm`) zapobiega przypadkowemu usunięciu danych
- **Odświeżanie listy** po operacjach CRUD utrzymuje stan aplikacji w synchronizacji z bazą danych
- Zastosowanie **HttpClient** z Dependency Injection pozwala na czystą komunikację klient-serwer

11 Pre-rendering

Pre-rendering to renderowanie komponentu na serwerze przed wysłaniem HTML do przeglądarki.

11.1 Z pre-renderingiem (domyślnie)

```
@page "/prerender-test"
@rendermode InteractiveServer
```

Działanie:

- Komponent renderuje się na serwerze
- Następnie na kliniecie po załadowaniu Blazzor
- Metody OnInitialized i OnInitializedAsync wykonywane 2 razy
- Szybsze LCP, lepsze dla SEO

11.2 Bez pre-renderingu

```
@page "/prerender-test"
@rendermode @(new InteractiveServerRenderMode(prerender: false))
```

Działanie:

- Serwer wysyła placeholder HTML

- Komponent renderuje się tylko na kliencie
- Metody lifecycle wykonywane 1 raz
- Wolniejsze LCP, ale mniej obciążenia serwera

12 Tryb Auto

`InteractiveAuto` adaptacyjnie wybiera rendering:

- Pierwszy raz: Server + WebSocket (SSR hydration)
- Po czyszczeniu cache: WebAssembly (bez WebSocketu, offline)

Kod

```
@page "/games"
@rendermode InteractiveAuto
```

Obserwacja

1. Load 1: DevTools → Network tab → `ws://` (WebSocket 101)
2. Load 2 (cache cleared): Brak WebSocketu, WASM renderuje lokalnie

Różnica: Server oszczędza bandwidth, WASM daje offline capability.